BOA VIEWS: ENABLING MODULARIZATION AND SHARING OF BOA QUERIES

Che Shian Hung

A Thesis

Submitted to the Graduate College of Bowling Green State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2019

Committee:

Robert Dyer, Advisor
Michael Decker
Robert C. Green II
Raymon Kresman
The Mining Software Repositories (MSR) field is well established, and recently has seen a focus on moving analysis techniques to a larger scale analyzing thousands of projects. Several tools exist to support these efforts, such as the Boa language and infrastructure. While Boa has seen extensive use by over one thousand users, in its current form it is not always possible to perform the entire analysis task within the infrastructure, often requiring some post-processing in another language. To address this problem, we use the notion of views from the relational database field and designed a language extension and runtime infrastructure in Boa that we call materialized views. Materialized views provide output reuse to Boa users, so that the results of prior Boa queries can be reused by users. This allows for computing results not previously possible within Boa and provides more sharing and reuse of MSR queries. To evaluate views, we performed two partial reproductions of prior MSR studies utilizing Boa’s dataset and infrastructure with Boa and compare our results to the prior studies. This shows the usability of the new infrastructure, allowing analyses in Boa that were not previously possible as well as providing a previously hand created gold dataset for identifier splitting as a reusable view for other MSR researchers. We also verified the caching behavior using the queries from one of the case studies. The results show that caching works as expected and can drastically improve the runtime performance.
ACKNOWLEDGMENTS

This research was supported in part by the U.S. National Science Foundation (NSF) under grants 1512947 and 1518776. Without their support, this research will not come true.

I would like to express special thanks to my advisor Robert Dyer, who not only had faith in me but also guided me along the way.

I am also grateful to other committee members Michael Decker, Robert Green, and Raymon Kresman, who always support me and encourage me during the course of this research.

I am thankful for all the support from other faculty members, my friends, and Bowling Green State University.

I want to thank to my family, who believes in me all the time.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 2</td>
<td>BACKGROUND AND MOTIVATION</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Overview of Boa</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Boa’s Query Language</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 MapReduce and Boa’s Runtime Infrastructure</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2.1.3 Boa’s Web Interface</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2.2 Problem: Supporting An End-to-end Mining Task</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2.3 Proposed Solution: Views</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>PROPOSED SOLUTION</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Directed Acyclic Graphs with Views</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>3.2 Views Syntax</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>3.2.1 Creating Sub-queries</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>3.2.2 Referencing Table</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>3.2.3 Table Filtering</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>3.2.4 Traverse Tables and Extract values from Rows</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>PROTOTYPE IMPLEMENTATION</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Type Resolution</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>4.2 Boa Compiler</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>4.2.1 Lexer and Parser</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>4.2.2 Type Checker</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>4.2.3 Output Protobuf</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>4.2.4 Code Generation</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>4.2.5 Static Tables</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>
4.3 Runtime

4.3.1 HDFS Layout 41

4.3.2 Oozie and Workflows 42

4.3.3 Cycles 47

CHAPTER 5 EVALUATION

5.1 Case Study 1: Splitting Identifiers 49

5.1.1 Boa Queries 50

5.1.2 Case Study Result 59

5.2 Case Study 2: Impact of Developer Turnover on Software Repositories 59

5.2.1 Boa Queries 60

5.2.2 Case Study Results 70

5.3 Performance Analysis on Caching Behavior 76

5.3.1 Analysis Method 76

5.3.2 Analysis Result 77

CHAPTER 6 RELATED WORKS

6.1 Database Views 80

6.2 Repository Mining Frameworks 82

CHAPTER 7 FUTURE WORK

7.1 Language Extensions 87

7.2 Runtime Optimizations 90

7.3 Boa Website Improvements 91

CHAPTER 8 CONCLUSION

BIBLIOGRAPHY

95
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example of Boa query that counts the number of times files appear in bug-fixing revisions.</td>
</tr>
<tr>
<td>2.2</td>
<td>The process of the MapReduce program when the example query in 2.1 is executed.</td>
</tr>
<tr>
<td>2.3</td>
<td>The screenshot of writing Boa query from Figure 2.1 on Boa web interface</td>
</tr>
<tr>
<td>2.4</td>
<td>The screenshot of the job description after clicking Run Program button from Figure 2.3</td>
</tr>
<tr>
<td>2.5</td>
<td>A set of procedures to find the source code that contain potential bugs through bug-fixing revisions without using views. The first two steps can be performed in Boa, but the rest of the steps have to be performed with external analysis tools.</td>
</tr>
<tr>
<td>2.6</td>
<td>By utilizing views in Boa, any complex MSR research questions can be solved solely with Boa.</td>
</tr>
<tr>
<td>3.1</td>
<td>An invalid data flow graph that containing a loop among program nodes.</td>
</tr>
<tr>
<td>3.2</td>
<td>A DAG to find the buggy files by examining past bug fixing behavior.</td>
</tr>
<tr>
<td>3.3</td>
<td>A Boa query implementing the DAG in Figure 3.2 to find buggy files.</td>
</tr>
<tr>
<td>3.4</td>
<td>An Boa query revised from Figure 3.3. The view Churn is defined in the scope of Filter as nested views.</td>
</tr>
<tr>
<td>3.5</td>
<td>A diagram that shows the structure of external view paths. The name of the components are listed at the top with color mapping. The first path references the table with job number. The second path references the table with username and tag name.</td>
</tr>
</tbody>
</table>
3.6 An Boa query revised from Figure 3.3, assuming the view FixingRevision is defined as another Boa job with job id 12345 and tag name FixingRevision assigned by user rdyer. ........................................ 19

3.7 A diagram that visualizes the change of tables after filterings. ........................................ 21

4.1 A sequence diagram shows the processes of type resolution and the interaction between poller and Boa compiler. ........................................ 24

4.2 An example of storing the sub-queries into trees. Given a Boa query at the left, the relationship between SubView nodes can be established like the trees at the right hand side. The dashed edges are used to link the trees to the main query. ........... 29

4.3 A diagram extended from Figure 3.7. When the last column is filtered, instead of returning an empty table, an array of EmptyTuple will be returned instead. .......... 31

4.4 The protobuf structure for Boa output. ........................................ 32

4.5 An example of visualizing Boa output in protobuf format from Figure 4.4. The example output type is defined at the top, and the tree at the bottom shows the protobuf structure for the rows. ........................................ 33

4.6 An example of generating jar file for each query in a Boa job. ................................. 34

4.7 An example Boa query that is used to demonstrate the changes we made to generate code to support views. The query itself does not serve any particular purpose but demonstrating code generation. ........................................ 36

4.8 The tuple definition generated from line 2 in Figure 4.7. ........................................ 37

4.9 The tuple definition generated from line 5 in Figure 4.7. ........................................ 37

4.10 An example Boa code that traverses three static tables. The table comes from the output variable o in job 12345. Each table is applied with a string filter. ........ 40

4.11 The generated Java code for the Boa code from Figure 4.10. ........................................ 40

4.12 The original Boa jobs layout on HDFS. ........................................ 42

4.13 The new output layout on HDFS, assuming job id 23456 represents the Boa job showed in Figure 3.4. ........................................ 43
4.14 The generated workflow DAG for the main query from Figure 3.4. The workflow contains two sub-workflow actions due to referencing tables from queries Filter and FixingRevision. 44

4.15 The generated workflow DAG for query Filter from Figure 3.4. The workflow contains one sub-workflow action due to referencing a table from the query FixingRevision. 45

4.16 The generated workflow DAG for query FixingRevision from Figure 3.4. The workflow contains no sub-workflow action because the query does not reference any tables. 46

4.17 An example directed dependency graph. The initial dependency relationship is shown with the solid edge. After changing view name Foo from Q1 to Q4 and creating query Q5, the dashed edges are added to the dependency relationship, which turns out containing a cycle among Q2, Q3 and Q4. To simplify the DAG, the data flow edges between queries and Boa dataset are omitted. 47

4.18 An example directed dependency graph with a cycle among the sub-queries sv1, sv2, and sv3. The text 12345 represents the job id of the query. To simplify the DAG, the data flow edges between queries and Boa dataset are omitted. 48

5.1 The DAG for splitting identifiers from gold set. The color is used to differentiate different query files, and the shape icons are used to differentiate different output variables. The outputs participate the data flow are marked as bold. 51

5.2 The Boa query file GoldSet.boa for case study 1, which import Binkley's gold set into Boa by emitting each entry to three output variables. The query job is assigned with job id 100. 52

5.3 The Boa query file WordLists.boa for case study 1, which import the word lists into Boa by emitting each entry to the corresponding output variable. The query job is assigned with job id 200. 54
5.4 The Boa query file **GreedySplit.boa** for case study 1. The query applies greedy algorithm along with three word lists to the identifiers from gold set. .......................... 55

5.5 The Boa query file **BoaIdentifiers.boa** for case study 1. The query collects the identifiers for each corresponding types from Boa dataset. The query job is assigned with job id **300**. ......................................................... 57

5.6 The Boa query file **BoaGreedySplit.boa** for case study 1. The query collects the identifiers and the corresponding types from Boa dataset. ................................. 58

5.7 The DAG for reproducing Foucault’s research on open source projects. The three queries on the left hand side first filters out the small projects. The query **Turnover-Metrics** then computes the turnover metrics for the remaining projects. The three queries at the right hand side reference and analyze the turnover metrics to answer the research questions. ................................. 61

5.8 The Boa query file **ProjectCommitTime.boa** for case study 2. The query collects the latest and the earliest commit time for each project. The query job is assigned with job id **1000**. ......................................................... 62

5.9 The Boa query file **ProjectDeveloperFilter.boa** for case study 2. The query collects the projects id based on the number of people contributing to the project. The query is assigned with job id **2000**. ......................................................... 62

5.10 The Boa query file **TurnoverProjectFilter.boa** for case study 2. The query filters out the small projects from the case study. This query job is assigned with job id **3000**. ......................................................... 63

5.11 The Boa query file **TurnoverMetrics.boa** for case study 2. The query computes the turnover metrics from the retaining projects. This query job is assigned with job id **4000**. ......................................................... 65

5.12 The Boa query file **TurnoverRQ1.boa** for case study 2. The query identifies the significance of turnover by computing the ratio of externals across projects’ history and finding the conversion rate for each project. ................................................. 67
5.13 The Boa query file TurnoverRQ2.boa for case study 2. The query studies the patterns the contribution from persistent, internal and external developers by computing the ratio of each turnover activities to total activities.

5.14 The Boa query file TurnoverRQ3.boa for case study 2. The query computes the correlation and the confidence intervals between turnover activity metrics and the density of bug-fixed commits.

5.15 The number of projects for the ratios of external newcomers external leavers throughout projects' life time.

5.16 The number of projects for each conversion rate level.

5.17 The number of projects for each highest activity metric. Each category contains the number of projects that have the highest total activity metric throughout project life times.

5.18 The number of projects for the higher newcomer activities and higher leaver activities.

5.19 The number of projects having positive correlation with bug-fixing revisions for each activity metric.

5.20 The number of projects having negative correlation with bug-fixing revisions for each activity metric.

5.21 The DAG for the first caching scenario in case study 1. The gray nodes represent the caching queries for the experiment.

5.22 The DAG for the second caching scenario in case study 1. The gray nodes represent the caching queries for the experiment.

7.1 An example of Boa query that references a view with AST type Project to find the top five largest projects have ten committers.

7.2 An example of Boa query declaring and using an argument variable to count the number of commits over years.

7.3 An example of Boa query referencing Figure 7.2 several times and set the argument variable to various languages.
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>A table of new tokens added to the lexer. The second column shows the definition for each token, in EBNF format.</td>
</tr>
<tr>
<td>4.2</td>
<td>A table of new and modified AST nodes in the parser. The second column shows the grammar for each AST node, in EBNF format. The first four AST nodes are new. For node <code>Index</code>, the second alternate was added to support wildcards to filter tables.</td>
</tr>
<tr>
<td>4.3</td>
<td>A table of major modifications on AST nodes during type checking phase. The second column briefly describes the changes for each node in type checking visitor.</td>
</tr>
<tr>
<td>4.4</td>
<td>The generated variable declaration Java code for the example query from Figure 4.7.</td>
</tr>
<tr>
<td>4.5</td>
<td>The generated Java code in the mapper function for the example query from Figure 4.7.</td>
</tr>
<tr>
<td>5.1</td>
<td>The runtime performance for each caching scenario. The runtime performance is recorded in seconds, and the standard deviation is round to the third decimal point.</td>
</tr>
<tr>
<td>5.2</td>
<td>The runtime performance for each query in case study 1. The runtime performance is recorded in seconds, and the standard deviation is round to the third decimal point.</td>
</tr>
</tbody>
</table>
CHAPTER 1 INTRODUCTION

In the Mining Software Repository (MSR) field, project and software artifacts, such as source code, are mined to discover useful insights and support software development and the project team. Currently, there are many tools designed to help MSR researchers more easily mine open source projects, such as Boa [11–13], GHTorrent [16], Sourcerer [27], and PyDriller [36]. Among these existing tools, Boa is the only tool that provides its own domain specific query language, a web-interface built for easily submitting queries and viewing results, and ultra-large-scale datasets based on several open-source software repositories such as GitHub and SourceForge. Boa’s datasets consist of snapshots of these software repositories. The queries are automatically transformed into distributed Hadoop MapReduce [9] programs and run on a small cluster. The language provides features such as visitors to allow users to more easily analyze the source code stored in the repositories, by walking the Abstract Syntax Trees (AST). Boa programs are designed to analyze a single project at a time, and send data to an aggregator for the final output.

Despite its benefits, Boa has some limitations. For example, if a query needs to use the result of a prior global analysis (e.g., the average/median/etc across all projects in the corpus) users have only two options. They can either take the result of the prior query and hard code it into a second Boa query, or they can use another language/tool and post-process the result of the first query. This limits the ability to use Boa as an end-to-end analysis tool. It also has the side effect of limiting reuse across users.

To address this problem, we introduce the notion of materialized views into the Boa language and infrastructure. A view in the relational database world is the result of a stored query. A materialized view is a static snapshot of a view, cached by the database for performance reasons. Views can be used in future queries and shared across users. A materialized view in Boa is very similar in concept: it is the statically cached result of a prior Boa query, which is re-usable in future queries. We provide a simple language extension to support the notion of declaring and using views in the query language and extend the runtime infrastructure. The runtime relies on
the open-source Apache Oozie [22] workflow scheduler. Each user-defined view is transpiled into an Apache Hadoop MapReduce program. The compiler generates workflows based on the dependencies among the views and schedules them with Oozie. If a view already has cached output, the workflow skips executing it.

To evaluate views in Boa, we partially reproduced two prior MSR studies [15, 20] using Boa’s new view features and compare our results to the prior works. We also evaluate the caching behavior to ensure caching works as expected, and execution times are faster with caching enabled. The results of these evaluations show the new infrastructure for views works and is useful for creating real-world MSR analyses. The views created for the studies are shareable with the community, thus demonstrating the reusability of a view.

In the next chapter, we introduce Boa and motivate the problem via a simple example. In Chapter 3 we propose our solution called views, and work through an example problem solved using the new view syntax. In Chapter 4 we explain details of the prototype implementation of views in the Boa compiler and runtime infrastructure. We then partially reproduce two case studies to show the usefulness of the new language features in Chapter 5. Prior research related to views and MSR mining tools are discussed in Chapter 6. Several future extensions are discussed in Chapter 7. Finally, we conclude the paper in Chapter 8.
CHAPTER 2 BACKGROUND AND MOTIVATION

In this chapter, we first introduce background information about the Boa language and infrastructure [11–13]. We briefly discuss the syntax of the Boa language, its runtime execution model relying on MapReduce [9], and the Boa web interface. Finally, we discuss limitations with the Boa infrastructure and motivate a proposed solution to the problem.

2.1 Overview of Boa

In this section we give an overview of Boa’s language and infrastructure, by first looking at the query language via a simple example and then investigating how that query executes.

2.1.1 Boa’s Query Language

The Boa language supports five primitive types: int, float, bool, string, and time. To allow users to handle different types of data and analysis, Boa also provides four compound types: array, map, set, and stack. For each Boa query, there must be at least one output variable, and each output variable always comes with an aggregator. Some aggregator examples in Boa are sum, set, collection, mean, maximum, and top. For the aggregators like maximum and top, users have to assign a weight for each output value and the aggregators will compute the result based on the weights. Boa also supports user-defined functions to provide code re-usability for the users. To mine project repository metadata and source code, Boa allows the users to create visitors to traverse through the repository metadata and abstract syntax tree (AST) from the source code. The users can also utilize the keywords before and after in the visit statements to manipulate the execution order of the statements.

An example Boa query is shown in Figure 2.1, which counts the number of times files appear in bug-fixing revisions. The input for the query is defined in line 1, which is given a variable name p and a Boa AST type Project. The output for the query is declared with an aggregator sum in line 2. Boa outputs consist of a set of key-value pairs, and the type of key-value pairs is defined in the output type definition. In the example, the key contains two indices with type string,
Figure 2.1 An example of Boa query that counts the number of times files appear in bug-fixing revisions.

and the type of the values is int. This means that the output will be a map of distinct keys to an integer value. A visitor example is shown from line 3 to line 8. The users can specify actions for the specific AST types in the visitor. In line 4, the keyword before causes Revision visit statement to execute before the node and its children are visited. Inside the visit statement contains an if statement, checking if the current revision is a bug-fixing revision. If the check returns true, we iterate the source files in the revision and count each file through an emit-statement. Each emission creates a key-value pair. In this case, the first string index in the emit statement captures the project id, and the second index denotes the file name. An integer 1 is sent to the output variable FixingFileCount, counting the file has appeared once.

2.1.2 MapReduce and Boa’s Runtime Infrastructure

MapReduce is a programming model for batch processing large datasets using a distributed cluster. The model contains four steps: splitting input, mapping, shuffling, and reducing. In the first step, the input is split into records and each record provided as input to a mapper function. Users provide custom mapper functions that take a single record, process it, and output zero or more key-value pairs. After mappers generate output, the shuffle phase occurs where the key-value pairs are sorted by key, then grouped together based on keys, and each unique key and all its associated values given as input to a reducer function. In the last step, the reducer function aggregates the data and produces key-value pairs as the final output.
Figure 2.2 The process of the MapReduce program when the example query in [2.1] is executed. The Splitting and Mapping columns represent input/output for the map function (provided by the user). The Shuffling and Reducing columns represent input/output for the reduce function (also provided by the user). The remainder of the process is handled by the MapReduce framework.

Once a Boa query is submitted, the query will be turned into a Boa job associated with a unique job number. During the compile time, the query will be translated into a MapReduce program in Java. The whole Boa query will be turned into a map operation, and the aggregators from the output variables will be used in the reducer in the reduce phase. In Boa, the MapReduce program will be executed on a Hadoop cluster.

Figure 2.2 demonstrates the process of the MapReduce program based on the query in [2.1] is executed. The goal of the query is to count the number of times files appear in bug-fixing revisions. The input consists of project snapshots. In the splitting phase, the inputs are split in the project base, and each mapper would work on a project from the dataset, producing a key-value pair for each emit statement. In the example, the key consists of a project id and a file name, and the value is a number 1. In the shuffling phase, the pairs with the same key are grouped together and sent to the reducers. Each reducer will apply a given aggregator \texttt{sum} to the pairs, summing up the numbers for the file and produces a new key-value pair. At the end of the process, the output generator writes each pair into a row and append all output from the reducers into one output file.

The MapReduce model contains high scalability and provides an efficient solution of data mining. Due to the fact that MapReduce operates on a distributed system, it can store and distribute
large amount of data among the cluster. MapReduce is efficient because each server on the cluster operates in parallel. Despite of the communication overhead between servers, as long as the task is not distributed to too many servers, MapReduce can process terabytes of data within minutes.

2.1.3 Boa’s Web Interface

Figure 2.3 The screenshot of writing Boa query from Figure 2.1 on Boa web interface

For users’ usability and accessibility, Boa provides a web interface for the users to run Boa query and observe the outputs. The website also provides documentation for Boa language and example queries for the users to start with. Whenever the user is about to submit a query, the user can select a specific dataset for the query. A screenshot of writing a Boa query on the web interface is shown in Figure 2.3. The query is from Figure 2.1 and the dataset for the query is set to the smaller version of snapshots from GitHub on September 2015. The Boa datasets are collected and generated from GitHub and SourceForge. The latest dataset contains different sizing options for testing purpose. For instance, the medium size contains one tenth number of projects from the full size, and the small size contains 0.2 percent of full dataset.

Once the user clicks the Run Program button, the query is then sent to Boa compiler and evaluator. Figure 2.4 shows the job status after running the query in Figure 2.3. As we can see,
the query has been successfully compiled and executed. The execution time is also given after the finished time. Each Boa job will be linked to the creator’s user account. The creator of the query can also control the accessibility of the job to either public or private through the web interface. When the job is public, other users can access the job and view the job status and output through the job URL. For every executed Boa job, the job is set to private in default setting. In the job status page in Figure 2.4, the user can click Make Public and change the accessibility to public. Once a job is set to public, the user can share the job with others through the URL link. If the user wants to view or download the output, he or she can click the tabs at the top of Figure 2.4 to do so. For
each executed Boa jobs, the output will be saved on the cluster, so the users can always view the output or even download the output from the website.

2.2 Problem: Supporting An End-to-end Mining Task

Even though Boa provides a great platform for the researchers to mine software repositories and answer MSR research questions, Boa is limited and insufficient to be used to solve more complex research questions. Due to the fact that each Boa query is turned into a MapReduce program, Boa will not help much if the research question requires several mapping or reducing operations to be resolved. Even if the users can write multiple Boa queries to run multiple MapReduce programs, not only the current input choices are limited, the structure and the format of the outputs has no re-usability in Boa. The users can only run the first MapReduce program in Boa, and then using other data analysis tools to perform further analysis on the query outputs.

Many of the research questions contain assumptions and required filtering. One possible MSR research question is to **find the buggy files by examining past bug fixing behavior** [31]. Bug-fixing revisions are the revisions that fix bugs. The solution can help the company or the project team to identify potential buggy source code for extra code review. To solve the problem, one solution is to filter out the projects we are not interested in, and then find the source code files appear the most among the bug-fixing revisions.

To solve this research question, the whole process can be seen in Figure 2.5. The steps are described as follow:

**Step 1a** Compute the fixing file counts, fixing revision counts (FRC), and average FRC across projects

**Step 1b** Download Boa output through Boa API

**Step 2a** Filter projects based on MSR criteria

**Step 2b** Download Boa output through Boa API

**Step 3a** Generate list of potentially buggy files

**Step 3b** Output result
In Step 1a, a Boa query is created to compute three outputs: fixing file counts, fixing revision counts (FRC), and average FRC. Fixing file count is the number of times the source files appeared in bug-fixing revisions. FRC is the number of bug-fixing revision in the project. Average FRC is the mean of FRC across projects. Each metric can be computed with one MapReduce program. In Step 2a, another Boa query is used to filter out unwanted projects, for the researchers might not be interested in all projects from the dataset. Since the researchers need to perform post-processing analysis tasks on queries’ results, they will most likely utilize Boa’s client API to download outputs, so that the researchers can process them with external tools like Python in Step 3a. This can
be a tedious step because the researchers have to manually merge and process Boa outputs, and eventually generate final result in Step 3b.

Among the steps, besides Step 1a and 2a, the rest of the steps are done outside of Boa framework. Besides using Boa, the researchers have to utilize other analysis tools to perform several steps just to solve a research question. The process can be time consuming and very painful. Even though the problem can be solved in other ways, it still requires external tools to solve the problem, not to mention if the research question requires more map and reduce operations to be solved. The researchers might need to put more effort to manually merge more Boa outputs. This points out that Boa fails to provide end-to-end analysis for the users.

2.3 Proposed Solution: Views

To tackle the problem, we introduce a new feature, materialized views, in Boa. Our goal for the feature is to allow the researchers to be able to resolve any MSR research questions solely with Boa. The concept of views comes from relational database, which allows the users to reuse previous query results. We want to apply the same idea to Boa, so that Boa users can reuse previous query results in a new Boa query. The views are materialized due to the nature of Boa runtime infrastructure, which automatically caches query results in HDFS. In addition, we want to support sub-queries in a Boa query, so that users can execute multiple MapReduce programs in one query instead of creating a Boa job for each query.

Figure 2.6 shows the process to solve a complex research question with views. We want the researchers to be able to resolve any complex MSR research questions solely with Boa. Once the query result becomes reusable, they can execute Boa query one after another, and each query might contain multiple sub-queries and reuse previous query results. The last step is marked blue because there is no need to perform further analysis on query outputs with external tools, so the researchers can just download the output results through Boa web interface. Since a query can support multiple sub-queries, we can always reduce the number $N$ from Figure 2.6 to one. Compare to Figure 2.5, the views feature should ease much pain during experiment. We also believe through reusing Boa outputs, Boa can become an end-to-end analysis tool for the researchers in MSR field. In the next
Figure 2.6 By utilizing views in Boa, any complex MSR research questions can be solved solely with Boa.

chapter, we will discuss our design and general approach to our solution.
CHAPTER 3 PROPOSED SOLUTION

In this chapter, we illustrate the design of views. Before going further, we would like to explain what exactly is a view and clarify the terms.

In the last chapter, we mentioned that a Boa query, or a Boa job, can contain multiple sub-queries. The query in the out most scope is the main query. A view refers to a Boa query, which contains one or more output variables. A table is a output produced from an output variable in a view. When a output is reused in other Boa query, we say the query is referencing a table from a view. Next, we will first introduce directed acyclic graphs (DAGs) and explain how DAG can be used to solve the problem with views, and then we will discuss the syntax for views.

3.1 Directed Acyclic Graphs with Views

![Diagram of Boa queries and dataset]

Figure 3.1 An invalid data flow graph that containing a loop among program nodes.

To have Boa become an end-to-end analysis tool, not only we have to support reusing outputs, we also have to handle the query dependencies at runtime. Whenever a table from query $X$ is referenced in query $Y$, a dependency relationship is established between two queries. If the table in query $X$ is missing, query $X$ must be executed again to generate the table before running query...
If somehow query $X$ fails to generate the table, query $Y$ should abort immediately.

The dependencies can be visualized in a directed graph. However, a valid view-referencing relationship must not contain any cycles. One invalid data flow graph is shown in Figure 3.1. Each node in the graph is a Boa query. The edges are the data flow between queries. For instance, query 2 references a table from query 1. A table in query 2 is referenced in query 3, and query 1 references a table from query 3. During runtime, each query would wait on the output from another query, forming an infinite loop. Therefore, to have a valid data flow graph, the dependency graph must be a directed acyclic graph (DAG).

Figure 3.2 A DAG to find the buggy files by examining past bug fixing behavior.

In the last chapter, we brought up an example research question, pointing out the current limitation in Boa. The goal of the example MSR question is to find the buggy files by examining past bug fixing behavior. To solve the question in Boa with views, only three Boa queries are needed. The corresponding data flow DAG is illustrated in Figure 3.2. The first Boa query contains
three outputs, which is derived from step 1a in Figure 2.5. The second query references two tables from query 1 to filter projects, and the output table is the retained projects. The third query reuses fixing-file-count table from query 1 and the retained-projects table from query 2 to rank the source files for the retained projects, which produces the final output for the research question. When each query is executed, the dataset is fed into the program as well. The data flows for the dataset are denoted with dashed directed edges in Figure 3.2. Through reusing outputs, we can create DAGs among Boa queries, helping Boa become an end-to-end analysis tool for the researchers. In the next section, we will explain new language features for views.

3.2 Views Syntax

In this section, we would like to introduce new syntax and new operations for views. During the design phase, we referred to several query languages such as SQL, R, and Spark. We strove to eliminate unnecessary view operations, minimizing the learning curve for both new and old users. Continued from the DAG we created for the research questions, Figure 3.3 shows the implementation of the DAG with new Boa syntax. The query contains two sub-queries. Each query’s color maps to a node in Figure 3.2. The blue portion is the first sub-query computing file count and revision metadata. The red sub-query uses metadata and revision count to filter out unwanted projects. The green portion shows the main query, which ranks the file count for the retained projects. Next, we will discuss the view operations and its usage.

3.2.1 Creating Sub-queries

Since a view represents a Boa query, there are two ways to create views. One is simply writing a Boa query, which becomes the main program of the query. The second approach is creating a sub-query. Line 1 and 19 demonstrate the syntax of creating views. The keyword view indicates the start of the sub-query, which is followed by a view name and a block containing a Boa query. In view FixingRevision, three output variables are defined: FixFileCount, FixRevisionCount, and AverageFRC. Another view Filter is defined with a output variable Retained. We can create nested views as well. A nested views version of the query is
view FixingRevision {
    FixFileCount: output sum[string][file: string] of int;
    FixRevisionCount: output sum[string] of count: int;
    AverageFRC: output mean of int;
    count := 0;

    visit(input, visitor {
        before n: Revision ->
            if (isfixingrevision(n.log)) {
                count = count + 1;
                foreach (i: int; iskind("SOURCE_", n.files[i].kind))
                    FixFileCount[input.id][n.files[i].name] << 1;
            }
    });

    if (count > 0) {
        FixRevisionCount[input.id] << count;
        AverageFRC << count;
    }
}

view Filter {
    Retained: output collection[pid: string] of int;

    v: table of avg: int = FixingRevision/AverageFRC;
    r: v._row;
    v >> r;

    visit(input, visitor {
        before n: CodeRepository -> {
            v2 := FixingRevision/FixRevisionCount[input.id];
            r2: v2._row;
            if (v2 >> r2 && r2.count > r._1
                && len(n.revisions) >= 100)
                Retained[input.id] << 1;
        }
    });
}

o: output top(5)[pid: string] of fileName: string weight count: int;

if (len(Filter/Retained[input.id]) > 0) {
    v := FixingRevision/FixFileCount[input.id];
    r: v._row;
    while (v >> r)
        o[input.id] << r.file weight r._2;
}

Figure 3.3 A Boa query implementing the DAG in Figure 3.2 to find buggy files.
shown in Figure 3.4. In line 2, view FixingRevision is defined in the scope of view Filter. Defining nested views could help the users organize the view path easier for future referencing. We will discuss more about referencing views in the next subsection.

In the scope of a view, the users cannot access anything that is defined in other view scopes, except referencing views. For instance, in Figure 3.4 users cannot access variables defined in view FixingRevision from the scope of view Filter and the main query. Same idea, view FixingRevision has no access to view Filter and the main query. Once the query is compiled, each Boa query is translated into an independent MapReduce program. Besides reusing output, we do not want to create more dependencies between MapReduce programs. However, in the future, we are planing to extend the language and defining a global static variable, allowing users to access from any scopes in the query.

3.2.2 Referencing Table

If the target table comes from the same Boa query, we say it is an internal table. To reference an internal table, users need to provide a relative view path (RVP) followed by a table name. The table name is the same as the output variable name. RVP consists of a set of view names. RVP can either start with current scope or the scope of main query. The only exception happens when the target table belongs to one of outer nested views or the table is defined after the current line. The later issue can be fixed by supporting forward declaration on views. We will discuss more about forward declaration in Chapter 7. In Figure 3.3 the blue boxes in view Filter demonstrate the path to reference tables from view FixingRevision. In this case, the RVPs start with the scope of the main query. The tables AverageFRC and FixRevisionCount are assigned to variables $v$ and $v2$. The main query also references two tables, Retained from view Filter and FixFileCount from view FixingRevision. An example of referencing with nested RVP is shown in line 36.

If the target views are from other Boa query, we call this referencing external views. The external views need to be referenced with a absolute view path (AVP) followed by a table name. The structure of external view paths is shown in Figure 3.5. AVP consists of a query root and
Figure 3.4 An Boa query revised from Figure 3.3. The view Churn is defined in the scope of Filter as nested views.
Figure 3.5 A diagram that shows the structure of external view paths. The name of the components are listed at the top with color mapping. The first path references the table with job number. The second path references the table with username and tag name.

A RVP. Query root represents the address of a Boa job. There are two kinds of query root: job id and tag name. Job id is an unique number given to every submitted Boa query. Tag name is a customized name the author set to a job. Figure 3.6 shows a revised version of Figure 3.3 assuming the view FixingRevision is defined in another Boa job by user rdyer, and the job is given a job id 12345 and a tag name FixingRevision. The revised Boa job in Figure 3.6 contains only a sub-query with view name Filter. In line 3 view Filter references an external table with job id to get the fixing file counts. In the reference path, J12345 is the query root, and AverageFRC is the table name. Since AverageFRC is defined in the main query in job 12345, there is no RVP required for the path. The beginning character J in the query root tells Boa compiler that the following string is a job id. While referencing external views, the RVP always starts from the scope of main query.

In Figure 3.6 line 18 shows an example of referencing an external table through tag name. In this case, the query root is @rdyer/ChurnRate, which rdyer is the username, and ChurnRate is the tag name. The beginning character @ is required to indicate the start of username. Similar to line 3 the reference path has empty RVP. The goal of having tag names is to allow each user to be
Figure 3.6 An Boa query revised from Figure 3.3, assuming the view FixingRevision is defined as another Boa job with job id 12345 and tag name FixingRevision assigned by user rdyer.

able to customize the queries according to different functionalities. Once a tag is set to a Boa job, the user can update or remove the tag through the web interface. If the user sets an existing tag to another job, the tag will be removed from the previous job and tagged to the indicated job. Our design allows duplicated tag names across the users but not the same user. For instance, user1 and user2 can both name one of their jobs FixingRevision, but user1 can only have one job tagged with FixingRevision at a time. This is also why the username is required while referencing with tag. Unlike job numbers, having tag names can significantly increase the readability of Boa jobs, providing more flexibility to MSR researchers.

While referencing views, users can enforce the type of referenced table in the statement. This not only allows users to keep track of the types, users can also provide addition field names in the type definition. In Boa, each referenced table becomes a table type, which contains the type for each column. An example of declaring table type is shown in line 21, Figure 3.3. The keyword table is required at the beginning of table type. Besides the keyword table and aggregator, the declaration of table type is similar to the declaration of output type. In addition, users can add field names into column types. In the example, since table AverageFRC only contains one column, the enforced table type consists a column type int with a field name avg. The field names are
optional. The field names can also apply to output variables. In Figure 3.3 line 2, the second column of table FixFileCount is given a field name file, and the third column is given is field name count. Without providing a field name, users can still access fields with default field names \( X \), which \( X \) denotes the order of the column, starting with 1. We will talk about the usage of the field names in the next subsection.

3.2.3 Table Filtering

The filtering operation allows the users to filter out unwanted column values during traversal. To filter the table, users can apply indices to perform filtering. One index is used to filter a column at a time. For instance, the first index filters the first column, and the second index filters the second column. In Figure 3.3 line 26 the referenced table FixRevisionCount has two columns with types string and int. Right after the reference path, an index \([\text{input.id}]\) is applied to the table. The index value \( \text{input.id} \) returns the current project id as a string. This filtering guarantees that next time, when a row is read from the table, the first column is always the current project id. Since the filtered column becomes a fixed value, each filter would decrease the number of columns by one. In line 36 table FixFileCount has three columns. After filtering with an index \([\text{input.id}]\), variable \( v \) only has the second and the third columns from original table. If we just want to skip filtering a column and allow any value in the column, users can use the wildcard symbol \( \cdot \) in the index. A downside of using wildcard index is that if the user filters a table with an underscore index, the user loses the access to that column due to shrinking columns. In the future, we will extend the language to address this issue.

Figure 3.7 illustrates how filter operations change the content of tables. Table \( v_1 \) contains four columns with types string, string, int, and int. Each column is assigned with an unique field name. The content of table \( v_1 \) represents the type for each column. Table \( v_2 \) applies two filters, wildcard and a string "foo", to table \( v_1 \). Thus, the table only contains rows with string "foo" in \( C_2 \) column. Table \( v_3 \) applies a filter with integer 3 to table \( v_2 \). Thus, any rows has other values than 3 in column \( C_3 \) is removed, storing the rest of the rows in table \( v_3 \). The gray columns in Figure 3.7 represent the filtered columns, and users no longer have access to these
v2 := v1[“foo”];
v3 := v2[3];

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;str&quot;</td>
<td>&quot;str&quot;</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;foo&quot;</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 3.7 A diagram that visualizes the change of tables after filterings.

columns. Therefore, table v2 has two columns: C3 and C4. Table v3 only has one column left, which is column C4. In our views, we also support filtering last column. We will discuss the return type and implementation detail in Chapter 4.

3.2.4 Traverse Tables and Extract values from Rows

Since the size of the tables can be enormous, instead of pre-loading the whole output file into the query at once, our strategy is to traverse the table row by row, reading each row into an iterator at a time. The benefit is to avoid pre-loading large data and save space. This approach can be applied to any tables regardless of table size.

In Boa, each row is a tuple type, which contains the column types from the table. Before traversing a table, we need to define an iterator with correct tuple type. The iterator type can be extracted with the attribute _row from the table. In Figure 3.6, line 19, the variable r is given a tuple type from the table v. The tuple type consists of two fields with types string and int. To read rows, we can use the right shift operator >> to read the next row into the iterator, which takes a table at the left-hand side and an iterator at the right-hand side. The right shift operator returns a boolean value, which can be used to indicate the availability of next row. To traverse the whole table, we can use a while loop to read in each row, such as while(v >> r). One usage example
is shown in line 20. The values in the row can be retrieved by using field names. If the user does not provide a field name for the column, the value can be retrieved with the default field name \( \_X \), and \( X \) denotes the order of column. An example of extracting values with customized field names is shown in line 21. The string value in the first column is extracted with field name \texttt{file}, and the integer in the second field is extracted with field name \texttt{2}.

Besides referencing, traversing and reading values from tables, we also support some handy helper functions such as \texttt{len} and \texttt{reset}. Given a table \( t \) and an iterator \( r \), users can write \texttt{len(t)} to get the number of rows in the table. One use case of function \texttt{len} is shown in line 17, Figure 3.6. The \texttt{reset} function can be applied as \texttt{reset(r)} to reset the iterator to the start of the table.

After discussing the new language for views, we can can put everything together and explain the query job in Figure 3.3. The sub-query \texttt{FixingRevision} is used to collect metadata about fixing files and bug-fixing revisions, which does not utilize views. The sub-query \texttt{Filter} first references table \texttt{AverageFRC} and read in a value. The reading operation only happens once because there is only one row in the table. In the visitor, it traverses each code repository, referencing table \texttt{FixRevisionCount} and apply filter to find the revision count for the current project. If the fixing revision count is greater than average and the number of revisions is greater than a hundred, it keeps the project and emits the project id to output variable \texttt{Retained}. In the main query, it first checks if the current project is filtered by use the \texttt{len} function on filtered table \texttt{Retained}. If the project passes the check, it then references table \texttt{FixFileCount} and filters the table with project id. At this point, the variable \( v \) contains a table with rows of filenames and the number of times each file appears in bug-fixing revision. To rank the files, it traverses the table, emitting the filenames with the counts as the weight. When the MapReduce is executed, the files will be ranked with the aggregator \texttt{top} in the reduce phase.

In the next chapter, we will discuss about our concrete implementation and how we build DAG in Boa.
CHAPTER 4 PROTOTYPE IMPLEMENTATION

In this Chapter, we are going to discuss about the implementation of view feature. To support views, we modified Boa compiler and the runtime component in Boa infrastructure. We will first discuss about type resolution, explaining how we retrieve external queries and why it is necessary. For the compiler, we will discuss about the implementation under each stage in the compiler (lexer, parser, type checker, and code generator) and the new output structure, and then we will discuss about the changes made to the runtime component along with an external tool Oozie. This thesis only focuses on the functionalities of views in Boa infrastructure, thus we have not modified the web interface and released views to Boa user. We will discuss about future changes on Boa web interface in Chapter 7.

4.1 Type Resolution

In the compiling process, type checking plays an important role to check if the types match in each statement. Since users can now reference tables from external queries, we need to retrieve correct type information for the external queries, so that we can make sure that users do not filter with the wrong type or read the wrong field in the query. We call the information retrieval process type resolution. Instead of performing type resolution every time we read in an external table, we decided to retrieve all external tables at once before compiling the query. To perform type resolution, we find the external query path on HDFS for each referenced query root in the query, for no matter how complicated an external reference path is, the external table can always be found in the Boa job pointed by the query root. For instance, given a reference path J12345/view1/view2/.../viewN/o, no matter how many nested sub-queries are in the RVP, once we found the address of the Boa job file, we can parse the query and use RVP to traverse the AST and locate the target table.

Figure 4.1 shows the process of type resolution and the interaction between poller and Boa compiler. The poller is an intermediate component between users and Boa. Once a Boa job is
submitted (Step 1), the poller will start performing type resolution immediately. To collect all table types information at once, the poller calls the compiler with tag `-views` to parse and traverse the query, collecting all external table paths in the Boa job (Step 2). Once all the external table paths are collected, the compiler returns the path strings back to poller (Step 3). As we mentioned before, each external path contains a query root. If the query root is job id, the id can be used to generate the job path on HDFS, for the job address is originally based on job id. However, if the query root consists of username and tag name, we need to find the corresponding job ids for such query roots (Step 4). Once all job id are found, the poller calls the compiler with tags `-viewSrcPath` and `-viewId` to compile the query (Step 5). The tag `-viewSrcPath` links each query root to a corresponding query path on HDFS. The tag `-viewId` links the query roots with tag name to the corresponding job ids. Before compiling the query, the compiler will parse the external queries and
Table 4.1 A table of new tokens added to the lexer. The second column shows the definition for each token, in EBNF format.

<table>
<thead>
<tr>
<th>Token</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW</td>
<td>'_row'</td>
</tr>
<tr>
<td>VIEW</td>
<td>'view'</td>
</tr>
<tr>
<td>TABLE</td>
<td>'table'</td>
</tr>
<tr>
<td>VIEWTABLE</td>
<td>J DECIMALNUMERAL (DIV IDENTIFIER)+</td>
</tr>
<tr>
<td></td>
<td>@ IDENTIFIER DIV IDENTIFIER (DIV IDENTIFIER)+</td>
</tr>
<tr>
<td></td>
<td>IDENTIFIER (DIV IDENTIFIER)+</td>
</tr>
</tbody>
</table>

store the ASTs into a map, so that the compiler can look up the map whenever needed. If the job id does not exist or the tag name is not found, the tag will be missing, and when the compiler finds an unknown query root, the compiler would abort immediately. If the job compiles successfully (Step 6), a workflow XML file will be generated for each query in the job directory, which we will discuss about workflow in the later sections. For this thesis, we have not supported users to label tag names on Boa jobs through web interface. Therefore, currently there is no need to retrieve job id from username and tag name (Step 4). However, we still put it in the process to show our approach.

4.2 Boa Compiler

Boa compiler is one of the main components in Boa because it translates the Boa language into MapReduce program in Java. The process of translation can be split into lexing, parsing, type checking, and code generation. Lexer reads in the characters from a program and convert different sequences into tokens. Parser converts sequences of tokens into AST nodes and builds an AST tree. Type checker makes sure the program types are used correctly. Code generator generates the translated code from the AST tree. Before discussing code generation, we also introduce the new output structure and explain with an example. During the compile process, if one of the stage fails, the process would immediately terminate. While modifying the compiler, we used JUnit to write test cases along the way to debug and ensure consistency and correct behavior.
4.2.1 Lexer and Parser

In Boa, we use ANTLR to generate tokens and AST tree. ANTLR is a parser generator tool written in Java. There are four new tokens added to the lexer, and the definition of each token is shown in Table 4.1. The tokens ROW, VIEW, and TABLE are simple keywords. The token ROW is used while extracting row type from a table. The token VIEW is used at the start of a sub-query. The third token TABLE is used in the type definition for views. The definition of the last token VIEWTABLE is defined in EBNF to represent referenced path, which includes three kinds of table path: AVP with job number as query root, AVP with username and tag name as query root, and RVP. Each path definition also includes the table name as the last matched identifier.

For the parser, a table of modified AST nodes is shown in Table 4.2. The first four AST nodes (SubView, Table, RowType, and TableType) are new to the parser. SubView is a kind of Statement containing AST of a sub-query. The class contains an Identifier, a Program, another parent SubView and a list of SubView children. The identifier captures the view name, and the program contains AST for the sub-query. The parent and children sub-queries are needed to restore table path in type checker. The second new AST node Table stores information of a table path, including job id, username, tag name, a list of view names, output variable name, etc. We can utilize these information to determine if the reference path contains job id, tag name, or just RVP. Another AST node RowType is used when extracting tuple type with .row selector. The type contains an Identifier, a Table, and a list of Index. If the recipient of the selector is an identifier, the identifier is used to store the recipient and the table is set to null. If the recipient is a table, the table is stored as the class member, leaving identifier as null. The list of indices is used to capture the filters on the recipient. An example syntax can be v["foo"]._row, which v is an identifier containing a table. The node RowType would then have v as the identifier, leaving table as null. The indices list would contain one index "foo". The fourth new AST node TableType is used to store type definition for a table. Each column type is stored as a Component in TableType. The Component class contains an Identifier and an AbstractType. The identifier is used to store the optional field name, and the abstract type
Table 4.2 A table of new and modified AST nodes in the parser. The second column shows the grammar for each AST node, in EBNF format. The first four AST nodes are new. For node Index, the second alternate was added to support wildcards to filter tables.

<table>
<thead>
<tr>
<th>AST Node</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubView</td>
<td>VIEW Identifier LBRACE Program RBRACE</td>
</tr>
<tr>
<td>Table</td>
<td>VIEWTABLE</td>
</tr>
<tr>
<td>RowType</td>
<td>Identifier (Index)* DOT ROW</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TableType</td>
<td>TABLE (LBRACKET Component RBRACKET)* OF Component</td>
</tr>
<tr>
<td>Index</td>
<td>LBRACKET Expression (COLON Expression)? RBRACKET (orig.)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

captures the type of the value. For instance, if a type definition is `table [pid: string] of int`, the TableType class then contains two components. The first component contains an identifier with token `pid` and a `string` as the abstract type. The second component has a null identifier and an abstract type `int`. To support wildcard(._) in table filters, we modified an existing node Index, adding a scenario that has only wildcard in the brackets into the grammar.

4.2.2 Type Checker

Boa is a statically strongly typed language. Therefore in the type checker, we have to keep track of variable types under the same scope. We also have to make sure the variables are used correctly, so that the user cannot perform operations such as `1 + true`. At the same time, we need to assign a corresponding Boa type for each visited AST node. To support views, we added a new Boa type `BoaTable`, which is used to initialize and assign to either a `Table` or a `TableType`. In `BoaTable` class, we use a list of `Object` to keep track of the filters that have been applied to the table. A `getRowType()` function is also provided in the class to get the corresponding tuple type from the table. For instance, if a table has type `table [string][int] of [string]` and has been filtered with one index, the `getRowType()` function will return a `BoaTuple` with types `int` and `string`. In the type checking phase, since we apply visitor pattern to examine and check each AST node, for the rest of the section, we will discuss several major changes the AST node. Table 4.3 also briefly describes the major changes on AST nodes to support views.

For each `SubView` node, we want to store the AST of sub-query into a view map, so that
Table 4.3 A table of major modifications on AST nodes during type checking phase. The second column briefly describes the changes for each node in type checking visitor.

<table>
<thead>
<tr>
<th>AST Node</th>
<th>Change Description</th>
</tr>
</thead>
</table>
| SubView    | 1. Recovering full RVP for each sub-query  
            | 2. Store the corresponding AST to a view map with RVP as key  
            | 1. Check the availability of each table by retrieving AST from RVP or AVP  |
| Table      | 2. Retrieve table type from AST  
            | 3. Return table as BoaTable  
            | 1. Check the filtered indices types with column types  |
| RowType    | 2. Shrink columns if any filtering applies  
            | 3. Return row as BoaTuple  |
| TableType  | Use type information to initialize and return BoaTable  |
| Factor     | 1. Type check filtered indices  
            | 2. Check the filtered indices types with column types  
            | 3. Check if the table can be filtered  
            | 4. Return filtered type as either BoaTable or BoaArray  |
| Index      | Check start expression to support wildcard filter  |
| Term       | 1. Type check right shift operator to read rows and perform bitwise shift  
            | 2. Return BoaBool while reading rows  |

whenever the user references a local table in sub-query, we can access and traverse the AST to retrieve the table type. The key stored in the map is the RVP of sub-query. In the last chapter, we mentioned two kinds of RVP. One uses the current scope as root, and one uses the scope of main query as root. In this case, we use RVP with the scope of main query as root, for sub-queries can be defined with the same name under different scopes. For instance, in the main query, two sub-queries are defined: sv1 and sv2. If the sub-queries both contain another sub-query with name sv3, the only way to differentiate the two is using the full version of RVP, sv1/sv3 and sv2/sv3, to differentiate ASTs in the map. However, SubView nodes does not contain RVP, so we use the parent-child relationship to build trees among SubView nodes to recover RVP for each node. An example of building SubView trees can be seen in Figure 4.2. The example transforms the query at the left to SubView trees at the right. The edges represent the parent-child relationship between nodes. The dashed edges are not part of the trees but used to link the trees back to main query. Given the trees, if we were to generate RVP for sub-query sv5, we just need to loop through the parents sv4 and sv3 to restore RVP (sv3/sv4/sv5). After recovering RVP, if the path has existed in the view map, a type check error is thrown because the RVP has been
used. Otherwise, the RVP and its associated query AST are stored in the view map.

**Boa Query**

```plaintext
o: output sum of int;
view sv1 {
    view sv2 {
        ...
    }
    ...
}
view sv3 {
    view sv4 {
        view sv5 {
            ...
        }
        ...
    }
    view sv6 {
        ...
    }
    ...
}
...
```

**SubView Trees**

![SubView Trees Diagram]

Figure 4.2 An example of storing the sub-queries into trees. Given a Boa query at the left, the relationship between SubView nodes can be established like the trees at the right hand side. The dashed edges are used to link the trees to the main query.

When a view is referenced with a RVP or an AVP, the view definition is stored in Table node. If a Table node does not contain a job number and view name, the table comes from a local sub-query. Otherwise, AVP represents the path of an external table. In the case of local sub-query, we need to use RVP from Table node to retrieve corresponding AST from the view map. If the root of RVP is not the main scope, we have to restore full RVP before looking up view map. After retrieving sub-query’s AST, we can get the table type from the output map stored in AST’s symbol table. If the Table consists of an AVP, we look up the query ASTs from type resolution phase. If the external query AST is found, given a RVP and the target table name, we utilize a visitor to
traverse query AST and return the target table type.

The new AST class, RowType, happens when users extract column types from a view (such as v.row). The user can also get the row type with filters (such as v["foo"].row). The returned row type is a BoaTuple consisting of the column types. The filter values become trivial because we only care about the columns that are left in the table. After we get the table type from looking up symbol table, we just need to decrease the number of columns in the table and create the tuple type from the filtered table. Another new AST class that captures type definition is TableType. When the node is visited, we need to use the type information provided in TableType to create and initialize a BoaTable.

The class Factor contains an operand and a list of operation nodes. The operation node can be a selector, a filter index, or a function call. If the type of the member operand is BoaTable and contains indices (such as t["foo"]), we need to check the validity of indices. When the user filters a table, each filtered index is turned into an Index node in AST, which contains a start expression and an end expression. The start expression indicates the main index value, and the end expression is used for slicing. One slicing example can be ary[1:10]. Since slicing is not valid for filtering tables, the index must not have end expression. If the filter contains a wildcard, both start expression and end expression are set to null. After validating index value, we need to check if the table has any column left and if the index type matched the column type. For Instance, given a table type table [string][int] of string, the following filtering behaviors are invalid: t["foo"][50]["foo2"]["foo3"] and t["foo1"]["foo2"]. The first example is not valid because the number of indices exceeds the number of columns in the table. The second example is invalid because the type of the second filtered index, string, does not match the type of second column int. In this case, the table can be filtered with at most three indices. If the last column is filtered, instead of shrinking table to an empty table, the filtered table returns an array of EmptyTuple. EmptyTuple is a type of Tuple that contains no values. The reason for using empty tuple is because there is no columns left in the table. Also, users are only interest in the number of rows matching the filters instead of the filtered values. Therefore, the
number of elements in the returned array is the same as the number of rows matching the filters, so that users can apply \texttt{len} function to get the number of rows. Figure 4.3 extends the filtering example from Figure 3.7. The example query filters the last column twice with integers 5 and 8 respectively. The returned values stored in variables \texttt{a1} and \texttt{a2} are shown at the bottom. Since there are two rows matching the filter 5, there are two elements in array \texttt{a1}. Similarly, since there is only one row matching the filter 8, array \texttt{a2} contains one element.

```plaintext
v2 := v1["foo"];v3 := v2[3];a1 := v3[5];a2 := v3[8];
```

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;foo&quot;</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>&quot;foo&quot;</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

\texttt{a1} \{EmptyTuple, EmptyTuple\} \texttt{a2} \{EmptyTuple\}

Figure 4.3 A diagram extended from Figure 3.7. When the last column is filtered, instead of returning an empty table, an array of \texttt{EmptyTuple} will be returned instead.

Another extension to the language is right shift operator, which takes a \texttt{BoaTable} from the left hand side and a \texttt{BoaTuple} from the right hand side. The operator can be type checked in Term nodes. However, not only we want to use the operator to read rows, we also want to preserve bitwise functionality of the operator. Therefore, we have to carefully handle some illegal cases such as \texttt{1 >> 3 >> v} or \texttt{4 >> r >> 2}. Since the reading row action can be used to indicate if there is remaining row or not, if the syntax is valid, the node is assigned with type \texttt{BoaBool}.

4.2.3 Output Protobuf

One of the major changes to support views is changing the output format from text to binary with protobuf format. The reason for changing output format is because of output reusability. Even
message Row {
  repeated Value cols = 1;
  required Value val = 2;
}

message Value {
  enum Type {
    INT = 0;
    FLOAT = 1;
    STRING = 2;
    BOOL = 3;
    TUPLE = 4;
  }
  required Type type = 1;
  optional int64 i = 2;
  optional double f = 3;
  optional string s = 4;
  optional bool b = 5;
  repeated Value t = 6;
  optional bool hasWeight = 7 [default = false];
}

Figure 4.4 The protobuf structure for Boa output.

though we can parse the output with original format every time we read an output, not only parsing can cause extra overhead, the format also lacks of type information for the table. Figure 4.4 shows the output’s protobuf structure. Each output table consists of several rows. Each row is translated into a Row message. The Row message contains two Value typed members: cols and val. Remember each emitted output is split into a key-value pair. The member cols denotes the multiple key fields of output type, and val stores the value field of output. The Value message represents a field in a row, which currently supports types integer, float, string, boolean and tuple. The tuple type is represented as the member t in line 18. This also happens when a Value message contains multiple Value messages. In line 19 the member hasWeight is used to check if the current value is a weight or not. Again, weight is an output combined items for aggregators like maximum and top.

Figure 4.5 shows an example of visualizing the protobuf format for a given output type. The example output type is defined at the top, and the translated row structure is visualized in a tree at
the bottom. In the protobuf tree, the bold texts represent the message members, and the texts inside parenthesis denote the types. The underlined types are non-terminals and contain one or more children in the tree. In the example, the output table contains three columns with types string, t1, and int. The first two columns are stored in cols, so cols contains two Value messages. The first Value contains a string. The second Value contains a tuple with type t1, which consists of three Value messages, each with type int, string, and bool. The last column is captured by val, which contains an integer. Next, we will discuss how to utilize this protobuf format in code generation.

```
type t1 = { int, string, bool };  
o: output sum [string][t1] of int;
```

Figure 4.5 An example of visualizing Boa output in protobuf format from Figure 4.4. The example output type is defined at the top, and the tree at the bottom shows the protobuf structure for the rows.

4.2.4 Code Generation

In the code generation phase, we examine AST nodes, generating the corresponding Java code for each node, and finally putting all the pieces together into a Java file. In the meantime, we also
utilize several templates to generate code. Previously, each Boa job contains only one MapReduce task, so the job is always translated into one MapReduce Java file. However, with views, a Boa job can contain multiple queries. Therefore, starting with the main query, we use a depth-first recursive approach to generate MapReduce Java file for each query one at a time, and then along with other class files, each Java file is built into a jar file.

Figure 4.6 An example of generating jar file for each query in a Boa job.

Using Figure 4.6 as an example, while the compiler is traversing main query’s AST, the compiler uses a map to capture the AST of sub-queries found in the current query. In this case, the main query contains one sub-query with view name sv1. After building jar file for the main query, the compiler recursively performs the same procedure on the sub-query (sv1) found in the map. While traversing its AST, the compiler finds two sub-queries, sv2 and sv3, under query sv1, and the same procedure repeats. In the example, the color of each query maps to the color of generated jar file. The path of each jar file also maps to the referenced RVP. Next, we will introduce
the new classes we added to generate for runtime execution.

There are three new classes added for runtime execution: EmptyTuple, Tuple, and TableReader. We have introduced EmptyTuple in the previous subsection. The empty tuple is used to as a dummy element when the last table column is filtered. It is also a subclass of the abstract class Tuple. Since the rows are treated as tuples, depends on the column types, the row (or tuple) can contain different members and different constructors. Tuple is a base class of any kinds of tuple. The abstract class contains a boolean variable def and a function fromRow. The def variable is used to check if the row is valid or not. When the table runs out of rows and the user reads the table into a tuple, the def variable is set to false. The function fromRow is called when we want to fetch the fields from Row to the tuple. This only happens inside TableReader class. Since the functions of protobuf message Row can vary depending on the table type, we use a tuple template and the table type information to generate the function body. The TableReader class is used to read rows from a table and pass the row into a tuple. Given a valid output path, the table reader not only uses a sequence reader to read rows from the output file, it also keeps track of the table filters. While reading a row, the table reader will compare the fields in each row to the filters until one matches. The TableReader class also contains a set of helper functions that supports retrieving values from rows to tuple, the class also contains functions such as length and reset to support len and reset built-in functions. Next, we will introduce how we generate code with different visitors.

To explain how we generate code for different pieces, an example Boa query shown in Figure 4.7 is used for the rest of the subsection. The query references an external table from job 12345. The table is stored in variable v1 with column types string, string, t, and int. The type t is a tuple type shown in line 2 which contains three fields with types int, string, and bool. The variable v2 is assigned with table v1 with two filters: a string "java" and a wildcard. An iterator r2 is used to traverse table v2. For each row found in table v2, the second field of the row iterator is emitted to output o with an index "sum". At the end of the query, the size of table v2 is emitted to output o with index "size".
1 o: output sum[string] of int;
2
type t = { i: int, string, b: bool };
3 v1: table [string][string][t] of lastColumn: int = J12345/sv/o;
4 v2 := v1["java"][_];
5 r2: v2._row;
6 ary := v2[_][3];
7 while (v2 >> r2) {
8   lastCol := r2.lastColumn;
9   o["sum"] << lastCol;
10 }
11 tableSize := len(ary);
12 o["size"] << tableSize;

Figure 4.7 An example Boa query that is used to demonstrate the changes we made to generate code to support views. The query itself does not serve any particular purpose but demonstrating code generation.

Remember besides the output statement, the Boa query represents the mapper function of the MapReduce task. Therefore, most of the code generation work happens in the mapper class. The mapper class contains a set of tuple classes’ definitions, variables declarations, a map function and other helper functions. Our implementation of code generation can be split into three parts: tuple class definition, variable declaration, and map function. Every tuple and variable used in the map function must first be defined as class members. Next, we will demonstrate how we declare variables and define tuples, and then discussing the new Java code to support views in the map function.

In the code generation phase, we utilize a tuple declaration visitor to identify tuples and generate tuple classes. For the example query in Figure 4.7, there are two tuple classes generated: BoaTup_0 and BoaTup_1. The generated tuple classes are shown in Figure 4.8 and Figure 4.9. Figure 4.8 shows the first tuple class BoaTup_0 generated from line 2. The class contains three fields with types Long, String, and Boolean, which maps to the three tuple field types in the query. The class also contains a constructor, taking a Value argument. When this constructor is called, we assume the Value is an tuple and carries three other Value messages. Therefore in the constructor, we can see three assignment statements, using the helper functions from
private class BoaTup_0 extends Tuple {
    Long _1;
    String _2;
    Boolean _3;
    ...
    BoaTup_0(final Value v) {
        this._1 = TableReader.valToLong(v.getT(0));
        this._2 = TableReader.valToString(v.getT(1));
        this._3 = TableReader.valToBoolean(v.getT(2));
    }
    ...
}

Figure 4.8 The tuple definition generated from line 2 in Figure 4.7.

private class BoaTup_1 extends Tuple {
    BoaTup_0 _1;
    Long _2;
    ...
    BoaTup_1(final Value v) {
        this._1 = new BoaTup_0(v.getT(0));
        this._2 = TableReader.valToLong(v.getT(1));
    }
    public void fromRow(final Row r, final int offset) {
        this._1 = new BoaTup_0(r.getCols(0 + offset));
        this._2 = TableReader.valToLong(r.getVal());
    }
    ...
}

Figure 4.9 The tuple definition generated from line 5 in Figure 4.7.

TableReader to translate the Value message into corresponding type and assign it to the tuple member.

Figure 4.9 shows the second generated tuple class BoaTup_1. The tuple class is translated from line 5 in the example query. Since the table v2 applies two filters on table v1, the row type of table v2 contains the last two columns of the origin table. Therefore, the tuple class contains two members. One with type BoaTup_0 and one with type Long, which maps to the types t and int in the query. Again, the tuple class also contains a constructor which takes a Value type argument. This constructor can be triggered when one of the column types is a tuple with types t,
another tuple, and int. In the constructor, since the first field is a tuple, instead of using the helper function from TableReader, we assign the field to a new instance of BoaTup_0, and passing the corresponding Value as argument, which triggers the constructor in Figure 4.8. The function fromRow is called when we found a valid row from a table. The function takes two arguments, a Row and an integer offset. The function would retrieve the fields from the row and assign them to the tuple fields. The offset value is the number of filters applied to the row, so that we can extract the correct field from the row. Since the last column of a row is stored as a required Value message val, we just need to use getVal() function to extract the field directly instead of using offset.

To generate code to declare variables, we use a variable declaration visitor to traverse the query AST, looking for VarDeclStatement nodes to declare variables. Table 4.4 shows the generated variable declaration Java code after traversing the query in Figure 4.7. In line 3 and line 4, the tables v1 and v2 are each translated to a Java variable with type TableReader. In line 5, the row is defined as a tuple type BoaTup_1. The variable r2 is assigned with an instance of BoaTup_1. The constructor will assign the tuple fields to null. For the variable ary, since the last column of table v2 is filtered, the last filter would return an array of EmptyTuple. In this case, we use an anonymous variable anonT_0 to capture the clone of table v2, so that we can apply filters to the temporary table without modifying v2. In line 8, since the column with field name lastColumn is an integer, the variable lastCol is translated into a long variable. Similarly, in line 11, since the operator len would return an integer as the size of the table, the corresponding Java code is also a long variable.

Table 4.4 The generated variable declaration Java code for the example query from Figure 4.7.

<table>
<thead>
<tr>
<th>Line #</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>TableReader v1;</td>
</tr>
<tr>
<td>4</td>
<td>TableReader v2;</td>
</tr>
<tr>
<td>5</td>
<td>BoaTup_1 r2 = new BoaTup_1();</td>
</tr>
<tr>
<td>6</td>
<td>TableReader anonT_0;</td>
</tr>
<tr>
<td>8</td>
<td>EmptyTuple[] ary;</td>
</tr>
<tr>
<td>11</td>
<td>long lastCol;</td>
</tr>
<tr>
<td></td>
<td>long tableSize;</td>
</tr>
</tbody>
</table>
Table 4.5 The generated Java code in the mapper function for the example query from Figure 4.7.

<table>
<thead>
<tr>
<th>Line #</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>v1 = new TableReader(0, &quot;J12345&quot;, &quot;sv&quot;, &quot;o&quot;);</td>
</tr>
<tr>
<td>4</td>
<td>v2 = v1.clone();</td>
</tr>
<tr>
<td></td>
<td>v2.addIndex(&quot;java&quot;).addIndex(null);</td>
</tr>
<tr>
<td>6</td>
<td>anonT_0 = v2.clone();</td>
</tr>
<tr>
<td></td>
<td>ary = anonT_0.addIndex(null).addIndex(3).filterToArray();</td>
</tr>
<tr>
<td>7,10</td>
<td>while (v2.fetch(r2)) {...</td>
</tr>
<tr>
<td>8</td>
<td>lastCol = r2._2;</td>
</tr>
<tr>
<td>11</td>
<td>tableSize = ((long)ary.length);</td>
</tr>
</tbody>
</table>

Figure 4.5 shows the generated Java code in the mapper function for the example query. In line 3, the new table reader instance is created and assigned to v1. The first construct argument is the starting position of the table. It is set to zero so that we can read from the beginning of the table. The rest of the arguments are the table’s referenced path. For the filter operation in line 4, the v2 table is first assigned with the clone of table v1, and the function addIndex in the TableReader class is used to add filter indices to the table. The argument of the function is a filter value. If the filter index is wildcard, the code generator would translate the wildcard to null, so the generated code would add a null index to the table filter. In line 6, the anonymous variable anonT_0 first captures the clone of table v2, and then the filters are added to the anonymous table and call the function filterToArray, returning and assigning an array of empty tuples to ary. In line 7, the row reading operation is translated to calling a fetch function from table v2. The argument r2 is passed into the function to read the next row. If the row is read successfully, the fetch function would return true. If the table runs out of rows, the function would return false. In line 8, the customized field name lastColumn is translated to the corresponding field (_2) in tuple BoaTup.1. For line 11, the len function is translated to the length member of the array ary. The translation of function len depends on the type in the argument. If the argument is a table (len(v1)), instead of calling the member, the generated code calls the length function instead ((long)v1.length()).
4.2.5 Static Tables

Besides the implementation of the new syntax and operations we mentioned previously, we also support referencing static tables. Static table is a kind of table that is used directly with the form of referenced view path. It allows users to traverse tables without using variables. Figure 4.10 shows an example of traversing three different static tables. Even though the three table comes from the same external table (J12345/o), they are three distinctive static tables because there are three different kinds of filters: "temp", "foo", and wildcard. The static tables can not only be used for traversal, we can also apply function on static tables such as len or reset. As long as the table has been defined, we can statically reference the table anywhere in the query. For instance, we can use static table inside an if statement to check the table size, such as if(len(J12345/sv/o) > 10). To generate Java code for static tables, we create an anonymous table for each static table. The generated code for the example Boa code is showed in Figure 4.11. The variable anonT_0 captures the static table with filter "temp". The anonymous variable anonT_1 captures the table with filter "foo", and the variable anonT_2 captures the one with the wildcard filter.

Figure 4.10 An example Boa code that traverses three static tables. The table comes from the output variable o in job 12345. Each table is applied with a string filter.

```java
J12345/o["temp"] >> r;
J12345/o["temp"] >> r;
J12345/o["foo"] >> r;
J12345/o["foo"] >> r;
J12345/o["foo"] >> r;
J12345/o[ ] >> r;
```

Figure 4.11 The generated Java code for the Boa code from Figure 4.10.
By using anonymous variables to capture static tables, users can traverse the tables without using a variable. Without using anonymous tables, if the table is not assigned to a variable, the table would be assigned with a new instance of table reader. No matter how many times the user reads from the same table, the row-reading statement would always read and assign the first row to the iterator.

The only constraint of having static table is that static tables cannot have dynamic filters. The dynamic filters are the filters using variables, such as $x$ and $\text{input.id}$. The reason for not supporting dynamic filter is because of the uncertainty of using dynamic filter. For instance, if we append a line of code, $J12345/\circ[x] >> r$; to Figure 4.10, and variable $x$ happens to be "temp", should the statement reads the third row or read the first row? To avoid this confusion and uncertainty, we disallow any dynamic filters on static tables.

4.3 Runtime

Once the Java code is generated successfully, as we showed in Figure 4.6, each Java file is built into a jar file. Remember the DAG we mentioned in Chapter 3, currently, each jar represents a query node in the DAG. To handle the dependencies between jar files at runtime, we utilize Oozie, a schedule system for Hadoop jobs, to execute Boa jobs. To run Oozie, we also have to create a workflow for each Boa query. However, before we start introducing Oozie and workflows, we would first explain the HDFS layout for each Boa job.

4.3.1 HDFS Layout

To support views, we made two major changes on HDFS layout. The first one is supporting multiple queries’ output in one Boa job. The second change is to create one output file per output variable. Figure 4.12 shows the original job layout on HDFS. Assuming there are three successfully executed jobs on the cluster (23456, 23457, and 23458), each job directory contains an output text file, part-r-00000. The output file contains the output from all the output variables defined in the job. However, this design can create extra overhead if we want to extract the output from a particular variable. The new layout is shown in Figure 4.13, which shows the layout for a particular
Figure 4.12 The original Boa jobs layout on HDFS.

job from Figure 3.4. The job contains two nested sub-queries, Filter and Retained. As we mentioned in Figure 4.6, the address of a sub-query directory follows its own RVP. In this case, since the sub-query FixingRevision is defined in the scope of query Filter, its directory is placed in Filter directory. Also, each query directory contains three items: an output folder, a query jar and a workflow XML file. The output folder consists of the output sequence files. Unlike the old layout, the result from each output variable is stored individually to support views, so that it is easier to open and read a particular table. The name of the output files also maps to the output variable name. The query jar is required to execute the query. Before, when a Boa job is submitted, after the query is compiled and built into a jar, the jar file is executed immediately. However, with views, we just create the jars and let Oozie execute them at runtime. To run the jar with Oozie, the workflow file is required, and we will discuss more about Oozie and workflows in the next subsection.

4.3.2 Oozie and Workflows

Oozie is a schedule system for Hadoop jobs, and we used Oozie to handle the dependencies between queries during runtime execution. The schedule is made up with a set of workflow jobs.
Figure 4.13 The new output layout on HDFS, assuming job id 23456 represents the Boa job showed in Figure 3.4.

Each workflow job is a XML file that contains a set of directed acyclic action nodes. The users can use the action node to execute a Java file, a sh file, or perform a MapReduce job, etc. As we shown in Figure 4.13, we created a workflow for each Boa query. Each workflow has at least one action node that executes the query jar. If the current query references any views, creating dependency with other queries, the referenced queries must be executed before running the current query. These actions are written as sub-workflow actions. Given the paths of a workflow file, the sub-workflow action can trigger another workflow file. These actions can also be visualized as the edges in the DAG. However, before running jar or triggering any workflows, we need to check if
the outputs for the job exist or not. If the outputs already exist, there is no need to rerun the query jar. Otherwise, the sub-workflow actions will be executed one at the time, and the query jar will be executed once all the sub-workflows actions are completed successfully.

**Figure 4.14** The generated workflow DAG for the main query from Figure 3.4. The workflow contains two sub-workflow actions due to referencing tables from queries Filter and FixingRevision.

Figure 4.14, Figure 4.15 and Figure 4.16 show the workflow DAGs for the example query from Figure 3.4. Each DAG represents a workflow file for a query. Figure 4.14 is the workflow DAG for the main query. Figure 4.15 represents the workflow for query Filter, and Figure 4.16 shows the workflow DAG for query FixingRevision. For simplicity, the root of the paths shown in the DAGs are the job directory on HDFS, and the code in each action node is pseudo-like to easily demonstrate the workflow DAG. Next, we will discuss each workflow DAG for the Boa job.

Starting from DAG in Figure 4.14, since the main query contains only one output variable o, we first check if the output file o.seq exists under a given path. If the output exists, the workflow ends and terminates the job. If the output does not exist, the check would return true and head to the sub-workflow actions. The sub-workflow actions run the workflows for the queries.
Filter / workflow.xml

If (!exist("Filter/output/Retained.seq"))
run("Filter/FixingRevision/workflow.xml")
delete("Filter/output") and run("Filter/Query.jar")
end

Figure 4.15 The generated workflow DAG for query Filter from Figure 3.4. The workflow contains one sub-workflow action due to referencing a table from the query FixingRevision.

that the current query is depending on. In this case, the main query depends on two queries, so there are two sub-workflow actions. One for sub-query Filter and another action for sub-query FixingRevision. The execution order does not matter, but the process can only be done in one thread. We will discuss about why later. Once the sub-workflow actions are finished, the workflow then moves on to jar execution node. In the jar execution node, we customized the action so that Oozie would delete the output folder before running the query jar. Executing the query jar would start a MapReduce job on the cluster. Once the job is finished, the workflow terminates.

For the workflow generated from Filter query, the pattern is very similar to the previous one. In Figure 4.15 the workflow first checks if the output Retained.seq exists or not. If the check returns false, the workflow terminates. If the output file is missing, since the Filter query references a table from sub-query FixingRevision, the workflow moves on to execute the sub-workflow action for query FixingRevision. After finishing the sub-workflow action, the workflow deletes the output folder and runs the jar file.

The third workflow DAG is slightly different, for the query does not reference any tables. Therefore, there is no need to perform sub-workflow actions. The workflow only need to run the query jar if the output is missing. In this case, there are three output files should be generated for
Filter / FixingRevision / workflow.xml

```xml
If (!exist("Filter/FixingRevision/output/FixFileCount.seq") ||
!exist("Filter/FixingRevision/output/FixRevisionCount.seq") ||
!exist("Filter/FixingRevision/output/AverageFRC.seq"))

end
```

Figure 4.16 The generated workflow DAG for query FixingRevision from Figure 3.4. The workflow contains no sub-workflow action because the query does not reference any tables.

the query, so if any of the output files is missing, the check should return true and move on to jar execution. The reason why we do not execute sub-workflow actions in parallel is because we might end up triggering the same workflow twice and causing conflict. In this case, the main workflow triggers two workflows from queries Filter and FixingRevision, and the workflow for query Filter triggers the workflow from query FixingRevision again. If the actions are executed in parallel, the cluster nodes could end up running the same Boa job at the same time, which could generate wrong or in-completed outputs.

If the user just submits the job, and the Boa job compiles successfully, Oozie would always starts with the workflow from the main query, and then using the depth-first approach to propagate the dependency DAG for the Boa job. While running workflows, if any of the actions or the MapReduce task fails, Oozie would automatically kill the current workflow and return to the parent workflow if there is one.
4.3.3 Cycles

In this subsection, we want to explain how we can guarantee having no cycles in the dependency DAG. According to our design of views, the cycle can only happen through the following two scenarios: updating view names, or query names, and referencing internal queries. Even though Oozie automatically checks if there is any cycles in the action DAG while running workflow, it does not check the sub-workflow actions with other workflows. Therefore, a loop can still happen among the workflows while using Oozie.

![Figure 4.17 An example directed dependency graph. The initial dependency relationship is shown with the solid edge. After changing view name Foo from Q1 to Q4 and creating query Q5, the dashed edges are added to the dependency relationship, which turns out containing a cycle among Q2, Q3 and Q4. To simplify the DAG, the data flow edges between queries and Boa dataset are omitted.](image)

The first scenario can be demonstrated in Figure 4.17. In the example, there are five queries: Q1, Q2, Q3, Q4, and Q5. The numbers represent the order of program creation. The query Q1 is created first, and the last created query is Q5. The initial dependency DAG is shown with solid edges. When Q2 is created, it references tables from Q1 by using view name Foo. However, if the user changes the view name Foo from Q1 to Q4, and a query Q5 is created and references a table from Q2, the dashed edges are added to the dependency DAG, which turns out containing a cycle among queries Q2, Q3, and Q4. To prevent this issue, we replace the view name with the corresponding job id in the compiling stage, so that in the generated jar file, the referenced path for the external
tables consist of job ids instead of view names. Therefore, the queries will always reference tables from the same query for every execution. This can also ensure purity in Boa language, which enforces producing the same outputs while running the same query.

Figure 4.18 An example directed dependency graph with a cycle among the sub-queries sv1, sv2, and sv3. The text 12345 represents the job id of the query. To simplify the DAG, the data flow edges between queries and Boa dataset are omitted.

Another scenario could happen when users accidentally create a cycle by referencing internal tables. One example is demonstrated in 4.18. In the example, the cycle happens among the sub-queries in Boa job 12345, for the sub-queries reference internal tables from each other. If Oozie runs the workflow file for job 12345, it will become an endless loop that is consistently reading each other’s workflow. To avoid the loops while referencing internal tables, for now we just don’t support forward declaration for sub-queries. In other words, the users can only reference the internal queries after it is defined, which will turn the directed dependency graph into topological order and guarantee no loops. In the future, we might support forward declaration on internal queries and provide other strategies to resolve the cycle issue.
CHAPTER 5 EVALUATION

To evaluate views, we perform two case studies by reproducing two existing research works in MSR field. We believe the selected research works are complicated enough to demonstrate the use of views. Each research work requires a set of procedures, which can also be divided into several tasks. We wrote Boa queries and utilized views to reproduce the research solely with Boa. We not only analyze the case study results, we also evaluate the caching behavior on one of the case studies.

We set up a cluster with Hadoop version 1.2.1. The cluster contains 15 compute notes and 1 master node. Each node is given 16 RAM and processed with 4 core Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz. Each core can run a at most a mapper or a reducer, and each map and reduce task is given 1 GB of storage space. The schedule system Oozie is installed with version 4.0.1.

Note that due to the addition of views to Boa, from this point forward every Boa job (aka, query) is actually also a view. Any job can be directly re-used by another job (assuming the user has permission to view that job) by referencing it via job number. Thus, many of the queries shown in this chapter may not look likes views but they are. We explicitly reference them via job numbers.

5.1 Case Study 1: Splitting Identifiers

To mine the natural language information from the source code, splitting identifier into words is required. Many splitting algorithms have been developed. To help the research community testing the algorithms, Binkley et al. [4] created a dataset (so called gold set) that contains 2,663 identifiers and their split form based on 8,522 human splitting judgements. Hill et al. [20] then performed an empirical study of different identifier splitting algorithms (Greedy, Samurai, and INTT, etc) on the gold set. Some splitting algorithms use hard words as input instead of the original identifiers. Hard words are the conservative split of a identifier based on certain hard evidences such as underscore (hard_word), digits (word10), and the transition from lower case to upper
5.1.1 Boa Queries

To reproduce the study, we decide to import the gold set into Boa and split the identifiers by using the greedy approach [14]. The approach requires a dictionary word list, an abbreviation list, and a list of stop words. Stop words are words that do not contain any useful information such as a or about. The dictionary contains 60,433 entries from Kevin Atkinson’s SCOWL word list (sizes 10 through 35) [24]. The abbreviation list contains 90 entries, which is also from SCOWL word list (sizes 10 through 35). To acquire better result, we defined and added a set of 27 technical abbreviations such as str and vars to the abbreviation list. The list of stop words was collected through online resource (https://www.ranks.nl/stopwords), which contains 665 entries in the list. Therefore, there are 61,215 entries in total in the word list for this case study. The gold set and word lists are imported to Boa by several python scripts, emitting each entry into corresponding Boa output variable.

The DAG for splitting identifiers from gold set is shown in Figure 5.1. There are five queries (or nodes) in the DAG. The name of the query is shown at the top of each node, which followed by view names if there is any. Each query contains at least one output. In this case, the color is used to differentiate different Boa query files. Among the queries in the DAG, three of them are from the same query file WordList.boa. The query contains two sub-queries: Abbreviation and StopWords. The shape icons are used to differentiate different output variables. For instance, the query GreedySplit contains two outputs Result and Accuracy. The outputs participating the data flow are marked as bold. In the DAG, the query GreedySplit references four tables, one from each query. Notice that even though this task does not require Boa dataset, each query is still fed with a Boa dataset, for the current infrastructure requires a Boa dataset to start the MapReduce process.

In the future, we will modify the infrastructure, allowing users to customize their own input dataset for each query.

In the gold set, each entry not only contains the identifier and split identifier, it also contains other information such as program name and the human judgements’ confidence level for each
Figure 5.1 The DAG for splitting identifiers from gold set. The color is used to differentiate different query files, and the shape icons are used to differentiate different output variables. The outputs participate in the data flow are marked as bold.

Each identifier can have up to five confidence scores, and the confidence score scales from 0 (a guess) to 2 (certain). The query **GoldSet** is shown in Figure 5.2. Since the query does not use Boa dataset at all, we used an if statement as a check at the beginning of the map function. The string 1061331 is an existing project id from the dataset, so that only the mapper node processing the project would execute the query code. Without the check, the query code would get executed as many times as the dataset size, which could potentially create much overhead and produce wrong results.

To import the gold set to Boa, we use three output variables to capture the information: Gold, ProgramLanguage, and ConfidenceLevel. The output Gold contains a key, the original identifier, the name of the program, the hard-split version of the identifier, and the annotator-split version of the identifier. The output ProgramLanguage stores the language information of the programs. The output ConfidenceLevel takes a key as the index, and the second column is
if (input.id == "1061331") {
    ProgramLanguage: output set [program: string] of language: string;
    ConfidenceLevel: output collection [id: int] of confidenceLevel: int;

    emit1 := function() {
        Gold[1]["::CreateProcess"]("mozilla-source-1.1")["Create-Process"] << "Create-Process";
        ProgramLanguage["mozilla-source-1.1"] << "cpp";
        ConfidenceLevel[1] << 1;
        ConfidenceLevel[1] << 2;
        ConfidenceLevel[1] << 2;

        Gold[2]["::DrawThemeTab"]("mozilla-source-1.0")["Draw-Theme-Tab"] << "Draw-Theme-Tab";
        ProgramLanguage["mozilla-source-1.0"] << "cpp";
        ConfidenceLevel[2] << 2;
        ConfidenceLevel[2] << 2;
        ConfidenceLevel[2] << 2;

        ...
    };

    ...
    emit1();
    emit2();
    ...
}

Figure 5.2 The Boa query file GoldSet.boa for case study 1, which import Binkley’s gold set into Boa by emitting each entry to three output variables. The query job is assigned with job id 100.
the confidence level given by an annotator. Since a Java method cannot be larger than 64K, we have to divide the gold set into groups and emit each group inside a function, so that the translated Java file will not violate the method size limit.

The query **WordLists** along with its sub-queries are shown in Figure 5.3. The sub-queries each contain an output **Words**. The sub-query **Abbreviation** imports the abbreviation word list to Boa, and another sub-query **StopWords** imports the stop word list to Boa. The main query imports the dictionary word list. Similar to the gold set query, we have to divide the dictionary words into small groups, so that the translated Java file will not exceed the method size limit. Instead of having each word list as an individual Boa job, the three word lists are put into one query file to realize modularization.

Given the gold set and three word lists, the last Boa job **GreedySplit** runs greedy algorithm on the gold set. The greedy algorithm strongly depends on the word lists. Given the hard words of a identifier, if the hard word is not on the word list, the greedy algorithm is then applied to the word and tries to split it. The algorithm first performs two searches on the identifier. The first one looks for the longest prefix, and the second search looks for the longest suffix. When a prefix or suffix is found on the word list during the search, the greedy algorithm is recursively applied to the remaining substring. Once both searches are done, we compare two searches by looking at the ratio of words found on list to the total soft words, and the string is split according to the winner search result. If the prefix and suffix do not appear in the word list, the beginning character is ignored from the string, and the algorithm checks the remaining substring until running out of characters.

The query file **GreedySplit.boa** is shown in Figure 5.4. In lines 7, 9, 11, and 63, the query statically references the gold set table and the three word list tables. Since we haven’t implemented the Boa website, the user cannot define tag names on Boa jobs, so currently users can only use job id as query root. To traverse the tables, two iterators are defined: **goldIterator** and **wordIterator**. Since the word list tables have the same table type, they share the same iterator **wordIterator**. After storing the words from the word lists into a set, the query then
Figure 5.3 The Boa query file `WordLists.boa` for case study 1, which import the word lists into Boa by emitting each entry to the corresponding output variable. The query job is assigned with job id **200**.
if (input.id == "1061331") {
  Result: output collection [original: string] of result: string;
  Accuracy: output collection of float;
  wordIterator: J200/Dictionary._row;
  goldIterator: J100/Gold._row;
  wordSet: set of string;

  while (J200/Dictionary >> wordIterator)
    add(wordSet, lowercase(wordIterator.word));
  while (J200/Abbreviation/Words >> wordIterator)
    add(wordSet, lowercase(wordIterator.word));
  while (J200/StopWords/Words >> wordIterator)
    add(wordSet, lowercase(wordIterator.word));

  count: float = 0;
  while (J100/Gold >> goldIterator) {
    if (len(goldIterator.hard) == 0) continue;
    hardWords := splitall(goldIterator.hard, "-");
    softWords: stack of string;
    foreach (i: int; def(hardWords[i])) {
      if (contains(wordSet, lowercase(hardWords[i]))) {
        push(softWords, hardWords[i]);
        continue;
      }
    }
    greedy(hardWords[i], softWords);
    result := pop(softWords);
    while (len(softWords) > 0)
      result = pop(softWords) + "-" + result;
    if (result == lowercase(goldIterator.anno))
      count += 1.0;
    Result[goldIterator.original] << result;
  }
  Accuracy << count / len(J100/Gold);
}

Figure 5.4 The Boa query file GreedySplit.boa for case study 1. The query applies greedy algorithm along with three word lists to the identifiers from gold set.
runs greedy algorithm on each hard word by calling function `greedy` in line 72. Since Boa does not support dynamic array type, we use stacks to keep track of splits dynamically. For each split hard word, the split results is built into stack `softWords`. Once an identifier is done splitting, we turn the soft words into a string by placing split marks in between. The final result is then emitted to the output variable `Result`. We also compares the result with the annotator-split version from the gold set to compute the accuracy of the algorithm.

Besides splitting identifiers from the gold set, we also apply the greedy approach to the identifiers found from Boa dataset. The DAG is similar to Figure 5.1. The only differences are the gold set query and the greedy query. Instead of using the gold set query as the dataset, we wrote a query `BoaIdentifiers` that mines all the identifiers from open source projects. Also, we created another query `BoaGreedySplit` that runs greedy algorithm to split identifiers from Boa dataset.

The query `BoaIdentifier` is shown in Figure 5.5, which traverses each project and counts the number of identifiers with the same type. Since this query directly accesses each project, there is no need to check the project id like the previous queries. The query contains 7 output tables. Each table collects the identifier for a specific type. Besides output table `DeclIDs`, the output tables contain three columns: project id, the identifier string, the total number of occurrence in the project. The output table `DeclIDs` has one more column which stores the type of the identifier.

The query `BoaGreedySplit` is shown in Figure 5.6. The query is similar to query `GreedySplit`. However, since Boa identifier dataset does not contain hard words as the gold set does, we use the function `generateHard` to split the identifier name to hard words. Since we are splitting all categories of identifiers from query `BoaIdentifiers`, we turn the splitting process into a function `ProcessID`, and the function is called for every identifier read from the iterator. Finally, since the Boa identifier table does not have the split version, we did not verify the result and compute accuracy in this query.

By observing the queries from the case study, we learn various benefits from having views. First, users can create their own dataset in Boa just like the gold set and word list queries, and they can reference the queries and perform analyze on the dataset easily. Second, using sub-queries
Figure 5.5 The Boa query file `BoaIdentifiers.boa` for case study 1. The query collects the identifiers for each corresponding types from Boa dataset. The query job is assigned with job id 300.
if (input.id == "1061331") {
  Result: output collection[original: string] of result: string;

  wordIterator: { word: string };

  wordSet: set of string;
  while (J200/Dictionary >> wordIterator)
    add(wordSet, lowercase(wordIterator.word));
  while (J200/Abbreviation/Words >> wordIterator)
    add(wordSet, lowercase(wordIterator.word));
  while (J200/StopWords/Words >> wordIterator)
    add(wordSet, lowercase(wordIterator.word));
  ...

  processID := function(id: string) {
    if (!def(id) || len(id) == 0) return;

    hardWords := generateHard(id);
    softWords: stack of string;
    foreach (i: int; def(hardWords[i])) {
      if (contains(wordSet, lowercase(hardWords[i]))) {
        push(softWords, hardWords[i]);
        continue;
      }

      greedy(lowercase(hardWords[i]), softWords);
    }

    result := pop(softWords);
    while (len(softWords) > 0)
      result = pop(softWords) + "-" + result;
    Result[id] << result;
  }

  while (J300/DeclIDs >> declIterator)
    processID(idIterator.id);
  while (J300/MethodIDs >> idIterator)
    processID(idIterator.id);
  while (J300/FieldIDs >> idIterator)
    processID(idIterator.id);
  while (J300/PackageIDs >> idIterator)
    processID(idIterator.id);
  while (J300/FormalsIDs >> idIterator)
    processID(idIterator.id);
  while (J300/LocalsIDs >> idIterator)
    processID(idIterator.id);
}

Figure 5.6 The Boa query file BoaGreedySplit.boa for case study 1. The query collects the identifiers and the corresponding types from Boa dataset.
can provide modularization. As we group the three word lists into one Boa job, the queries share the same query root. This eases the pain of reference multiple queries for the user. Third, having views allows resource sharing to the research community. As long as the Boa job is set to public, the customized dataset (such as queries GoldSet and BoaIdentifier) and the experiment result can be used by other researchers to perform further analysis.

5.1.2 Case Study Result

In the original study, [Hill et al.] tests the greedy splitting algorithm with different dictionary sizes: small(50,276 entries), medium(98,569 entries), and large(479,625 entries). The testing results show that using the large dictionary gives the best accuracy with 60%, and with medium dictionary, the greedy approach produces the worst accuracy with 51%. Somehow the small dictionary set gives better accuracy (56%) than using medium dictionary set. One explanation of this phenomenon can be the medium dictionary contains more longer and less common words than small dictionary.

After executing query GreedySplit, the greedy approach produces accuracy with 53%, which is better than the worst accuracy from the original study result. We believe the the accuracy is reasonable because our word list contains about 61,215 entries, which sits between the small and mediums dictionary size from the original study. Also, the difference can come from having different sets of words in the dictionary, for the greedy algorithm heavily depends on the words contained in the word list.

Again, since there is no correct split in Boa dataset, there is no accuracy for the second splitting experiment.

5.2 Case Study 2: Impact of Developer Turnover on Software Repositories

Developer turnover represents the flow of human resource in a project. Sometimes the developers can switch team, working on different modules but staying in the same project. They are represented as IN (Internal Newcomers) or IL (Internal Leavers). If the developers are new to the project or leaving the project, they are either EN (External Newcomers) or EL (External Leavers).
The developers are denoted as ST (Stayers) if they stay in the same modules team. To find out
the impact of new developers or leaving developers on project activities and its pattern, Foucault
et al. [15] collected the developer turnover metrics on five open source projects, and analyzed the
metrics to answer three research questions. The three research questions from the original study
are shown below:

**RQ1.** Using the concepts of external newcomers and leavers at the project level, is turnover
an important phenomenon (in terms of number of developers involved) in open-source software
projects?

**RQ2.** Looking deeply into the project at the module level, is there any patterns regarding the
contributions of persistent, internal and external developers?

**RQ3.** Using the turnover metrics at the module level, is there any relationship with the quality
of the software modules?

These research questions study the distribution of newcomers and leavers, the pattern of turnover
on the developers’ activities, and the impact of turnover on the quality of each module.

5.2.1 Boa Queries

In this case study, we reproduced the experiment by mining the turnover metrics from Boa
dataset. Besides mining the turnover metrics, three more queries are written to answer the research
questions from the original work. The DAG for this case study is shown in Figure 5.7, which
contain seven Boa queries in total. To simplify the DAG, the dataset node is omitted, but each
query is still fed in the same Boa dataset. Before computing turnover metrics, we first filter out
the small projects from the dataset, which is completed with three queries: **ProjectCommitTime**, 
**ProjectDeveloperFilter**, and **TurnoverProjectFilter**. For better re-usability, we generalize each
query as much as possible.

Figure 5.8 shows the query **ProjectCommitTime**, which collects the latest and the earliest
commit time for all the projects in the dataset. One of the output variables **LatestCommitTime**
takes the project id as index and the latest commit time as the output value. Since the output type
has not supported Boa type **time**, the time is converted to an int before emitting to the outputs.
Figure 5.7 The DAG for reproducing Foucault’s research on open source projects. The three queries on the left hand side first filters out the small projects. The query TurnoverMetrics then computes the turnover metrics for the remaining projects. The three queries at the right hand side reference and analyze the turnover metrics to answer the research questions.

If the projects does not contain any commits, the project and its commit time will not appear in the output tables. Therefore, this query not only can be used to extract the latest and the earliest commit times for the projects, this query can also be used to check if the project contains any revision or not.

The query ProjectDeveloperFilter is shown in Figure 5.9. For each project, the query uses a set to keep track of the users who committed to the project. After adding the encountered usernames to the set, depending on the size of the set, the project id is emitted to the output variable AtLeast with different index values. The output collects the project ids with at least certain number of developers on the project. In this query, the number of developer is scaled with 10. For instance, if the project has 23 developers in total, the project id would be emitted to the output with indices 10 and 20.

The third filtering query TurnoverProjectFilter references the output tables from the previous queries to filter out the small projects from the dataset. In this case study, the period length is set...
Figure 5.8 The Boa query file ProjectCommitTime.boa for case study 2. The query collects the latest and the earliest commit time for each project. The query job is assigned with job id 1000.

Figure 5.9 The Boa query file ProjectDeveloperFilter.boa for case study 2. The query collects the projects id based on the number of people contributing to the project. The query is assigned with job id 2000.
TurnoverRetainedProject: output collection of pid: string;

atLeast20 : table of pid: string = J2000/AtLeast[20];
latestTime : table of t: int = J1000/LatestCommitTime[input.id];
earliestTime: table of t: int = J1000/EarliestCommitTime[input.id];
latestTimeIterator : latestTime._row;
earliestTimeIterator: earliestTime._row;
pLength := 6;
countThreshold := 20;
yearThreshold := 2;
revisionCount: map[int] of int;

if (len(atLeast20[input.id]) > 0 && latestTime >> latestTimeIterator && earliestTime >> earliestTimeIterator) {
latestT := time(latestTimeIterator.t);
earliestT := time(earliestTimeIterator.t);
if (addyear(earliestT, yearThreshold) <= latestT) {
maxP := getPeriod(earliestT, latestT, pLength);
pass := true;
for (i := 1; i <= maxP; i++)
revisionCount[i] = 0;
visit(p, visitor{
before n: Revision -> {
period:= getPeriod(earliestT, n.commit_date, pLength);
revisionCount[period] = revisionCount[period] + 1;
stop;
}
});
for (i := 1; i < maxP; i++) {
if (revisionCount[i] < countThreshold) {
pass = false;
break;
}
}
if (pass)
TurnoverRetainedProject << input.id;
}
to **6 months**. The commit history has to be longer than **2 years**, and each period has to contain at least **20 commits**. Furthermore, the project must have at least **20 developers** working on the project before. With these constraints, we can filter out the projects that have little activities and low number of developers. The 2 year activity limit is necessary so that we can have enough number of periods to study the trend of the turnover metrics. The query is shown in Figure 5.10. From line 2 to line 4, the query references three external tables from queries ProjectCommitTime and ProjectDeveloperFilter. To fulfill the constraints, the query first filters out the projects having at least 20 developers in line 21 and the query filters out the projects having less than 2 years commit history in line 24. For the remaining projects, a visitor is used to sum up the number of commits in each period. The function `getPeriod` takes two time values and a period length as arguments, returning the number of period the second time value is in since the first time value. A map `revisionCount` is used to keep track of the revision count in each period. Once done counting, except the last period, if the rest of the period commit counts all pass the threshold value (20), the project id is emitted to the output variable `TurnoverRetainedProject`. The reason of ignoring the last period is because the length of the last period can vary from 1 day to close to 6 months, so we only focus on the previous periods.

After filtering out the projects for the case study, query TurnoverMetrics is used to compute the turnover metrics on the retaining projects. The query can be found in Figure 5.11. Due to the length of the query, only the output variables and the referenced tables are shown in the figure. For the output tables, the first column with type `string` is the project id. Except the output variable `PeriodCount`, the second column in the reset of the tables with type `int` denotes the period index. The annotations of the other column field names are shown in line 2. Three external tables are referenced in the query. Besides the project filter table, we also need the commit time tables to get the period index while computing the turnover metrics.

To compute the developer turnover metrics, we first use a visitor to collect the name of the developers along with the modified module names in each period. The module is determined by directory structure of the modified files. Along with the name of developers contributing in each
# output variables

2. Commit_M: output sum [string][int][m: string] of c: int;
3. BFCommit: output sum [string][int] of c: int;
4. BFCommit_M: output sum [string][int][m: string] of c:int;
5. Module: output set [string][int] of m: string;
7. D_MP: output set [string][int][m: string] of n: string;
8. EN: output collection [string][int][m: string] of n: string;
9. EL: output collection [string][int][m: string] of n: string;
10. IN: output collection [string][int][m: string] of n: string;
11. IL: output collection [string][int][m: string] of n: string;
12. ST: output collection [string][int][m: string] of n: string;
13. ENA: output collection [string][int][m: string] of a: float;
15. INA: output collection [string][int][m: string] of a: float;
16. ILA: output collection [string][int][m: string] of a: float;
17. STA: output collection [string][int][m: string] of a: float;
18. ENAsum: output sum [pid: string] of a: float;
22. STAsum: output sum [pid: string] of a: float;
23. ENCount: output sum [string][int] of c: int;
24. ELCount: output sum [string][int] of c: int;
25. INCount: output sum [string][int] of c: int;
26. ILCount: output sum [string][int] of c: int;
27. STCount: output sum [string][int] of c: int;
28. ENMCount: output sum [string][int][m: string] of c: int;
29. ELMCount: output sum [string][int][m: string] of c: int;
30. INMCount: output sum [string][int][m: string] of c: int;
31. ILMCount: output sum [string][int][m: string] of c: int;
32. STMCount: output sum [string][int][m: string] of c: int;
33. Activity: output sum [string][int][m: string][n: string] of c: int;

# table & iterators

36. retainedProject := J3000/TurnoverRetainedProject;
37. latestTime : table of t: int = J1000/LatestCommitTime[input.id];
38. earliestTime: table of t: int = J1000/EarliestCommitTime[input.id];
39. latestTimeIterator : latestTime._row;
40. earliestTimeIterator: earliestTime._row;
41. ...

Figure 5.11 The Boa query file TurnoverMetrics.boa for case study 2. The query computes the turnover metrics from the retaining projects. This query job is assigned with job id 4000.
period, these two metrics are emitted to the output variables $D_{MP}$ and $D_P$. In the same visitor, we also compute the developers’ activities in different modules per period. The way we assess the activities is to count the number of files changed from the developers. The metric is emitted to the output table $Activity$. Then, we follow Foucault’s approach to compute the turnover metrics (EN, EL, IN, IL, and ST) and the turnover activities metrics (ENA, ELA, INA, ILA, and STA). Each of them is also emitted into an output table for the query. The sum of each activity metric across projects are also computed as outputs for the second research question. We also output the metrics that count the number of turnover metrics in each module ($EN_{Count}$, $EL_{Count}$, etc). Besides the turnover metrics, we also compute the number of bug fixed revisions and emitted them to outputs $BugFixCommitCount$ and $BugFixCommitMCount$. The bug-fixed metric is required to answer one of the research questions. The reason why we did not compute them in a separated query to realize modularization is because we want to count the commits for each period and each module. Since both of them are specifically designed for this case study, we decided to compute them in the same query. Notice that the metrics mentioned above all contains a period index column. Since each project can have different number of periods, the output variable $PeriodCount$ contains the period count for each project, so that there is no need to recompute the period index while traversing the metrics.

To answer the first research question, we study the ratio of external newcomers and external leavers to all the developers in the same period. If the ratio is significant large, it confirms that turnover is a significant phenomenon among open source projects. Also, we compute the stayers conversion rate to see how much percentage of developers have been stayers across the project history.

The query $TurnoverRQ1$ shown in Figure 5.12 is written to find the ratio of external metrics and the stayers conversion rate. The query references 8 tables from query $TurnoverMetrics$. To compute the ratio of external newcomers and external leavers, we need the number of developers in each turnover metrics and the number of periods in the project. After importing the metrics into maps, we compute the ratio for each period by dividing the number of external newcomers
Figure 5.12 The Boa query file **TurnoverRQ1.boa** for case study 2. The query identifies the significance of turnover by computing the ratio of externals across projects’ history and finding the conversion rate for each project.
and external leavers by the total of the turnover metrics for the period. The ratio is computed and emitted to output \texttt{ENLratio} in line 50. For the stayers conversion rate, we reuse the stayer table and the developer table to compute the number of developers that have been stayers and the number of developers that have been on the project throughout the time. Since we do not need the period index and module information from the table, we filtered out the columns by using wildcards in line 6 and line 7. After importing the developers’ username into sets, we compute the stayers conversion rate for the project in line 54. The outputs \texttt{ENLratioStats} and \texttt{ConversionRateStats} count the frequency of the ratios and conversion rates every 0.1 level, so that we can observe the turnover impact among the projects.

To answer the second research question, we wrote query \texttt{TurnoverRQ2} to compute the ratio of each turnover activity metric to the total activities for each project. The query is shown in Figure 5.13. The query references 5 turnover total activities tables from \texttt{TurnoverMetrics}, which contains the total activities for a turnover metric across project life time. After reading the total activities values from the tables, the total activity is computed in line 54. If the total activity is zero, the project is filtered. In the if statement, we compute the ratio for each metric to the total activity. The function \texttt{outputMax} takes the activity ratios and highest activity ratio among five metrics, the function also find the highest activity ratio between newcomers and leavers. The result would emit an integer 1 to the corresponding outputs to study the pattern among the projects. The activity ratios are also emit to the output table \texttt{Ratio}, including the newcomer activity ratio and leaver activity ratio.

The third research question studies the relationship between the turnover metrics and the project quality. To assess the project quality, we compute the density of bug-fixed commits per module. The tables \texttt{Commit} and \texttt{BFCommit} are referenced to compute the bug-fixed density for each module. The Spearman correlation tests are performed to compute the correlation between the turnover activity metrics (ENA, INA, etc) and the bug-fixed commits’ density. The tests compute a correlation coefficient for each period, which is a number ranges from -1 to 1, representing the perfect negative and perfect positive correlation. A 0 correlation coefficient represents no correlation.
Figure 5.13 The Boa query file TurnoverRQ2.boa for case study 2. The query studies the patterns the contribution from persistent, internal and external developers by computing the ratio of each turnover activities to total activities.
among two series of data. To fully reproduce Foucault’s approach, we also compute 95% confidence intervals of the correlation coefficients for each turnover activity metrics. If the end values of confidence interval are both either positive or negative, then we have 95% of confidence to say the turnover activity metric has positive or negative impact on the quality of software modules. For each activity metric, we also count the number of times it has positive correlation or negative correlation with bug-fixed density.

The first output Correlation collects the correlation coefficients for the turnover activity metrics in each period. The second output variable CorrelationCI collects the confidence interval for the metrics in the first output table. Since the confidence interval is computed from the correlation coefficients for all the periods, the confidence interval output table does not have period column like the correlation table. To store the confidence intervals into table, a tuple type ci is defined with two floats in line 2. The tuple type is also used in the output definition in line 4. However, since Boa does not support having tuple values in the key-value pairs, we store all the confidence intervals in the index keys and use an dummy integer as values. For the rest of the output tables, each table counts either the positive correlation or the negative correlation between an activity metric and the bug-fixed density.

With views, Boa not only becomes an end-to-end analysis tool, users can also perform filtering more easily. For instance, after we compute the retained projects for this case study in query TurnoverProjectFilter, we only need to read the row from the project retained table to know if the project is filtered or not. One example is shown in line 23 from query TurnoverRQ1. Furthermore, the case study proves again that the feature views makes all the query outputs become new datasets, and the researchers can perform further analysis on the query outputs easily.

5.2.2 Case Study Results

In Foucault’s experiment, they studied the impact of developer turnover on five open source projects written in four different programming languages. The five projects are Angular.JS, Ansible, Jenkins, JQuery, and Rails. However, in the case study, we study the impact of developer turnover on open source projects in Boa dataset, which contains 7,830,023 projects in total. How-
Figure 5.14 The Boa query file `TurnoverRQ3.boa` for case study 2. The query computes the correlation and the confidence intervals between turnover activity metrics and the density of bug-fixed commits.
ever, only 1,676 projects are retained for the case study after query `TurnoverProjectFilter`. Since the scale of the dataset is way different, we answer the research questions by focusing on the overall trend of turnover metrics instead of studying the impact at the module level for each project. However, the outputs from query `TurnoverMetrics` do provide the turnover metrics at the module level. Even though the dataset is different, we will still explain and observe the differences in the mining results for each research question.

**RQ1.** By observing the number of external newcomers, external leavers, and stayers, Foucault et al. found that throughout the histories of the projects, at least 80% of developers are newcomers and leavers. Also, the conversion rate for the projects are between 8% and 19%, meaning that only relatively low number of newcomers become stayers throughout the time. These results show that developer turnover is an important phenomenon among open source projects.

\[\text{Number of Projects for Each Level of External Ratio}\]

![Figure 5.15](image)

**Figure 5.15** The number of projects for the ratios of external newcomers external leavers throughout projects’ life time.

Since there are 1,676 open source projects in the dataset, we focus on the overall trend of average external ratio as well as the conversion rate. The average external ratio is the ratio of external newcomers and external leavers throughout project histories. The number of projects in each ratio level is shown in Figure 5.15. Most of the projects have average external ratio from 0.2 to 0.4. There are relatively few projects having high external ratio. This shows that in average,
the developers in the project have roughly one third of changes of leaving the project, and there is same opportunity that having external newcomer to the projects.

![Conversion Rate Distribution](image)

**Figure 5.16** The number of projects for each conversion rate level.

![Activity Metric Distribution](image)

**Figure 5.17** The number of projects for each highest activity metric. Each category contains the number of projects that have the highest total activity metric throughout project life times.

The conversion rate result is shown in Figure 5.16. Among 1,676 projects, almost 500 projects have conversion rate between 30% and 40%, and most of the project conversion rates fall between 20% and 50%, which is greater than the conversion rates obtained in the original studies. This shows that in average, about one third of chances that the newcomers would become the stayers.
in the project. Also, there are 2 projects having conversion rate with 100%, meaning that the newcomers all become stayers throughout project life times. In this research question, we also believe that turnover is an important phenomenon because open source projects tend to have certain external ratios and conversion rate.

**RQ2.** Through analyzing the turnover activity metrics, several patterns are identified among the open source projects. [Foucault et al.](#) found out that in Angular.JS, most of the activities come from stayers and external leavers. In Ansible, all categories contributes similar levels of activities. For the projects Jenkins, JQuery and Rails, internal newcomers tend to contribute more activities than other categories of developers, and external newcomers and external leavers are more focused on certain modules and do not contribute more than half of the module activities.

For this research question, we compare projects activities metrics throughout the project life times. The result of comparing each activity metric is shown in Figure 5.17. Among 1,676 projects, the stayers contribute the most in about 75% of the projects. This makes sense because the stayers have more knowledge to the projects compare to developers in other categories. However, the external leaver is the least activity metric among the projects, but this does not mean that external leavers contribute less activities to the projects, for the analysis only looks at the number one metric in the projects, meaning that external leavers can contribute the second most activities in

Figure 5.18 The number of projects for the higher newcomer activities and higher leaver activities.
many projects.

To compare the newcomer activities and leaver activities, we perform the same analysis on these two groups of metrics, and the result is shown in Figure 5.18. In this analysis each category includes external and internal metrics, and the result shows that the newcomers contribute slightly more than leavers among the projects.

**RQ3.** After computing the correlations between turnover activities and bug-fixed density, Foucault et al. noticed that most of the projects have a positive correlation between external newcomers’ activities and the bug-fixed density. Also, the stayers’ activities have a strong positive correlation with the bug-fixed density. However, none of the projects’ external leavers’ activities has significant relationship with the bug-fixed density.

In this case study, we also compute the correlations and the confidence intervals between turnover activities and bug-fixed density. Based on the confidence intervals, we count the projects having positive and negative correlations for each turnover activity metric. The positive correlation result is shown in Figure 5.19. Contrary to the original study, the stayers contribute much less bug-fixed revisions than other metrics. However, for each metric, no more than 50 projects have positive correlations with bug-fixed density. Figure 5.20 shows the negative correlation between
turnover activity metrics and bug-fixed density. The stayer activities in more than 400 projects have negative correlation with bug-fixed density. This might due to the high stayer activities observed from Figure 5.17. Given the fixed number of bug-fixed revisions, the more commit activities the developers have, the less bug-fixed density the team results.

Even though some of the results do not match the original study, noticed that we applied the analysis on 1,676 open source projects, and Foucault et al. only focuses on five open source projects. Therefore, our analysis results are more general to all open source projects, but the turnover impact for a project can vary on a case-by-case basis.

5.3 Performance Analysis on Caching Behavior

In this section, we discuss how we evaluate caching behavior and its evaluation results.

5.3.1 Analysis Method

In views, caching happens when the outputs have existed for the referenced job, so there is no need to re-execute the query jar to re-generate the outputs. Therefore, we assume the caching performance should be better than the runtime performance without caching. In this experiment, we uses the queries from the first case study to test the caching performance, and we will split identifiers on the gold set.
To test the caching performance, we designed three testing scenarios. For each scenario, we will execute query **GreedySplit** and collect 10 runtime performances to have reliable result for each scenario. The first scenario is to estimate the runtime performance for running the queries without caching, and all the queries are executed for each run.

The second scenario is to execute queries according to Figure [5.21]. The gray nodes represent the cached query outputs. The outputs for query **WordLists** and its sub-queries **WordLists/Abbreviation** and **WordLists/StopWords** are cached, and only the queries **GoldSet** and **GreedySplit** are re-executed.

In the third scenario, besides caching the outputs for query **WordLists** and its two sub-queries, we also cache the outputs for query **GoldSet**. Only the query **GreedySplit** gets re-executed. The scenario DAG is shown in Figure [5.22].

5.3.2 Analysis Result

The runtime performance for each scenario is shown in Figure [5.1]. The first scenario has the highest runtime performance with median 147 seconds, and the third scenario has the fastest runtime performance with median 29 seconds. Noticed that the cached job **WordLists** includes...
Figure 5.22 The DAG for the second caching scenario in case study 1. The gray nodes represent the caching queries for the experiment.

Table 5.1 The runtime performance for each caching scenario. The runtime performance is recorded in seconds, and the standard deviation is round to the third decimal point.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Median</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Caching</td>
<td>147</td>
<td>146.6</td>
<td>2.270</td>
</tr>
<tr>
<td>Cache WordLists</td>
<td>56.5</td>
<td>56.4</td>
<td>0.966</td>
</tr>
<tr>
<td>Cache WordLists &amp; GoldSet</td>
<td>29</td>
<td>28.9</td>
<td>0.876</td>
</tr>
</tbody>
</table>

two sub-queries **Abbreviation** and **StopWords**. The runtime performances are very consistent with low standard deviation. These results testify our assumption, showing that with more cached queries during the run, the faster the runtime performance. This is reasonable because having more cached queries means there are less queries to execute.

The runtime performances for the 5 Boa query from case study 1 is shown in Figure 5.2. Compare the result of the third scenario to the performance for **GreedySplit** query, the performance difference (7 seconds) is the workflow overhead by running Oozie. If we add up the execution times for queries **GreedySplit** and **GoldSet** and compare it to the second scenario, the workflow overhead increases up to 14.5 seconds. Lastly, if we add up the all the individual query performance and compare to the first scenario, the workflow overhead becomes 32 seconds. Therefore, the more queries executed in a Oozie job, the more workflow overhead it produces. This makes
Table 5.2 The runtime performance for each query in case study 1. The runtime performance is recorded in seconds, and the standard deviation is round to the third decimal point.

<table>
<thead>
<tr>
<th>Query Name</th>
<th>Median</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoldSet</td>
<td>21</td>
<td>21.3</td>
<td>0.483</td>
</tr>
<tr>
<td>WordLists</td>
<td>31</td>
<td>30.5</td>
<td>0.707</td>
</tr>
<tr>
<td>WordLists/Abbreviation</td>
<td>21</td>
<td>20.9</td>
<td>0.568</td>
</tr>
<tr>
<td>WordLists/StopWords</td>
<td>21</td>
<td>20.9</td>
<td>0.738</td>
</tr>
<tr>
<td>GreedySplit</td>
<td>21</td>
<td>21.2</td>
<td>0.422</td>
</tr>
</tbody>
</table>

It makes sense because running more queries means there are more workflows to process. Even though using views creates workflow overhead, we believe that the benefits of having views (output reusing, modularization, and dataset sharing) outweigh the workflow overhead.
CHAPTER 6 RELATED WORKS

In this chapter, we examine prior research related to views in Boa. We organize it into two categories: 1) prior research in the database community relating to views and 2) prior research in the mining software repositories/software engineering community related to large-scale software repository mining.

6.1 Database Views

Among the view related database works, view maintenance has been discussed the most, and many approaches and struggles have been presented. Srivastava and Rotem [37] presented a parameterized approach to combine the users’ and systems’ needs to maintain views. Gupta et al. [18] introduced counting algorithm and DRed algorithm for maintaining recursive and non-recursive views. Griffin and Libkin [17] proposed an approach which is based on equational reasoning to incrementally maintain the updates from base relations to materialized views. Ross et al. [33] presented an view maintenance plan that exploits additional views to reduce the time cost of maintaining views. Mistry et al. [29] proposed a materialized view maintenance plan that uses both transient and permanent materialized views in their maintenance technique to achieve the lowest estimated maintenance cost. Lee et al. [26] proposed optimal delta evaluation to perform efficient incremental view maintenance and reduce the accessing overhead to views and base relation. In Boa, we don’t need to worry about updating base relation and views. Once the Boa dataset and views are generated, they will be stored on the cluster and become immutable.

Besides improving the algorithms and reducing the cost of incremental view maintenance, the following four studies introduced different types of view maintenance. Due to maintenance overhead, in some occasions, deferred view maintenance might be needed. Colby et al. [7] provided an algorithm that will perform view maintenance of the user’s choosing time. The algorithm not only strives to minimize the execution time, it also avoids the state bug that might happen with immediate view maintenance. The second case happens when the users update the entries directly from
views instead of base relation, the changes then need to be updated to the base relations. Lechtenbörger and Vossen [25] proposed an optimized approach to handle this case, which exploits the expressions from views to compute minimal view complements. Furthermore, view complements can be used to create temporal views for other maintaining purposes. In the third study, Johnson and Shkapenyuk [23] developed an efficient algorithm to propagate updates to maintain views in a streaming warehouse, which requires processing different update actions in real time. In 1997, Gupta et al. [19] performed a study that compares adaptation techniques and re-materialization techniques when the definition of a view is updated. Adaptation methods use the provided query, the old view, and the base relations to generate the updated view. The study shows that adaptation techniques can perform significantly faster than re-materialization techniques, but in some cases re-materialization techniques outperform adaptation techniques.

Most views maintaining focuses on conventional relation model. However, some studies provided different solutions of maintaining views on object-oriented model. Abiteboul and Bonner [1] introduced virtual classes and imaginary objects to explain how view mechanism should work in object-oriented database. Alhajj and Polat [2] came up an idea that adds a modification list to each base object to handle incremental updates on views, so that the system no longer needs to recompute a view from scratch every time it is accessed. Ali et al. [3] proposed MOVIE as a solution to incrementally maintain views in ODMG-compliant object database. According to the experiment result, MOVIE can perform more than a hundred times faster than re-materialization. However, in certain cases, MOVIE costs nearly as much as re-materialization. Again, none of these view maintenance solutions are applicable to our case.

Instead of optimizing and maintaining views, the following studies show how views can be used to optimize queries. In 1995, Chaudhuri et al. [5] proposed an efficient approach that optimizes queries. The algorithm would analyze the queries and the existing materialized views in the database. If there is a more efficient query that utilizes views and performs the same task, the optimizer will run the efficient query instead. However, the approach only applies with query with no aggregates or group-by clause. For aggregated queries, Cohen et al. [6] presented a sound but
not complete algorithm to optimize the query performance in 1999. Not only searching for usable materialized views in the database, the algorithm also utilizes the results from previous queries to rewrite the aggregate query. In the future, we can perform similar optimization that modifies Boa query with existing views to enhance performance.

6.2 Repository Mining Frameworks

Many MSR frameworks have produced invaluable contribution on software developments, pointing out the importance of historical project data. Each MSR framework can serve different purposes, such as mining the projects to look for better solution with a given input, mining projects’ commits to compare and study the impact, or gathering projects data into a database for MSR community, etc.

The following four mining frameworks focus on finding better solution for the users from different aspects, such as software artifacts, API usage, etc. Hipikat [8] is a tool that gathers information from a project’s archives, forming an implicit group memory to recommend artifacts to users. When a newcomer is trying to perform a task in the project, he or she can run a query to Hipikat. Hipikat would return a set of suggested artifacts related to the task. The artifacts include bugs and features in the project, any messages posted on forums related to the project, etc. Since not all the open source projects meets Hipikat mining requirements, Hipikat focus particularly on the Eclipse extensible integrated development environment project. MAPO [39] is a mining framework that mines API usages from open source repositories. When the developers are searching for a certain API usage through the existing source code search engines, the number of search results can be too many and too complicated to learn the correct usage. To mine the API usages, given a query with the API information, MAPO utilizes several existing code search engine such as Koders and CodeBase to gather data. The framework then extracts the method calls and performs a set of processes and filtering, returning with a short list of API usages to the user. Sourcerer [27] is an infrastructure mining and searching open source code not only with text but also from the structural aspects. Sourcerer not only targets on well-known repositories like Sourceforge, it also utilizes code crawlers to gather source code from other websites. Once
the project is downloaded, Sourcerer would keep a local copy of each project release, and then a feature extractor would be used to parse and extract relations and fingerprints into its relational database. Sourcerer uses Lucene as a text search engine for indexed searches. Sourcerer also contains a Ranker component to rank the entities with non-text based approach. Comparing to Google search engine, the searching and retrieval performance is improved after using Sourcerer. BUMPER [30] is a tool for mining bugs and fixes to provide solution to buggy code. Since there is not much linking relationship between the bug tracking tools and version control systems, when a developer encountered an existing bug, he or she has to rely on classical search engines or search through version control system to find solution. BUMPER connects to all bug tracking and version control systems, and it downloads the bugs and fixes data into one database. The developers can use web-based interface provided in BUMPER to run queries, mining the bug reports and find any existing or similar solutions to the buggy code. The evaluation result shows that compare to Google search engine, using BUMPER is faster for the developers to find a solution for a bug.

The following three mining tools focus on mining code changes. They are all designed and built as Eclipse plug-in tools. Chianti [32] is a tool performing impact analysis on Java programs. The tool compares two versions of a project and identifies the affected tests after the change. Chianti can identify the affecting changes that modify one of the test cases. Chianti can help developer discover certain changes that are responsible for a test failure. DynaMine [28] mines the revision history and discovers error patterns in software systems. Given a software system, DynaMine first identifies coding patterns from the incremental changes between each revision, storing the patterns in a database. DynaMine then mines the database, performing dynamic analysis on the patterns to diagnose the faulty patterns. Given a pattern selected by the user, DynaMine also provides relevant existing patterns and violation statistics for the pattern. APFEL [41] processes CVS archives and stores fine-grained changes as tokens in a database. The changes include method calls, exceptions, and variable usages. APFEL is inspired from Hipikat. Hipikat doesn’t support such fine-grained changes in the project artifacts. APFEL applies token-based approach to tokenize source code and identify the type of changes on each element. Even though APFEL has its own limitation with its
parsing approach, APFEL is an useful tool to mine developers usage on fine-grained level changes.

The following two frameworks specifically mine the commits from software repositories. RepoGrams [34] is a tool that mines Git repositories and visualizes the project commits. RepoGrams aims to support software engineering (SE) researchers to easily compare and verify their selected projects for their researches. For better comparison, RepoGrams supports two normalization modes, twelve built-in metrics for visualization and several interaction options for the users to mine the projects. The authors pointed out Boa’s usefulness in locating an initial set of target projects for SE researchers, along with the weakness saying that Boa doesn’t support such visualization like in RepoGram to help the researchers visually compare projects’ commits. PyDriller [36] is a MSR framework written in Python. To extract repository information with PyDriller, the user has to provide the Git path, and the framework will extract and return the commits in the repository at runtime. Given a specific start date and end date, PyDriller also supports API to filter commits for the users. To enhance runtime performance, the repository objects such as Commit and Modification are lazily loaded. Compare to another Python framework GitPython, PyDriller can produce the same result with 50% less LOC in average. The main difference between PyDriller and Boa is that PyDriller can mine every current project on GitHub, but Boa only supports mining the snapshots from Java projects on GitHub or SourceForge.

The following two frameworks act as data gathering tools, providing repository dataset for MSR researchers. GHTorrent [16] is a web-based project that mirrors the event streams from GitHub. GHTorrent uses multiple hosts to collect repository data through GitHub’s API. The user can run MySQL or MongoDB queries to download data. GHTorrent aims to provide software repository dataset to the research community. The difference between GHTorrent dataset and Boa dataset is that GHTorrent collects the event stream, but Boa dataset provides AST nodes for the user to traverse and mine the repositories. Perceval [10] is a data gathering tool that supports more than 20 data sources, including version control systems, issue tracking systems, mailing and social media tools. Perceval can be used as either a stand along tool or a python library. The data gathering result will be returned in JSON format, which can be used to perform analysis with other
tools easily. Besides data gathering, Perceval also supports data visualization with GrimoireLab toolset, which also allows filtering functionality for the users to perform mining analysis. Unlike Perceval, Boa only supports data sources from GitHub and SourceForge. However, Boa provides a domain specific language that allows the users to write Boa queries to perform customized analysis.

Spark [40] is an interactive data analysis tool, which utilizes resilient distributed dataset (RDD) objects to reuse data with map and reduce operations. With RDD stored in memory, the dataset no longer needs to be reloaded from the disk for each MapReduce job, which produces significant overhead when running iterative jobs. Their study shows that Spark can perform interactive machine learning jobs ten times faster than Hadoop’s MapReduce job. However, when it comes to perform MSR tasks, Boa outperforms Spark because the size of dataset is extremely large. It is better to store the dataset on disk instead of in memory.

MetricMiner [35] is a web application that mines software repositories with metrics. It runs on a cloud infrastructure. MetricMiner is built on rEvolution, a command-line tool that clones the repository data and stores them into a database. Once the repository is cloned, MetricMiner automatically analyzes the data and generates several metrics such as cyclomatic complexity and lack of cohesion of methods (LCOM). The researchers can run SQL query in MetricMiner to mine the data. MetricMiner also supports statistical analysis on the output by dynamically generate and execute R script. To achieve better run-time performance, each task in MetricMiner is executed asynchronously. In Boa, the researchers can write Boa queries to compute different metrics for each project as well. However, without supporting views, Boa cannot perform statistical analysis on the query result.

GreenMiner [21] is the first hardware based mining software repositories that mines the software energy consumption on mobile applications. Usually, there are not much energy performance data existing as project artifacts. Given a mobile application, Green Miner would run a set of test cases and record the energy usage. After testing, the framework aggregate the energy consumption data and visualize the result. The users can also define its own set of tests to test the applications. Though the case studies, not only GreenMiner can be used to mine energy consumption of a mo-
bile application, the framework can be used to debug the application for better power usage. Boa
dataset does not contain any trace of energy usage for the projects, thus cannot be used to compute
energy consumption.

Candoia [38] is a platform and an ecosystem for building and sharing MSR tools, and it works
like an appstore, and each MSR tool is an app. The goal of Candoia is to provide portable, adopt-
able, and customizable MSR tools for the researchers. The platform not only supports dataset
generation, it also allows the users to build, install, and share the customized MSR tool as an app
on Candoia. Furthermore, Candoia provides a set of well-known domain specific languages for
customization. Each language is used to express certain domains in the app. For instance, HTML
and CSS are used for layout and visualization. JSON is used to describe structural information,
and Boa is used to express MSR logic. The evaluation engine in Candoia is inspired from Boa
query engine. Candoia uses an interpreter version of Boa. The engine runs on a single node and
uses processes and threads to run MSR apps in the parallel manner.
CHAPTER 7  FUTURE WORK

In this chapter we discuss some potential future extensions and optimizations for views.

7.1 Language Extensions

Forward declaration of views. When referencing internal views, currently users can only reference in a top down order. In the future, we also want to support forward declarations on views so users do not have to define the views before referencing them. Once we support forward declarations of views, users could accidentally create cycles in the DAG. Therefore, we also have to implement cycle checking to support this optimization. If any cycle is detected, the current job would fail. With this support, users could define views anywhere in the program, which gives them more flexibility and allows them to organize/structure the program in ways that make more sense.

```
1 top5: output top(5) of string weight int;
2 v: table [pid: string] of Project = J23456/tenCommitterProject;
3 r: v._row;
4 astCount := 0;
5 for (v >> r) {
6   visit(r._2, visitor {
7     ...
8   });
9   top5 << r.pid weight astCount / 1000000;
10  astCount = 0;
11 }
```

Figure 7.1 An example of Boa query that references a view with AST type `Project` to find the top five largest projects have ten committers.

Supporting AST types. So far our views only support `int`, `float`, `string`, `bool`, and `tuple` types. If views can support AST types such as `Expression` and `Statement`, users can reference Boa jobs that contain AST types in the output, creating workflows to analyze code behavior and solve more complex MSR problems. An example Boa program referencing views with AST type is shown in 7.1. The example program references a view containing two columns. The first column is the
project id with type `string`, and the second column is the project AST from the dataset. As the view name `tenCommitterProject` suggests in line 2, the view contains projects having ten committers. In the main program, we can utilize the view to find the top five projects containing the most number of AST nodes, which is tracked by the variable `astCount`. With this extension, users can pass AST types through views to solve more complex problems, such as the one shown in the example.

```plaintext
1 lang: argument of string default "java";
2 counts: output sum[int] of int;
3 exists (i: int; lowercase(input.programming_languages[i]) == lang)
4     visit(input, visitor {
5         before n: Revision -> counts[yearof(n.commit_date)] <<= 1;
6     });
```

Figure 7.2 An example of Boa query declaring and using an argument variable to count the number of commits over years.

```plaintext
1 o: output collection[int] of float;
2 view1 := @hungc/JavaYearlyCommits(lang="python");
3 view2 := @hungc/JavaYearlyCommits(lang="c++");
4 view3 := @hungc/JavaYearlyCommits(lang="ruby");
5 view4 := @hungc/JavaYearlyCommits(lang="c#");
   ...
```

Figure 7.3 An example of Boa query referencing Figure 7.2 several times and set the argument variable to various languages.

**Supporting arguments on views.** Another potential extension to the feature is adding arguments while referencing views, so the intermediate programs can produce output according to the arguments’ value. Whenever an argument is defined, the default value needs to be defined as well to deal with a case that has no argument provided. An example view usage with arguments is shown in 7.2 and 7.3. The goal of the first query is computing the number of commits under Java files over years. However, since the same research question can be replicated to programming languages other than Java. In the first query, an argument variable `lang` is defined with type `string`, and the default value is "java", which will be applied when executing current query
and when the view is referenced without specifying the argument. In the second query, the view of the first query is referenced four times, each with different \texttt{lang} argument. Once the second query is executed, Boa workflow will check if the output for each argument value exists. If yes, the output file will be read in directly. Otherwise, the first query will be executed again with new argument value. This extension significantly increases the reusability because the user do not need to write many Boa queries with much duplicated code. They can treat each view as a MapReduce function with customized arguments.

**Global static variables.** Based on our current design, the scopes of views are independent to each other. Inside a view, even if the views are defined in the same Boa query, the user cannot access any variable belonging to another view. This also makes sense because each view is a different MapReduce program. However, if there exists a way of sharing resources between views, users can use global variables across views to solve problems. One possible approach to achieve this is to extend the Boa language to support global state variables in the main program, so that in the scope of views, users can access and modify the variables. One problem of this design is the execution order of the views in the same Boa query. We need to decide if users can customize the execution order or it will follow the top-down order or other approaches to decide the execution order at run time.

**Filtering without dropping column.** While filtering views, users can use wildcards to filter the column with any values. However, once the filter has been applied, the column cannot be retrieved again. For example, given a table \texttt{t1} with type \texttt{table [string][string][string] of int}, and another table \texttt{t2} is assigned as following: \texttt{t2 := t1["foo1"]_["foo2"]}. While reading rows from \texttt{t2}, we are basically traversing \texttt{t1} with the following conditions: the first column has to be "foo" and the third column has to be "foo2". However, since \texttt{t2} now only contains one column, we can only access the last column from \texttt{t1} but not the second column. This can be a problem when the user wants to access the column that is filtered with wildcard. With our current design, one way to get around this problem is reading rows from \texttt{t1} and check each column value manually. In this way, they can still access every columns in the table, but the
code can be very tedious. To fix this problem, we are planning to add another symbol such as ‘.’ to the filter. The symbol ‘.’ works the same as wildcard except keeping the column from the table. When the user defines \( t_2 \) as \( t_1["foo1"][][]["foo2"] \), the filter still allows any values in the second column. The only difference is that now table \( t_2 \) contains two columns instead of one. The first column is the second column from \( t_1 \), and the second column is the last column from \( t_1 \). In this way, users can apply filters but still preserve the access to intermediate columns at the same time.

7.2 Runtime Optimizations

**Filtering.** Currently, our filtering approach is to check each row and see if the values match the indices from the filters or not. However, since users mostly likely don’t know the actual values stored in the table, they might end up providing a filter that won’t match any rows. If this is the case, traversing the whole table can cause much overhead, giving time complexity \( O(n) \). One solution is to apply bloom filter on the table columns. For every table, we can create a bit vector for the first column. When the user applies a filter to the table, we can use the bit vector to test the filter index. Bloom filter eliminates false negatives, so if the test fails, meaning that the index has never appeared in the column. When the user reads in a row, the table can return false immediately without traversing.

**Table length function.** Boa supports the \( \text{len()} \) function on tables, which takes a table as an argument and returns the number of rows in the table. However, the way we implement the function is currently very inefficient. Currently, we clone the current table iterator and scan the entire table every time the function is called. Therefore, given a table stored in a variable \( v \), when the function \( \text{len}(v) \) is called 10 times, the whole table will be traversed 10 times during execution, producing time complexity \( O(n) \), which \( n \) denotes the number of rows in the table. One solution to improve performance is to keep track of the number of rows for the table during the output phase. When we are writing rows into output, we can use the first row to store the length and other meta data. Next time, when a table is referenced, we can extract the length information from the first row directly without traversing the table. Also, while reading rows, we just need to make sure to start reading
from the second row. This optimization improves the time complexity from $O(n)$ to $O(1)$.

**Cache management.** Since users now can reuse outputs, the availability of outputs becomes important. If the output is not found, Boa needs to rerun the query to reproduce one, and more queries might get involved. This can drastically increase runtime overhead and also increase the number of worker nodes used for the Job. However, the storage space on the cluster is limited. To save some storage space, we need to implement a caching system that would keep the frequent used table outputs and delete the ones that are less used.

7.3 Boa Website Improvements

**Support Tag Names.** Currently we only support using job ids as query roots to reference external views. We want to also support tag names as query roots in the future, so that Boa users can tag a job and increase the readability of referenced view paths. To achieve this, we have to update the web interface, giving users an interface to create and modify the tag names.

**Views catalog.** Currently users can re-use any of their prior jobs as a view. However, besides the user’s own queries, there is no way for a user to discover available queries from others that can be reused. We would like to provide a view catalog for users on Boa’s web interface, so that they can view the query description, the reference path and the content of the query. The view catalog can also provide filtering so that users can filter Boa jobs by certain users and dates. If the tag name feature is enabled, we can also support search-by-tags function in the views catalog. This feature should encourage MSR researchers reusing prior queries in Boa and sharing their own queries for the wider MSR community.

**Static code analysis.** When users are writing queries on the Boa website, we can perform static code analysis on view paths and diagnose the returned table types. We can use the characters ‘@’ and ‘J’ to indicate the start of an absolute view path. As users writing the path, we can search through the database, returning a list of possible usernames, tag names, job ids, nested views and output variables to users. The interface should also warn the user if the provided view path is not valid. Additionally, if a given path is valid, we can retrieve the type information for the specific table and illustrate the type information to the user while typing. We believe this can help them
make significant less mistakes as they are working under Boa environment.
CHAPTER 8 CONCLUSION

In this paper, we introduced a new feature, materialized views, in an ultra-large-scale mining software repositories tool named Boa. Boa provides a domain specific language and a web interface for researchers to mine open source repositories. However, Boa fails to provide output re-usability and end-to-end analysis capabilities. This forces users of the tool to use other tools and post-process the output from Boa. To solve this problem, we implemented views to provide output reuse and support sub-queries in Boa programs. We designed a set of operations and functions specifically for views, that allow users to easily specify what code is a view, give it a name, and allow a type-safe way of referencing the output from a prior view. We also provide syntax to easily scan the output (which is in tabular format). To handle the runtime execution and data dependency among queries, we utilize Oozie to execute workflows generated for each query. To enhance runtime performance, we also support output caching to prevent unnecessary query execution. With these new features, a single Boa query can now have multiple phases, with intermediate output generated at each step and consumed in the next. This makes the language more like a dataflow language.

To evaluate views, we partially reproduced two prior MSR cases studies. In the first case study, we applied the greedy splitting algorithm to split the identifiers from gold set and Boa dataset, and we produced similar accuracy to the original study. For the second case study, we mined the developer turnover from open source projects in the Boa dataset, and the turnover metrics are used to study its impact on project activities and project qualities. We found that the stayers tend to contribute to the projects more than other categories do, but the stayer activities have more negative impact on bug-fixed density in the projects. We also test the caching behavior with queries from the first case study. The testing results show that with more cached queries during the run, the faster the performance.

Views in Boa allow users the ability to write better, more modular code. Users can create sub-views that perform a single task, and then utilize the output of that sub-view in later views/queries.
These views are also re-usable, allowing users to share their queries and the resulting data, so that other users can directly reuse the output. As an example, someone could write a filtering view based on MSR community best practices, and every other users can simply re-use the output from that view to filter the dataset prior to performing their specific analysis.

We have identified several future extensions to views to make it easier to use them. We also identified several possible performance optimizations to help improve the runtime performance.


