Boa Views: Easy Modularization and Sharing of MSR Analyses

Che Shian Hung
Bowling Green State University
Bowling Green, Ohio
hungc@bgsu.edu

Robert Dyer
Bowling Green State University
Bowling Green, Ohio
rdyer@bgsu.edu

ABSTRACT

Mining Software Repositories (MSR) has recently seen a focus toward ultra-large-scale datasets. Several tools exist to support these efforts, such as the Boa language and infrastructure. While Boa has seen extensive use, 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. This limits end-to-end reproducibility and limits sharing/re-use of MSR queries. To address this problem, we use the notion of views from the relational database field and designed a query language and runtime infrastructure extension for Boa that we call materialized views. Materialized views provide output reuse to Boa users, so that the results of prior Boa queries can be easily reused by others. This allows for computing results not previously possible within Boa and provides for increased sharing and reuse of MSR queries. To evaluate views, we performed two partial reproductions of prior MSR studies utilizing Boa's dataset and infrastructure 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 runtime performance.

CCS CONCEPTS

• **Software and its engineering → Distributed programming languages**; • **Computing methodologies → Distributed programming languages**.

KEYWORDS

Boa, materialized views, end-to-end analysis, modularity, re-use

ACM Reference Format:
Che Shian Hung and Robert Dyer. 2020. Boa Views: Easy Modularization and Sharing of MSR Analyses. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387480

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387480

1 INTRODUCTION

There are many tools designed to help Mining Software Repository (MSR) researchers more easily mine open source projects, such as World of Code [19], Boa [6–8], GHTorrent [11], Sourcerer [18], and PyDriller [25]. 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. Boa queries are automatically transformed into distributed Hadoop MapReduce [5] 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. Boa programs are designed to analyze a single project at a time, and send data to aggregators 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 re-use 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 reusable 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 [15] 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 [10, 14] 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 re-usability of a view.

In the next section, we introduce Boa and motivate the problem via a simple example. In Section 3, we propose our solution called

views, and work through an example problem solved using the new view syntax. We then partially reproduce two case studies to show the usefulness of the new language features in Section 4. Prior research related to views and MSR mining tools are discussed in Section 5. Finally, we conclude the paper in Section 6.

2 BACKGROUND AND MOTIVATION

In this section, we first provide background about the Boa language and infrastructure [6–8]. Then we discuss some limitations with the Boa infrastructure and motivate a solution to the problem.

2.1 Overview of Boa

Here we give an overview of Boa's language and infrastructure, by first looking at the query language via a simple example and then explaining how that query executes.

Boa's Query Language The Boa language supports five primitive types: *int*, *float*, *bool*, *string*, and *time* and five compound types: *tuple*, *array*, *map*, *set*, and *stack*. Boa also provides a set of domain-specific types such as `Project`, `CodeRepository`, and `Revision` as well as nine AST types such as `Declaration`, `Statement`, and `Expression`.

Boa queries take a single `Project` as input, process it, and generate output using `output` variables. The output variables aggregate multiple output values from all projects to generate the final result. Some example aggregators are: `sum`, `set collection`, `mean`, `maximum(N)`, and `top(N)`.

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

3 visit(p, visitor {
4   before rev: Revision ->
5     if (isfixingrevision(rev.log))
6       foreach (i: int; iskind("SOURCE_", rev.files[i].kind))
7         FixingFileCount[p.id][rev.files[i].name] << 1;
8 });
```

Figure 1: Example Boa query counting number of times source files appear in bug-fixing revisions.

As an example, consider the query shown in Figure 1 that counts the number of times source files appear in bug-fixing revisions. The input for the query is given the alias `p` and defined on line 1. The output for the query is named `FixingFileCount` and uses the `sum` aggregator on line 2. Lines 3–8 declare a visitor and visits the entire input tree. The visitor declares what action to take when reaching nodes of a given type. For example, on line 4 the visitor will run the code on lines 5–7 when first reaching any node of type `Revision`, before visiting the children of that node.

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.

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.

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 demonstrates the process of the MapReduce program based on the query in 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 `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.

2.2 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.

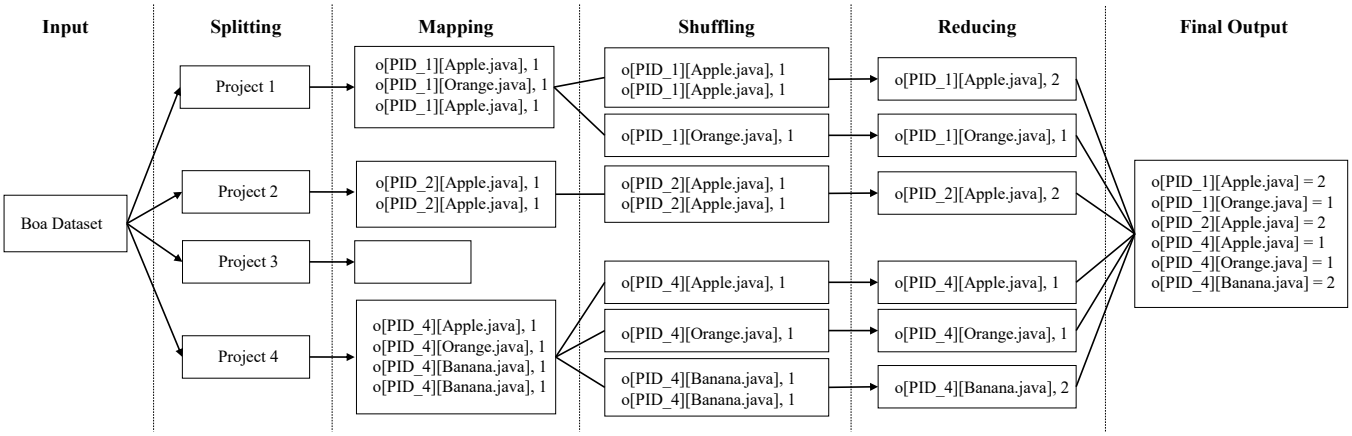
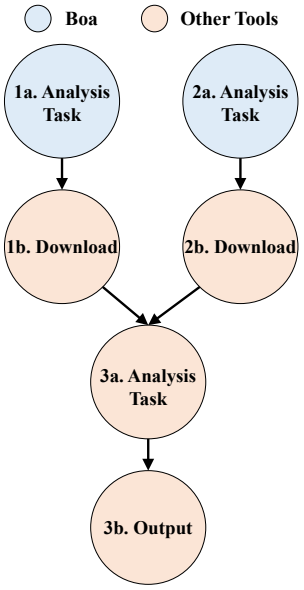


Figure 2: The MapReduce program when the example query in Figure 1 executes. The Mapping column represents the map function (the query provided by the user). The Reducing column represents input/output for the reduce function (output aggregator). The MapReduce framework handles the rest.



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 common MSR criteria
Step 2b Download Boa output through Boa API

Step 3a Generate list of potentially buggy files
Step 3b Output final result

Figure 3: A set of tasks to find source code containing potential bugs through bug-fixing revisions. The first two steps can be performed in Boa, but the remaining steps have to be performed with post-processing the output.

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** [22]. 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 3. The steps are shown on the right.

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.

3 APPROACH: VIEWS IN BOA

To tackle the problem, we introduce a new feature in Boa called *materialized views*. Our goal for the feature is to allow researchers to be able to resolve any MSR research questions solely within Boa, provide them with better modularization capabilities, and to enable more sharing of MSR queries. The concept of views comes from relational databases, which allow users to reuse previous query results. We borrow that notion here, allowing users to specify queries as views, and provide a runtime caching mechanism that effectively makes them into materialized views.

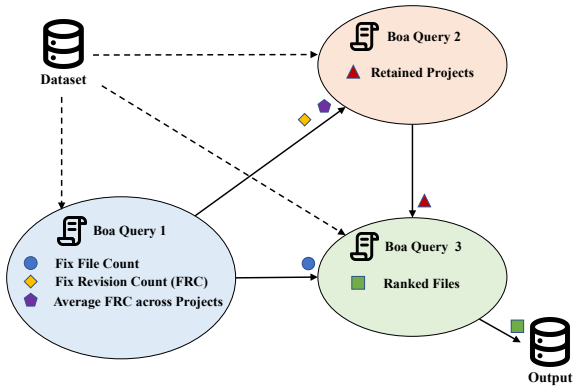


Figure 4: A DAG to implement the analysis of Figure 3.

With views, all of the tasks shown in Figure 3 can now be done in a single Boa query (aka job), or broken into multiple Boa sub-queries written by the same or different users. Boa can now become an end-to-end analysis tool for researchers in the MSR field. In this section, we illustrate the design of views.

With views, Boa queries can contain multiple sub-queries. The query in the outer-most scope is the *main query*. From this point on, all Boa queries/jobs are actually views and we use the terms interchangeably. Each view contains one or more *output variable*. When a view re-uses the output from another view, we call it a *table* and say the view is referencing the table.

Whenever a table from query `X` is referenced in query `Y`, a dependency relationship is established between the queries. If the table in query `X` is missing, query `X` must be executed again to generate the table before running query `Y`. If somehow query `X` fails to generate the table, query `Y` should abort immediately.

The dependencies should form a directed acyclic graph (DAG). A valid view-referencing relationship must not contain any cycles, otherwise during execution each query would wait on the output from another query, forming a deadlock.

Let's consider the example analysis to **find the buggy files by examining past bug fixing behavior**. To perform this analysis in Boa with views three Boa queries are needed. The corresponding DAG is shown in Figure 4. The first query contains three outputs, which are derived from step 1a in Figure 3. The second query references two tables from query 1 and filters projects, outputting the retained projects. The third query reuses the `FixFileCount` table from query 1 and the `RetainedProjects` table from query 2 to rank source files for the retained projects, producing the final output for the research question. Next we show the source code.

3.1 Views Syntax

In this section, we introduce new syntax to support views. Figure 5 shows a Boa query that implements the DAG in Figure 4. The query contains two sub-views. Each view's color maps to a node in Figure 4. The blue portion is the first sub-view computing file count and revision metadata. The red sub-view uses metadata and revision

```
1  view FixingRevision {
2    FixFileCount: output sum[string][file: string] of int;
3    FixRevisionCount: output sum[string] of count: int;
4    AverageFRC: output mean of int;
5    count := 0;

6    visit(input, visitor {
7      before n: Revision ->
8        if (isfixingrevision(n.log)) {
9          count = count + 1;
10         foreach (i: int; iskind("SOURCE_", n.files[i].kind))
11           FixFileCount[input.id][n.files[i].name] << 1;
12       }
13   });

14   if (count > 0) {
15     FixRevisionCount[input.id] << count;
16     AverageFRC << count;
17   }
18 }
19 view Filter {
20   Retained: output collection[pid: string] of int;

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

24   visit(input, visitor {
25     before n: CodeRepository -> {
26       v2 := FixingRevision/FixRevisionCount[input.id];
27       r2: v2._row;
28       if (v2 >> r2 && r2.count > r._1
29         && len(n.revisions) >= 100)
30         Retained[input.id] << 1;
31     }
32   });
33 }
34 o: output top(5)[pid: string] of fileName: string weight count: int;

35 if (len(Filter/Retained[input.id]) > 0) {
36   v := FixingRevision/FixFileCount[input.id];
37   r: v._row;
38   while (v >> r)
39     o[input.id] << r.file weight r._2;
40 }
```

Figure 5: Boa query implementing the DAG in Figure 4.

counts to filter unwanted projects. The green portion shows the main query, which ranks the file count for the retained projects.

Creating Sub-Views 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 view of the query. The second approach is creating a sub-view. Line 1 and 19 demonstrate the syntax of creating sub-views. The keyword `view` indicates the start of the sub-view, which is followed by a view name and a block containing an entire Boa query. Each view defines their own output variables. We can create nested views as well. Defining nested views could help users organize the view path easier for future referencing. We will discuss more about referencing views later. Views create a block-level scope, so variables, function, etc do not pollute the namespace of another sub-view. When compiled, each (sub)view is translated into a MapReduce program. We utilize a workflow

```
1  view Filter {
2    Retained: output collection[pid: string] of int;
3    v := J12345/AverageFRC;
     ...
15 }
16 o: output top(5)[pid: string] of fileName: string weight int;
17 if (len(Filter/Retained[input.id]) > 0) {
18   v: table sum[file: string] of int =
       @user/FixingRevision/FixFileCount[p.id];
19   r: v._row;
20   while (v >> r)
21     o[input.id] << r.file weight r._2;
22 }
```

Figure 6: Boa query revised from Figure 5, assuming the view `FixingRevision` is defined as another Boa job with job id `12345` and tag name `FixingRevision` assigned by user `user`.

scheduler (such as Oozie [15]) to manage the DAG of MapReduce programs generated. The generated workflow checks for cached outputs before running views. We don't expire cache entries and leave cache expiration policy as future work.

Referencing Tables 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 in the view. A RVP consists of a set of view names. A RVP can either start with the current scope or the scope of main query. In Figure 5, the blue boxes in view `Filter` demonstrate RVPs to reference tables from view `FixingRevision`. In this case, the RVPs start with the scope of the main query.

If a target table comes from other query (possibly another user's), we say it is an *external table*. External tables need to be referenced with an absolute view path (AVP) followed by a table name. An AVP consists of a *query root* and a RVP. Query root represents the address of a Boa job. There are two kinds of query root: explicit job id and user/tag name. Job id is a unique number given to every submitted Boa query. Tag name is a customized name users can give to a job (note: this feature is being proposed here, but not yet supported in the web interface).

Figure 6 shows a revised version of Figure 5, assuming the view `FixingRevision` is defined in another Boa job by user `user`, and the job is given a job id `12345` and a tag name `FixingRevision`. On line 3, view `Filter` references an external table via 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. When referencing external views, RVP always starts from the scope of the main view.

Figure 6 line 18 shows an example of referencing an external table via tag name. In this case, the query root is `@user/ChurnRate`. The goal of having tag names is to allow each user to be 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. This design allows duplicated tag names across users but not across jobs for a single user. This is 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.

Tables have a type similar to their corresponding output types. Users can explicitly give the type (or it can be inferred) as shown in Figure 5 on line 21. The only difference is the keyword `output` is replaced with `table`. One reason to give the type is the user can then assign meaningful names to various columns of the table. When traversing the table they can access individual cells using those names. Since names are optional, they can also access the cells positionally with syntax such as _1, _2, etc.

Table Traversals Since the size of the tables might 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. In the future we could optimize based on actual table size.

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 from a table variable with the attribute `_row`. In Figure 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. The right shift operator returns a boolean value, which can be used to indicate if a row was read. To traverse the whole table, we can use a while loop to read in each row, such as `while(v >> r)` (line 20). The values in the row can be retrieved by using field names or positionally. An example is shown on line 21 where a string value in the first column is extracted with field name `file` and the integer in the second field is extracted with field name _2.

Table Filtering Filtering allows users to filter out unwanted rows during traversal. 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, etc. In Figure 5 line 26, the referenced table `FixRevisionCount` has two columns with types `string` and `int`. An index `[input.id]` is applied to the table. This filtering guarantees that when a row is read from the table, the first column must match current project id. Since the filtered column becomes a fixed value, each filter decreases the number of columns by one. On line 36, table `FixFileCount` has three columns. After filtering with index `[input.id]`, variable `v` only has the second and the third columns from original table. If we just want drop a column, we can use a wildcard index [_].

4 EVALUATION

To evaluate views, we reproduce two prior MSR studies and evaluate the caching behavior on one of the studies. We believe the selected studies are complicated enough to demonstrate the use of views. We wrote Boa queries and utilized views to reproduce the research entirely within Boa. We set up a Boa cluster with Hadoop version 1.2.1 and Oozie version 4.0.1, containing 1 master and 15 compute nodes. Each identical node has a 4 core Intel Xeon 3GHz and 16GB memory. Map and reduce tasks are given 1 core and 1 GB/each.

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 described in this section may not look likes views - but they are. We explicitly reference them via job numbers.

4.1 Case Study 1: Identifier Splitting

For certain tasks, splitting source code identifier into words is required. Many splitting algorithms have been developed. To help the research community testing the algorithms, Binkley et al. [2] created a dataset (so called *gold set*) containing 2,663 identifiers and their split form based on 8,522 human splitting judgements. Hill et al. [14] then performed an empirical study of different identifier splitting algorithms (*Greedy*, *Samurai*, *INTT*, etc.) on that gold set. Some splitting algorithms use hard words as input instead of the original identifiers. Hard words are the conservative split of an identifier based on certain points such as underscores, digits, and the transition from lower to upper case.

4.1.1 Boa Queries. To reproduce the study, we imported the gold set into Boa and split the identifiers by using the greedy approach [9]. The approach requires a dictionary word list, an abbreviation list, and a list of stop words. Stop words are words that do not contain useful information such as `a`. Our dictionary contains 60,433 entries and the abbreviation list contains 90 entries from Kevin Atkinson's SCOWL word lists [16] (sizes 10 through 35). To acquire better results, 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 from Ranks[1] and 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 converted into Boa queries by several simple python scripts, emitting each entry into a corresponding Boa output variable.

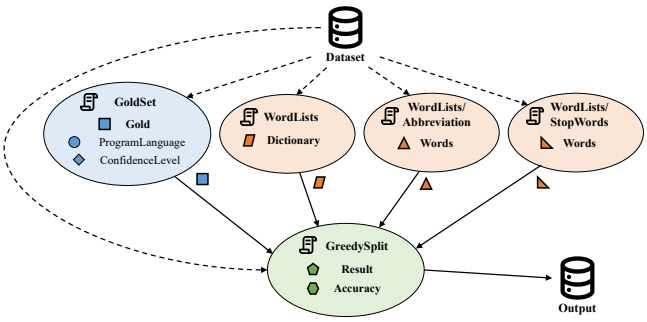


Figure 7: The DAG for splitting identifiers from gold set.

The DAG for splitting identifiers from gold set is shown in Figure 7. 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 sub-view names if any. Each query contains at least one output. In this case, the color is used to differentiate 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-views: `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 in 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 the Boa input 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 could modify the infrastructure to allow 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 identifier. Each identifier can have up to five confidence scores, and the confidence scores scale from 0 (a guess) to 2 (certain). The query **GoldSet** is (partially) shown here:

```
1 Gold: output collection [id:int][original:string][program:string][
      hard:string] of anno:string;
2 ProgramLanguage: output set [program:string] of language:string;
3 ConfidenceLevel: output collection [id:int] of confidenceLevel:int;
4 if (input.id == "1061331") {
5   Gold[1]["::CreateProcess"]["mozilla-source-1.1"]["Create-Process"
      ] << "Create-Process";
6   ProgramLanguage["mozilla-source-1.1"] << "cpp";
7   ConfidenceLevel[1] << 1;
8   ConfidenceLevel[1] << 2;
9   ConfidenceLevel[1] << 2;
    ...
12916 }
```

Since the query does not use the Boa dataset at all, we used an if statement as a check at the beginning of the map function to ensure the code ran only once. The string `1061331` is a random project id from the dataset, so only the mapper node processing that project will execute this query. Without the check, the query would execute as many times as the input dataset size (millions of projects), which could potentially create much overhead and also produce incorrect 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 the confidence level given by a annotator.

The query **WordLists** contains two sub-queries, each containing an output `Words`. The sub-query `Abbreviation` provides the abbreviation word list and `StopWords` provides the stop word list. The main view provides the dictionary word list. Instead of having three separate views, the word lists are contained in a single view for better modularization. Given the gold set and these word lists, the last view **GreedySplit** runs the greedy algorithm.

4.1.2 Case Study Result. In the original study, Hill et al. test the greedy splitting algorithm with different dictionary sizes: small (50,276 entries), medium (98,569 entries), and large (479,625 entries). The testing showed that using the large dictionary gives the best accuracy with 60%, small gives second best with 56%, and medium produces the worst accuracy with 51%.

After executing `GreedySplit`, our greedy approach produces accuracy of 53%, which is better than the worst accuracy from

[1] https://www.ranks.nl/stopwords

the original study result. We believe the accuracy is reasonable since our word list contains 61k entries, which sits between the small/medium dictionary sizes from the original study. Also, the difference may come from having different words in the dictionaries, as the greedy algorithm heavily depends on the word list.

4.2 Case Study 2: Developer Turnover

Developer turnover represents the flow of human resources in a project. Sometimes developers switch teams, working on different modules but staying in the same project. They are represented as IN (Internal Newcomers) or IL (Internal Leavers). If developers are new to/leave the project, they are EN (External Newcomers) or EL (External Leavers). Developers are denoted as ST (Stayers) if they stay in the same module/team. To find the impact of new/leaving developers on project activities, Foucault et al. [10] collected developer turnover metrics on 5 open source projects and analyzed the metrics to answer three research questions:

RQ1 Using the concepts of external newcomers and leavers at the project level, is turnover an important phenomenon in open-source software projects?

RQ2 Looking deeply into projects at the module level, are 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?

4.2.1 Boa Queries. We partially reproduced the prior work by first mining turnover metrics from Boa's dataset. Three more queries were written to answer the research questions above. The DAG for this case study is shown in Figure 8 and contains seven Boa queries in total. Before computing turnover metrics, we first filter out small projects from the dataset.

ProjectCommitTime Collects the first/last commit times (converted to integers) for each project. The output variable `LatestCommitTime` takes the project id as index and provides the last commit time. If a project does not contain any commits, the project will not appear in the output tables. Therefore, this query can also be used to check if a project contains any revisions at all.

ProjectDeveloperFilter For each project, this query uses a set to track all unique committers. Then the project id is emitted to the output variable `AtLeast` with different cutoff values. The output collects project ids with at least a certain number of committers. In this query, the number of committers is output by 10s. For instance, if the project has 23 developers total, the project id would be emitted to the output with indices 10 and 20.

TurnoverProjectFilter This query uses the previous two views to filter out small projects from the dataset. In this case study, the period length is set 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 filter out projects that have little activities and low number of developers.

TurnoverMetrics After filtering the projects, this query computes the turnover metrics. First it collects the name of the developers along with the modified module names, within each period. The module is determined by directory structure of the changed files. 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. 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 emitted into output variables. We also compute the number of bug-fixing revisions in each module per period.

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

TurnoverRQ1 The query references 8 outputs from view `TurnoverMetrics`. To compute the ratio of external newcomers and leavers, we need the number of developers in each turnover metric 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 and leavers by the total of the turnover metrics for the period. 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 persistent developers. After importing the developers' usernames into sets, we compute the stayers conversion rate for the project. The outputs `ENLratioStats` and `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.

TurnoverRQ2 Computes the ratio of each turnover activity metric to the total activities for each project. The query references 5 outputs from `TurnoverMetrics`. After reading the total activities values from the tables, the total activity is computed. If the total activity is zero, the project is filtered. We compute the ratio for each metric to the total activity. The activity ratios are emit to the output table `Ratio`, including the newcomer and leaver activity ratios.

TurnoverRQ3 To assess project quality, we compute the density of bug-fixing commits per module. The tables `Commit_M` and `BFCommit_M` are referenced to compute the bug-fixing density for each module. Spearman correlation tests are performed to compute correlation between the turnover activity metrics (ENA, INA, etc) and the bug-fixing commits' density. The tests compute a correlation coefficient for each period (a number from -1 to 1) representing the perfect negative and perfect positive correlation. A 0 correlation coefficient represents no correlation. 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% 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 or negative correlation with bug-fixing density.

4.2.2 Case Study Results. Foucault studied the impact of developer turnover on 5 open source projects written in four different programming languages. Here we study the impact of developer turnover on open source projects in the Boa dataset, which contains 7,830,023 projects. However, only 1,676 projects are kept after

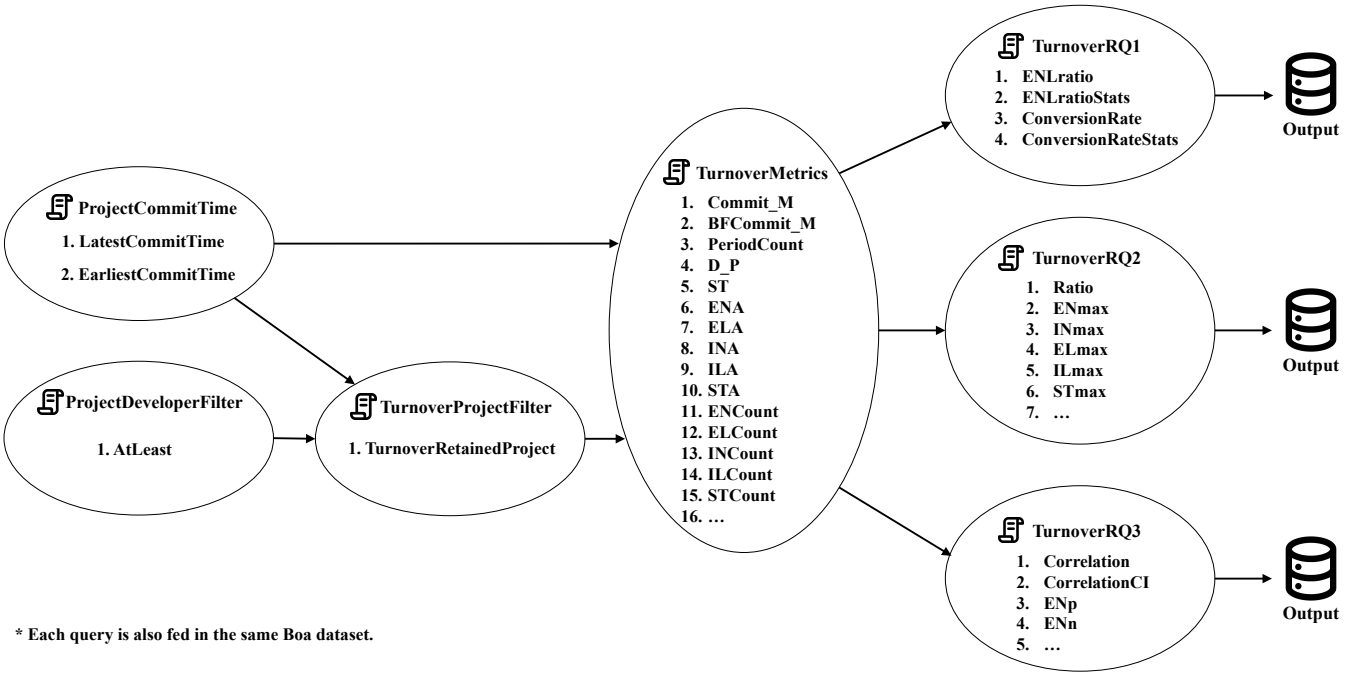


Figure 8: The DAG for reproducing Foucault et al. [10]. Three queries on the left filter projects. Query `TurnoverMetrics` computes turnover metrics. Three queries on the right analyze the turnover metrics to answer the RQs.

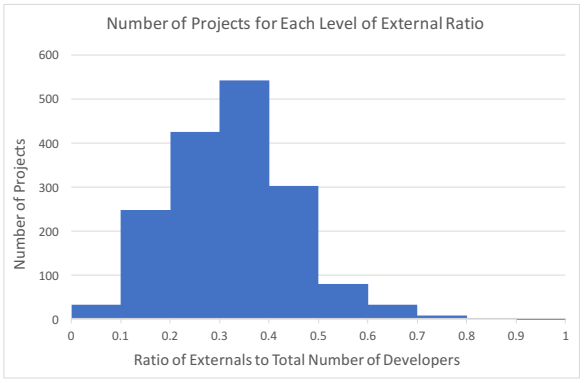


Figure 9: Number of projects for average external ratios.

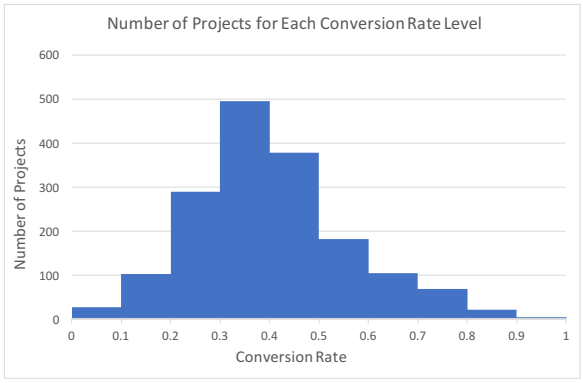


Figure 10: Number of projects at each conversion rate.

filter `TurnoverProjectFilter`. Since the scale of the dataset is larger, 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 metrics at the module level.

RQ1 By observing the number of external newcomers, leavers, and stayers, Foucault et al. found at least 80% of developers are newcomers and leavers. Also, conversion rates for projects were 8–19%, meaning a low number of newcomers became stayers. Since there are 1,676 projects in our dataset, we focus on the trend of average external ratio as well as the conversion rate.

Average external ratio is the ratio of external newcomers and leavers to all developers. The number of projects in each ratio level is shown in Figure 9. Most projects have a ratio from 0.2–0.4. There are few projects having very high ratio.

The conversion rates are shown in Figure 10. Among 1,676 projects, almost 500 projects have a conversion rate between 30–40%, and most of the project conversion rates fall between 20–50%, which is greater than the conversion rates obtained in the original studies. These rates are higher than the prior study.

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

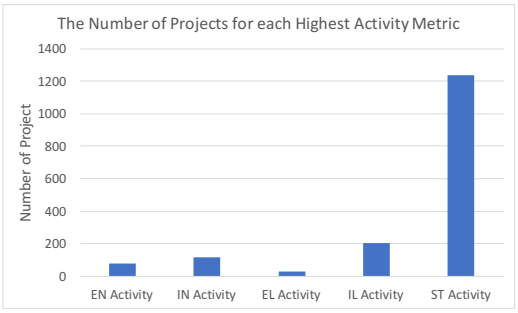


Figure 11: Number of projects for each highest activity metric. Each category contains number of projects that have the highest total activity metric throughout project life times.

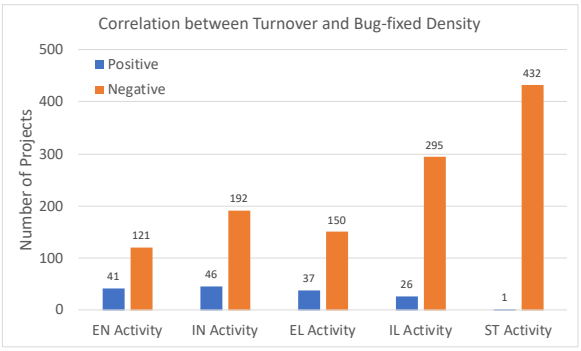


Figure 12: Number of projects having correlation with bug-fixing revisions for each activity metric.

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 11. Among 1,676 projects, the stayers contribute the most in about 75% of the projects. This makes sense as the stayers have more project knowledge compared to developers in other categories. External leavers is the smallest activity metric among the projects, but this does not mean that external leavers contribute less activities as the analysis only looks at the number one metric in the projects, meaning that external leavers can contribute the second most activities in many projects.

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

In this case study, we also compute the correlations and the confidence intervals between turnover activities and bug-fixing density. Based on the confidence intervals, we count the projects having positive and negative correlations for each turnover activity metric. The result is shown in Figure 12. Contrary to the original study, the stayers contribute much less bug-fixing revisions than other metrics. However, for each metric, no more than 50 projects have positive correlations with bug-fixing density. The stayer activities in more than 400 projects have negative correlation. This might be due to the high stayer activities observed from Figure 11. Given the small number of bug-fixing revisions, the more commit activities the developers have, the less bug-fixing density the team results.

Even though some of the results do not match the original study, notice that we applied the analysis on 1,676 open source projects, and Foucault et al. only focuses on 5 open source projects. The turnover impact for a project varies on a case-by-case basis so we believe our analysis results simply show a more general trend.

4.3 Performance Analysis on Caching Behavior

In this section, we evaluate performance and caching behavior.

4.3.1 Analysis Method. With views, caching happens when the outputs already exist 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 use the queries from the first case study to test the caching performance. To test the caching performance, we designed three testing scenarios. For each scenario, we executed query `GreedySplit` 10 times.

The first scenario estimates the runtime performance for running the queries without any caching enabled, so all queries execute on each run. The second scenario executes where the outputs for query `WordLists` and its sub-queries `WordLists/Abbreviation` and `WordLists/StopWords` are cached, and only queries `GoldSet` and `GreedySplit` re-execute. In the third scenario we also cache outputs for `GoldSet`, so only `GreedySplit` re-executes.

Table 1: Runtime performance for each caching scenario

Scenario	Median	Average	Std. Dev.
1. No Caching	147.0	146.6	2.270
2. Cache WordLists	56.5	56.4	0.966
3. Cache WordLists & GoldSet	29.0	28.9	0.876

4.3.2 Analysis Result. The runtime performance for each scenario is shown in Table 1. The first scenario is the slowest with median 147 seconds and the third scenario is the fastest with median 29 seconds. Notice that the cached job `WordLists` includes two sub-queries `Abbreviation` and `StopWords`. These results verify our assumption, that more cached queries during the run means faster performance. This is reasonable as having more cached queries means there are less queries to execute.

Table 2: Runtime performance for individual queries

Query Name	Median	Average	Std. Dev.
GoldSet	21.0	21.3	0.483
WordLists	31.0	30.5	0.707
WordLists/Abbreviation	21.0	20.9	0.568
WordLists/StopWords	21.0	20.9	0.738
GreedySplit	21.0	21.2	0.422

The runtime performances for the 5 Boa queries from case study 1 are shown in Table 2. Compare the result of the third scenario to the performance for `GreedySplit` query, the performance difference (8 seconds) is the workflow overhead of running Oozie. If we add execution times for queries `GreedySplit` and `GoldSet` and compare to the second scenario, the workflow overhead increases to 14.5s. Lastly, if we add up all the individual query performances and compare to the first scenario, the workflow overhead becomes 32s. Therefore, the more queries executed in an Oozie job, the more workflow overhead it produces. This makes sense as running more queries means there are more workflows to process. Despite this overhead, we believe the benefits of having views (output reuse, modularization, and dataset sharing) outweigh the cost.

5 RELATED WORKS

We examine related prior research along two dimensions: 1) views in databases and 2) large-scale software repository mining.

Database Views Many works have investigated view maintenance problem in databases. Srivastava and Rotem [26] presented a parameterized approach to combine the users' and systems' needs to maintain views. Gupta et al. [13] introduced the counting and DRed algorithms for maintaining recursive and non-recursive views. Griffin and Libkin [12] proposed an approach based on equational reasoning to incrementally maintain the updates from base relations to materialized views. Ross et al. [23] presented a view maintenance plan that exploits additional views to reduce the time cost of maintaining views. Mistry et al. [20] 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. [17] proposed optimal delta evaluation to perform efficient incremental view maintenance and reduce the accessing overhead to views and base relation. In Boa all queries are deterministic, so we don't need to worry about view updates. Once views generate output, it is stored on the cluster and immutable.

Other studies show how views can be used to optimize queries. Chaudhuri et al. [3] 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. [4] presented a sound but not complete algorithm to optimize the query performance. 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 could perform similar optimizations to modify Boa queries with existing views to enhance performance.

Repository Mining Frameworks GHTorrent [11] mirrors the event streams from GitHub. GHTorrent uses multiple hosts to collect repository data though GitHub's API. Users can download MySQL or MongoDB dumps for analysis. GHTorrent and Boa provide different data. Analyses using GHTorrent can be in SQL or access MongoDB while in Boa they use Boa's DSL. Boa automatically scales analyses and with views can now provide multi-phase analysis and sharing of intermediate queries and results.

World of Code [19] is an infrastructure for easily updating a large collection of open source software repositories that supports research/tools relying on version control systems (such as Git). While similar in that both provide large amounts of open source data and tools for researchers to analyze it, Boa views allows easily sharing and accessing other user's results and analyses, hopefully leading to more shared analyses and a faster pace of research.

Software Heritage Project [1, 21] is available as downloads in several formats or on Amazon Athena for scalable processing. The project aims to archive open source software and development artifacts with over 80m projects. Unlike Boa views however, users can not easily share their analyses with each other and would have to coordinate a shared Athena instance.

Sourcerer [18] provides a SQL database for over 18k projects, containing metadata and source code elements. Since it is SQL based, many end-to-end analyses can be written using it by utilizing SQL table joins. Sourcerer could provide a notion of views via the SQL storage engine. However unlike Boa views, views created by one user would not be readily available to all other users.

MetricMiner [24] 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. 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 view support, Boa cannot perform statistical analyses on the aggregated data.

6 CONCLUSION

In this paper, we introduced the notion of materialized views in Boa. Views enable query output reuse and enable modularization via subqueries. 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. To enhance runtime performance, Views cache to prevent unnecessary query execution. Views in Boa allow users to write MSR analyses that run at a large scale in a modular fashion. Views are also re-usable, allowing users to share their queries and the resulting output, so that other users can directly reuse them.

We evaluated views via partial reproductions of two prior MSR studies: evaluating a greedy identifier splitting algorithm on a hand-curated gold set and mining developer turnover metrics. Both showed results similar to the prior studies. We also tested the caching behavior and verified that performance improves with more cached queries. In the future we plan several language extensions to ease use, as well as investigating modifications to the web-based frontend of Boa to support views and view sharing.

ACKNOWLEDGMENTS

This work supported by the US National Science Foundation (NSF) under grants CNS-18-23294, CNS-15-12947, and CCF-15-18776.

REFERENCES

[1] Jean-Francois Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. 2018. Building the Universal Archive of Source Code. *Commun. ACM* 61, 10 (2018), 29–31.

[2] David Binkley, Dawn Lawrie, Lori Pollock, Emily Hill, and K. Vijay-Shanker. 2013. A Dataset for Evaluating Identifier Splitters *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 401–404.

[3] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE '95)*. IEEE Computer Society, Washington, DC, USA, 190–200. http://dl.acm.org/citation.cfm?id=645480.655434

[4] Sara Cohen, Werner Nutt, and Alexander Serebrenik. 1999. Rewriting Aggregate Queries Using Views *(PODS '99)*. ACM, New York, NY, USA, 155–166. https://doi.org/10.1145/303976.303992

[5] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[6] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 422–431. http://dl.acm.org/citation.cfm?id=2486788.2486844

[7] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2015. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 7 (Dec. 2015), 34 pages. https://doi.org/10.1145/2803171

[8] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2020. Boa: Mining Ultra-Large-Scale Software Repositories. http://boa.cs.iastate.edu/. Accessed: 2019-07-12.

[9] Henry Feild, David Binkley, and Dawn Lawrie. 2006. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *IASTED International Conference on Software Engineering and Applications (SEA'06)*. ACTA Press, Calgary, AB, Canada, 6.

[10] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C. Murphy, and Jean-Rémy Falleri. 2015. Impact of Developer Turnover on Quality in Open-source Software *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 829–841. https://doi.org/10.1145/2786805.2786870

[11] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's Data from a Firehose *(MSR '12)*. IEEE Press, Piscataway, NJ, USA, 12–21.

[12] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. *SIGMOD Rec.* 24, 2 (May 1995), 328–339. https://doi.org/10.1145/568271.223849

[13] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. *SIGMOD Rec.* 22, 2 (June 1993), 157–166. https://doi.org/10.1145/170036.170066

[14] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. 2014. An Empirical Study of Identifier Splitting Techniques. *Empirical Softw. Engg.* 19, 6 (Dec. 2014), 1754–1780. https://doi.org/10.1007/s10664-013-9261-0

[15] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. 2012. Oozie: Towards a Scalable Workflow Management System for Hadoop *(SWEET '12)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2443416.2443420

[16] Atkinson K. 2004. Spell checking oriented word lists (scowl). http://wordlist.sourceforge.net/

[17] Ki Yong Lee, Jin Hyun Son, and Myoung Ho Kim. 2001. Efficient Incremental View Maintenance in Data Warehouses *(CIKM '01)*. ACM, New York, NY, USA, 349–356. https://doi.org/10.1145/502585.502644

[18] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (01 Apr 2009), 300–336. https://doi.org/10.1007/s10618-008-0118-x

[19] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus. 2019. World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE Press, Piscataway, NJ, USA, 143–154. https://doi.org/10.1109/MSR.2019.00031

[20] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithivasan Ramamritham. 2001. Materialized View Selection and Maintenance Using Multi-Query Optimization. *Sigmod Record* 30 (Jan. 2001), 307–318. https://doi.org/10.1145/375663.375703

[21] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2019. The Software Heritage Graph Dataset: Public Software Development under One Roof *(MSR '19)*. IEEE Press, Piscataway, NJ, USA, 138–142. https://doi.org/10.1109/MSR.2019.00030

[22] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for Inspections: Hit or Miss? *(ESEC/FSE '11)*. ACM, New York, NY, USA, 322–331. https://doi.org/10.1145/2025113.2025157

[23] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. *SIGMOD Rec.* 25, 2 (June 1996), 447–458. https://doi.org/10.1145/235968.233361

[24] F. Z. Sokol, M. F. Aniche, and M. A. Gerosa. 2013. MetricMiner: Supporting researchers in mining software repositories. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Press, Piscataway, NJ, USA, 142–146. https://doi.org/10.1109/SCAM.2013.6648195

[25] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[26] Jaideep Srivastava and Doron Rotem. 1988. Analytical Modeling of Materialized View Maintenance *(PODS '88)*. ACM, New York, NY, USA, 126–134. https://doi.org/10.1145/308386.308425