Distributed and automated mining of GitHub repositories

Martin Steinhauer
University of Salerno
Salerno, Italy
martin.steinhauer@studenti.unisa.it

***Abstract*—The mining of software repositories (MSR) became a very important source for multiple research in the last decades. Especially GitHub, a very popular online platform for hosting private and public Git repositories, is one the most common sources for mining all different types of data. However, there is still no standardized pipeline which allows to extract those data automated and fast. Several tools exist to fulfill a specific task, but none of them implement neither pipelining platform nor a complete parallelization approach.**
This project aims to generate a pipeline for first mining all refactorings and then calculate the metric change introduces by this operation in an automated and distributed way by using a map-reduce pattern built on the Apache Spark framework and describes the major pitfalls in distributed repository mining.

***Index Terms*—MSR, distributed mining, GitHub, Apache Spark**

I. Introduction

This project originally was intended to analyze and predict the effects of refactoring operations done in open source software projects that are publicly available on GitHub. Previous work showed the impacts on software quality measured by classic metrics in the field of static code analysis [1] by using a statistical model. Therefore, Lin2019a built a metric tuple that measured metrics like lines of code, McCabe's cyclomatic complexity [2], or class level metrics [3]. To improve the prediction accuracy, our goal is to create a dataset fetched from open source software projects GitHub which contains detected refactoring operations extracted from source code and associated metrics before and after this refactoring operation. To avoid bias, we want to create a very large dataset that covers the most refactoring operations. We assume that repositories with the most commits will have the most refactoring operations. During the first project phase, we soon discovered several problems with mining software code at such a large scale in Git.
Because of these issues, we decided to focus the project entirely on the mining and generation of the dataset in a distributed and generalized way by using Apache Spark. We will also evaluate the performance of our distribution approach by comparing the runtime between the single- and multithreaded approach.

II. Related Work

The reproducibility of git-based research is not easy. In particular, they pointed out the lack of frameworks those studies are built which makes it very hard to rerun those analyses or adapt them to your research questions [4]. Since then, some new frameworks evolved which enable an easy traversing of git repositories.
Common used frameworks to analyze the commit history of software repositories are RepoDriller [5] and PyDriller [6], [7]. Whereas both tools simplify the traversing of Git commits, PyDriller is able to calculate some basic metrics like complexity and method count. The metric calculation is done within the library and could not easily be replaced with custom metric logic. Additionally, PyDriller states to be multithreaded and therefore improve the performance and lower the calculation time. A closer look into the library shows that this statement is only partially true: PyDriller is based on the library GitPython which is restricted by visiting only one commit at a time. This is caused by the underlying file-based Git archive that enables thread-safety by locking the files during e.g. check out operations. Hence it is only possible to parallelize the processing in the file dimension but not in the time dimension (the commit history). In other words, only the iteration over files within one commit and then checkout the next commit which can be parallelized. The same theory applied to RepoDriller which relies on the Java-based alternative JGit [8]. And besides, both tools allow no multi-repository mining and require additional orchestration and process management.
Other projects trying to avoid direct file system access and transferring repositories in an additional database before the first analysis. One well-known project is the GHTorrent project [9], [10] which allows various file-independent queries after importing the repository into a MySQL or MongoDB database. Unfortunately, the file content is not considered within this approach and only tooling for generating and importing is covered by them.
The project 'Public git archive' was introduced by researchers at sourced and allows us to analyze repositories from GitHub at a very large scale. They provide a lot of tools written in Go to query and also introduce a new storage format called Siva to reduce the stored size of forked repositories. Through a publicly available index file no additional processing is needed for developers. After the company was sold, the public available index file was taken down due to high costs and apparently was lost completely [11].

III. Challenges in large scale Git mining

The last section showed in detail existing problems with existing tools in contrast to research. Subsequently, we will identify four issues in large scale Git mining.

A. Reproducibility challenge

As seen in related work, it is very important to build a tool that allows other researches to easily reproduce the dataset generated by the mining tool. Therefore, hosted datasets are a very problematic approach since it requires additional (financial) resources of the project holder and also is dependent on the life cycle of the initial project. By the decision of deprecating the project, all data could be gone like in the case of sourced. Mining data directly from GitHub may not resolve the problem of deleted or switched to private repositories and also introduce some overhead in terms of cloning and preprocessing but decreases the dependence to a hosting company or the creator of the project. Consequently, we will focus on the raw Git archive as a source of data.

B. Multi-dimensionality of Git data

At first glance, a GitHub repository looks like a tree-like timeline which contains different revisions and file versions along that axes. But from an algorithmic perspective, that can rapidly become an issue in terms of performance as every axis results in an iteration loop to reach at least each file once. This could also be described as multi-dimensionality of the Git data, as each additional dimension added to the mining process, will result more computation time and without considering the actual computation cost for detecting refactorings or calculating software quality metrics. Depending on the mining problem, we identified the following dimensions:

- **Time dimension:** Time dimension describes the total size of the analyzed commits. Git normally consists of one or more branches containing one main branch, denoted as *master*. When traversing over the master branch, one will get all branches that have been merged into the master. In our case, we assume that for refactoring detection it would be sufficient to detect all valuable refactorings that have been considered as useful by merging them back into the master branch. Therefore, the time dimension of the refactoring detection could be seen as linear. When an analysis problem requires visiting every commit in the repository, the time dimension is considered as the total count of commits within the whole repository.
- **File dimension:** Defined by the file count one commit has in total. This amount of files can vary significantly and often depend on the project size, age of the project or the used programming language. Additionally, it is important to note that the file dimension is different for each commit as a file can be added or deleted within a commit. If the latest commits has only a few files and all others have many, the file dimension will affect the performance more as if the last commit have many files and the others have only a few.
- **Structural node dimension:** Similar to the file dimension, the total number of interesting nodes, for example, classes, methods or code lines, are counted in the content of each file. Considering object-oriented languages, a file normally contains one class, but also could consist of more than one class declaration. If the mining problem is dependent on method or class member declarations, those would also be considered as a number of interesting nodes per file.
- **Metric dimension:** The last dimension is very specific particularly regarding the mining problem. In our example, be would like to extract refactoring operations and calculate software quality metrics. Therefore, the metric dimension is highly coupled with the structural node dimension and defined by the method the metric is collected. RefactoringMiner [12] uses an abstract UML algorithm to detect refactorings, while CK [13] is using an abstract syntax tree (AST). Those algorithms also add on top of the file or structural dimension, depending on how they are implemented and designed.

C. Parallelization challenge

One of the biggest challenges is building a mining tool, that allows scaling of the mining process along the presented dimensionality axes. Both presented tools, PyDriller and RepoDriller, do not or only partially support the parallelization. While RepoDriller does not do any kind of parallelization, PyDriller allows scaling along the file dimension axis. Files within a single commit can be analyzed by multiple threads and increase the performance and decrease the computation time as the use of those libraries shows [14]. As the multi-dimensionality of GitHub repositories outlines, we can not presume that repositories contain many files. When a repository contains fewer files but has a large revision history, the runtime of the mining process will get worse.
As already explained, the structural node dimension and the metric dimension are highly dependent on the type of analysis that should be executed. To make this approach generalizable, we focus on parallelization along the time and file dimension axes. As seen in PyDriller, the file dimension can easily be parallelized. In opposite, the commit axis is rather difficult: Since Git writes lock files when accessing the repository to avoid file inconsistencies by modifying the repository, only one commit can be visited concurrently. This limitation reflects in JGit and GitPython, which has all file-based actions marked as non-threadsafe. This limitation results in libraries not implementing a time dimension parallelization approach.

D. Generalization challenge

Even though we are conducting this distributed approach in the context of detecting refactoring operations, it is designed to be generalizable and can be reused for other types of mining problems. Since mining problems can vary from detecting developer behavior [], social aspects [] up to high technical

questions like code evolution [] and smell detection [], it is not easy to build a pipeline that matches all type of problems. Therefore, this pipeline concentrates on technical, content-based repository mining.
Additionally, libraries are not standardized in their input and output design. While RefactoringMiner takes the repository and commits as an input, CK is designed not designed Git-specific and takes just the source directory as an input. This requires the pipeline to adopt the corresponding libraries in terms of in- and outputs of each pipeline step and even more important to keep track of the memory usage, since the libraries itself write outputs to disk, while within a pipeline, results sometimes are stored in memory. This causes a very careful design to avoid out of memory exceptions.

IV. Distributed mining approach

So far we have seen four major challenges when mining repositories automated and in a large scale. This section will present a distributed approach for detecting refactoring commits and calculate software quality metrics before and after each refactoring. Therefore we are using a MapReduce-inspired pattern for data distribution and Apache Spark for pipeline implementation. MapReduce was originally invented by Google and targets big data analysis and calculations by splitting the computation process into a map and a reduce phase [15] on a cluster of computation nodes. Those concepts are reimplemented and available as open source software in Apache Hadoop [16]. Spark improves this pattern by introducing resilient distributed datasets (RDDs) which kept mainly in memory, which results in a faster processing speed than the traditional, hard-drive storage based Hadoop system. Also, Spark allows a better fault tolerance due to a directed acyclic graph (DAG) for computation tasks. In case one task in the graph fails, Spark can easily restart that specific task without canceling the whole computation.

A. Architectural overview

The pipeline is basically split into three major computation steps: The first will extract the refactoring operations from a list of given repositories as an input. Spark then clones one repository instance in the first step and scan for commits along the master branch. The list of interesting commits is split into batches of a definable size of *k*-commit and sent by shuffling to each of the other worker nodes. Depending on the number of worker nodes - computer instances that are used by Spark to distribute the computation - each node will clone the repository from GitHub, if it does not already exist on the node.
In the second step, each worker node runs an instance of RefactoringMiner on the partial list of commits. The previously described number of *k*-commits depends on the size of memory each worker node has available, as the results are kept in the heap until one batch hash finished the computation. Therefore it is important to adjust *k* accordingly: Is *k* set to high, Spark will fail caused by out of memory exception. Is *k* to small, the performance will decrease as instantiating an instance of RefactoringMiner is an expensive operation. This step outputs a list where each row consists of the repository name, the commit id of the refactoring operation and the type of the refactoring operation (e.g. move method, extract method...). The intermediate results are written as structured data using the Apache Parquet file format [17] to reduce storage size and improve performance.
The third step takes the shuffled list of refactoring commits produced in the second step and starts again with checking if the repository exists. If not, it gets cloned to that worker node. Now for every batch of refactoring commits an instance of CK is created and the parent commit of the given commit is identified. After that, both commits are checked out and CK calculates the metrics before and after the commit. Additionally, to reduce the computation time and avoid useless data processing, those files that are marked as changed are identified by the Git diff command and the metric calculation is only executed for that list of modified files. As a result, another parquet file is generated containing information about the repository, the involved commits, the refactoring operations (flattened by comma separation), the *side* of the operation (left = commit before refactoring, right = commit after refactoring), the file name, and the metrics.

B. Implementation details

Both libraries have not been very suitable to fit into our defined in and output format. Therefore they have been forked and slightly modified. RefactoringMiner only provides a method to detect a range of refactoring operations between two commit ids, though it found under certain circumstances more commits than actually existed in this range. Another method that identifies refactoring operations only in one commit was also not suitable in terms of performance and object instantiation. So we added a method that could accept a list of commits without instantiating each time a whole Git repository object.
As CK accepts only a source directory, we also forked and modified this library to take a list of interesting files within that directory. That allows the calculation of changed files only within the repository.
Besides, the calculation of metrics is designed as abstract metric processors, which improves the simplicity and extensibility of adding custom metric calculation libraries.

C. External limitations

During our development phase, we discovered some issues caused by the JGit library: Git in general has the ability to convert the linefeed before and after the checkout. This mechanism is for interoperability when using Unix-like and windows systems for development. This feature can be turned on and off globally by the setting *core.autocrlf*. Sometimes when using JGit - and mostly when *core.autocrlf* is overridden by a *.gitattributes*-file - for checking out commits the case occurs, that the files are not converted back and are shown (correctly) as modified in Git. This causes JGit to throw an exception that changes have been made and a subsequential checkout is prevented due to merge conflicts. This error is

handled by our pipeline by doing a reset and clearing the cache.
Another problem with JGit - and Git as well - is caused by Unix-like systems and as Spark is mostly executed on Linux, this issue can not be ignored. Windows file system is case-insensitive, which means it does not recognize any difference between the same filename in upper- or lowercase while Unix-like systems are case-sensitive. Checking out two commits when the filename got changed from upper- to lower-case for example will cause Git to detect a merge conflict. This issue can not be simply resolved and the only convenient solution was to delete the repository.

V. Performance evaluation

After pipeline implementation, we run some performance tests on the pipeline and test how the distribution approach affects the total runtime of a GitHub project. Please consider that those performance tests are not complete and should just be seen as an indicator of parallelizing the time dimension of a project. As discussed earlier, different shaped projects should be used, especially with varying size along the time and file dimension.

A. Evaluation environment

For this evaluation, we used some smaller Java projects. Since no comparable reference implementations exist, we focus on the RefactoringMiner step within the pipeline and evaluate with different settings for the parallel-jobs parameter which controls how many RefactoringMiner instances are created simultaneously and describe an equivalent for parallelization. The other parameter available in the pipeline is called parallel-jobs and sets how many projects should be analyzed in parallel. Both values are capped by the number of executors in the cluster. We analyzed each selected GitHub project with parallel-jobs = 1, parallel-jobs = 2, parallel-jobs = 4 and parallel-jobs = 8 and measured the time. We also considered the cloning time for repositories. As especially larger projects will take up to many hours in non parallelized settings, so we decided to omit those projects concerning energy waste and environmental protection.
The analysis itself was run on up to 8 worker nodes which were placed in virtual machines running on a local server with 64GB system memory and four deca-core CPUs (Intel Xeon E-7 4870). Each worker node gets one Spark core assigned to prevent multiple repositories on one worker node blocked by the Git repository. Therefore, when running 8 worker nodes, each Spark jobs can reserve a maximum system memory of 8GB. The server runs one hard-drive, which may have to be considered in terms of performance since in a real cluster each worker node can access the disk at full speed and does not have to share it with other tasks.

B. Results

Table I shows the results of the described evaluation process. The runtime of the *rxjava*-project is decreased by up to 43.8% and up to 45.7% in the *guava*-project. The *gson*-project experienced a small increase in runtime by 26,7%.

TABLE I
Computation time

Repository	Executors	Runtime (min)
reactivex/rxjava	1	48
reactivex/rxjava	2	40
reactivex/rxjava	4	34
reactivex/rxjava	8	27
google/gson	1	1.5
google/gson	2	1.7
google/gson	4	1.9
google/gson	8	1.9
google/guava	1	35
google/guava	2	27
google/guava	4	21
google/guava	8	19

Additionally, in Table II we listed according to our preliminary analysis the count of commits as well as the minimum and maximum file counts retrieved from the first and last commit.

TABLE II
Repository Dimsionality Metrics

Repository	File Min	File Max)	Commit Count
reactivex/rxjava	4	1862	5752
google/gson	134	207	1485
google/guava	65	3173	5282

VI. Conclusion

In this report, we showed that current implementations only can parallelize in the file dimension and lack in parallelization of the time dimension. We also demonstrated that tools like RepoDriller or PyDriller are suitable for mining a single repository, but do not provide an automated or scalable way to mine multiple projects at the same time. We implemented a pipeline that models the capability of distributed repository mining based on Apache Spark and improved the computation time.

A. Limitation

In this section, we want to discuss our approach critically. As already mentioned, this approach should be seen as a proof of concept of how git mining can be viewed from the perspective of a big data problem and distributed over multiple computers. We also have seen, that smaller projects with only a few commits can impact the runtime negatively. This is caused by the distribution overhead that Spark introduces by splitting and shuffling the initial data.
Therefore, several difficulties came along the way when integrating existing libraries into the pipeline which caused a lot of different issues. Each computation step handles the analysis of the Git repository in its way due to different data input for each library. A more generalized way of defined input and output data would make the integration much easier, but without modifying the original library that would be hard to realize. Otherwise, generalization will affect performance in a negative way. If high performance is desired than each step is

hard to generalize when using file-based repositories.
The approach of cloning each repository is also not the best solution in terms of performance and data usage. In our case, cloning even big repositories will not affect the computation time that much as this was mostly about 5 minutes. But when using a slower internet connection, this value will definitively affect the overall performance negatively. Additionally, checking out each commit revision will not result in the best performance, and due to the described issues within libraries like JGit the management of parallel file-based repositories introduces a lot of external error sources.
The last point is the implementation of each computation step. The DAG gives Spark the ability to analyze and optimize the data it computes very efficiently with known database techniques like query optimization on structural data. Since the only structured data sits between each heavy computation step and for the DAG, each mining step like mining refactorings or calculating metrics is just a black box. So no further optimization can be done.

VII. Further work

As this report shows only a proof of concept, a lot of additional and further work can be done to optimize the computation time and generalize the pipeline for other use cases. In further work, we want to propose an extended approach that replaces the file-based Git repository by a structured database each repository is imported once at the beginning of the pipeline and then can be reused and even better distributed across the Spark cluster. This could be considered like the GHTorrent project, but also storing the file content of the source file in the database which eliminates reading data from disk on each worker node and can be optimized by Spark.
There are also small projects that use graph databases for storing the AST of a file instead of just storing the plain file content. In general, graph databases look like a very convenient way for modeling not only the AST of a file but also social aspects and interactions between developers, files, and the timeline.

References

[1] B. Lin, C. Nagy, G. Bavota, and M. Lanza, "On the Impact of Refactoring Operations on Code Quality Metrics," *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pp. 594–598, 2019.

[2] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[3] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[4] G. Robles, "Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories Proceedings," *Proceedings - International Conference on Software Engineering*, pp. 171–180, 2010.

[5] "mauricioaniche/repodriller: a tool to support researchers on mining software repositories studies." [Online]. Available: https://github.com/mauricioaniche/repodriller

[6] "ishepard/pydriller: Python Framework to analyse Git repositories." [Online]. Available: https://github.com/ishepard/pydriller

[7] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911, 2018.

[8] "JGit — The Eclipse Foundation." [Online]. Available: https://www.eclipse.org/jgit/

[9] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose," *IEEE International Working Conference on Mining Software Repositories*, pp. 12–21, 2012.

[10] G. Gousios, "The GHTorent dataset and tool suite The GHTorent Dataset and Tool Suite," no. June, 2015.

[11] "Getting connection refused · Issue #171 · src-d/datasets." [Online]. Available: https://github.com/src-d/datasets/issues/171

[12] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[13] M. Aniche, *Java code metrics calculator (CK)*, 2015, available in https://github.com/mauricioaniche/ck/.

[14] C. Gote, I. Scholtes, and F. Schweitzer, "Git2net - Mining time-stamped co-editing networks from large git repositories," *IEEE International Working Conference on Mining Software Repositories*, vol. 2019-May, no. i, pp. 433–444, 2019.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, may 2010, pp. 1–10.

[17] "Apache Parquet." [Online]. Available: https://parquet.apache.org/