tABLE OF cONTENTS

[Chapter 1 – Evaluation 2](#_Toc397616264)

[1.1 Introduction 2](#_Toc397616265)

[1.2 Analyzing a real project with DyeVC 2](#_Toc397616266)

[1.3 Performance evaluation 6](#_Toc397616267)

[1.4 Threats to Validity 9](#_Toc397616268)

[1.5 Final Considerations 9](#_Toc397616269)

# – Evaluation

## Introduction

The purpose of this chapter is related to the research questions posed in <Include link to Introduction>:

* Q1: Which clones were created from a repository?
* Q2: What are the dependencies between different clones?
* Q3: Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into my work?
* Q4: Is it computationally feasible to gather this information from all known repositories, keeping them available to be used when needed?

To assess the feasibility of our approach in answering these questions, we performed two experiments. First, we conducted a *post-hoc* analysis in a real project, to check if DyeVC can help answering questions Q1-Q3. Next, we took some real projects of different sizes and from different sources, to evaluate the feasibility of processing huge amounts of information using our approach, in a reasonable time, in order to answer question Q4.

This chapter is organized as follows: Section 1.2 describes the *post-hoc* analysis experiment. Section 1.3 describes the performance evaluation of our approach. Section 1.4 discusses some threats to validity of the experiment. Lastly, Section 1.5 presents the final considerations of this chapter.

## Experiment Planning

The o

## Analyzing a real project with DyeVC

To demonstrate that our approach can help answering questions Q1-Q3, we conducted a *post-hoc* analysis using a real open source project. We used the JQuery project[[1]](#footnote-1), a project that began in 2006 and had 6,222 commits by the time of the evaluation. We used the repository history and reconstructed it, simulating the actions that occurred in the past. We do not replicate the repository history here, due to its size, but the repository is public available at GitHub. We used comments generated automatically by Git to help us depicting specific flows that happened in the past. For example, the comment *“Merge branch 'master' of https://github.com/scottjehl/jquery into scottjehl-master”* tells us that there was a user named *scottjehl* and that the merge operation was done at a branch called *scottjehl-master* on that user’s repository.

Due to the operating mode of Git, some details are missing, but these do not compromise our analysis. The first one is the moment when a clone arises or deceases. This information does not exist anywhere in the repository. We inferred the creation of clones looking at the commit messages in the repository history (a commit by developer X led to the creation of a clone named X). Clones created at a given time stayed alive for the rest of the analysis.

The second missing detail is that although we had the commit dates and times in the repository history, these dates and times were not guaranteed to be correct. This is because in DVCS’s we do not have a central clock. Each commit is registered with the local time at the machine where the clone is located, which could lead us to have a commit in the history with a predecessor in the future, depending on when and where each one of them were performed. This missing detail is not so important, because the precedence between two commits is not depicted from their commit times, but from the pointers that Git maintains from a commit to its ancestors. We can use these dates, but not as an authoritative information.



Figure 1 - Topology view showing first monitored repository



Figure 2 - aakoch’s commit history showing commits pending to be pushed

We chose a moment in time when three developers were involved, performing commits and merging changes in the repository. We created three clones for these developers, named after their author names and commit messages: *jeresig*, *adam*, and *aakosh*. Figure 1 shows the topology view when *aakosh* had 121 commits pending to be pushed to the central repository (hereafter represented as *central-repo*). Figure 2 shows part of aakosh’s commit history and how DyeVC represents commits pending to be pushed as green nodes in the graph.



Figure 3 - Topology view showing the three monitored repositories

Later on, *aakoch* pushed his commits to *central-repo* and both *adam* and *jeresig* commited some changes. Figure 3 shows the topology view after registering *adam* and *jeresig* to be monitored by DyeVC. Here, we can see that *aakoch* was synchronized with *central-repo*, whereas *adam* and *jeresig* had some pending actions. At this point, we are able to revisit questions Q1 and Q2:

**Q1:** *Which clones were created from a repository?* DyeVC’s topology view (Figure 3) shows all the clones where there is an instance running, and discovers other clones connected to them, even if there is no instance running.



Figure 4 - Adam’s tracked branches

**Q2:** *What are the dependencies between different clones?* DyeVC’s topology view (Figure 3) shows the dependencies between the peers in the topology, as well as the number of commits ahead or behind in each of these dependencies.

*Adam* had 121 commits to pull from *central-repo*, what is corroborated by the details of his tracked branches (Figure 4). He also had a non-tracked commit pending to be pushed. Non-tracked commits are not shown in the tracked branches view, but we can see them in commit history views, painted in gray. Figure 5shows the collapsed commit history for *jeresig*, where we can see *adam’s* non tracked commit with hash a2bd8 (we know this is an *adam’s* commit by comparing the id of the repository in the message details with *adam’s* repository id in Figure 3).



Figure 5 – Jeresig’s collapsed commit history

The repository history leads us to think that *jeresig* is a core developer in the project, because he performed most of the merges to master branch. Looking at Figure 3, we see that he had 26 commits pending to be pushed to *central-repo*. These 26 commits can be seen at *aakoch’s* commit history (Figure 6), as red commits, once they could not be pulled by *aakoch* until *jeresig* pushed them to *central-repo*.There was also a commit in *central-repo* pending to be pulled by *jeresig*. If we look back at Figure 5 we see that the only yellow commit is a0887, made by *aakoch*. This tells us that *jeresig* pulled changes from *central-repo* at a moment before *aakoch* pushed commit a0887. If we look at Figure 7, we see that all the pending commits (those that were pending to be pushed and those that were pending to be pulled) are related to the same branch *master*. This tells us that, if *jeresig* wanted to push these commits to *central-repo*, he would have to perform both push and pull operations. This analysis helps us revisit and answer Q3.



Figure 6 - Aakoch’s commit history



Figure 7 - Jeresig’s tracked branches

**Q3:** *Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into my work?* New commits in tracked branches of peers can easily be found looking at Level 3 information (tracked branches, shown in Figure 4 and Figure 7). This view will show to which branch these commits are related and how many new commits exist. If we want to look at each commit individually, we can look at Level 4 information (commit history, shown in Figure 2 and Figure 6) and notice the yellow nodes. Additionally, Level 4 information is used to find new commits in repositories that are not peers (red nodes), or new commits in non-tracked branches (gray nodes).

## Performance evaluation

In order to answer question Q4, we evaluated the time spent to perform the most common DyeVC operations, by analyzing repositories of different sizes and hosted in different Git servers. The results are shown in Table 1.

Table 1 shows the monitored projects (name and hosting service), the repository metrics – in terms of number of commits, disk usage, number of files and number of peers in the measured topology –, and the time spent by DyeVC to run some background and foreground operations. All measurements were done in the same period of the day and from the same machine, a Core Duo CPU running at 2.53 GHz, with 4GB RAM running Windows 8.1 Professional 64 bits, connected to the internet at 35 Mbit/s.

We measured the main operations of our approach: “Insert 1st”, invoked when the user includes the first repository of a given system to be monitored. “Insert 2nd”, invoked when the user includes a repository to be monitored in a system that already have registered repositories. “Commit History”, invoked when the user requests to see the commit history of a given repository. “Topology”, invoked when the user wants to see the topology of repositories of a given system. “Check Branches”, invoked periodically to check all the monitored repositories, searching for ahead or behind commits. “Update Topology”, invoked periodically to update the topology information in the central database. This last operation updates the existing repositories, their peers, and the existing commits, marking in which repositories each commit is found.

Table 2 shows the correlation between each repository metric and the measured operations, according to the *Pearson coefficient* (PEARSON, 1895). This correlation coefficient measures the linear correlation between two variables X and Y and ranges from −1 to 1. Values of 1 or -1 mean that a linear equation can describe the correlation between X and Y perfectly (either positive or negative, respectively). A value of 0 means that there is no linear correlation between X and Y.

Table 1 - Time (in seconds) spent by DyeVC in several operations for repositories with different histories and sizes

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Repository | Hosting | Repository metrics | | | | Foreground operations | | | Background operations | | | |
| # commits | MB | # files | # peers | Commit History | Memory Usage[[2]](#footnote-2) | Topology | Insert 1st | Insert 2nd | Check Branches | Update Topology |
| DyeVC | github.com | 187 | 1.0 | 539 | 6 | 3.5 | 15 | 2.7 | 12.4 | 16.1 | 1.7 | 4.4 |
| Sapos | github.com | 702 | 7.0 | 685 | 11 | 5.6 | 19 | 3.2 | 20.8 | 22.6 | 1.8 | 5.2 |
| jgit | eclipse.org | 2,979 | 10.0 | 1,595 | 3 | 18.4 | 512 | 3.4 | 42.4 | 46.0 | 5.9 | 6.8 |
| egit | eclipse.org | 3,775 | 27.0 | 1,478 | 3 | 21.3 | 559 | 3.7 | 49.6 | 46.6 | 4.2 | 7.3 |
| jquery | github.com | 5,518 | 20.0 | 253 | 3 | 65.0 | 1,121 | 4.1 | 40.0 | 37.4 | 1.4 | 9.4 |
| Tortoise Git | code.google.com | 6,166 | 85.0 | 3,220 | 3 | 68.0 | 492 | 4.2 | 39.0 | 36.0 | 1.6 | 9.6 |
| Gitextensions | github.com | 6,417 | 448.0 | 1,549 | 3 | 73.0 | 1,529 | 17.0 | 155.8 | 129.0 | 1.6 | 10.6 |
| drupal | drupal.org | 23,922 | 84.4 | 9,290 | 3 |  |  | 18.0 | 102.0 | 95.0 | 2.0 | 18.0 |
| Expresso Livre | gitorious.org | 25,822 | 141.0 | 20,729 | 3 |  |  | 18.2 | 110.0 | 102.0 | 2.1 | 19.3 |
| Git | github.com | 35,260 | 98.0 | 2,656 | 3 |  |  | 19.4 | 196.0 | 158.6 | 3.4 | 40.0 |

Table 2 - Pearson coefficient between measured operations and repository metrics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | # commits | Size | # files | # peers |
| Insert 1st | 0.79 | 0.65 | 0.30 | -0.44 |
| Insert 2nd | 0.82 | 0.65 | 0.36 | -0.46 |
| Check Branches | 0.00 | -0.28 | -0.13 | -0.25 |
| Update Topology | 0.94 | 0.17 | 0.33 | -0.35 |
| Commit History | 0.95 | 0.62 | 0.41 | -0.61 |
| Topology | 0.86 | 0.61 | 0.59 | -0.39 |

Looking at Table 2, we can see that the number of commits in the repositories is the metric with the Pearson coefficient nearest to 1. Generally, operations took longer in repositories that had more commits. Size also presents a high Pearson coefficient, but this is caused by the set of projects we chose, where the majority of repositories with a greater number of commits also had greater sizes. According to Table 1, the slowest operations were “Insert 1st” and “Insert 2nd”, due to the amount of data sent over the Internet to update the database. The only operation with no significant variation in response times was “Check Branches”. Amongst the foreground operations, the “Topology” operation had a significant increase in its response time, but with lower values than the “Commit History” operation. This is because the latter deals with much finer grain data than the former. In fact, the application was not able to show the commit history for repositories with more than 6.4K commits, due to out of memory errors. The maximum Java heap size during the experiment was configured to 1.5GB and this was the memory usage for the project *Gitextensions*. This is a scalability limitation of our approach. The increasing memory usage is due to two factors: First, the commit graph has to be entirely in memory to be plotted. Second, the X position of nodes in the graph are calculated based on node ancestry, but the Y position is calculated in order to minimize the number of lines crossing during merges and splits in the graph. In order to do so, we used the *Dijkstra’s algorithm* (DIJKSTRA, 1959), for which memory usage also scales with the number of nodes.

## Threats to Validity

## Final Considerations

Bibliography

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, v. 1, n. 1, p. 269–271, 1 Dec. 1959. Accessed: 29 jun. 2014.

PEARSON, K. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London*, v. 58, n. 347-352, p. 240–242, 1 Jan. 1895. Accessed: 22 jun. 2014.

1. https://github.com/jquery/jquery.git [↑](#footnote-ref-1)
2. Memory usage was measured in MB during the execution of “Commit History” operation. [↑](#footnote-ref-2)