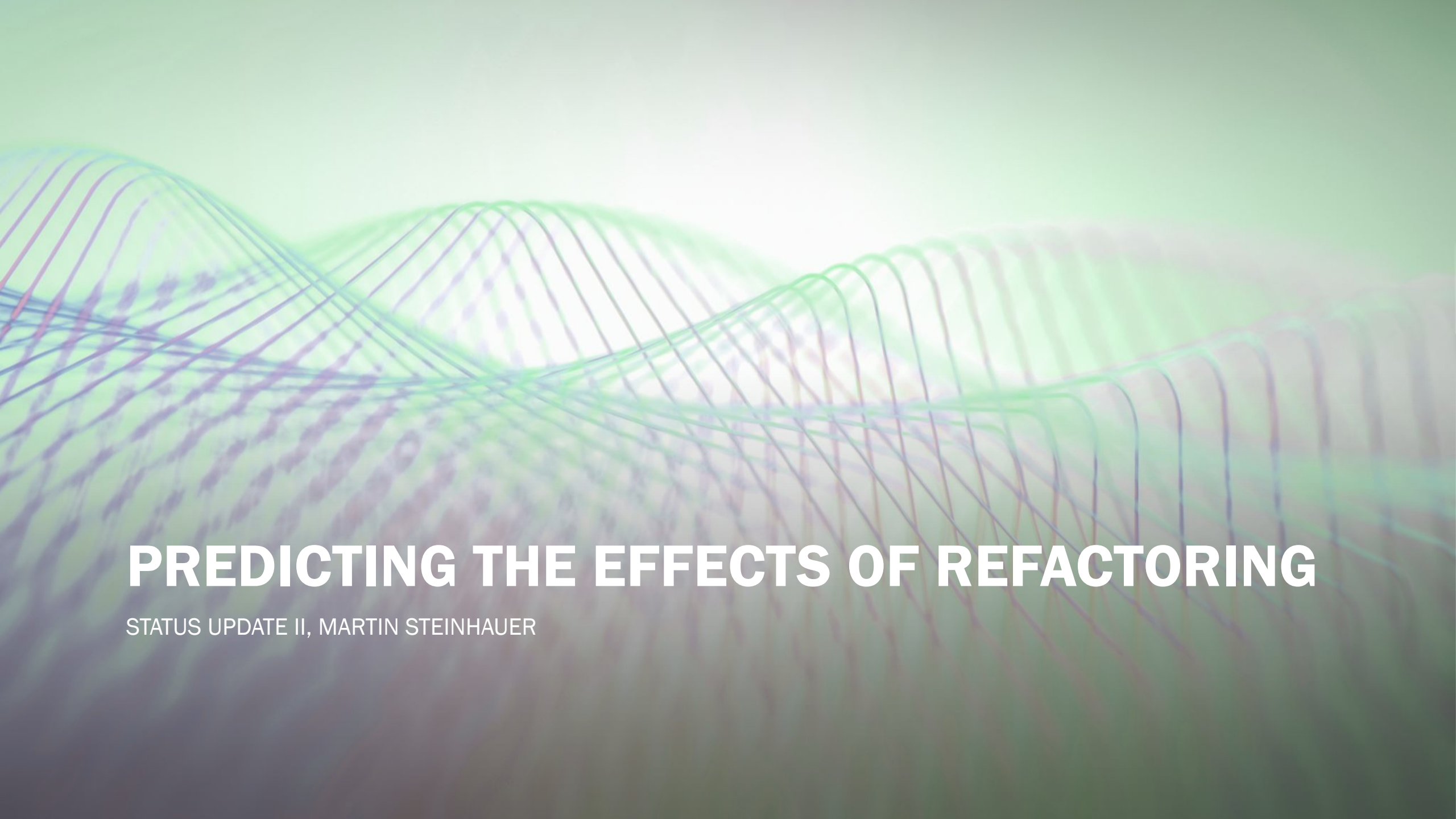


PREDICTING THE EFFECTS OF REFACTORING

STATUS UPDATE II, MARTIN STEINHAUER

OVERVIEW AND GOALS

- Predicting refactoring effects using machine learning
  - Through product metrics before and after a refactoring commit based on history git data
  - Approach based on the paper „On the Impact of Refactoring Operations on Code Quality Metrics“ [7]

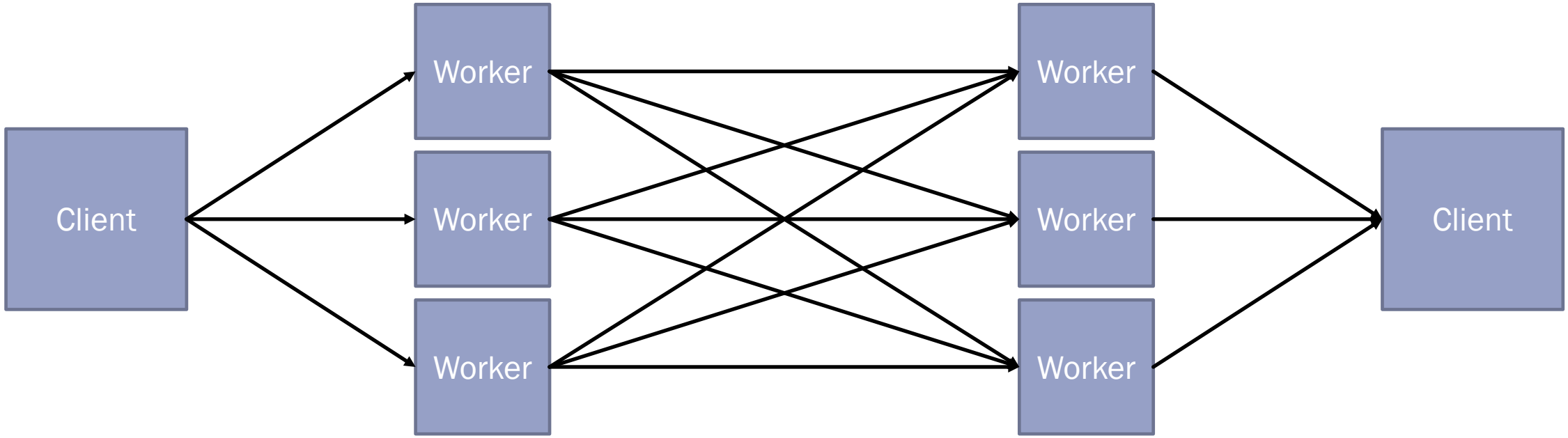
- Goals
  - Create a large dataset containing enough refactoring operations
  - Determine which features are suitable to predict how well an refactoring operation is performing
  - Generate a prediction model based those features and evaluate it

- Main research questions to be answered
  - **Which features of a commit-refactoring combination are suitable to predict the effect of the refactoring method?**
  - **How well does the trained machine learning model perform?**

STEPS

1. Select suitable repositories having an adequate number of commits and refactoring operations
2. Create a dataset with several product metrics and source code measures extracted from the previously selected repositories
3. Explore the dataset and determine which features are strong and suitable to build a prediction model
4. Train the machine learning model
5. Evaluate the model performance
6. Write documentation

DATASET GENERATION PIPELINE EXTENSION



List of top k Github repos

Download and mining with RefactoringMiner

List of refactoring commit hashes

Calculate metrics with Repodriller

Dataset of project metrics

PITFALLS

THE JAVA 9 PITFALL

- Spark is written in Scala based on Java JDK 8
- RefactoringMiner is available through Maven, so just add the dependency and done

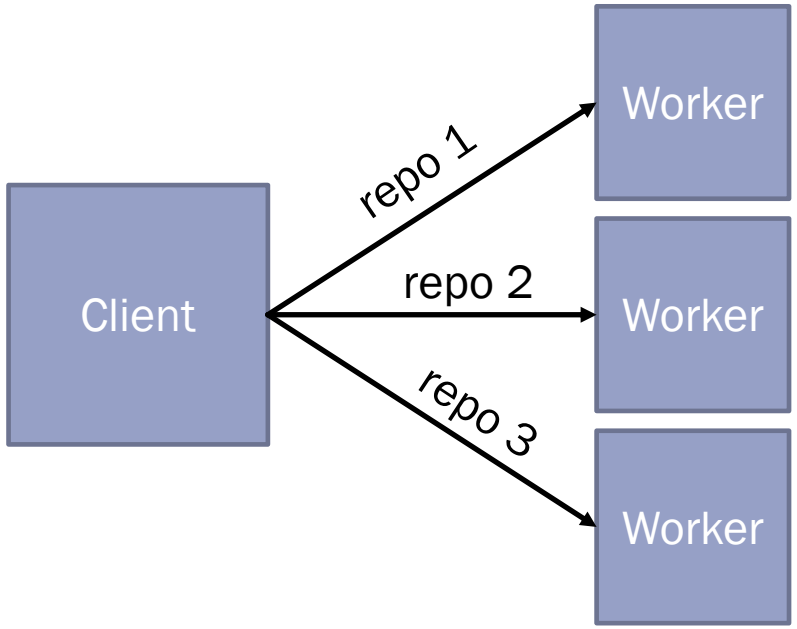
- No, because RefactoringMiner jar's are compiled by Java 9 or higher, therefore it can not be used directly
  - Solution: Clone the repository, compile with JDK 8 and install it to the local Maven repository

SCALABILITY PITFALL

- Original idea: Distribute one repository to each node within the cluster
  - No need to further adapt the RefactoringMiner [2] API
  - Each repository must be cloned only once to one of the worker nodes in the cluster

- But that does not work. Why? Isn't Spark intended to handle big data?
- Yes, but:
  - The RefactoringMiner API provides an async callback, so all refactoring data have to be stored in heap of the JVM
  - Spark cluster consist of powerful computation nodes, but storing about 100k of commit objects in heap is very much data
  - Additional pitfall: Approach to bridge the callback to an iterator in combination with flatMap, but Spark flushes the data only when the iterator has finished

ONE REPOSITORY PER WORKER IMPLEMENTATION



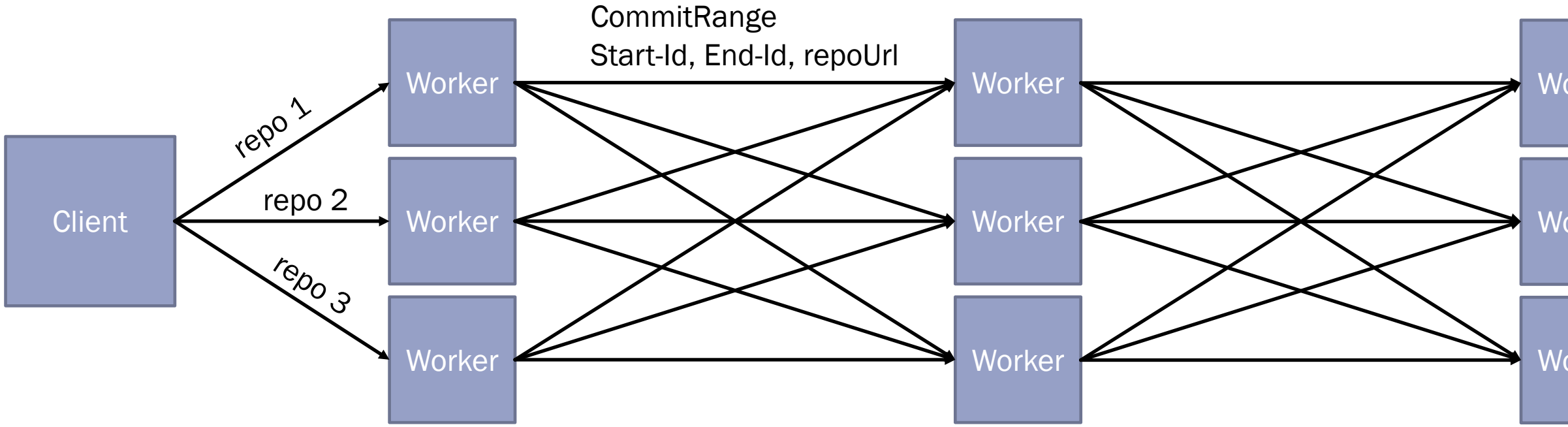
All refactoring commits kept in heap!

SCALABILITY PITFALL

- Second idea: Using a divide&conquer approach by splitting the commits with overlapping first and last element at a batchsize of k commits
  - Only the repository url and the range has to be transferred to each worker
  - The worker clones the repository automatically when it not already exists (this is time consuming!)
  - Parallelization of the repository mining itself -> could improve performance even when mining a single repository
  - RefactoringMiner provides a method that analyzes commits between start and end commit id

- Pitfall:
  - Heap out of space error still exists (sometimes). Why?
  - Debugging showed that batchsize after splitting is not always the same size (e.g. k = 1000, some commit ranges has a size of up to 50k commits per batch)
  - Guessed reason: Git history is not “linear”, even when traversing only one branch
  - The underlying library jGit traverses the tree structure recursively until the commit id is found. So when parent count is higher than 1, it not always chooses the “right” parent it chose when splitting the commits

K COMMIT PER REPOSITORY



scan repository and split commit ids into batches of size k

scan repository and split commit ids into batches of size k
Each worker must keep only k refactoring commits in heap

SCALABILITY PITFALL

- Third idea: Using a divide&conquer approach by splitting the commits with overlapping elements, but storing all commit hash id's
  - Only the repository url and a list of commit hashes have to be transferred
  - The worker clones the repository automatically when it not already exists
  - Parallelization of the repository mining itself -> could improve performance even when mining a single repository

- Pitfall:
  - RefactoringMiner API allows to analyze one commit at a time, but not a list. So every time jGit [6] has to initialize the repository again.
  - Therefore, an extension is needed to accept a list and only initialize jGit once. So RefactoringMiner [4] got forked and adapted!
  - The right balance of batchsize has to be found (out of heap vs. speed)
  - So far, none other pitfalls discovered ☺
  - The computation time (Gradle project) by using a Spark cluster could be dropped from 3,4h to
    - 1,7h with four workers
    - 50 minutes with eight workers

METRICS CALCULATION

- After extracting refactoring commits (consisting of commit id and the detected refactoring operation), Spark calculates in a second step the metrics before and after the refactoring operation
- Commit id's are sufficient since the parent commit can be found via the parent commit
  - RefactoringMiner ensures that no merge commits are analyzed (which could have more than one parent)

DROPPING REPODRILLER

- RepoDriller is essentially only a wrapper for jGit
- Like RefactoringMiner, the API was not very suitable for the Spark Mapreduce-style
- Implemented the metric calculation with jGit [6] and CK [5]
  - Detect changes with the git diff command
  - Locate the file id's within the repository
  - Load the file content and calculate the metrics on all involved java files
  - Metric calculation is kept abstract to simply plugin any other metric processing library or extend with additional metrics

- jGit is very powerful library to traverse repositories and has a lot of online resources when original documentation is missing

PARQUET FILE OUTPUT

- The pipeline outputs files in the parquet file format which
  - is column-based
  - allows compression through run-length encoding (small!)
  - Storing typed data with schema
  - Faster to query under certain circumstances

repository	commit_id	type
https://github.com/gradle/gradle	f666121b2aa03501e5758c09b97162592af103f2	MOVE_CLASS
https://github.com/gradle/gradle	f666121b2aa03501e5758c09b97162592af103f2	MOVE_CLASS
https://github.com/gradle/gradle	f666121b2aa03501e5758c09b97162592af103f2	CHANGE_ATTRIBUTE_TYPE
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	CHANGE_ATTRIBUTE_TYPE
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	CHANGE_PARAMETER_TYPE
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	RENAME_PARAMETER
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	CHANGE_PARAMETER_TYPE
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	CHANGE_PARAMETER_TYPE
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	RENAME_PARAMETER
https://github.com/gradle/gradle	c73bbb70cad1e4ae73ee6dfafdb48be8bfd701f5	CHANGE_PARAMETER_TYPE

Showing 1 to 10 of 29,006 entries

Previous 1 2 3 4 5 ... 2901 Next

WRAP UP

Current state:

- We have a fully working Spark pipeline which allows the extraction of refactoring operations and according metrics
- The pipeline requires a list of Github repositories as input
- Refactoring extraction can be done in a distributed and highly parallelizable way

CURRENT AND NEXT STEPS

- Generate a big dataset (is done at the moment, about 500-1000 of biggest repositories)
- Look at the data
- Try to find a suitable machine learning model to do predictions

REFERENCES

[1] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” pp. 483–494, 2018.

[2] https://github.com/tsantalis/RefactoringMiner

[3] https://github.com/mauricioaniche/repodriller

[4] https://github.com/im-a-giraffe/RefactoringMiner

[5] https://github.com/mauricioaniche/ck

[6] https://www.eclipse.org/jgit/