

GraphRepo: Fast Exploration in Software Repository Mining

Alex Serban, Magiel Bruntink, Joost Visser




Software Repository Mining

- analyse the data available in software repositories, such as **version control** repositories, **bug tracking systems**, etc., to uncover interesting and actionable information about software systems
- Examples of tasks:
 - **Commit analysis** – analyse the type of commits (e.g., bug solving, new features, etc.)
 - **Coupled change analysis** – analyse which modules are changed together (e.g., files, methods)
 - **Code search** – source code search engines
 - **Programs translation** – convert one program from a language to another
 - etc.

Typical Repository Mining Workflow



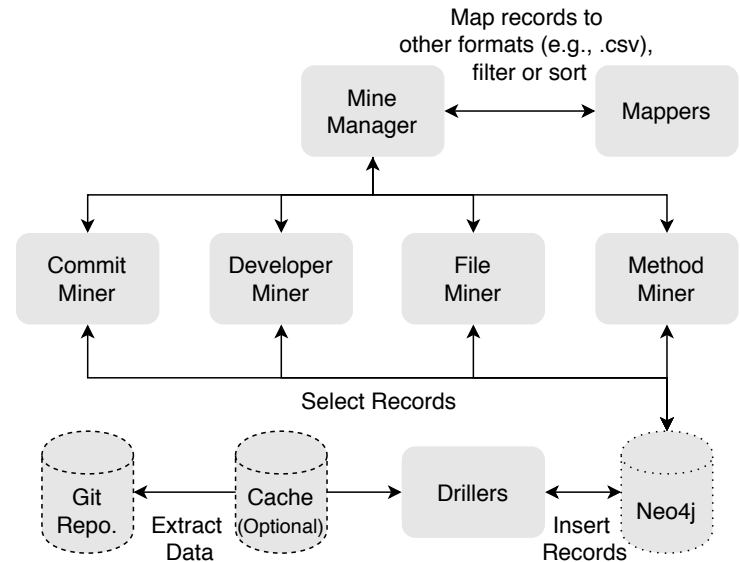
- Define the information one is seeking (e.g., some research questions)
 - Select a number of repositories to analyse
 - Extract the information needed and further process it
- 
- Whenever new questions arise, the information has to be extracted again from repositories
 - The information extraction process is costly

GraphRepo: Fast Exploration in Repository Mining

- GraphRepo extracts and inserts the code related information from repos in Neo4j
 - This allows real time exploration (e.g., answer new questions fast), scalability and data sharing and reuse (e.g., with snapshots)
 - GraphRepo also provides a large set of interfaces with the Python ecosystem (for interoperability), e.g., with PySpark or Scikit-Learn
- The advantage is that once the data is indexed, it can be queried in real time
 - GraphRepo is suitable for scenarios:
 - where the same repositories are reused for multiple analyses,
 - where the data has to be updated continuously (freshness) and maintained consistent
 - where real time exploration is desired

GraphRepo Architecture

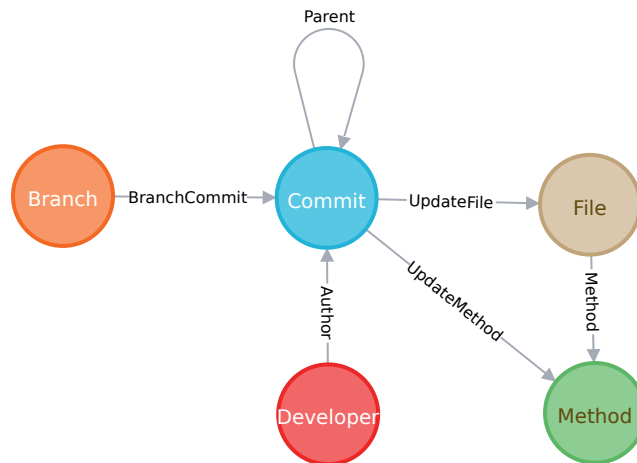
- Mappers provide a way to further process the data (e.g., convert it to other formats) or interoperate with other frameworks (e.g., PySpark)
- Miners are default components to query the data in Python. They include default queries and can be extended easily
- Drillers parse data from repositories and index it in Neo4j. They also provide various caching mechanisms



Graph Schema



- The schema is generic and contain the 4 universally available entities in repos: Developers, Commits, Files and Methods
- Additionally we represent branches as nodes to allow faster selection by branch (although this info. can not always be reconstructed)
- The update relationships hold metadata about the updates, e.g., the #loc added or removed, the source code before and after the update, the method complexity after updates, etc.
- All repos are indexed in the same database. This allows the analysis of teams of developers working on multiple projects



Benchmarks

Table 1: Project details, where the number of nodes and relationships for each project is extracted from Neo4j.

ID	Name	Technologies	Start - End Dates	#Devs.	#Commits	#Files	#Methods	#Nodes	#Relationships
P1	Hadoop	Java	01.01.2018 - 01.01.2019	107	2359	6613	43817	52897	127837
P2	Jax	Python	01.01.2019 - 01.05.2020	182	3721	299	7963	12166	42640
P3	Kibana	Java/Javascript	01.06.2018 - 01.06.2019	232	5826	26827	15227	48113	130642
P4	Tensorflow	C++/Python/Go	01.12.2019 - 01.05.2020	568	11403	11549	51335	74856	213368

Table 2: Benchmark results. The Driller column is present for informative purposes and represents the time needed to extract the data using PyDriller.

PID	Driller	Insert & Index	Most Costly Insert		Q1		Q2		Q3		Q4		Q5	
	Time	Time	Query	Time	Time	#Changes	Time	#Methods	Time	#Files	Time	#Methods	Time	
P1	13m	7m	UpdateFile	5m	21ms	62	3s	143	3s	443	34ms	1262	40ms	
P2	8m	5m	UpdateFile	3m	22ms	129	1s	192	4s	60	24ms	929	39ms	
P3	35m	13m	UpdateFile	9m	21ms	36	1s	59	1s	3083	53ms	3048	67ms	
P4	1h59m	51m	UpdateFile	34m	43ms	77	2s	208	5s	4782	95ms	15545	2s	

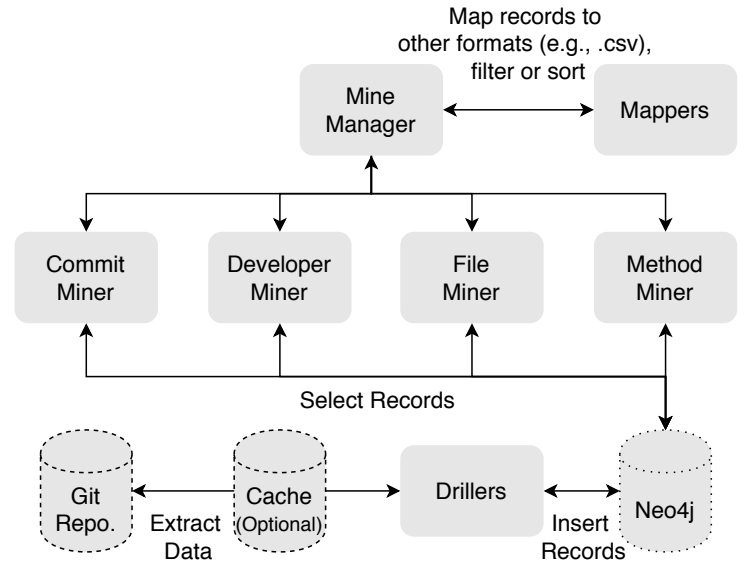
Table 3: Query description and complexity in nr. of lines of code (#loc) for benchmark queries. The #loc includes both the initialization of GraphRepo and the result mapping.

ID	Description	#loc
Q1	Select all nodes and relationships for a project.	2
Q2	Select the evolution of a file's #loc over time, for a specific file.	3
Q3	Select the evolution of method complexity over time, for all methods in a file.	7
Q4	Select all files edited grouped by file type, for a specific developer.	4
Q5	Select the average complexity of methods edited by a developer in all her commits.	4

Note: All benchmarks ran on a cloud instance with 2 vCPUs and 4 GB RAM. Indexing was performed with a batch size of 50 and all the available information was stored in Neo4j. This includes the source code before and after a commit, for each file edited in a commit. This corresponds to the "Most Costly Insert" UpdateFile (Table 2) query. When the full source code is not indexed (only the file diffs are indexed), the insert performance increases with ~90%.

Advantages of using GraphRepo

- Query performance (real-time exploration of repositories), and scalability
- Easy to extend and interoperate with Python's rich ecosystem (by developing new miner and mappers)
- Easy to maintain data consistency and reuse the data across experiments
- Easy to reproduce experiments, by only sharing the GraphRepo config files and any custom Mapper



Demo & QA

For more information see:

- the project's documentation: <https://graphrepo.readthedocs.io/en/latest/>
- the project's repo: <https://github.com/NullConvergence/GraphRepo>