Lightweight Collection and Storage of Software Repository Data with DataRover

Thomas Kowark, Christoph Matthies, Matthias Uflacker, and Hasso Plattner
Hasso Plattner Institute, University of Potsdam
August-Bebel-Str. 88
Potsdam, Germany
{firstname.lastname}@hpi.de

ABSTRACT

The ease of setting up collaboration infrastructures for software engineering projects creates a challenge for researchers that aim to analyze the resulting data. As teams can choose from various available software-as-a-service solutions and can configure them with a few clicks, researchers have to create and maintain multiple implementations for collecting and aggregating the collaboration data in order to perform their analyses across different setups. The DataRover system presented in this paper simplifies this task by only requiring custom source code for API authentication and querying. Data transformation and linkage is performed based on mappings, which users can define based on sample responses through a graphical front end. This allows storing the same input data in formats and databases most suitable for the intended analysis without requiring additional coding. Furthermore, API responses are continuously monitored to detect changes and allow users to update their mappings and data collectors accordingly. A screencast of the described use cases is available at https://youtu.be/mt4ztff4SfU.

CCS Concepts

•Information systems → Extraction, transformation and loading; •Software and its engineering → *Software evolution;*

Keywords

Data Collection, Data Mapping Definition, Link Discovery, API monitoring, Data Storage

1. INTRODUCTION

Software development teams are nowadays not only supported by issue trackers and source code management systems. A variety of software-as-a-service solutions for tasks like project management, continuous integration, or code analysis, can be set up with only a few clicks. The information stored in these systems is vital for researchers to reason about the effects of software engineering practices and their effects on product and process metrics. In order to perform such analyses the data first needs to be **e**xtracted from the different source system, **t**ransformed into a single, holistic representation, and **l**oaded into a suitable target database, in a so-called ETL process.

In data collection, it is challenging to keep up with the development of new systems and the evolution of existing tools. Connection plugins or scripts need to be created and maintained either on-demand or by maintainers of plugins for systems like Sonarqube[1], in order to be able to collect the required data. With regards to data storage, the suitability of database systems and data formats is influenced by the intended analysis. If social graphs formed by developers are of interest, a graph database that is shipped with built-in graph analysis algorithms is superior to a document store lacking those features. The document store, on the other hand, allows for simpler text analysis. Thus, in addition to providing source code for data collection, the same input data needs to be transformable to different output formats depending on the analysis use case.

In this paper, we introduce **DataRover**. The system reduces the implementation effort for ETL workflows by separating API connection and querying from data storage. Minimalistic connector implementations solely perform API connection and querying. The returned results are transformed into property graphs according to user-defined mappings, which are created through a graphical front end based on sample JavaScript Object Notation (JSON[2]) responses. By that, users can link ground data from different sources and store it in a schema and database most suitable for their use case without programming knowledge. DataRover further monitors API responses to detect changes in document structure or endpoints and, thus, allows to change connector implementations or data mappings in a timely manner.

The paper first presents DataRover's target users and their desired use cases (Section 2). Subsequently, we walk through a sample parsing procedure (Section 3) and discuss the system's architecture along with its implications for the development of connector implementations and storage components in Section 4. Initial evaluation ideas and results are shown in Section 5. Finally, we discuss related work, the current state of implementation, and future work.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970286

[1]http://www.sonarqube.org
[2]http://www.json.org

2. USERS

Our main user group are researchers in the area of software repository mining (MSR). In their work, they analyze the influence of software development practices on the outcome and progress of projects. To this end, they need to collect artefacts created by software development teams, process them (cleansing, link detection, etc.) and then test their hypotheses by executing queries on the created dataset. For validation purposes, they also have to perform this import-process-analyze workflow for data originating from projects which employed a different collaboration infrastructure.

Beyond researchers, the people involved in the software development processes (developers, project managers, etc.) have a valid interest in such analyses to introspect and improve their development practices in a data-driven manner. While their infrastructures rarely change completely, updates or replacement of existing and addition of new collaboration tools also require at least a review of the data extraction scripts and systems; in the worst case new tools have to be developed.

For both groups, their primary task is not programming or maintaining these systems, but analyzing the gathered data. Hence, effort in terms of programming, setup overhead, and general amount of interactions with the system should be minimal. This requires the tool to provide a simple front end for selecting and configuring parsing implementations, as well as defining how returned data is stored in the desired target database. Furthermore, the implementation should reduce the amount of programming required to update an existing or create a new parsing implementation or connection to a new database management system. We discuss both aspects of our system in the following chapters.

3. DATA COLLECTION & LINKAGE

We now outline how researchers and and practitioners can setup DataRover. As a use case, we recreate the work done for [7] and collect issue tracker and repository data from the "Ruby on Rails" Github project and combine it with Travis-CI[3] build information. As an extension of the original study, we further integrate question and answer data from Stackoverflow and link it to the developers and contributors of the Rails project. A screencast of this use case is provided at https://youtu.be/mt4ztff4SfU. The created datasets are available at https://bit.ly/kowark-ase-16-data.

3.1 Job Definition

Data collection *Missions* are defined by specifying the database type that results will be written to, as well as the database url, and access credentials. Currently users can select between Neo4j graphs, MongoDB document stores, and different schemas for an SAP HANA database. Support for further relational databases is planned. Afterwards, users can select *Explorers* from the list of available implementations. Each explorer internally defines which input data is required from the user and a simple input form is rendered accordingly. For the Github and Travis-CI explorers, the repository name and owner suffices as information. If the to be parsed project is private, access can be granted using OAuth authorisation. Finally, users can define how often parsing tasks are performed, ranging from continuous parsing to only daily explorer runs. The mission control overview shows all selected explorers, their parameters, and status messages created throughout data collection and storage runs on a single screen (see Figure 1).

[3]http://api.travis-ci.org/

3.2 Mapping Selection

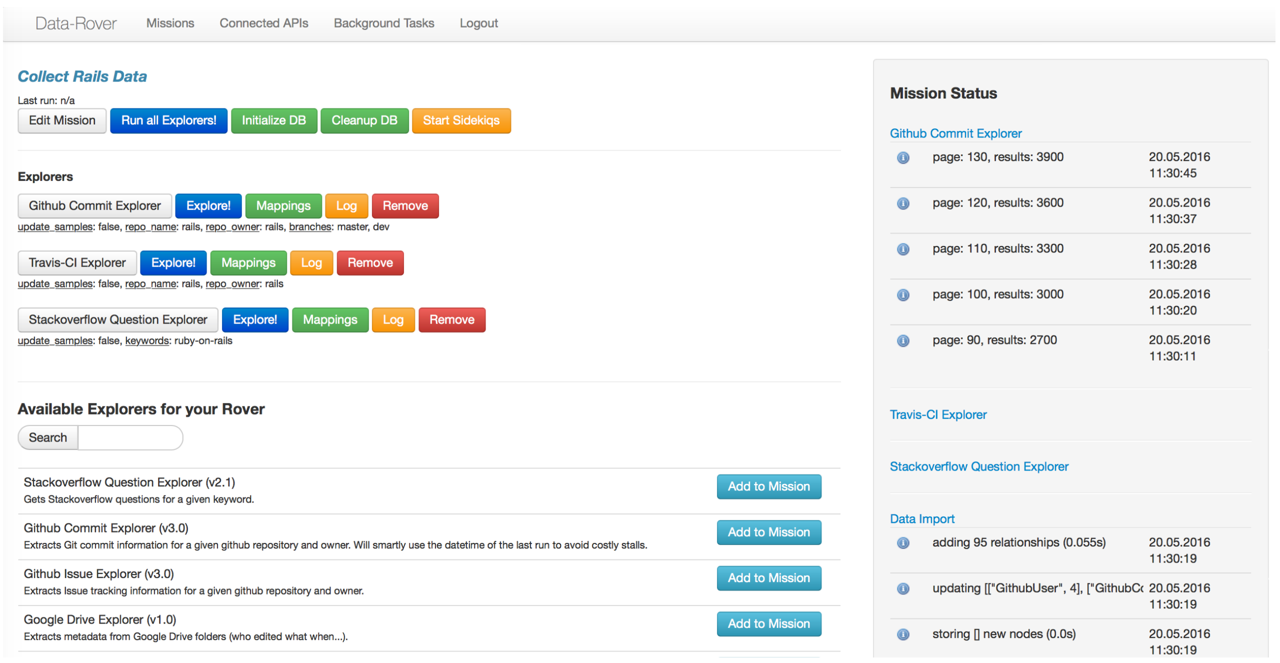
Once an explorer is chosen and configured with the required input data, users are asked to define the data mapping. Each mapping can be named and stored in order to be reused as a template. The interface for mapping definition is displayed in Figure 2. Based on a JSON sample that is created from previous API calls (see Section 4 for details), users define which data schema they want to achieve. To this end, they define classes for elements, choose to ignore keys, move elements between JSON objects, or define links between nodes. To better understand the resulting data structures, a graphical representation is available on demand (see Figure 3). Mapping templates can be loaded through a dropdown list. The "opt-in" flag specifies whether elements, for which no mapping is defined, are ignored or stored automatically (see Section 4).

In the example use case, one possible option is to create a flat object hierarchy that contains commits as nodes. Committer and author information is added as attributes of these nodes instead of links to nodes of type "GithubUser". Thus, the memory footprint of the resulting data set can be reduced, which might be necessary for larger projects containing thousands of users. Build information from Travis-CI is in this case also be added as an attribute, e.g., as a boolean flag stating that the build was successful. To achieve this, the mapping for builds does not create a separate node of type "Build", but identifies the corresponding GithubCommit node using the commit url, and adds the desired attribute.

For other analyses, graph structures are a better choice, as they, for example, allow to detect developer groups around certain files and folders through network analysis. In this case, users define a mapping that creates nodes for each element and establishes links between them instead of storing the same information twice. Builds could thereby also become nodes of their own, and links to the respective commits are established.

3.3 Mission Extension

As an extension point of the original study, Stackoverflow data can be used to provide a proxy for someone's expertise with a given language or framework. To add this data, users first need to select the respective explorer and define keywords of interest (e.g., "ruby-on-rails"). Secondly, they have to create a mapping that specifies a link between the Stackoverflow data and the Github/Travis-CI data. This can be achieved by creating a foreign key field on the `display_name` property of the Stackoverflow user object. If a user registered their Github and Stackoverflow accounts with the same username, the system creates the respective link between the two user objects. Similar to the Travis-CI/Github linkage, the memory footprint can be reduced by merging the Stackoverflow user information into an existing Github user node. Beyond explicit links, users can instruct DataRover to scan strings like the body of Stackoverflow questions for URLs. If hyperlinks are found, that reference, for example, a Github Issue or a Github Commit, they are also created within the resulting dataset.


Data-Rover Missions Connected APIs Background Tasks Logout

Collect Rails Data

Last run: n/a

Edit Mission | Run all Explorers! | Initialize DB | Cleanup DB | Start Sidekiqs

Explorers

Github Commit Explorer | Explore! | Mappings | Log | Remove

update_samples: false, repo_name: rails, repo_owner: rails, branches: master, dev

Travis-CI Explorer | Explore! | Mappings | Log | Remove

update_samples: false, repo_name: rails, repo_owner: rails

Stackoverflow Question Explorer | Explore! | Mappings | Log | Remove

update_samples: false, keywords: ruby-on-rails

Available Explorers for your Rover

Search

Stackoverflow Question Explorer (v2.1)
Gets Stackoverflow questions for a given keyword. — Add to Mission

Github Commit Explorer (v3.0)
Extracts Git commit information for a given github repository and owner. Will smartly use the datetime of the last run to avoid costly stalls. — Add to Mission

Github Issue Explorer (v3.0)
Extracts Issue tracking information for a given github repository and owner. — Add to Mission

Google Drive Explorer (v1.0)
Extracts metadata from Google Drive folders (who edited what when...). — Add to Mission

Mission Status

Github Commit Explorer

●	page: 130, results: 3900	20.05.2016 11:30:45
●	page: 120, results: 3600	20.05.2016 11:30:37
●	page: 110, results: 3300	20.05.2016 11:30:28
●	page: 100, results: 3000	20.05.2016 11:30:20
●	page: 90, results: 2700	20.05.2016 11:30:11

Travis-CI Explorer

Stackoverflow Question Explorer

Data Import

●	adding 95 relationships (0.055s)	20.05.2016 11:30:19
●	updating [["GithubUser", 4], ["GithubCc	20.05.2016 11:30:19
●	storing [] new nodes (0.0s)	20.05.2016 11:30:19

Figure 1: Screenshot of the mission control overview page of DataRover. Users can edit mission parameters, add explorers from the catalogue, jump to configuration pages for each explorer, and get latest status information from each explorer and the data storage system on the right hand side. Explorers can also be started individually or all at once through the "Explore!" and "Run all Explorers!" buttons.

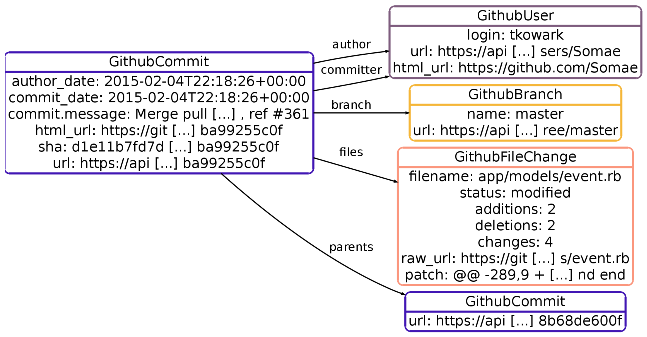

GithubCommit
author_date: 2015-02-04T22:18:26+00:00
commit_date: 2015-02-04T22:18:26+00:00
commit.message: Merge pull [...] , ref #361
html_url: https://git [...] ba99255c0f
sha: d1e11b7fd7d [...] ba99255c0f
url: https://api [...] ba99255c0f
author
committer
branch
files
parents
GithubUser
login: tkowark
url: https://api [...] sers/Somae
html_url: https://github.com/Somae
GithubBranch
name: master
url: https://api [...] ree/master
GithubFileChange
filename: app/models/event.rb
status: modified
additions: 2
deletions: 2
changes: 4
raw_url: https://git [...] s/event.rb
patch: @@ -289,9 + [...] nd end
GithubCommit
url: https://api [...] 8b68de600f

Figure 3: Graphical representation of the "Linked Mapping" for Github repository data.

3.4 API Changes

Once a data collection job is set up, it will continuously run until users decide to stop it. If during this time the API of a tool changes, explorers will produce errors, of which the users are notified via email. They can then either select a new explorer that is suitable for the new API version or need to implement one themselves. If changes only manifest in the returned data, e.g., if a field is added to the result, users will also get notified, but only have to change the mapping definition instead of altering the parser definition. Once the mapping is updated, the explorers are restarted and parsing continues as expected.

In summary, the front end limits user interaction to providing input parameters for the explorers and adapting data mappings to their needs. By using template mappings, and the "opt-in" flag, the amount of mapping entries that have to be provided manually is limited to only fields of direct interest to the user.

4. IMPLEMENTATION

DataRover is a web application written in Ruby on Rails. Its architecture is presented in Figure 4. In the following, we describe implementation details of the main classes: explorers, mappings, and data stores.

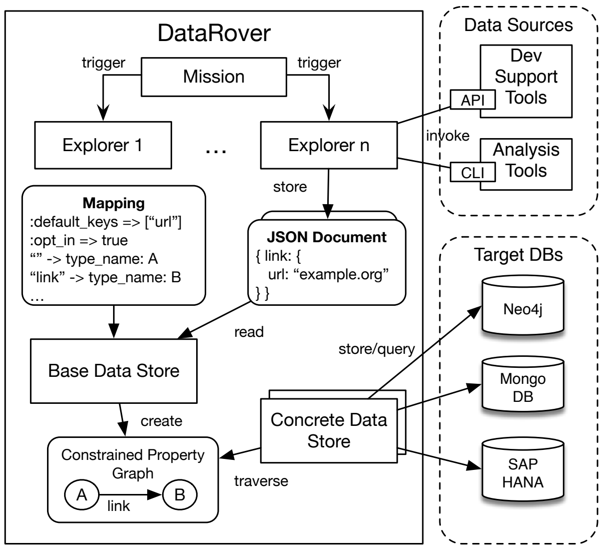
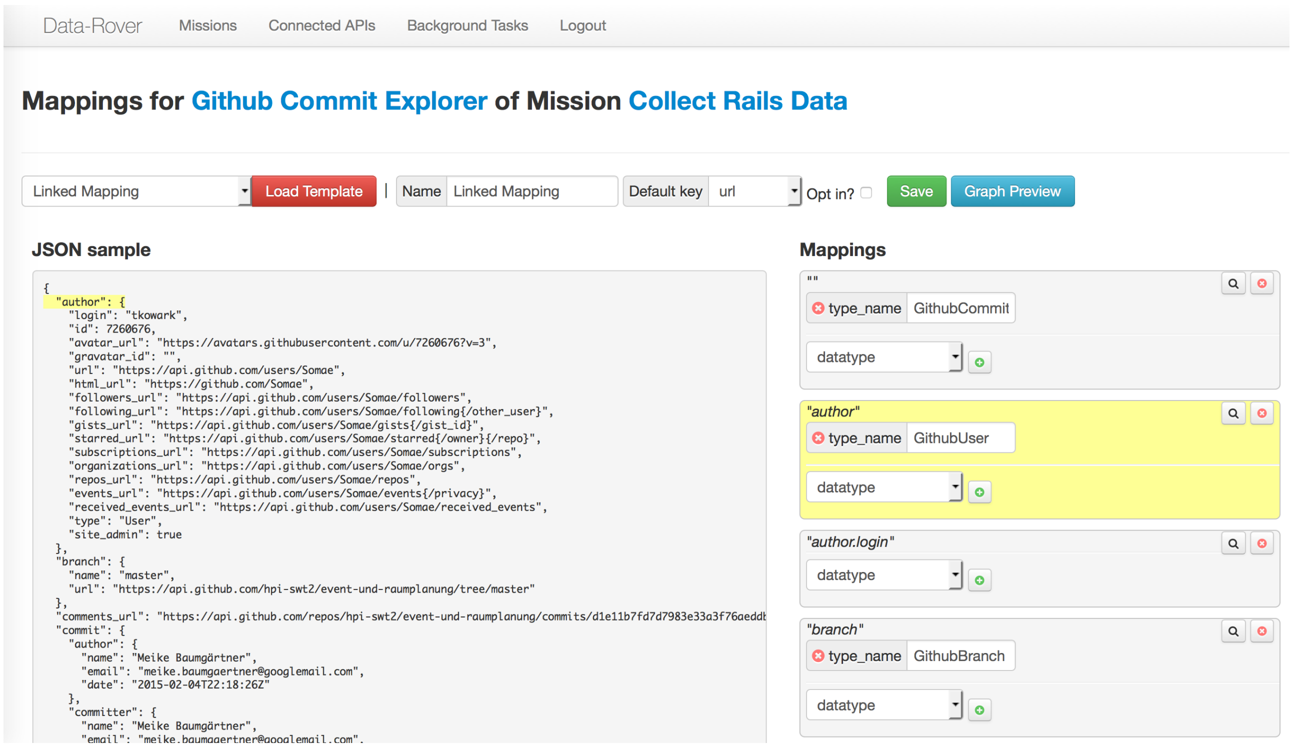


Figure 4: Overview of the internal classes (angled corners) and data (round corners) involved in storing software repository information with DataRover.



Mappings for Github Commit Explorer of Mission Collect Rails Data

Linked Mapping | Name: Linked Mapping | Default key: url | Opt in? ☐

```
{
  "author": {
    "login": "tkowark",
    "id": 7260676,
    "avatar_url": "https://avatars.githubusercontent.com/u/7260676?v=3",
    "gravatar_id": "",
    "url": "https://api.github.com/users/Somae",
    "html_url": "https://github.com/Somae",
    "followers_url": "https://api.github.com/users/Somae/followers",
    "following_url": "https://api.github.com/users/Somae/following{/other_user}",
    "gists_url": "https://api.github.com/users/Somae/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/Somae/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/Somae/subscriptions",
    "organizations_url": "https://api.github.com/users/Somae/orgs",
    "repos_url": "https://api.github.com/users/Somae/repos",
    "events_url": "https://api.github.com/users/Somae/events{/privacy}",
    "received_events_url": "https://api.github.com/users/Somae/received_events",
    "type": "User",
    "site_admin": true
  },
  "branch": {
    "name": "master",
    "url": "https://api.github.com/hpi-swt2/event-und-raumplanung/tree/master"
  },
  "comments_url": "https://api.github.com/repos/hpi-swt2/event-und-raumplanung/commits/d1e11b7fd7d7983e33a3f76aedd
  "commit": {
    "author": {
      "name": "Meike Baumgärtner",
      "email": "meike.baumgaertner@googlemail.com",
      "date": "2015-02-04T22:18:26Z"
    },
    "committer": {
      "name": "Meike Baumgärtner",
      "email": "meike.baumgaertner@googlemail.com",
```

""
type_name: GithubCommit
datatype

"author"
type_name: GithubUser
datatype

"author.login"
datatype

"branch"
type_name: GithubBranch
datatype

Figure 2: Screenshot of the mapping definition front end of DataRover. Users can select JSON element on the left side and either edit an existing or create a new mapping. For the overall collection of mappings, it is possible to define whether unmapped elements are by default ignored or stored, as is. The "Graph View" button triggers the creation of a graphical representation of the resulting dataset (see Figure 3).

4.1 Explorers

Each explorer is a subclass of an abstract explorer class. To implement a new explorer, only a single method that performs the respective API call using the parameters specified by the user has to be implemented. For example, the explorer for git repository data on Github queries the API for all commits in a given repository and continues to collect details (i.e., file changes and patches) about the single commits, which are not included in the initial response.

Each returned element (i.e., JSON object containing information about commits) is stored in a "DataStore" instance. These instances are initialized with a mapping and later perform data transformation, linking, and storage. The explorer implementation itself is oblivious to this process. Hence, for example, the whole parser for Github Commits only contains code for client creation, a call to retrieve all branches, retrieving a list of commits, and the details for each commit. Overall, 10 lines of code (LoC) suffice when using the Octokit[4] client gem for Ruby. Creating similar implementations for other APIs are straightforward as Ruby even provides internal mechanisms for XML to JSON transformation (*Hash.from_xml(xml_doc).to_json*). Other potential data sources are third party tools and libraries, e.g., for source code analysis. For them, the implementation includes encapsulating preparations of the source code repository, configuring the tool, calling it through the command line interface (CLI), parsing the output, and transforming it into JSON. For the metric_fu tool suite available for Ruby on Rails[5], this amounts to roughly 200 LoC, albeit most of this performs repository setup and cleaning and is therefore independent from changes to the tool.

4.2 Data Mappings

For the JSON documents retrieved by an explorer, a mapping needs to be present which, in turn, comprises a set of mapping entries. Each mapping entry either directly refers to a JSON key or specifies a regular expression. In the latter case, the mapping is applied for all matching elements. This, for example, is useful to avoid duplicate mapping definitions for all elements ending with "_at" within Github API responses. Within mapping entries, any of the options presented in Table 1 can be specified. DataRover checks wether combinations of options are invalid or contradictory (e.g., using type_name and datatype for the same element) and warns the user accordingly. Unmapped JSON elements are ignored if the "opt_in" flag is set for the entire mapping. Otherwise, the elements will be stored using the corresponding JSON keys as relation, or attribute names. Type names have to be provided by users for all objects nested within the JSON document. Finally, a "default_key" can be defined to avoid having to specify "url" as a uniquely identifying attribute for all objects.

4.3 Data Stores

Based on the provided mapping, the *DataStore* base class performs transformation of JSON objects into constrained

[4]https://github.com/octokit/octokit.rb

[5]https://github.com/metricfu/metric_fu

Table 1: Reference of mapping entry options.

Mapping Key	Meaning
:type_name	Type of a resulting node. If the foreign_key option is also set, type of the linked node.
:primary_key	JSON key that uniquely identifies a node, e.g., a Github URL or an email address
:datatype	Allows to denote special types like long strings or dates, which require different column types or preprocessing for certain target databases.
:ignore	Skips this field in graph creation
:move_to	Target element to which a JSON element and its sub-elements are moved
:rename_to	Allows renaming an attribute or relation
:foreign_key	Uniquely identifying attribute of a node class which this element refers to
:contains_links	Strings will be parsed for URLs that identify other nodes
:no_updates	Only a relation is created but the linked object is not updated.
:no_inserts	Relations are only inserted if the linked object is already present in the database.

directed property graphs in which nodes are labeled with a class, have attributes, and are linked by relations, which itself do not contain attributes. Once the graphs are built, they are stored into concrete database systems according to rules defined within database specific data store implementations. For graph databases, the resulting property graph is stored, as is. For relational databases, a table for each object class is created and relations are persisted in separate join tables. To avoid creation of duplicates upon multiple parsing runs, the primary_key property is used by the data stores to query for existing elements, which in turn are upserted with the newly retrieved data. Thus, each concrete data store has to also implement a query interface allowing for node lookups based on primary key and attribute values.

4.4 API Monitoring

DataRover monitors the queried APIs. Contrary to the approach presented in [6], our system does not use a proxy server, but directly compares each returned JSON object against previously received versions. This allows to detect whether new elements have been added to the responses or if data types of any elements changed. Null-values, empty strings, or empty arrays do not overwrite existing samples. If elements are no longer detected in the JSON responses, they are not removed from the sample, as it is possible that this behavior is specific to a certain project and does not apply to all explorers. This method can also be used to bootstrap mappings when creating new explorers. Upon their first invocation without a valid mapping, responses are collected to form a sample JSON file. From this file, a sample mapping is created containing an entry for each detected JSON key. Users can then modify these mappings as needed.

5. EVALUATION

Evaluation of our system is ongoing. To show that it simplifies ETL processes for MSR researchers, we plan to perform an evaluation according to Kitchenham [4], i.e., define requirements for an ETL tool, identify shortcomings of existing solutions, and discuss how our tool overcomes them based on exemplary use cases. Beyond ease of use, the following aspects are also relevant for assessing data rover.

Table 2: Query times and memory consumption for flat and linked mappings on the Ruby on Rails repository data. System: 2,3 GHz Macbook Pro, 16GB RAM, Mac OS X 10.11.5. Queries: average of 20 runs, fastest and slowest removed. Databases: Neo4j Community 2.3.2 and PostgreSQL 9.4.1.

Schema	Memory	Query Runtime
Linked, no "opt-in" (Neo4j)	261.08 MiB	492ms
Linked (Neo4j)	121.15 MiB	280ms
Flat (Neo4j)	132.57 MiB	221ms
Flat (PostgreSQL)	24.50 MiB	30ms
Linked (PostgreSQL)	33.88 MiB	54ms

5.1 Data Footprint

For a first estimate of different data schemas on memory consumption and query performance, we extracted commit data from the Ruby on Rails Github project[6] and stored it once using a flat mapping, and using a linked version, with and without the "opt-in" flag being set. As of May 20, 2016, the repository contains around 58.000 commits created by close to 3000 collaborators. Using the flat mapping and omitting any unnecessary elements, only one node per commit is created containing the email addresses of the committer and author, as well as the commit timestamps. Using the graph-like mapping, separate nodes for users and commits are created. As an example query, the amount of commits contributed by each author is determined.

The results for a Neo4j database are shown in Table 2. Without the "opt-in" flag, the resulting data set is about twice as big as the minimal one. In terms of total data size, this difference seems marginal at first, but scaling up to multiple repositories of similar or even larger size, it can decide whether an analysis is performable on a personal laptop or a larger system is required. The flat mapping does not further reduce memory footprint as each commit node needs to store duplicate information, which outweighs the overhead for node, relationship, and index creation for users. Query times for the flat mapping, on the other hand are faster, as no relationships need to be considered. Storing the same data in a PostgreSQL database, the flat mapping not only provides faster query times but also reduces memory footprint significantly. We plan to extend these analyses to data sets created from multiple sources and more complex queries in the future.

5.2 Data Import

The flexibility in data storage comes at the cost of additional computing that is necessary to transform each returned JSON object, lookup objects in the target database by their primary keys, and perform the upsert operations. To compare our solution to state-of-the-art implementations, we plan to perform identical data import tasks with the systems mentioned in Section 6 and DataRover and measure the overall completion time. Through data mappings identical data schemas can be created, thus, allowing to compare

[6]https://github.com/rails/rails

data quality, as well. For the presented use case, we can already conclude that, in comparison to the response times of the APIs, the data import overhead is negligible. Hence, future evaluation will mainly focus on cases where data import costs slow down the overall process significantly, such as the import of large, local Git repositories.

5.3 Link Discovery

To assess the quality of link discovery, a gold standard dataset of Stackoverflow members and their corresponding Github accounts needs to be created and compared against the links DataRover detected through foreign key mapping and link detection in texts. From this data, precision and recall allow to quantify link quality and compare it to competing systems. For the sample use case, links for 2320 of the total 3075 Github user accounts could be detected using the foreign key approach presented in Section 3.

Through these different evaluations, we aim to not only assess whether DataRover actually allows MSR researchers and practitioners to collect, link, and store software repository data in a more convenient manner with less programming overhead, but also determine if the resulting datasets allow for faster analysis by reducing query times or memory consumption.

6. RELATED WORK

Import of ground data is a necessity for all software repository mining (MSR) tools. Mainly, this task is achieved through plugin architectures, which allow for distributed development of data collectors and on-demand addition of new features. A downside of this approach is the strong connection to the target platform. An Alitheia Core [3] plugin for parsing Bugzilla data, for example, requires approximately 300 lines of code, most of which implements integration of the data into the platform and therefore is not reusable for other tools. A different approach was taken by Ghezzi et al., who developed SOFAS, a service-oriented platform for software repository mining [2]. Here, data collectors are standalone services who return data in RDF format according to the Software Evolution ontology family. While RDF data can be transformed into other formats to enable different use cases, e.g., by using RDF-to-relational mapping systems like ontop [5], the additional data processing step is time consuming, especially for large software repositories. To process such large repositories, BOA was developed [1]. It frees users entirely from data importing tasks and lets them focus on large scale analysis. Unfortunately, this setup also prevents single users from directly extending and using the platform for their own purposes, e.g., importing data from industry projects. Our system therefore extends the existing landscape of MSR tools with lightweight data collection and transformation. The configurable mappings further enable reuse of existing analyses by creating data schemas similar to the ones used by the aforementioned tools.

7. SUMMARY

In this paper, we presented how DataRover stores software repository data by mapping input JSON objects to constrained property graphs. Using this approach, minimalistic data sets tailored to specific use cases can be created, which reduces memory consumption and increases query performance. The system's linking mechanisms further allow to add new data sources on demand in order to extend existing analyses. With the built-in API monitoring mechanism, necessary adaptations to explorer implementations or mapping definitions are detected in a timely manner.

7.1 Availability

The system is publicly available at http://bitbucket.org/tkowark/data-rover and distributed under MIT license. The presented use cases can already be performed and additional explorers for Jira issue trackers, organisation directories of large companies, and plain websites are available. Deployment is automated through Capistrano[7] tasks, which create a running DataRover instance on a selected target system. To create such a system, a Vagrant[8] file can be used to install and setup all necessary operating system packages within a virtual machine.

7.2 Future Work

Besides cleaning up the implementation and providing bugfixes, future development will focus on creating explorers for additional source systems and supporting further target databases. With this foundation, we aim to deploy DataRover within real companies, to observes how the instant availability of holistic, interlinked software development data is used by development teams in practice.

8. REFERENCES

[1] R. Dyer, H. Nguyen, H. Rajan, and T. Nguyen. Mining Source Code Repositories with Boa. In *SPLASH 2013*, October 2013.

[2] G. Ghezzi. *SOFAS, Software Analysis as a Service. Improving and rethinking software evolution analysis.* PhD thesis, University of Zurich, 2012.

[3] G. Gousios and D. Spinellis. A Platform for Software Engineering Research. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 31–40, 2009.

[4] B. A. Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *SIGSOFT Softw. Eng. Notes*, 21(1):11–14, Jan. 1996.

[5] M. Rodriguez-Muro, R. Kontchakov, and M. Zakharyaschev. Ontology-based data access: Ontop of databases. In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, pages 558–573, 2013.

[6] S. M. Sohan, C. Anslow, and F. Maurer. Spyrest: Automated restful API documentation using an HTTP proxy server (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 271–276, 2015.

[7] P. Wagstrom, C. Jergensen, and A. Sarma. A Network of Rails: A Graph Dataset of Ruby on Rails and Associated Projects. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 229–232, Piscataway, NJ, USA, 2013. IEEE Press.

[7]http://capistranorb.com
[8]http://vagrantup.com/