EMSE manuscript No.
(will be inserted by the editor)

World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS data

Yuxing Ma · Tapajit Dey · Chris Bogart · Sadika Amreen · Marat Valiev · Adam Tutko · David Kennard · Russell Zaretzki · Audris Mockus

Received: date / Accepted: date

Abstract Open source software (OSS) is essential for modern society and, while substantial research has been done on individual (typically central) projects, only a limited understanding of the periphery of the entire OSS ecosystem exists. For example, how are the tens of millions of projects in the periphery interconnected through technical dependencies, code sharing, or knowledge flow? To answer such questions we: a) create a very large and frequently updated collection of version control data in the entire FLOSS ecosystems named World of Code (WoC), that can completely cross-reference authors, projects, commits, blobs, dependencies, and history of the FLOSS ecosystems and b) provide capabilities to efficiently correct, augment, query, and analyze that data. Our current WoC implementation is capable of being updated on a monthly basis and contains over 24B git objects. To evaluate its research potential and to create vignettes for its usage, we employ WoC in conducting several research tasks. In particular, we find that it is capable of supporting trend evaluation, ecosystem measurement, and the determination of package usage. We expect WoC to spur investigation into global properties of OSS development leading to increased resiliency of the entire OSS ecosystem. Our infrastructure facilitates the discovery of key technical dependencies, code flow, and social networks that provide the basis to determine the structure and evolution of the relationships that drive FLOSS activities and innovation.

Keywords Software mining · Software supply chain · Software ecosystem

Yuxing Ma, Tapajit Dey, Sadika Amreen, Adam Tutko, David Kennard, Audris Mockus
Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville, US
E-mail: yma28@vols.utk.edu, tdey2@vols.utk.edu, samreen@vols.utk.edu, atutko@vols.utk.edu, dkennard@vols.utk.edu, audris@utk.edu

Chris Bogart, Marat Valiev
Carnegie Mellon University, US
E-mail: cbogart@andrew.cmu.edu, mvaliev@andrew.cmu.edu

Russell Zaretzki
Department of Business Analytics & Statistics,
University of Tennessee, Knoxville, US
E-mail: rzaretzk@utk.edu

1 Introduction

Tens of millions of software projects hosted on GitHub and other forges attest to the rapid growth and popularity of Free/Libre Open Source Software (FLOSS). These online repositories include a variety of software projects ranging from classroom assignments to components, libraries, and frameworks used by millions of other projects. Such large collections of projects are currently archived in public version control systems, and, if made available for analysis conveniently, would represent a unique opportunity to study FLOSS at large and answer both theoretical and practical questions that rely on the availability of the entirety of FLOSS data. In particular, this infrastructure, referred to as World of Code (WoC) and described below, allows researchers to conduct a census of open source software that would provide types and prevalence across projects, technologies, and practices and serve as a guide to setting policies or creating innovative services. Our infrastructure facilitates the discovery of key technical dependencies, code flow, and social networks that provide the basis to understand the structure and evolution of the relationships that drive FLOSS activities and innovation. Such a large database of software development activities can serve as a basis for "natural experiments" that evaluate the effectiveness of different software development approaches. If preserved, it will also facilitate future anthropological studies of software development [15].

Our objective in the current study is to describe a prototype of an infrastructure that can store the huge and growing amount of data in the entire FLOSS ecosystem and provide basic capabilities to efficiently extract and analyze that data at that scale. Our primary focus is on the types of analyses that require global reach across FLOSS projects. A good example is a software supply chain where software developers correspond to the nodes or producers, relationships among software projects or packages represent the "chain", and changes to the source code represent products or information (that flow along the chain) with corporate backers representing "financing."

Several formidable obstacles obstruct progress towards this vision. The traditional approaches for obtaining the repository of a project or a small ecosystem does not scale well and may require too many resources and too much effort for individual researchers or smaller research groups. Thus, the community needs a way to scale and share the data and analytic capabilities. The underlying data are also lacking in context necessary for meaningful analysis and are often incorrect or missing critical attributes [42]. Keeping such large datasets up-to-date poses another formidable challenge.

In a nutshell, our approach is a software analysis pipeline starting from discovery and retrieval of data, storage and updates, and transformations and data augmentation necessary for analytic tasks downstream. Our engineering principles are focused on using the simplest possible techniques and components for each specific task ranging from project discovery to fitting large-scale models. The result is a conceptual implementation loosely following the microservices architecture [45], where the design and performance of the loosely coupled components can be independently evaluated, each service can utilize a database that is optimal for its needs, and the most computationally-intensive components are extremely portable to ensure they run on any high-performance platform. Specifically, our prototype appears to capture almost the entirety of the publicly available source code in ver-

sion control systems and the latency of updates on the existing hardware platform does not exceed one calendar month, which is pretty fast given the size of the dataset and the complexity of the task (See Sections 3.1 and 3.2 for more details). Furthermore, we built a tool on top of the infrastructure and provided two types of API to enable wide data access for users.

We begin with an overview of related work in Section 2. The architecture of the prototype implementation of the infrastructure is discussed in Section 3. We facilitate wide access to the large data collection by developing a tool on top of our infrastructure, which is described in Section 4, along with an evaluation of query performance. We present a couple of applications in Section 5, demonstrating the tremendous value of this infrastructure to numerous software analytic tasks. We also provide a tutorial about how to use the WoC infrastructure, using an example on Java language trend analysis in Section 6. We discuss various ways of improving the existing infrastructure in Section 7, discuss a few existing limitations in Section 8, and conclude our paper in Section 9.

2 Related Work

While we are not aware of a complete census of FLOSS with an analysis engine, several large-scale software mining efforts exist and may be roughly subdivided into attempts at preservation, data sharing for research purposes, and construction of decision support tools.

Software development is a novel cultural activity that warrants preservation as a cultural heritage. The software source code, the only representation of software that contains human readable knowledge, needs to be preserved to avoid permanent loss of knowledge [15]. Software Heritage [15] is a distributed system involved in collecting and storing large amount of open source development data from various open source platforms and package hosts. It currently has software from GitHub, GitLab, Debian, PyPI, etc., and contains 73M projects, 1.7B commits, and 15.6B source files. The main drawback of this particular effort is the lack of focus on enabling applications to software analytics. The API provided allows for quick query of every historical particle in a software project and meets the preservation need, however, it does not grant the access to the full relationships (e.g., the set of projects containing a given commit) among these particles across entire collection of software. Quick access to these relationships is crucial in conducting software analytics such as identification of dependencies among artifacts and authors as well as code spread in the open source community.

One potential value of archiving software lies in the reuse of software artifacts. For example, Nexus [1] repository manager, allows developers to share software artifacts in a standard way and provides support for building and provisioning tools (e.g. Maven) to access necessary components such as libraries, frameworks and containers.

Commercial efforts, such as BlackDuck or FOSSID[1] have proprietary collections they use to determine if their clients have included open source software within their proprietary software code. It is generally not clear how complete

[1] blackducksoftware.com,fossid.com

these collections are nor if the companies involved might consider opening them for research purposes.

In addition to source code and binaries, large scale collection of other software development resources could be integrated with the source code data. For example, GHTorrent [25, 26, 28–30] attempts to record every event for each repository hosted on GitHub and provides multiple approaches (SQL request and MongoDB data dump) for data access. The primary limitation is that the collected metadata is specific to GitHub and it does not include the underlying source code as well. Therefore, obtaining dependencies encoded within the source code cannot be accomplished. FLOSSmole [32] collects open source metadata from various forges as a base for academic research but only focuses on software project metadata.

Another platform is Candoia [53–56] which provides software development data collections abstraction for building and sharing Mining Software Repository (MSR) applications. In particular, Candoia contains many tools for artifact extraction from different VCSs and bug databases and it also support projects written in different languages. On top of these artifacts, Candoia created its general data abstraction for researchers to implement ideas and build tools upon. This design increased portability and applicability for MSR tools by enabling application on software repositories across hosting platforms, VCSs and bug recording tools. The approach is focused on the design and benefits of creating a specialized software repository mining language. While it abstracts a number of repository acquisition tasks, it also makes it more difficult to handle operational data problems that tend to occur at much lower levels of abstraction and tend to be too idiosyncratic for generalized abstraction. The main drawbacks of Candoia are that it only supports limited programming language (JS and Java) based projects, and ecosystem-wide research might be difficult to implement since Candoia relies on users to provide software related data (e.g., targeted software repository URL) and eco-system wide compliance is generally low.

Other platforms are aimed at improving reproducibility by providing a repository of datasets for researchers to share their data. These include PROMISE Repository [51], Black Duck OpenHub [52], and SourcererDB [46]. PROMISE Repository is a collection of donated software engineering data. It was created to facilitate generations of repeatable and verifiable results as well as to provide an opportunity for researchers to extend their ideas to a variety of software systems. Black Duck OpenHub is a platform that discovers open source projects, tracks the development and provides the functionality of comparison between softwares. Currently, it is tracking 1.1M repositories, connecting 4.2M developers and indexing 0.4M projects. SourcererDB is an aggregated repository of 3K open source Java projects that are statically analyzed and cross-linked through code sharing and dependency. On top of providing datasets, it also provides a framework for users to create custom datasets using their projects.

Apart from providing datasets (repository) for potential users, platforms such as Moose [16], RepoGrams [49], Kenyon [6], Sourcerer [5], and Alitheia Core [27] are more focused on facilitating building and sharing MSR tools. Moose is a platform that eases reusing and combining data mining tools. RepoGrams is a tool for comparing and contrasting of source code repositories over a set of software metrics and assists researchers in filtering candidate software projects. Kenyon is a data platform for software evolution tools. It is restricted to supporting only software evolution analysis. Sourcerer is an infrastructure for large scale collec-

tion of open source code where both meta data and source code are stored in a relational database. It provides data through SQL query to researchers and tool builders but is only focused on Java projects. Alitheia Core is a platform with a highly extensible framework and various plug-ins for analyzing software on a large database of open source projects' source code, bug records, and mailing lists.

Furthermore, there were efforts to standardize software mining data description for enhanced reproducibility [33]. None of the listed platforms focus on both collection and analysis of the dependencies of the entirety of FLOSS source code version control data. Further, they contain either limited collections (e.g. only GitHub, no source code, have only donated data, or do not contain an analysis engine). For example, it is not possible to answer simple questions such as "In which projects has a file been used?", "What projects/codes depend on a specific module?", "What changes has a specific author made?" etc.

Some large companies have devoted substantial effort to develop software analysis platforms for the entire enterprise, aiming to improve the quality of software they build and to help the enterprise achieve its business goals by providing recommendations to software development organizations/teams, monitoring software development trends, and prioritizing research areas. For example, Avaya, a telecommunications company, built a platform [31], which collects software development related data from most of its software development teams and third parties and enabled systematic measurements and assessments of the state of software. CodeMine [12], is a software platform developed by Microsoft that collects a variety of source code related artifacts for each software repository inside Microsoft. It is designed to support developer decisions and provide data for empirical research. We hope that similar benefits can be realized with the WoC platform targeted to the entire FLOSS community.

Large scale software mining efforts also include domain specific languages. Robert Dyer et al. developed Boa [17–21, 47], both as a domain specific language and as an infrastructure, to ease open source-related research over large scale software repositories. The approach is focused on the design and benefits of an infrastructure and language combination. However, the lack of explicit tools to deal with operational data problems make it of limited use to achieve our aims. Their collection procedures -discovery, retrieval, storage, update, and completeness issues (for example, only certain languages are supported)- are not the primary focus of this effort. The tools to deal with operational data problems common in version control data are also lacking in Boa.

The system described in this paper is loosely modeled after a system described a decade ago [40, 41]. In comparison, at that time, git was just beginning to emerge as a popular version control system, but now it dominates the FLOSS project landscape. The number of software forges and individually hosted projects was much larger then in contrast to the consolidation of forges and the overwhelming dominance of GitHub. Furthermore, the scale of the FLOSS ecosystem is more than an order of magnitude larger now and it continues to experience very rapid growth. WoC could not, therefore, reproduce that design closely and, instead, is focused on preserving the original git objects and on creating a design that enables both efficient updating of this huge database and ways to cross-reference it so that the complete network of relationships among code and people is readily available.

3 Building the WoC Infrastructure

The process of mining individual git repositories is complex to begin with [8], but becomes even more difficult on a large scale [24]. Specifically, using operational data from software repositories requires resolution to three major problems [42]: the lack of context, missing attributes or observations, and incorrect data. This makes critical tasks such as debugging and testing complex and time consuming. To cope with these big data challenges we employed both vertical and horizontal prototyping [3, 9, 36, 48] before building the complete infrastructure.

In this section, we present a prototype WoC implementation. It has four stages: project discovery, data retrieval, correction, and reorganization as shown in Figure 1, which is typical of most big data systems, that use the layered data approach where the initial layers accumulate and process raw data and the later layers produce cleaned/augmented data.

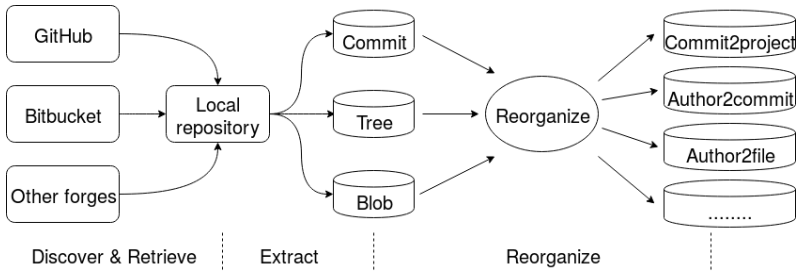

Fig. 1 Overarching data flow

3.1 Project Discovery

Millions of projects are developed publicly on popular collaborative platforms/-forges such as GitHub, Bitbucket, GitLab, and SourceForge. Some of the FLOSS projects can be identified from the registries maintained by various package managers (e.g. CRAN, NPM) and Linux distributions (e.g. Debian, Fedora). Most other project repositories, however, are hosted in personal or project-specific sites. A complete list of FLOSS repositories is, therefore, difficult to compile and maintain since new projects and forges are constantly being created and many older forges disappear continually. There is also a tendency for the FLOSS repositories to migrate to (or be mirrored on) several very large forges [38]. A number of older forges provide convenient approaches to migrate repositories to other viable forges before being shut down. This consolidation has alleviated some of the challenge of discovering all FLOSS projects [41], though the task remains nontrivial. We discuss several approaches to project discovery below. To package our project discovery procedure we have created a docker container[2] that has the necessary scripts.

Using Search API: Some APIs may be used to discover the complete collection of public code repositories within a forge. The APIs are specific to each forge and

[2] https://github.com/ssc-oscar/gather

come with different caveats. Most APIs tend to be rate limited (for user or IP address) and the retrieval can be sped up by pooling the IDs of multiple users.

Using Search Engine: Search engines (e.g. Google or Bing) can supplement the discovery of FLOSS project repositories on collaborative forges when the forge does not provide an API, or when the API is broken. The primary drawback is the incompleteness of the repositories discovered.

Keyword Search: Some forges provide keyword based search of public repositories, which is a complementary approach when a forge does not provide APIs for the enumeration of repositories and the results returned from search engines are lacking.

Using these and other opportunistic approaches help ensure that they complement each other in approximating the publicly available set of repositories though it does not guarantee the completeness. We expected that various ways of crowdsourcing the discovery (with incentives to share a project's git URL) would help increase the coverage in the future.

3.2 Project Retrieval

This data retrieval task can be done in parallel on a very large number of servers but requires a substantial amount of network bandwidth and storage. The simplest approach is to create a local copy of the remote repositories via git clone command. As of May 2019, we estimate over 73M unique repositories (excluding GitHub repositories marked as forks, repositories with no content, and private repositories). A single thread shell process on a typical server CPU (we used Intel E5-2670) with no limitations on network bandwidth clones between 20K and 50K repositories (the time varies dramatically with the size of a repository and the forge) in 24 hours. To clone 73M repositories in one week would, therefore, require between two and five hundred servers. However, we do not possess dedicated resources of that size and, therefore, optimize the retrieval by running multiple threads per server and retrieving a small subset of the repositories that have changed since the last retrieval. Specifically, we use five Data Transfer Nodes of a cluster computing platform[3].

3.3 Data Extraction

Code changes are organized into commits that typically change one or more source code files within the project. Once the repository is cloned as described above, we extract the Git objects[4] from each repository and store them in a single database.

3.3.1 ***Data Model***

Git [10] is a content-addressable filesystem containing four types of objects. The reference to these objects is a SHA1[5] [22] calculated based on the content of that

[3] No. node: 300, Bandwidth up to 56 Gb/s

[4] https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[5] https://en.wikipedia.org/wiki/SHA-1

object. A few typical Git objects are described below.
commit: A commit is a string including the SHA1's of commit parent(s) (if any), the folder (tree object), author ID and timestamp, committer ID and timestamp, and the commit message.
tree: A tree object is a list that contains SHA1's of files (blobs) and subfolders (other trees) contained in that folder with their associated mode, type, and name.
blob: A blob is the compressed version of the file content (the source code) of a file.
tag: A tag is the string (tag) used to associate readable names with specific versions of the repository.

Figure 2 illustrates the relationships among the Git objects described above. The snapshot at any entry point (commit) is constructed by following the arrows from left side to right side. Each commit points to a tree(folder), and each tree points to blobs(files in this folder) inside it and its subtrees(subfolders).

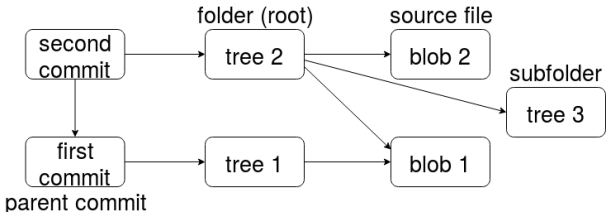

Fig. 2 Git objects

3.3.2 ***Object Extraction***

While a standard Git client allows extraction of raw git objects, it displays them for manual inspection. For bulk extraction of the git objects, first we list all objects within the git database, categorize them, and then create a bulk extractor based on a portable pure C implementation of *libgit2*[6]. We run listing and extraction using 16 threads on each of the 16-CPU node on a cluster[7]. The process takes approximately two hours for a single node to process 50K repositories. The extraction procedure represents a *microservice*.

3.4 Data Storage

The collection of public Git repositories as a whole replicates many of the same git objects hundreds of times [41]. Without removing this redundancy, the required storage for the entire collection exceeds 1.5PB, and it also makes analytics tasks virtually impossible without extremely powerful hardware. Many reasons for this redundancy exist, such as pull-based development, usage of identical tools or libraries, and copying of code.

To avoid redundancy of git object among repositories, we store all git objects into a single database. The database is organized into four parts corresponding to

[6] https://libgit2.org/
[7] CPU: E5-2670, No. node: 36, No. core: 16, Mem size: 256 GB

each type of git object. Each part is further separated into a cache and content. The cache is used to rapidly determine if the specific object is already stored in our database and is necessary for data extraction described above. Furthermore, the cache helps determine if a specific repository needs to be cloned. If the heads (the latest commits in each branch in .git/refs/heads) of a repository are already in our database, there is no need to clone the repository altogether.

Cache database is a key-valued database, with the twenty byte Git object SHA1 being the key and the packed integer (indexing the location of the object in the corresponding value database) being the value. The value database consists of an offset lookup table that provides the offset and the size of the compressed git object in a binary file (containing concatenated compressed git objects). While this storage allows for a fast sweep over the entire database, it is not optimal for random lookups needed, for example, when calculating diffs associated with each commit. For commits and trees, therefore, we also create a key value database where key is SHA1 of the git object and value is the compressed content of the said object. Cache performance is relatively fast: a single thread on Intel E5-2623 is capable of querying of 1M git objects in under 6 seconds, or over 170K git objects per second per thread. This can be multi-threaded and run on multiple hosts, thus reaching any desired speeds with expanded hardware.

Needless to say, with 24B objects occupying over 80TB we need to use parallel processing to do virtually anything. Thankfully, we can use SHA1 itself to split the database into pieces of similar size. We, therefore, split each of the database into 128 slices based on the first seven bits of Git object SHA1. This results in 128 key-offset cache databases for all four types of objects, 128 content databases as flat files for the four types of objects, and 128 key value databases for commits and trees: 128*(4+4+2) databases with each capable of being placed on a separate server to speed up parallel tasks. The individual databases containing content range from 20MB for tags up to over 0.5TB for blobs. The largest individual cache databases are over 2Gb for tree object SHA1s.

Databases are fragile and may get corrupted due to hardware malfunction, internet attack, pollution/loss by unrecoverable operation, etc. To enhance the robustness and reliability and to avoid permanent data loss, we maintain three copies of the databases: two copies on two separate running servers and one copy on a workstation that is not permanently connected to Internet. In the future, we will consider keeping a copy using a commercial cloud service.

Furthermore, due to the size of the data and complexity of the pipeline, some of the objects may have been missed or have been retrieved but are not identical to originals. Techniques to validate the integrity of the data at every stage of the process are necessary. We therefore, include numerous tests to ensure that only valid data gets propagated to the next stage.

In particular, the errors when listing and extracting objects are captured and the operation is repeated in case a problem occurs. The extracted objects are validated to ensure that they are not corrupt and also to ensure that they are not going to damage the database or the analytics layer. To validate correctness, the object is extracted per git specifications and recreated from scratch. The SHA1 signature is compared to ensure it matches that of the original object. A substantial number of historic objects have issues due to a bug in git that has since been fixed. Furthermore, a much smaller number of objects also had issues that we assume are either caused by problematic implementations of git or problems in operation

(zero-size objects that may be occasionally created when git runs out of disk space during a transaction).

Despite the scrubbing and validation efforts, some of the data may still be problematic or missing, therefore a continuous process of checking the database for missing or incorrect data is needed. We plan to add a missing object recovery service that identifies missing commits, blobs, and trees, and retrieves and stores them (in case they are still available online).

3.5 Update

The process of cloning all GitHub repositories takes an increasing amount of time with the growth in size of existing repositories and the emergence of new ones, given fixed hardware. Currently, to clone all git repositories (over 90M including forks), we estimate the total time to require six hundred single-thread servers running for a week and the result would occupy over 1.5PB of disk space. Fortunately, git objects are immutable and we can leverage that to simplify and speed up the updates. More generally, to get acceptable update times, we use a combination of two approaches:

- Identify new repositories, clone and extract Git objects
- Identify updated repository and retrieve only newly added Git objects

The work flow is illustrated in Fig. 3.

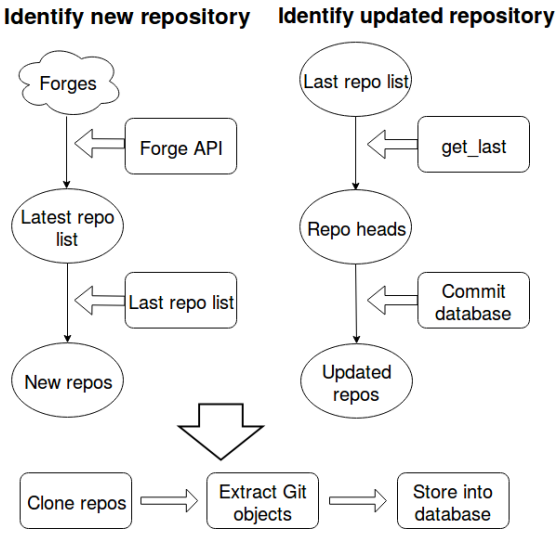

Fig. 3 Update workflow

In fact, only approximately three million new projects were created and an additional two million updated during Dec, 2018.

3.5.1 ***Procedures for new repositories***

Forge-specific APIs are utilized to obtain the complete list of public repositories as described above. A comparison with prior extract yields new repositories. The list may include renamed repositories and forks. We can exclude forks for GitHub, since it is an attribute returned by GitHub API. Other forges contain fewer repositories, so the forks are not large enough to be a concern.

3.5.2 ***Procedures for updated repositories***

First we need to identify updated repositories from the complete list of repositories. Since we are not sure how GitHub determines the latest update time for a repository, we use a forge-agnostic way of identifying updated repositories. We modified the *libgit2* library so that we can directly obtain the latest commit of each branch in a Git repository for an arbitrary Git repository URL, without the need to clone the repository. If any of the heads contain a commit that is not already in our database, the repository must have had updates and needs to be obtained.

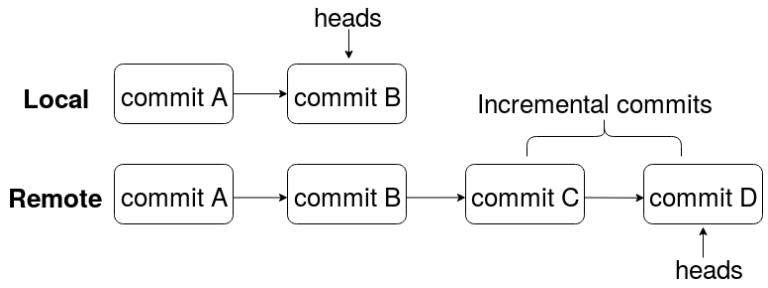

Fig. 4 Incremental commits

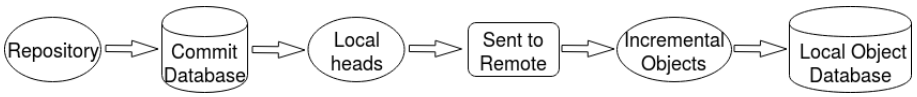

Fig. 5 Future workflow

We are working on a strategy to reduce the amount of bandwidth needed to do the updates. Instead of cloning an updated repository, we'd like to retrieve only incremental Git objects (see Fig. 4) that are generated during the time gap between two consecutive updates. This can be easily done via git fetch for a git repository, but since we do not keep the original git repository and it is time consuming to pre-populate it with git objects, we plan to customize git fetch protocol by inserting additional logic in order to use our database backend that comprises git objects from all repositories. The procedure consists of two steps:

1. Customize git fetch protocol[8] to work without git's native database.
2. Keep track of the heads for each project that we have in our database so that we can identify latest commits to the modified git fetch.

For the second step, the database backend will use the project name as input and provide the list of heads for the project. These heads are then sent to the remote so that the set of latest commits (and related trees/blobs) will be calculated out and transferred back as illustrated in Figure 5. By following this strategy, we could drastically speed-up mining incremental Git objects from repositories in each update.

3.6 Data Reorganization

Objects in Git are organized in a way for fast reconstruction of a repository at each commit/revision. In fact even the seemingly simple operation of identifying what files changed in a commit is computationally intensive. Furthermore, there is no consideration for the projects, files, or authors as first-class objects. This limits the usability of the git object store for research and suggests the need for an alternative data design. Since our objective is to obtain relationships among projects, developers, and files, we have created an alternative database that allows both a rapid lookup of these associations and sweeps through the entire database that make calculations based on such relationships.

3.6.1 ***Analytic Database***

The scale of the desired database limits our choices. For example, a graph database [9] like neo4j would be extremely useful for storing and querying relationships, including transitive relationships. However, it is not capable (at least on the hardware that we have access to) of handling hundred's of billions of relationships that exist within the entire FLOSS. In addition to neo4j, we have experimented with more traditional database choices. We evaluated common relational databases MySQL and PostgreSQL and key value databases or NoSQL [35] databases MongoDB, Redis, and Cassandra. SQL like all centralized databases [2] has limitations handling petabyte datasets [37, 50]. We, therefore, focus on NoSQL databases [43] that are designed for large scale data storage and for massively parallel data processing across a large number of commodity servers [43].

For the specific needs of the cache database and for key value stores for the analytics maps we use a C database library called TokyoCabinet (similar to berkeley_db) using a hash-indexed as described above, to provide approximately ten times faster read query performance than a variety of common key value databases such as MongoDB or Cassandra. Much faster speed and extreme portability lead us to use it instead of more full-featured NoSQL databases.

[8] git fetch downloads only new objects from the remote repository

[9] a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data

3.6.2 ***Maps***

Apart for the general requirement to be able to represent global relationships among code, people, and projects, we also consider the basic patterns of data access for several specific research tasks as use cases in order to design a database suitable for accomplishing research tasks within a reasonable time frame. The specific use cases are:

1. Software ecosystem research would need the entire set of repositories belonging to a specific FLOSS sub-ecosystem, e.g., the set of all repositories that use Python language.
2. Developer behavior research would need to identify all projects that a specific developer worked on, the files they authored, and software technologies they used.
3. Code reuse research would need to identify all projects where a specific piece of code occurs and determine how it got there.

To support the first task, a mapping from file names to project names would be necessary. The second task would require author to project, file, and to content of the versions of the file authored by that developer (in order to access the source code and identify what components or libraries were employed). The last task would require a map between blobs (that contain snippets of code) and projects. It would also require a map between blobs and commits in order to identify the time when the specific piece of code was introduced.

We have identified a number of objects and attributes of interest here: projects, commits, blobs, authors, files, and time. The complete set of possible direct maps for an arbitrary pair is 30. Since author and time are properties of the commit and are not properties of projects, blobs, or files, it makes sense to place commit at the center of this network database. The author-to-file map can then be constructed as a composition of author-to-commit and commit-to-file maps; and author-to-project map can be constructed via author-to-commit and commit-to-project maps. We also need to associate file names with the corresponding blobs since a single commit may create multiple files. Out of the 12 maps[10], only 10 need to be instantiated because commit-to-author and commit-to-time maps are embedded as the properties of the commit object.

In addition to having the commit at the center, for certain tasks we also needed to have a blob-to-file map as well. For example, we want to identify module use in Python language files. First, we need to identify relevant files via suitable extension (e.g., .py), then we can determine all the associated commits via file to commit map. These commits, however, may involve other files and if we use commit to blob map to identify associated blobs, we would get blobs not just for python, but also for all files that were modified in commits that touched at least one python file. The file-to-blob map allows us to reduce the number of blobs that need to be analyzed dramatically.

In addition to these basic maps we create additional maps, such as the author ID to author ID map for IDs that have been established to belong to the same person (see Section 5.2), and project to project maps to adjust for the influence

[10] bidirectional maps between the commit and five objects/attributes and between file and blob

of forking. Project-to-project maps are based on the transitive closure of the links induced between two projects by a shared commit. Explicit forks that can be obtained as a GitHub project property do not generalize to other forges and, even on GitHub, represent only a fraction of all repositories that have been cloned from each other and then developed independently. Project-to-project map also handles instances where repositories exist on multiple forges or when they are renamed.

As with the original data we utilize multiple databases and use compressed files for sweep operations and TokyoCabinet for random lookup. We separate maps into 32 instead of 128 databases we use for the raw objects since maps tend to be much smaller in size than, for example, blobs. For commits and blobs we use the first character of SHA1 for database identification. For authors, files, and projects, we use the first byte of FNV-1a Hash [11]. Both approaches yield quite uniform distribution over bins.

As noted above, the maps from commit to metadata are not difficult to achieve because most of the metadata is part of the content of a commit object. However, git blobs introduced or removed by a commit are not directly related to the commit and need to be calculated by recursively traversing trees of the commit and its parent(s). A Git commit represents the repository at the-state-of-world and contains all the trees (folders) and blobs (files). To calculate the difference between a commit and its parent commit, i.e., the new blobs, we start individually from the root tree that is in the commit object, traverse over each subtree and extract each blob. By comparing two sets of blobs of each commit, we obtain the new blobs for the child commit. This step requires substantial computational resources, but the map from the commit to the blobs authored in a commit is used in numerous research scenarios and, therefore, is necessary. On average, it takes approximately one minute to obtain changed files and blobs for 10K commits in a single thread. With 1.5B commits, the overall time for a single thread would take 104 days, but it needs to be done only on approximately 20-40M new commits generated each month.

4 Architecture for research workflows

To make WoC more easily usable in a wide variety of research scenarios, we have designed an architecture to help simplify, support, and evaluate the implementation of research tasks. This section describes that architecture, along with critical performance benchmarks to inform the users on the computational tasks for alternative implementations.

4.1 Architecture

The research workflow architecture is illustrated in Figure 6. The figure shows the application layer, built on top of the three lower layers:
Application Layer: This layer is where the research tasks are implemented by use of WoC. We provide a library of applications to illustrates various types of research analyses that can be implemented using WoC.

[11] http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-1a

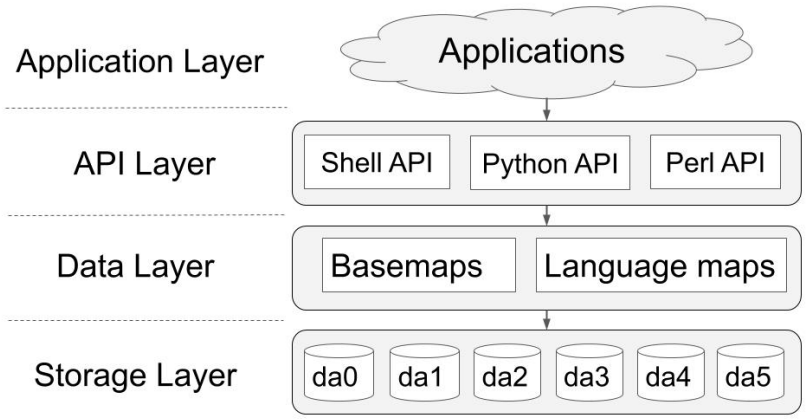

Fig. 6 Architecture of the Tool

API Layer: The applications may use Shell or Python API, or may reuse or modify Perl files (used to support Shell API) to access and process the WoC data. A more detailed description of the Shell and Python APIs can be found in Section 4.2.

Data Layer:

As described above, to be able to identify the relationships rapidly we constructed several types of relationships (or basemaps) that cross-reference the git objects and other properties. In particular, we treat *project, commit, blob, author, file name* as the first class objects and map them to their properties (e.g., time, parent commit, head commit, child commit, etc.). In addition to these basemaps, we also construct technical dependencies that are derived from importing external dependencies for several languages (Language Maps). These dependencies are calculated based on each version of a file. The data is described in more detail in Section 4.3.

Storage Layer All the data are hosted on six servers, which are connected to each other through NFS (network file system). Users can login to any of the servers (da0 to da5) and and run their applications on multiple servers.

4.2 API

We support three primary APIs for WoC users to access the dataset: Shell, Python, and Perl. Presently, running an application requires being logged in to one of the hosting servers.

4.2.1 Shell API

For the lowest level access we provide Shell API that is modeled after core philosophy of Unix[12]: have a set of specialized commands that are connected in a

[12] https://en.wikipedia.org/wiki/Unix_philosophy

workflow through their standard input and output and via creation of files, E.g., according to Doug McIlroy: “Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features”. The entire application workflow can be built using exclusively shell and standard Unix utilities such as ‘join’, ‘sort’, ‘cut’, ‘uniq’, ‘sed’, and ‘wc’ with added specialized commands to extract data from key-value databases. The key information for this API is the knowledge of how to use shell and standard Unix command and the description of the databases. To enable this approach we also provide all databases as key-sorted (and compressed) text files that can be used with ‘grep’, ‘join’, or ‘sort’ to produce any desired queries. We also add a random lookup operation `getValue mapname` to access values of a key object in the provided `mapname`. In addition, we add the command `showCnt type` to access the content of each git object given in the standard input where `type` is one of `tag, tree, commit, blob`. A few examples are listed below:

– Checking the content of a Git object given a SHA:

```
# (on da3) e.g., show a commit SHA's content:
echo e4af89166a17785c1d741b8b1d5775f3223f510f | showCnt commit
# Output Formatting:
# Commit SHA;Tree SHA;Parent Commit SHA;Author;Committer;Author Time;
    Commit Time
e4af89166a17785c1d741b8b1d5775f3223f510f;
    f1b66dcca490b5c4455af319bc961a34f69c72c2;
    c19ff598808b181f1ab2383ff0214520cb3ec659;Audris Mockus <audris@utk.
    edu>;Audris Mockus <audris@utk.edu>;1410029988 -0400;1410029988 -0400
```

– Given an object, check its related objects:

```
# (on da3) e.g., show the names of the projects associated with a given
    commit SHA:
# ‘‘getValue" command takes a database name as an argument and keys
    presented as standard input and produces key-value pairs as output.
echo e4af89166a17785c1d741b8b1d5775f3223f510f | getValue /da0_data/
    basemaps/c2pFullP
# Output Formatting: Commit SHA;ProjectNames
e4af89166a17785c1d741b8b1d5775f3223f510f;W4D3_news;chumekaboom_news;
    fdac15_news;fdac_syllabus;igorwiese_syllabus;jaredmichaelsmith_news;
    jking018_news;milanjpatel_news;rroper1_news;tapjdey_news;
    taurytang_syllabus;tennisjohn21_news
```

4.2.2 Python API

At the top level of abstraction, we provide Python API via package **oscar**[13] that implements the key notions of author, file, project, commit, blob, and tree as the corresponding classes. The enumeration below describes Python classes that were created by wrapping up data objects 1. Each of the classes has a couple of methods attached to access corresponding properties. For the methods that contain slash(/), the method before slash returns actual data in string, while the one after return a generator of corresponding python instances. E.g. `Author.commit_shas()` returns

[13] https://github.com/ssc-oscar/oscar.py

a list of the SHAs of commits that the person authored; `Author.commits()` returns a generator of Commit objects built from those SHAs.

1. Author('...') - initialized with a combination of name and email, e.g. "Albert Krawczyk <pro-logic@optusnet.com.au>"
   - .commit_shas/commits - all commits by this author
   - .project_names - all projects this author has committed to
2. Blob('...') - initialized with SHA of blob
   - .commit_shas/commits - commits creating or modifying (but not removing) this blob
3. Commit('...') - initialized with SHA of commit
   - .blob_shas/blobs - all blobs in the commit
   - .child_shas/children - the commit that follows this commit
   - .changed_file_names/files_changed
   - .parent_shas/parents - the commit that this commit follows
   - .project_names/projects - projects this commit appears in
4. Commit_info('...') - initialized like Commit()
   - .head
   - .time_author - the commit time and its author
5. File('...') - initialized with a path, starting from a commit root tree. This represents a filename, regardless of content or repository; e.g. File(".gitignore") represents all .gitignore files in all repositories.
   - .commit_shas/commits - All commits that include a file with this name
6. Project('...') - initialized with project name/URI
   - .author_names - all author names in this project
   - .commit_shas/commits - all commits in this project

4.2.3 Perl APIs

While the Python API provides high level of abstraction, it is not very computationally efficient. In order to provide an intermediate level of efficiency between that of Python and Shell APIs, we also provide a way to implement applications or their components in Perl language. For example, the shell commands `getValue` and `showCnt` are both implemented in Perl. The Perl API instead of creating classes of objects as in Python, it handles the maps directly. To support writing WoC workflows in Perl we provide a variety of utility functions in package 'WoC.pm.' We also have, over the course of evolving WoC, created a number of applications that can be used as templates and modified by the users for their needs. For example, we can parse the content of the commit to obtain its tree, parent commit, author, and time:

```
use WoC;
my ($tree, $parent, $authName, $authEmail) = ("","","","");
my ($pre, @rest) = split(/\n\n/, $code, -1);
for my $l (split(/\n/, $pre, -1)){
  $tree = $1 if ($l =~ m/^tree (.*)$/);
  $parent .= ":$1" if ($l =~ m/^parent (.*)$/);
  ($authName, $authEmail) = gitSignatureParse($1) if ($l =~ m/^author (.*)$/);
}
($auth, $ta) = ($1, $2) if ($auth =~m/^(.*)\s(-?[0-9]+\s+[\+\-]*\d+)$/);
$parent =~ s/^:// if defined $parent;
```

We also have examples on how to parse, for example, a python source code to obtain the dependencies defined by the import statements (a segment is shown below):

```
for my $l (split(/\n/, $code, -1)){
 if ($l =~ m/^\s*import\s+(.*)/) {
  my $rest = $1;
  $rest =~ s/\s+as\s+.*//;
  my @mds = $rest =~ m/(\w[\w.]*[\,\s]*)*/;
  for my $m (@mds) { $matches{$m}++ if defined $m};
 }
 if ($l =~ m/^\s*from\s+(\w[\w.]*)\s+import\s+(\w*)/) {
  if ($2 ne ""){ $matches{"$1.$2"} = 1; }
  else{ $matches{$1} = 1; }
 }
}
```

For more detail please refer to the tutorial page of our repository[14].

4.3 Description of the WoC Data

We use abbreviated object names for WoC data and basemaps as shown in Table 1. As noted above, types of basemaps are created to represent relationships among these objects, which are illustrated in Figure 7. Notice that some maps are missing in Figure 7, because initially we built maps with commit being the core, and other maps were built as certain research tasks the users were attempting to do would benefit from them. The basemaps are stored in TokyoCabinet databases for random queries and key-sorted compressed text files of these basemaps are also created to enable quick sweeps over the whole dataset and to enable the shell API.

In addition to the basemaps, programming language based maps are created to enable language oriented analytic and applications. These contain mappings that list repositories, and the modules they depended on, at a given UNIX timestamp under a specific commit. The format of each entry in these maps are like the following, where `module1;module2;...` represent the modules that repository depended on at the time of that commit:

`commit;repository_name;timestamp;author;blob;module1;module2;...`

So far, 12 maps are ready including C, C#, Java, JavaScript, Python, R, Rust, Go, Swift, Scala, and Fortran. It is likely that more language maps will be added in the future.

Table 1 Objects Terminology

Object Abbreviation	Annotation	Entity Type
a	author	string
b	blob	SHA
c	commit	SHA
f	file name	string
p	project	string

[14] https://bitbucket.org/swsc/lookup/src/master/

[15] 'File' in this figure refers to 'File name'

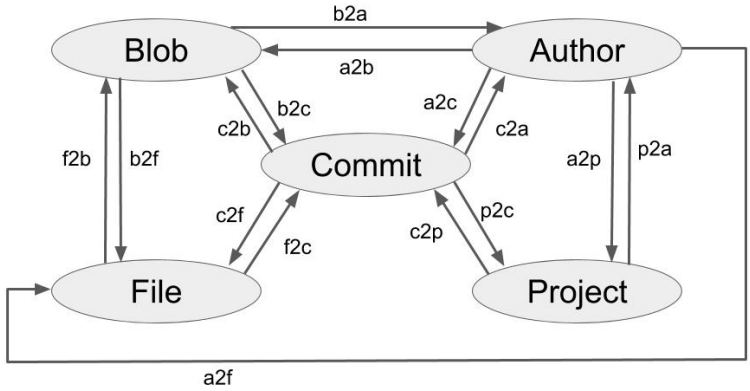

Fig. 7 Maps between primary objects[15](Basemaps)

4.4 Performance Benchmark

The anticipated workflow of a specific research task involves a set of queries that proceed from selecting an initial sample of interest such as a set of files related to a specific language, a set of projects or authors with certain properties or other collection. This is typically followed up by one or more network operation such as identifying blobs associated with the selected files, projects associated with the initial set of developers and so on. These tasks can typically be implemented in numerous ways, each leading to different computer memory, disk IO, and computational overheads. To help users decide upon the the best way to proceed and, more generally, to gauge the time needed for their desired workflow, we set up experiments to test our WoC infrastructure performance on such queries. Our existing basemaps should meet users' need in most cases by a query of a single map (e.g. author to commit). However, in cases where a map is not ready (e.g. file to project in Figure 7), users might need to combine/join two or more maps to achieve their goal. We, therefore, tested the performance of both single map queries and combined map queries, and present the results below.

Since the file[16] to project map is not pre-computed, we can start from the file to commit map to test single map query performance and then join the results with the commit to project map to test the combined map query. We randomly selected 100, 1K, 10K, 100K, and 1M file names from our dataset, and used the Python and Shell APIs without any other task being run on the server to find the corresponding commits in which the files were modified and the projects those commits belong to. We collected the time it took to run each test and show them in Figure 8 for the single map queries, and Figure 9 for the combined map queries.

From Figures 8 and 9, we see that the run time increases linearly as the task size increased, highlighting the scalability of the WoC infrastructure. We also found Shell API to be three to four times faster than Python API (Figure 8 and right part of Figure 9), for the same query. One hypothesis is the interpreted nature of

[16] By file, we refer to the file name (including folder path) in the rest of our paper.

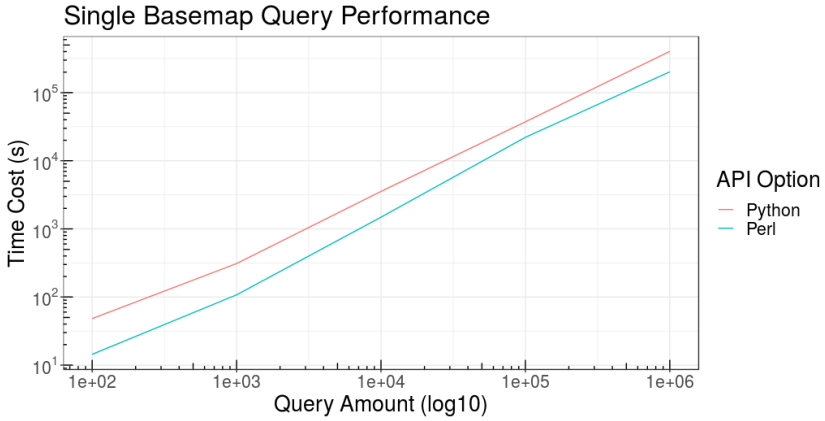

Fig. 8 Single Map Query Performance

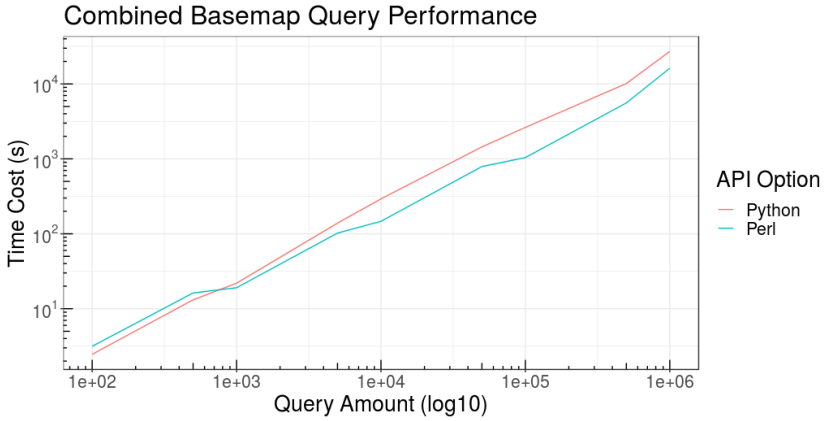

Fig. 9 Combined Maps Query Performance

Python. Specifically, the data access parts of Shell API are implemented in Perl. While Perl is an interpreted language just as Python, many of the functions are implemented natively in C language, while in Python more performance-critical code is interpreted.

It is worth noting that the x-axis on Figure 9 represents the number of queries, which in this scenario is the sum of the number of file to commit queries and the number of commit to project queries.

We tested the performance of the tool for 100 to 1M queries. If a research workflow involves the initial sample of objects for a very large part of the WoC database, we recommend leveraging the database in the form of compressed text for key-value basemaps instead, because as the number of random access queries increases, it exceeds the time it takes to sweep the entire database using efficient shell commands such as grep. In fact, a single sweep of the file to commit compressed data only takes 38 hours while 1M queries of the file to commit basemap takes 56 hours using Shell API.

5 Applications

To evaluate if the experimental platform is capable of supporting research tasks conducted as a part of actual investigations and to provide a set of vignettes for other researchers, we conducted two types of studies. First, we implemented several basic and involved research tasks that require the entirety of FLOSS data as a part of the investigation. Furthermore, we also recruited three researchers external to our group to either conduct investigations of their own utilizing WoC or to provide us with their research problems that can only be solved by using WoC. Below we report both the experiences and results from these experiments.

5.1 Use of programming languages

Language popularity may influence developers decisions as it may affect the market for their software as well as their job prospects. For example: What language-specific API should developer provide for their component? What language should the developer use to implement their product?

To plot, for example, Java language use trend we use WoC to identify all files with .java extension. Then, via file-to-commit map, obtain the complete set of commits authoring these files. Commit dates are used to plot the time trends of language-specific commits, authors (property of a commit), projects (via commit to project map) and, if desired, lines of code changed. The entire process is highly parallelizable since each map is separated into 32 instances and can be processed independently. The entire calculation, while not interactive on our hardware, can be performed in tens of minutes. For illustration, we show the ratio of the number of commits over the number of developers (a measure of productivity) each year in Fig. 10. The ratio decreases for most languages, perhaps because as a language becomes more popular, the less experienced contributors join and lower the average productivity.

5.2 Correcting Developer Identity Errors

One of the particularly troubling data quality issues with version control systems is developer name disambiguation. Often, names and emails of developers are missing, incomplete, misspelled or duplicate [7,23]. Performance of any disambiguation algorithm depends on the distribution of the actual misspellings in the underlying data. In order to design and evaluate corrective algorithms, it is important to study a large collection of actual data and unearth patterns of irregularities that compromise data quality. WoC contains a nearly complete collection of git author ids (name and email combinations) and is, thus, more representative of such irregularities than any specific project.

To obtain author IDs we use author-to-commit map containing roughly 30 million distinct author IDs. Common error patterns include organizational ids and emails (Mozilla, Linux, Google etc), names of tools and projects (OpenStack, Jenkins, Travis CI), roles such as (admin, guest, root etc.) and words that preserve anonymity (student, nobody, anonymous etc) as a part of their credentials. We also found a large number developer IDs to be misspelled.

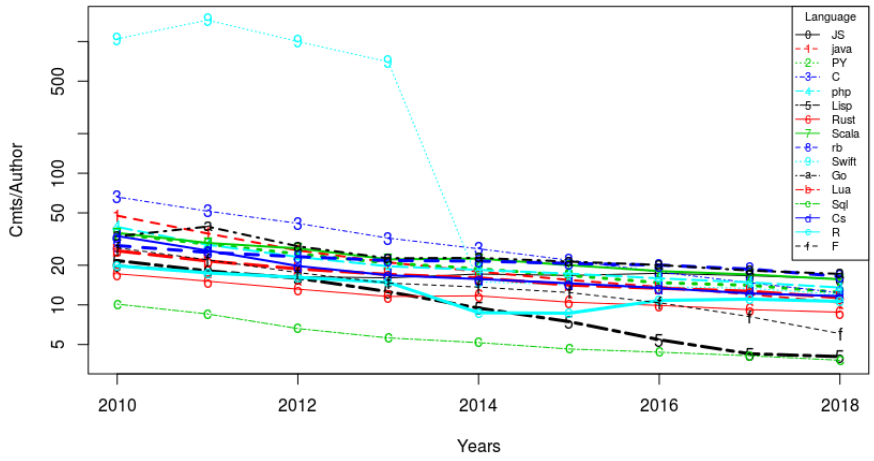

Fig. 10 Productivity by Language

Traditional identity correction approaches rely on the misspelling patterns of author ID (the full name and email) [7, 57, 58]. With WoC data, we can enhance the traditional string matching with behavioural comparison, by creating similarity measures between author IDs using files modified by developers, time patterns of commits, and writing styles in commit messages. For illustration — two author IDs that modify a similar set of files may suggest that these IDs belong to the same developer. To implement file-based similarity, we used author to commit and commit to file maps to obtain the set of files modified by a single author ID. Then file-to-commit and commit-to-author maps were used to calculate similarity using weighted Jaccard measure. Commit message text was used to fit a Doc2Vec [34] model to associate each author ID with their writing style. Traditional and behavioural similarities were used to train highly accurate machine-learning model [4].

This experiment demonstrates the utility of WoC data for designing tools to solve common and vexing data quality problems when constructing developer networks. It is also an example of how WoC can be enhanced by incorporating such techniques and providing corrected data to researchers.

5.3 Cross-ecosystem comparison studies

A second research group used the database to gather comparative statistics about different software ecosystems. The purpose was to supplement other comparative data about those ecosystems in support of a study of how ecosystem tools and practices influence development behavior. The ecosystem study involved a survey, interviews, and data mining over 18 ecosystems whose repositories listed more than 1.2M packages. Some questions about ecosystem practices could be mined from

metadata available elsewhere; for example detailed information about dependencies, release frequency, and version numbering practices can be easily extracted from libraries.io[17]. However deeper questions about project content would have been out of reach without WoC; independently building the mechanism to collect all of these projects, building a database of blobs, files, projects, and authors, and comparing them using various metrics would have been too much work for too little gain without the availability of this research platform.

5.3.1 ***File cloning across ecosystems***

One such statistic is rate of file cloning. It was theorized that in ecosystems with more flexible support for dependencies and a tolerance for the risk of breaking changes, developers would be more likely to use dependency management tools to make use of functionality from other projects, rather than copying those files in directly; hence in such ecosystems we should find relatively few commits adding a blob that already exists in any other project available through the ecosystem's dependency management system.

Using WoC, this analysis was straightforwardly accomplished by joining blob-to-commit and commit-to-project mappings, filtering for blobs that appeared in multiple projects, and identifying pairs with one commit in the time frame, and at least one older commit. Such blobs were discarded when the files were very small (since these often turned out to be empty or trivial files duplicated by chance or by tools) resulting in a set of duplicates that, on visual inspection of a sample, did appear to represent genuine examples of reuse-by-cloning.

Contrary to our expectations, the ecosystem with the most propensity for cloning was the one with the modern and flexible dependency system: npm. Despite the strengths of npm's dependency management system, there is a strong tradition of copying dependencies like jQuery into projects rather than letting npm retrieve them. Figure 11 summarizes the findings for a selection of ecosystems.

5.3.2 ***Developer migration across ecosystems***

Another metric of interest was developer overlap between ecosystems. Our ecosystem comparison had included a survey of values and practices in the 18 ecosystems of interest, and we hypothesized that ecosystems might be similar if many developers were actually working in both ecosystems, or had migrated from one to the other.

This question was answered by joining author-to-commit and commit-to-project data for the 1.2M projects in our study, and relying on the identity matching technique described in Sec 5.2.

Over all pairs of ecosystems, we found a sizable correlation between similarity of average responses on ecosystem *practice* questions (things like frequency of updating, collaboration with other projects, means of finding out about breaking changes), and overlap in committers to those ecosystems (Spearman $\rho = 0.341, p < .00001, n = 16$ ecosystems). Interestingly, perceived *values* of the ecosystem (such as a preference for stability, innovation, or replicability) do *not* seem to align with developer overlap ($\rho = -0.05, p = 0.44$). While more research is needed, we

[17] https://libraries.io/

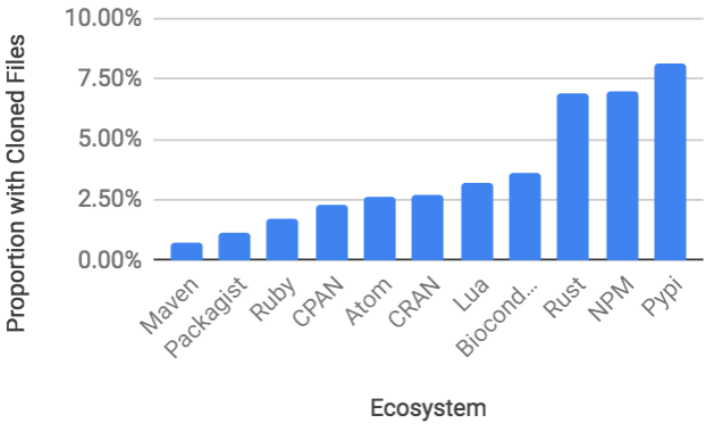

Fig. 11 Proportion of repository packages that added at least one cloned code file over 1kb in 2016.

hypothesize that developers may carry practices over from other languages and platforms they have used in the past, in a sometimes cargo-cult-like way, despite recognizing that a new ecosystem is designed to accomplish different ends.

In our very large-scale, wide-ranging study, these questions of developer migration and cloning were of great interest, but would likely have been too expensive to pursue alongside other lower-hanging fruit, absent WoC's prepared set of precomputed maps between files, blobs, authors, projects, and timestamps. The dataset with its analytical maps was not designed with these particular ecosystem comparison in mind, but its design happens to make such ecosystem questions relatively easy to answer.

5.4 Python ecosystem analysis

An external researcher wanted to use WoC to investigate open source sustainability by identifying source code repositories for packages in PyPI ecosystem and to measure package usage directly. While over 90% of npm packages provide repository URLs, less than 65% of Python Package Index (PyPI) packages do.

The researcher obtained all packages from PyPi and calculated blob SHA1s for *setup.py* file of the first PyPI releases of each package. We filter out resulting 101584 blobs to exclude empty or uninformative blobs (blobs that appear in more than one commit using blob-to-commit map). The 54218 informative blobs are then mapped to 54062 unique commits and commits to 51924 unique projects (adjusted for forking as described in Section 3.6). Repositories were recovered for 96% of the 54218 original packages in approximately 20 minutes of computation. To ensure that these repositories are, in fact, used to version control corresponding packages, they can be matched via additional blobs for *setup.py* and other files obtained from PyPi for that package.

Another problem being solved by this researcher was identifying which of the seemingly abandoned projects may be "feature complete," i.e. already have the

intended scope and do not require further maintenance [11]. Feature complete projects should be widely used in contrast to abandoned projects. Proxies of project usage, e.g., GitHub stars or forks can be used to identify such projects [11]. WoC, however, lets us measure the extent of use directly. As described in Section 5.1, all commits modifying Python files are identified (file-to-commit map) and the resulting commits are mapped to projects (commit-to-project map). Blobs associated with these commits (commit-to-blob map) are then used to extract imports from these files. The entire procedure could be completed in approximately four hours using the parallelism of the analytic maps (32 databases) and blob content maps (128 databases).

The reported usage was compared to project development activity, i.e the total number of adoptions versus the total number of commits. In some cases, usage was not accurately reflected in the number of commits. Common examples are packages providing console scripts and CMS-like projects. In the former case, packages are not reused in programmatic code and thus don't get into statistics. In the latter case, website builders often do not publish their code and thus such usage remains unobserved. Therefore, while the number of public reuses provides some extra information about package use, it should be adjusted for package type.

5.5 Repository filtering tool

Millions of repositories on GitHub and other forges also include projects that are completely unrelated to software development. GitHub is widely used for education and other tasks such as backing up text files, images, or other data. Researchers investigating education may need to focus on tutorials, while other researchers may need a sample of actual software development projects. Furthermore, a way to select specific subsets of software development projects in order to conduct, for example, "natural experiments" would also be highly beneficial. WoC can support such project segmentation tasks in a variety of ways. An external education researcher wanted to understand the impact of self-administered programming tutorials. To do that, WoC was used to identify developers who participated in tutorials by searching the set of projects in WoC via keywords related to education: "assignment", "course", "homework", "class", "lesson", "tutorial", "syllabus", "mooc", "udacity". The search yields over 1M projects. While it is only a small fraction of all projects in WoC but it represents a large sample in absolute terms. Further filtering was needed to find developers who also worked on actual software projects to measure the impact of self-administered tutorials. The project-to-commit map identified 605K users of tutorials and, when these users were mapped to all projects they participated in, we determine that only half of them contribute to non-tutorial projects. These 300K individuals are potential subjects of tutorial-impact study. Further information (such as their commit activity and project participation) can be obtained from WoC and combined other data, be used in this research. WoC can be extended with other approaches to segment projects[18]. For example, identification of projects with sound software engineering practices [44] relies on a combination of factors easily obtainable in WoC, such as history, license, and unit tests.

[18] Section 5.2 shows how WoC can also be used to improve them

5.6 Other Applications

A number of research publications have utilized the WoC database, including:

- The relationship between dependencies of NPM packages, collected using the WoC infrastructure, and their popularity was discussed in [14].
- The effort contribution and demand patterns of the contributors to the NPM ecosystem was discussed in [13].
- The investigation of what attributes drive the adoption of a software technology was discussed in [39].

6 Archetypical Usage of WoC

To increase the utility of this project to a wider research community, we would like to prioritize easy access to the World of Code to other interested parties. In this section, we provide a brief introduction and an overview of the World of Code and how to use it. Moreover, there are some resources already in place that were designed to assist in this process, which can be found in a public repository[19].

After describing WoC and its applications, in this section we demonstrate how to actually use WoC to implement a specific analysis. A couple of approaches presented here leverage the WoC tool to implement the Java language trend analysis, as described in Section 5.1.

1. Identify Java files based on '.java' extension, collect commits that changed these files, and deduplicate the commits. Now we have all commits where one or more java files were created/modified. The source code of the custom `lsort` command is presented in Appendix B.

```
#start from basemap dump(``file to commit" dump, P represents version),
for i in {0..31}; do zcat /da0_data/basemaps/gz/f2cFullP.$i.s | awk -F ";"
      "/.java;/{print $2 }" done | ~audris/bin/lsort 10G -u | gzip >
     JavaCommits.gz
```

2. For each commit in commit collection, we can use either Python or Perl API to find related author and commit time, and then calculate the number of authors and commits by year – the trend

```
# Using Python
import gzip
from datetime import datetime
from collections import defaultdict

year2commit_count = {}
year2commit_count = defaultdict(lambda: 0, year2commit_count)
year2author_count = defaultdict(set)
java_commits = gzip.open("JavaCommits.gz", "r")
for commit in java_commits:
    time, author = Commit_info(commit).time_author
    year = datetime.fromtimestamp(int(time)).year
    year2commit_count[year] += 1
    year2author_count[year].add(author)
```

[19] https://github.com/ssc-oscar/lookup

```
    print(year2commit_count)
    for year, authors in year2author_count.items():
        print("Year: "+ str(year) + "# of authors: " + str(len(authors)))
```

```
    # Using Perl
    # we can run /da3_data/lookup/showCmt.perl on every commit and extract
        author and time info from there
    # A simpler way is to utilize basemap c2taFullP.{0..31}.tch (i.e., the
        basemap from commit to author and commit time) by calling Cmt2ATShow.
        perl (see source code in Appendix)
    zcat JavaCommits.gz | perl Cmt2ATShow.perl | gzip > JavaYearAuthor.gz
    # count records for each year, we get the number of commits by year. E.g.,
         for year 2014:
    zcat JavaYearAuthor.gz | grep "^2014;" | wc -l
    # after deduplication, count records for each year and we get the number
        of authors by year. E.g., for year 2014:
    zcat JavaYearAuthor.gz | sort -u | grep "^2014;" | wc -l
```

In fact, directly using language maps is more efficient when implementing this analysis, since language specific information have already been extracted from base maps and stored as language maps for use.

```
# Alternatively, we use language map: c2bPtaPkgPjava, which consists of commit,
    blob, project name, time, author, etc.
zcat c2bPtaPkgPjava.{0..31}.gz | cut -d\; -f3,4 | gzip > JavaYearAuthor.gz
# now follow the similar approach in Perl example shown above to get the final
    result
```

7 Future work

To have an impact on research practice, the WoC prototype needs to be exposed via reliable services that help with research and do not overwhelm the platform. Currently, we only have Python and Perl API available. However, more languages will be supported in the future. Comparatively small pre-extracted relations will be stored into relational database to extend our accessibility to users who are used to SQL. WoC should also accommodate additional data and computational procedures needed for discovering, correcting, cleaning, augmenting, and modeling the underlying data. Processing hundreds of terabytes of data on powerful clusters may be out of reach for most research groups. Therefore, to accommodate massive queries WoC would require more powerful hardware. Such hardware can be obtained from cloud vendors, but the costs of hosting and analyzing data on these platforms might be high. An alternative might be a few high-throughput services that work on the hardware we currently employ.

The differentiating features of WoC are the completeness of the collection and access to global relationships. Specifically, two basic services would be difficult to replicate outside WoC, yet be capable of high throughput on the limited hardware. First, a reporting service that considers prevalence of certain features, such as languages, tools, and other technologies as well as the information about contributors might provide services akin to those provided by a population census. The second basic service would focus on identifying all entities linked to a specific entity, such

as files modified by a developer, all repositories containing a specific code, or all files that use a specific module or technology. These two capabilities, in conjunction with MSR technology already in use, would provide both, population-level data and complete links within entire FLOSS ecosystem. It would then be up to researchers to retrieve additional data on individual projects based on the stratified samples from the first service or derived from the relationships obtained from the second service.

8 Limitations

We tried to make the assumptions and rationale for specific decisions clear within each section but it is important to reiterate at least some of the limitations. Despite a large size (the collection contains over 1.45B commits), there is no guarantee it closely approximates the entirety of public version control systems as the project discovery procedure is only an approximation. Our focus on git (due to the simplified global representation) excludes older version control systems that have not been converted to git yet. We regularly identify issues with data being incomplete due to collection, cleaning, or processing and we are working on an approach to continuously validate and correct it. The particular design decisions were focused on the particular computing capabilities that were available to us at the time and could/should be revisited as the prototype evolves. The entirety of research tasks that WoC provides is not exhausted by the few examples we have investigated and certain tasks may require different solutions. We do, however, think that the micro-services approach allows for simpler addition/extension/replacement of components as needs or opportunities arise than would be possible with a more monolithic architecture.

How to reliably clean, correct, integrate, and augment the collected data so that the resulting analyses accurately reflect the modeled phenomena is a concern. To ensure the performance of the analytics layer certain objects are filtered from it. For example, some of the public repositories are created to test the performance/-capabilities of git and contain many millions of files/blobs in a single commit. Such commits are excluded from the analytics layer to speed-up the commit-to-file and commit-to-blob maps. The nature of the data may also create performance problems. For example, the most common blob is an empty file. Mapping such blobs to all commits that create them or to all files does not make sense, since there are millions of commits that have created empty files. These performance-related modifications may affect some arguably superficial analyses, e.g., what are the commits with the largest number of files? We explicitly highlight these modification in the WoC code to minimize potential confusion.

Reproducibility may pose an issue in a constantly updated database. Since git objects are added incrementally and order in which they are stored is preserved, we can reconstruct any past version of the object store. For the analytic layer, which depends on the set of git objects available at the time, we create versions, where each of the maps described above is tagged with a version identifying the state of git object store. Preserving these past versions ensures reproducibility of the results obtained from them.

The research use cases presented do not constitute an empirical evaluation of WoC usability but, instead, focus on presenting vignettes that are effective for

these scenarios. Some of these vignettes went through several iterations until the simplest and fastest implementations were obtained.

9 Conclusions

We introduce WoC: a prototype of an updatable and expandable infrastructure to support research and tools that rely on version control data from the entirety of open source projects and discuss some of the research problems that require such global reach. We discuss how we address some of the data scale and quality challenges related to data discovery, retrieval, and storage. We enable wide data access to collected data source by providing a tool built on top of the infrastructure, which scales well with completion to query in linear time. Furthermore, we implement ways to make this large dataset usable for a number of research tasks by doing targeted data correction and augmentation and by creating data structures derived from the raw data that permit accomplishing these research tasks quickly, despite the vastness of the underlying data. Finally, we evaluated WoC by conducting actual research tasks and by inviting researchers to undertake investigations of their own. In summary, WoC can provide support for diverse research tasks that would be otherwise out of reach for most researchers. Its focus on global properties of all public source code will enable research that could not be previously done and help to address highly relevant challenges of open source ecosystem sustainability and of risks posed by this global software supply chain. Transforming the WoC prototype into a widely accessible platform is, therefore, our immediate priority.

All source codes can be found in a public repository.[20]

Acknowledgment

This work was supported by the National Science Foundation NSF Award 1633437.

References

1. Nexus repository. `https://www.sonatype.com/nexus-repository-oss`. Accessed: 2019-01-02
2. Abadi, D.J.: Data management in the cloud: Limitations and opportunities. IEEE Data Eng. Bull. **32**(1), 3–12 (2009)
3. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 359–370. ACM (2004)
4. Amreen, S., Mockus, A., Bogart, C., Zhang, Y., Zaretzki, R.: Alfaa: Active learning fingerprint based anti-aliasing for correcting developer identity errors in version control data. arXiv preprint arXiv:1901.03363 (2019)
5. Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. Science of Computer Programming **79**, 241–259 (2014)
6. Bevan, J., Whitehead Jr, E.J., Kim, S., Godfrey, M.: Facilitating software evolution research with kenyon. ACM SIGSOFT software engineering notes **30**(5), 177–186 (2005)

[20] https://github.com/ssc-oscar/Analytics

7. Bird, C., Gourley, A., Devanbu, P., Gertz, M., Swaminathan, A.: Mining email social networks. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06, pp. 137–143. ACM, New York, NY, USA (2006). DOI 10.1145/1137983.1138016. URL http://doi.acm.org/10.1145/1137983.1138016
8. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P.: The promises and perils of mining git. In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, pp. 1–10. IEEE (2009)
9. Budde, R., Kautz, K., Kuhlenkamp, K., Züllighoven, H.: Prototyping. In: Prototyping, pp. 33–46. Springer (1992)
10. Chacon, S., Straub, B.: Pro git. Apress (2014)
11. Coelho, J., Valente, M.T., Silva, L.L., Shihab, E.: Identifying unmaintained projects in github. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM (2018)
12. Czerwonka, J., Nagappan, N., Schulte, W., Murphy, B.: Codemine: Building a software development data analytics platform at microsoft. IEEE software **30**(4), 64–71 (2013)
13. Dey, T., Ma, Y., Mockus, A.: Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem. arXiv preprint arXiv:1907.06538 (2019)
14. Dey, T., Mockus, A.: Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 66–69. ACM (2018)
15. Di Cosmo, R., Zacchiroli, S.: Software heritage: Whyandhowtopreservesoftwaresourcecode. ipres 2017 (2017)
16. Ducasse, S., Gîrba, T., Nierstrasz, O.: Moose: an agile reengineering environment. In: ACM SIGSOFT Software engineering notes, vol. 30, pp. 99–102. ACM (2005)
17. Dyer, R.: Task fusion: Improving utilization of multi-user clusters. In: Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity, SPLASH SRC, pp. 117–118 (2013)
18. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 35th International Conference on Software Engineering, ICSE'13, pp. 422–431 (2013)
19. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: an enabling language and infrastructure for ultra-large scale msr studies. The Art and Science of Analyzing Software Data pp. 593–621 (2015)
20. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: Ultra-large-scale software repository and source-code mining. ACM Trans. Softw. Eng. Methodol. **25**(1), 7:1–7:34 (2015)
21. Dyer, R., Rajan, H., Nguyen, T.N.: Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences, GPCE, pp. 23–32 (2013)
22. Eastlake 3rd, D., Jones, P.: Us secure hash algorithm 1 (sha1). Tech. rep. (2001)
23. German, D., Mockus, A.: Automating the measurement of open source projects. In: Proceedings of the 3rd workshop on open source software engineering, pp. 63–67. University College Cork Cork Ireland (2003)
24. Gorton, I., Bener, A.B., Mockus, A.: Software engineering for big data systems. IEEE Software **33**(2), 32–35 (2016)
25. Gousios, G.: The ghtorrent dataset and tool suite. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp. 233–236. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2487085.2487132
26. Gousios, G., Pinzger, M., Deursen, A.v.: An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, pp. 345–355. ACM (2014)
27. Gousios, G., Spinellis, D.: Alitheia core: An extensible software quality monitoring platform. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pp. 579–582. IEEE (2009)
28. Gousios, G., Spinellis, D.: Ghtorrent: Github's data from a firehose. In: Mining software repositories (msr), 2012 9th ieee working conference on, pp. 12–21. IEEE (2012)
29. Gousios, G., Vasilescu, B., Serebrenik, A., Zaidman, A.: Lean ghtorrent: Github data on demand. In: Proceedings of the 11th working conference on mining software repositories, pp. 384–387. ACM (2014)

30. Gousios, G., Zaidman, A.: A dataset for pull-based development research. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 368–371. ACM (2014)
31. Hackbarth, R., Mockus, A., Palframan, J., Weiss, D.: Assessing the state of software in a large enterprise. Journal of Empirical Software Engineering **10**(3), 219–249 (2010)
32. Howison, J., Conklin, M., Crowston, K.: Flossmole: A collaborative repository for floss research data and analyses. International Journal of Information Technology and Web Engineering (IJITWE) **1**(3), 17–26 (2006)
33. Kim, S., Zimmermann, T., Kim, M., Hassan, A.E., Mockus, A., Gîrba, T., Pinzger, M., Jr., E.J.W., Zeller, A.: Ta-re: an exchange language for mining software repositories. In: ICSE'06 Workshop on Mining Software Repositories, pp. 22–25. Shanghai, China (2006). URL `http://dl.acm.org/authorize?804411`
34. Le, Q., Mikolov, T.: Distributed representation of sentences and documents. In: Proceedings of the 31 st International Conference on Machine Learning, vol. 32. JMLR, Beijing,China (2014). URL `https://cs.stanford.edu/~quocle/paragraph\_vector.pdf`
35. Leavitt, N.: Will nosql databases live up to their promise? Computer **43**(2) (2010)
36. Lichter, H., Schneider-Hufschmidt, M., Zullighoven, H.: Prototyping in industrial software projects-bridging the gap between theory and practice. IEEE transactions on software engineering **20**(11), 825–832 (1994)
37. Luhn, H.P.: A business intelligence system. IBM Journal of research and development **2**(4), 314–319 (1958)
38. Ma, Y., Dey, T., Smith, J.M., Wilder, N., Mockus, A.: Crowdsourcing the discovery of software repositories in an educational environment. PeerJ Preprints **4**, e2551v1
39. Ma, Y., Mockus, A., Zaretzki, R., Bradley, R., Bichescu, B.: A methodology for analyzing uptake of software technologies among developers. arXiv preprint arXiv:1908.11431 (2019)
40. Mockus, A.: Large-scale code reuse in open source software. In: ICSE'07 Intl. Workshop on Emerging Trends in FLOSS Research and Development. Minneapolis, Minnesota (2007). URL `papers/ossreuse.pdf`
41. Mockus, A.: Amassing and indexing a large sample of version control systems: towards the census of public source code history. In: 6th IEEE Working Conference on Mining Software Repositories (2009). URL `papers/amassing.pdf`
42. Mockus, A.: Engineering big data solutions. In: ICSE'14 FOSE (2014). URL `papers/BigData.pdf`
43. Moniruzzaman, A., Hossain, S.A.: Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. arXiv preprint arXiv:1307.0191 (2013)
44. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Software Engineering **22**(6), 3219–3253 (2017)
45. Newman, S.: Building microservices: designing fine-grained systems. " O'Reilly Media, Inc." (2015)
46. Ossher, J., Bajracharya, S., Linstead, E., Baldi, P., Lopes, C.: Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, pp. 183–186. IEEE (2009)
47. Rajan, H., Nguyen, T.N., Dyer, R., Nguyen, H.A.: Boa website. http://boa.cs.iastate.edu/ (2015)
48. Rosch, E.: Principles of categorization. Concepts: core readings **189** (1999)
49. Rozenberg, D., Beschastnikh, I., Kosmale, F., Poser, V., Becker, H., Palyart, M., Murphy, G.C.: Comparing repositories visually with repograms. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 109–120. ACM (2016)
50. Russom, P., et al.: Big data analytics. TDWI best practices report, fourth quarter **19**(4), 1–34 (2011)
51. Sayyad Shirabad, J., Menzies, T.: The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada (2005). URL `http://promise.site.uottawa.ca/SERepository`
52. Software, B.D.: Black duck open hub. `https://www.openhub.net/`. Accessed: 2018-12-18
53. Tiwari, N.M., Upadhyaya, G., Nguyen, H.A., Rajan, H.: Candoia: A platform for building and sharing mining software repositories tools as apps. In: MSR'17: 14th International Conference on Mining Software Repositories (2017)
54. Tiwari, N.M., Upadhyaya, G., Rajan, H.: Candoia: A platform and ecosystem for mining software repositories tools. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 759–764. ACM (2016)

55. Upadhyaya, G., Rajan, H.: On accelerating ultra-large-scale mining. In: Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track, pp. 39–42. IEEE Press (2017)
56. Upadhyaya, G., Rajan, H.: On accelerating source code analysis at massive scale. IEEE Transactions on Software Engineering (2018)
57. Winkler, W.: String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage (1990)
58. Winkler, W.E.: Overview of record linkage and current research directions. Tech. rep., BUREAU OF THE CENSUS (2006)

Appendix A Source Code for Cmt2ATShow.perl

```
#!/usr/bin/perl -I /home/audris/lib64/perl5 -I /home/audris/lib/x86_64-linux-gnu/perl
use strict;
use warnings;
use Error qw(:try);
use TokyoCabinet;
use Compress::LZF;

sub toHex {
  return unpack "H*", $_[0];
}
sub fromHex {
  return pack "H*", $_[0];
}

my $split = 1;
$split = $ARGV[1] + 0 if defined $ARGV[1];

my %c2at;
for my $sec (0..($split-1)){
  my $fname = "$ARGV[0].$sec.tch";
  $fname = $ARGV[0] if ($split == 1);
  tie %{$c2at{$sec}}, "TokyoCabinet::HDB", "$fname", TokyoCabinet::HDB::OREADER,
        16777213, -1, -1, TokyoCabinet::TDB::TLARGE, 100000
      or die "cant open $fname\n";
}

while (<STDIN>){
  chop ();
  my $c = fromHex($_);
  my $ss = pack 'H*', substr ($_, 0, 2);
  my $sec = (unpack "C", $ss)%$split;
  if (defined $c2at{$sec}{$c}) {
    my ($time, $author) = split(/;/, $c2at{$sec}{$c});
    my @parts = localtime($time);
    my $year= $parts[5] + 1900;
    print $year.";".$author."\n";
  }
}
for my $sec (0..($split-1)){
  untie %{$c2at{$sec}};
}
```

Appendix B Source Code for the custom `lsort` command in tutorial

```
#!/bin/bash
export LC_ALL=C
export LANG=C
sz=${1:-10G}
shift
sort -T. -S $sz --compress-program=gzip $@
```