CodeDJ: Reproducible Queries over Large-Scale Software Repositories

Petr Maj[1]
Czech Technical University in Prague, Czech Republic

Konrad Siek[1]
Czech Technical University in Prague, Czech Republic

Alexander Kovalenko
Czech Technical University in Prague, Czech Republic

Jan Vitek
Czech Technical University in Prague, Czech Republic
Northeastern University, Boston, MA, USA

Abstract

Analyzing massive code bases is a staple of modern software engineering research – a welcome side-effect of the advent of large-scale software repositories such as GitHub. Selecting which projects one should analyze is a labor-intensive process, and a process that can lead to biased results if the selection is not representative of the population of interest. One issue faced by researchers is that the interface exposed by software repositories only allows the most basic of queries. CodeDJ is an infrastructure for querying repositories composed of a persistent datastore, constantly updated with data acquired from GitHub, and an in-memory database with a Rust query interface. CodeDJ supports reproducibility, historical queries are answered deterministically using past states of the datastore; thus researchers can reproduce published results. To illustrate the benefits of CodeDJ, we identify biases in the data of a published study and, by repeating the analysis with new data, we demonstrate that the study's conclusions were sensitive to the choice of projects.

2012 ACM Subject Classification Software and its engineering → Ultra-large-scale systems

Keywords and phrases Software, Mining Code Repositories, Source Code Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.6

Supplementary Material *Software (ECOOP 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.2.13`

Funding This work is supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No.CZ.02.1.01/0.0/0.0/15_003/0000421 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 695412).

1 Introduction

With over 190 million public projects, GitHub is our largest source of empirical data about how software is developed. It is a treasure trove that must be mined if we want to distill insights from its contents. Manual inspection is limited to small-scale case studies; even automated analysis tools struggle with the sheer amount of data available. The software engineering community has taken up this challenge, researchers examine increasingly larger numbers of projects in order to test hypotheses and derive knowledge about the software development process. Examples of such studies include investigations of testing practices [12], changes to licensing over time [18], popularity trends [4] and configuration settings [17].

[1] These authors contributed equally.

These works use samples of GitHub ranging from 15K to 100K projects filtered to exclude projects considered as lacking in size, popularity, originality or importance.

For any scientific study of software, selecting the projects that make up the input of that study is fraught with risks. Any given choice can introduce unwanted and sometimes undetected bias. This bias may, in turn, taint the conclusions of the work. Much like the task of polling voters before an election, choosing a subset of a larger population must be done carefully. In polls, the goal is to ensure appropriate representation of likely voters. The chosen subset excludes citizens who are either not eligible or unlikely to vote, and balances the various population groups. At the same time, for reasons of cost and practicality, the size of this subset is kept as small as possible. Even when pollsters are careful, the accuracy of predictions varies. In software engineering, we often look for some properties of "real" code – where our definition of the term is sensitive to context and research goals. One may exclude course assignments because the errors made by beginners are not relevant to deployed software; on the other hand, if our goal is to shine a light on acquisition of programming skills, then that kind of code may be exactly what is needed. Picking the right set of inputs is thus the first challenge any researcher in the field must address.

With software, Nagappan et al. warned us that more is not always better [14]. Their observations hold now more so than back in 2013 as anyone can create a GitHub repository at no cost and house almost anything there. Manual inspection found that 37% of hosted projects are not used for software development [11]. Thus, the quality of data gathered from software repositories should always be questioned. A stark illustration why skepticism is in order comes from the finding that ten common source corpora have up to 68% of bit-for-bit identical file duplicates [1]. Furthermore, the same paper showed that clones impacted the accuracy of results obtained with these corpora. We argue that more is worse: as the number of projects to scrutinize grows, it becomes harder to check whether their data is clean, consistent and well-formed. Consider the case of text files accidentally misidentified as code [15], an error that went unnoticed for three years and was "fixed" by partially invalidating the original paper's conclusions [2]. As a result of this state of affairs, researchers spend significant effort collecting and curating meaningful suites of open source projects. Unfortunately, manual curation cannot track the constantly changing software landscape.

In this paper, we aim to address a seemingly simple yet eminently practical question, *how does one find software projects in large-scale software repositories?* The assumption underlying our work, our hypothesis, is that it is possible to select thousands of projects from millions by formulating queries on attributes found in the projects' metadata and on easily computed properties of their source code. To be concrete about the kinds of queries we envision, consider looking for the one hundred most popular projects predominantly written in Java, developed in the five years before the introduction of Lambdas by at least two developers with five years of experience. Furthermore, let's ensure that the selected projects have no more than 5% duplicate files between each other. While the search interface provided by software repositories may allow to query for projects by language, there is no way to compute this query automatically without retrieving all projects.

This paper reports on the status of CodeDJ, an infrastructure for querying large-scale software repositories. In its current incarnation our system is geared towards processing data from any git-based software repository. For our experiments, we specifically target GitHub. The three main engineering challenges we contend with are the sheer size of the data source, the constant updates to its data, and the narrow, rate-limited, interface for accessing projects. In addition, a key design requirement is reproducibility; not only should queries execute deterministically, but the infrastructure should be able to replay a historical query with identical results. Thus, researchers may take any query from the literature, even years after it was originally run and its output was used in a publication, and match its results. Furthermore, researchers should be able to modify a historical query and run it based on the information available at any point in the past.

To address these challenges and requirements, CodeDJ is architected in two distinct subsystems. Interaction with the data source is mediated by Parasite, a time-indexed datastore that automatically and continuously queries GitHub for data about projects. Parasite is responsible for data acquisition and keeping that data up-to-date over time. Every datum is logically time-stamped to enable reproducibility. To ensure that CodeDJ can scale, Parasite can be split up into multiple distinct substores based on the projects' main language. The second subsystem, an in-memory database named Djanco, handles user-written queries. For each query, Djanco determines the portion of the datastore that is required, loads the data, and executes the query. Queries evaluate with project metadata in memory while source code remains on disk. The query syntax is based on data frame manipulation interfaces popular in data science, such as dplyr [19], and is expressed in Rust. We claim the following contributions:

- The design of CodeDJ, a scalable infrastructure for querying large-scale software repositories that supports reproducibility and continuously updated data sources.
- A prototype implementation of Parasite and Djanco written in Rust that shows scalability to millions of projects.
- A dataset consisting of 3.6 million software projects written in 17 languages obtained from GitHub.
- A case study illustrating that the choice of projects can invalidate the conclusion of a research project.

Equally important is what we don't do. We do not provide guidance on how to use our infrastructure. The determination of what is the *right* input for a given analysis is problem specific and the choice remains something individual researchers must grapple with. We have not shown scalability of our infrastructure to the whole of GitHub, we are comfortable with datastores of up to 10 million projects. A larger size may require more work. We do not support interactive queries, our infrastructure was designed with the understanding that queries can take hours to run. We did not focus on optimizing query evaluation by, e.g. parallelizing their execution. Lastly, we do not index any artifacts other than code. Adding images, configuration files and documentation is possible but was not considered one of our targets.

Availability. CodeDJ is an open source infrastructure. Readers interested in repeatability, will find our reproduction package at:

```
https://github.com/PRL-PRG/codedj-ecoop-artifact
```

The source code of Parasite and Djanco are on GitHub at:

```
https://github.com/PRL-PRG/codedj-parasite
https://github.com/PRL-PRG/djanco
```

As our datastore is too large to easily share, Sec. 3.3.4 discusses how external users can run queries on our servers. Another alternative is for users to set up their own CodeDJ instance and gather their own data to execute queries. Our reproduction package contains a complete walk-through of the set up procedure. Of course, users must publish their dataset to enable reproducibility.

2 Related Work

Table 1 gives a high-level comparison with eight systems with aims similar to ours. The first column (*Active*) indicates if the system is actively maintained. Some research projects have fallen into disrepair and their web pages are unreachable. The second column (*Updated*) indicates if continuous updates are supported. Given the rate of addition to GitHub, most systems struggle to keep up. The third column (*Reproducible*) indicates if results are reproducible. Reproducibility is only relevant when the data is updated, systems built on a single static snapshot trivially support reproducibility. The fourth column (*Consistent*) indicates that the data is consistent. Inconsistencies arise when some earlier data (such as parent commits) are missed. The fifth column (*Queries*) describes the nature of the query interface exposed to users. Some systems have a simple filtering mechanism for a fixed set of attributes, such as the language of the project, others have their own query language. In our case, we express queries in Rust. The sixth column (*Sources*) indicates where the data comes from. Mostly this is GitHub, but the Apache Software Foundation and various other sources have also been used in the past. The seventh column (*Size*) is an estimate of how many projects are available. Finally the last column (*Contents*) indicates if source code can be queried. Most systems only include metadata about projects due to the size of the code.

Table 1 Systems comparison.

	Active	Updated	Reproducible	Consistent	Queries	Sources	Size	Contents
Stress [8]	–	–	Y	Y	Filter	Apache	211	–
Flossmetrics [9]	–	–	Y	Y	Filter	Many	2.8**K**	–
Orion [3]	–	–	Y	Y	Own	Many	185**K**	Y
Boa [7]	Y	–	Y	Y	Own	Java	380**K**	Y
Black Duck	Y	Y	–	Y	Filter	Many	680**K**	–
Sourcerer [16]	–	–	Y	Y	Filter	GitHub	4.5**M**	–
GHTorrent [10]	Y	Y	–	Y	SQL	GitHub	157**M**	–
GitHub	Y	Y	–	–	Filter	GitHub	190**M**	Y
CodeDJ	Y	Y	Y	Y	Rust	GitHub	3.6**M**	Y

Stress. This system aims to help choose projects in a reproducible manner [8]. Its corpus consists of 211 projects which can be filtered on 100 pre-computed attributes such as bug tickets or lifetime. The corpus can be sorted and sampled randomly. Queries can be exported so they can be repeated later. Source code is not available for querying. Stress is inactive. CodeDJ scales to larger corpora and allows to specify richer queries. In terms of reproducibility, we support updates to the corpus.

Flossmetrics. This work analyzed 2800 open source projects and computed statistics about various aspects of their development process, such as number of commits and developers [9]. Information from additional sources such as project mailing lists and issue trackers was included. Queries could be formulated on metrics such as COCOMO effort, core team members, evolution and dynamics of bugs. Filtering based on these criteria was supported. The project is inactive and it did not support updates.

Orion. This system aimed to enable retrieving projects using complex search queries linking different artifacts of software development, such as source code, version control metadata, bug tracker tickets, developer activities and interactions extracted from the hosting platform [3]. The project is no longer maintained, it scaled to about 185K projects. CodeDJ is designed to scale to larger corpora and offers a more flexible query interface.

Boa. This system focuses on semantics queries over Java programs [7]. A corpus of 380K Java projects can be queried using a dedicated query language that supports automatic parallelization and pluggable mining functions. Source code can be queried in sophisticated ways as Boa is able to parse and analyze Java. A larger corpus of 7.5M projects can be queried on project summaries. Boa provides reproducibility by ensuring its queries are deterministic with respect to the dataset's version, which are created and archived infrequently (i.e. 2013, 2015, 2019, 2020). CodeDJ differs from Boa in that it is language agnostic and geared towards project selection, as opposed to project analysis. Furthermore, CodeDJ provides full reproducibility in the presence of a continuously evolving dataset.

Black Duck Open Hub. A public directory of open source software[2] that offers search services for discovering, evaluating, tracking, and comparing projects. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of code bases. CodeDJ allows researchers to write their own queries and supports reproducibility.

SourcererCC. The aim of this project is to detect code clones [16]. The tool scales to large datasets and can detect near-identical code at various granularities. It has been used to analyze cloning across large corpora of Java, JavaScript, Python, C and C++ projects on GitHub [13]. It can be used by researchers to detect duplication in their samples which is a source of bias. The project's web page appears to be inactive.

GHTorrent. This database of metadata about GitHub projects offers an SQL interface for queries [10]. It monitors GitHub events to constantly update the available data. The limitation of the approach is that GitHub's events do not have all commit details and file contents, thus these are not stored by GHTorrent. In our experience, the database is not always consistent, this may be due to missed events. We have attempted to upload queries through the public SQL interface but the queries timed out.

GitHub. This service provides two ways to query metadata and contents. A REST API can be used for requesting information about projects and listing them, its search queries provide filtering capabilities across a small set of fixed attributes. A web API provides extended filtering options such as searching within repositories written in a particular language. These interfaces are rate-limited and thus return partial results. The results are non-deterministic and non-reproducible as projects may be added and deleted at any time. CodeDJ provides a view of a subset of GitHub on which we support reproducibility and our queries are richer and deterministic.

We would be remiss if we failed to mention the Software Heritage Archive which aims to preserve all publicly available source code; currently upwards of 9.5B source files, 2B commits and 150M projects [6]. It only allows retrieval of single objects. The authors point to the fragility of current arrangements and the dynamic nature of source code repositories

[2] `https://www.openhub.net`

makes it difficult to reproduce studies that use them. We have encountered this ourselves: we see projects deleted from GitHub, changing names, or visibility. In the future, CodeDJ can be extended to query the heritage corpus as well as other repositories.

3 An Infrastructure for Querying Large-Scale Repositories

The goal of CodeDJ is to allow researchers to formulate queries that evaluate attributes of projects hosted on GitHub and return data about projects matching a specified predicate.

3.1 Design considerations and system architecture

The design of CodeDJ flows from four high-level principles that we motivate next:

- **Consistent, eventually:** The sheer size and churn in data sources such as GitHub means that obtaining a snapshot of the whole data source is not practical. But, it is often the case that a slightly out-of-date view is sufficient for most investigations. We choose to refresh entire projects atomically at irregular intervals. Thus, any individual project is consistent, but for any group of projects, the lower bound on their refresh times is the last consistent time point (`git` histories can be destructively updated, allowing for post factum inconsistencies, we ignore these).
- **Code-centric, language agnostic:** We aim to support queries on project metadata and file contents written in any programming language. To reduce space requirements, the only source artifacts we store is code, deduplication is used to remove redundancy, and metadata is trimmed where possible.
- **Flexible query interface:** Popular data science tools such as dplyr [19] or Spark [20] offer a mix operations inspired by database query languages extended with general purpose capabilities. Inspired by these, we propose an interface expressed in Rust as a library with operations for selecting, grouping, filtering and sampling data. The benefits of our approach over, say, SQL, is that queries are type-safe and benefit from the full generality of the Rust language.
- **Reproducible by design:** The importance of reproducibility cannot be overstated [5], consider [15] which recorded the names of the most starred projects seven years ago, without author names it is not possible uniquely to identify projects, and even with their full names, reconstructing a historical star count is not possible. CodeDJ is designed so it is possible to run any query with the information that the datastore had at an arbitrary point in the past. For this purpose the datastore is time-indexed, strictly append-only.

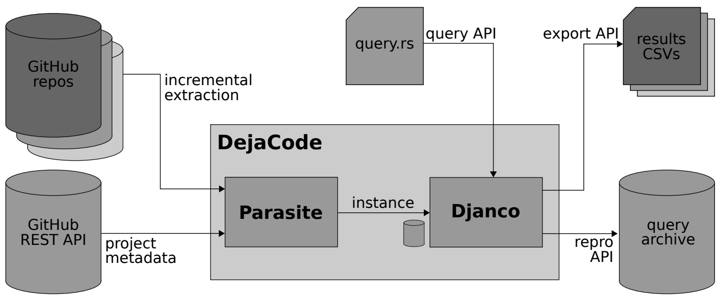

Figure 1 System overview.

Fig. 1 overviews the architecture of CodeDJ. The system is structured around two components, Parasite, a datastore that tracks GitHub, and Djanco, an in-memory database with a Rust interface. Parasite is set up to continuously extract information from GitHub using its REST API for some data and cloning project repositories for other data. The information obtained from the data source is deduplicated and stored in a dedicated format on disk. At irregular intervals projects are refreshed, and the new information is appended to the datastore. When an end-user query is submitted for execution, it comes as a Rust function calling the Djanco query API, a database instance is created for that query. The database will load the data needed for query execution from Parasite. The output of a query is some results, usually as a text file and a record of that query in a reproducibility archive.

The remainder of this section describes our implementation, the design of the query interface and our support for reproducibility.

3.2 The Parasite datastore

Parasite is a dedicated, perpetually running application whose task is to synchronize its on-disk representation with GitHub. This task is complicated by these four constraints:

- **Scalable:** We expect to grow to hundreds of millions of projects, the disk format must be space efficient and its in memory format must be compact and fast to access.
- **Peaceful co-existence:** We must abide by GitHub's terms of service. Parasite must be economical in both the number requests to the GitHub API calls and raw `git` operations.
- **Time-indexed:** Every datum in the store must be associated with its acquisition date, this feature must have a minimal overhead so as not to increase our footprint.
- **Robust:** Backups are not possible due to limited resources, the datastore must thus be resilient to corruption.

Our description focuses on three aspects, the data acquisition process, the data storage format and the interface exposed to Djanco. We also explain how we meet the above constraints.

3.2.1 Acquisition

While, in theory, the GitHub API is sufficient to fulfill all our needs, the fact that GitHub defends itself against denial of service attacks limiting users to 5,000 requests per hour causes a practical problem. As every commit requires one request, the interface is too restrictive to collect data within a reasonable amount of time. Therefore, instead of relying on the API alone, Parasite combines a number of interfaces:

- **Git:** we use the `git clone` command to retrieve source code files and commit histories from repositories;
- **GitHub:** we use the REST API for project metadata (stars, watchers, issues, etc.), information that cannot be obtained through `git` alone;
- **GHTorrent:** instead of querying GitHub for projects directly, we seeded Parasite with the URLs of projects obtained from GHTorrent.[3]

Parasite continuously downloads data from its data sources on a per-project basis. The projects known to Parasite are maintained in a priority queue. Projects are visited in inverse order of last access time. Thus, given any group of projects, the lower bound on the time they were last visited determines the last point when Parasite had a consistent view of those projects modulo destructive `git` history rewrites.

[3] While GHTorrent has over 100M URLs, they are not all valid. Out of 5.5M URLs we visited, only 3.6M were usable, the remaining are either duplicates, have been deleted, or have become private.

When a project is visited, the download procedure begins. First, the project's metadata is retrieved via a call to the REST API. This yields a JSON file with metadata and sundry information. The metadata is stripped of non-essential information (such as URLs for various REST API requests) and stored. The project's current and last known URLs are compared to detect renaming and the new URL is recorded if a change occurred. Next, the project's heads are checked against the heads in the datastore. Each head corresponds to a branch in `git`. If any of the heads changed, the project is cloned and data about new commits and the contents of changed files are extracted and stored. We clone projects because using the REST API to get new commits is slow and rate limited. We clone repeatedly at each visit, caching projects is not feasible due to space limitations (in the future, we plan to cache the most active projects to reduce the amount of data unnecessarily transferred via full clones).

Once a local copy of a project exists, we determine which *substore* that project belongs to and append new commits and files to it. Substores are partitions of the dataset that Parasite uses to organize its disk structures around. Projects are matched to a single substore by properties such as size (e.g. a substore for small projects) or dominant language (e.g. a substore of Python projects).

When processing a chain of commits, a simple optimization is achieved by observing that if we find a commit that is already in the datastore, then all of its parent commits must also already be present. The final step is to record the time of the visit, and move to next project in the queue. Any error during the processing, terminates the visit and the project is flagged as potentially invalid.

Parasite is written in Rust using `libgit2`. It has been parallelized at project-level granularity and scales up to 32 threads. With more threads, the bottleneck shifts from local repository analysis to network bandwidth and ultimately to the GitHub rate limit. When adding projects, Parasite processes 244 projects per thread per hour. As GitHub limits are attached to users (identified by tokens), Parasite supports rotating multiple tokens which allow us to sustain a download rate of 7821 projects per hour using 32 threads. Since Parasite is still in accretion mode, we cannot report on the update rate alone, but we expect it to be limited by GitHub to a rate of 120K active project updates per day per token.

Table 2 Current dataset composition.

	Records	Size	Ratio
Users	4.8M	200M	<0.01%
Projects	3.6M	4.9G	0.2%
Commits	167M	88G	3.2%
Paths	848M	80G	2.9%
Files	463M	2603G	93.7%

Parasite has visited 3.6M projects composed from all non-fork C++ and Python projects available in GHTorrent and a random subset of 50K projects in 17 popular languages. In total, the datastore has 3.6M projects and occupies 2.8TB on disk. Table 2 shows that the majority of the datastore is taken by source code.

3.2.2 Storage

The storage format of Parasite is designed to ensure a low disk footprint, to scale to hundreds of millions of projects. The store is append-only to allow reverting to historic states and to simplify recovery from data corruption. Parasite can be thought of as storing *records*.

Records of same kind are backed by a single *record file*. Records compose together to form *entities*. The following entities are stored by Parasite:

- **Projects:** A project is identified by unique `git` clone URL, it has a set of *heads* (one per branch) and other information from GitHub metadata.
- **Commits:** A commit is identified by its SHA hash, it has a message, changes, parents, an author, a committer, and a time.
- **Paths:** A path is identified by the hash of its string value.
- **Users:** A user is identified by their email.
- **Snapshots:** A snapshot of a file containing source code is identified by its hash.

Records are the smallest unit of information in the datastore, the only way to update an entity is to add a new record. The decomposition of entities to records has been designed along the lines of what information can be updated in isolation. Entities are assigned unique numeric *identifiers* based on their contents. One of the key internal data structures in Parasite are the multiple *mappings* from entity hashes to identifiers. These mappings are used for deduplication.

Deduplication is crucial as up to 94% of files can be duplicates [13]. Mappings are costly as they must be kept in memory. For our corpus, the deduplication mappings for all entities require 89GB. While not a concern at this time, as our dataset grows, mappings will become a bottleneck. To decrease their size, we split Parasite into *substores*. Each substore manages a disjoint partition of the projects. We perform deduplication only within substores. This means that mappings are smaller at the price of some duplication across substores. Our implementation assigns projects to substores based on their size and dominant language; small projects (less than 10 commits) are kept distinct from projects written in targeted languages. A drawback of this design is that identifiers are not unique, if multiple substores must be accessed, extra care must be taken when merging their contents. On the other hand, this compartmentalization has immediate benefits: In terms of robustness, different substores can be stored in different locations and a loss of one does not impact the others. In terms of performance, queries can trivially skip reading irrelevant substores. We measured the duplication across substores at only 5.1%.

As source code (snapshots) dominate the datastore, Parasite internally splits snapshots by language, storing each language separately. This improves reading times for queries that filter by language.

Parasite avoids storing information that is expensive to update and that can be computed readily. For instance, the relation between commits and their project is not stored; it can be recovered from project heads and commit parents. To further reduce footprint, larger records are compressed. For snapshots, the compression ratio is 70%.

To quickly find the latest records for a particular entity, Parasite computes indices, which are stored in dedicated *index files* that provide, for each entity, the location of the latest version of its constituent records. These index files are updated in place as new records are added which exposes them to the risk of being inconsistent. If this occurs, they can always be recomputed from scratch. As of this writing, all indices in the datastore comprised 0.6% of our disk footprint.

To ensure that it is possible to associate a time with every datum on disk, Parasite introduces the notion of a *savepoint*. Since the store is append-only, time-indexing in Parasite boils down to simply associating a time to the current position of each substore. For consistency, savepoints can only be created between visits of projects. They are thus both a mechanism for reproducibility and robustness. Any query can be re-executed at any savepoint and will see the same information. The datastore can be rolled back to a savepoint in case of data corruption.

3.2.3 Interfaces

Parasite has two interfaces, one for data acquisition and another for reading data.

For monitoring purposes data acquisition exposes a detailed breakdown of running tasks, their progress and the usage of GitHub resources. Parasite has both an interactive text-based interface and a command-line interface for automation via scripts. These interfaces allow to create savepoints, verify integrity of the datastore and repair data corruption by reverting to previous savepoints. Parasite monitors available memory to keep as many mappings in memory as it can. Most of the datastore management can be done without reloading any mappings; the initial load takes 26 minutes.

The read interface allows to access records. Iterators are created relative to a savepoint and return records in the order they were added up to that savepoint. Many records are never superseded, for these iterator return values can be used as such. For records that can be overridden with newer values, iterators return updates in reverse chronological order. For projects, Parasite assembles their information; this takes some time as URLs, heads, update status, substore, and metadata must be loaded first, assembly discards all but the most recent versions. Iterators are geared towards sequential access to all elements, but the index files kept by Parasite can be used for random access as well.

3.3 The Djanco database

The Djanco database acts as an intermediary between Parasite and the end-user. It provides a robust query engine that manages loading and pre-processing data and a domain-specific language to express queries easily and concisely. Finally, it supports replaying historical queries. Djanco is designed under the following simplifying assumptions:

- **Single-user:** Djanco is used by a single user for a single query at a time; any parallelism is internal and transparent.
- **Determinism:** Queries are fully replayable on the basis of parameters explicitly provided by the end-users such as random seeds, timestamps, and data source.
- **Read-only:** Queries cannot update the datastore, changes are limited to local objects and are not persisted.
- **Fixed-schema:** Djanco only contains data and metadata pertaining to GitHub.

The need for Djanco comes from the structure of Parasite. The datastore is designed to allow continuous updates and to decrease footprint. This complicates answering research questions. For instance, Parasite elides the relation from a project to its commits. A simple question such as how many commits there are in a project requires recomputing that relation by looking up one the of project's branches and its most recent commit. From that commit, one can follow the parent commits and recursively enumerate them all. Then, repeat for all branches. The database layer computes relations such as these and caches data persistently to speed up queries.

The rationale for a dedicated database rather than an off-the-shelf one are threefold. First, and most arguably, our experience using MySQL on a related project suggested that scalability to large data size (2.8TB and growing) can lead to significant execution overheads. Secondly, we can leverage the assumptions above to implement a domain-specific database as many features of traditional databases (transactions, locks, a general schema) are superfluous. Instead, we implement a solution specialized to our schema that lazily loads selected data from the datastore. Finally, some of our queries are difficult to express in the relational model. Queries can become lengthy and involve multiple joins, nesting and views, which makes them difficult to debug and maintain.

3.3.1 Instances

A Djanco database *instance* is logically created for each end-user query. Each instance is irrevocably tied to a specific slice of the datastore. This slice is defined by two parameters: the substores that indicate which projects to load, and a timestamp indicating a savepoint to be checked out from each substore. If multiple datastores are used, the database joins and deduplicates them. The Djanco schema is shown in Fig. 2, it defines five different

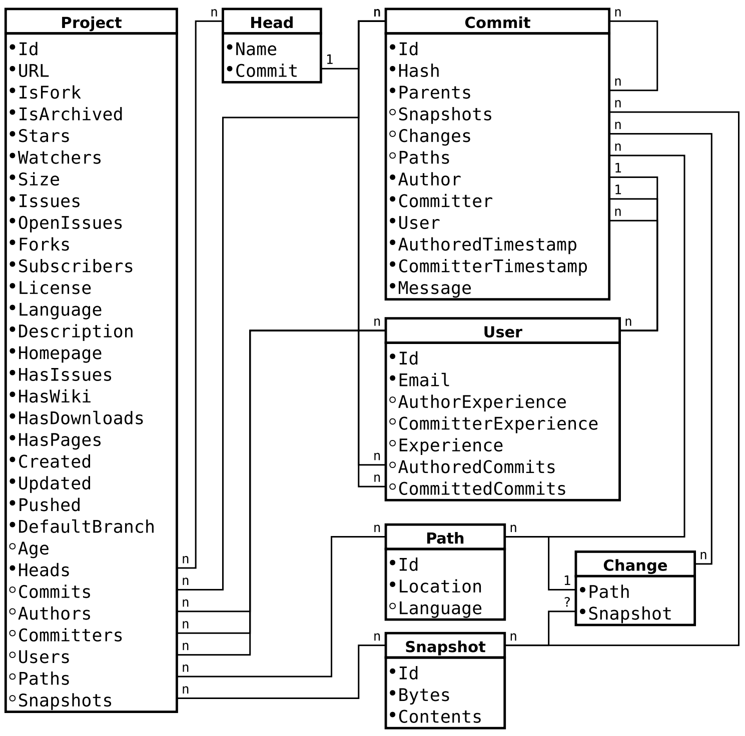

Figure 2 Djanco schema (computed attributes marked ∘).

entities: projects, commits, paths, users and snapshots. Each with their own attributes and convenience methods. Even though Djanco derives its schema from Parasite, there is not a one-to-one correspondence between them. While Parasite tends towards generality and frugality, Djanco instead tends towards expressivity and convenience. For instance, Parasite stores project metadata in JSON, while Djanco parses the format, extracts useful information at sensible types. The basic information about projects is their ID and URL. The metadata includes:

- the language as determined by GitHub;
- the numbers of stars, watchers, subscribers, issues, and forks;
- dates for creation, most recent update, and most recent push;
- the license, description, and homepage URL;
- which web services are active: issues, wiki, downloads, pages;
- size in bytes;
- name of the default branch (e.g. “master” or “main”);
- whether the project is archived or a fork.

Djanco provides a method to calculate the age of a project as the span of the time between its first and most recent commit. Finally, it provides methods to retrieve relations between a project and other entities: heads, commits, users, authors, committers, paths, and snapshots. Except for heads, all the relations need to be computed.

Commits have IDs, hashes, messages, as well as timestamps at which they were authored and pushed. Each commit is associated with users, having an author and a committer. A commit also has a list of changes: a change is a modification to a file represented by a path in the repository and the contents of the file after the change. Finally, commits reference a list of zero or more parent commits in the commit tree.

Users have IDs and emails. In addition, experience is computed for authors and committers as the timespan between their first and last commit. Users also have a method to acquire the list of commits they authored or committed.

Paths represent file system locations within the project (e.g. `"src/main.c"`). They are identified by a synthetic ID and contain a string representing the path. A method to guess the language of a file from its extension is provided. Snapshots are the stream of bytes that are contents of a file at some point in time. For instance, if a file is edited during a commit, the contents of that file before and after the edit are two separate snapshots.

3.3.2 Queries

Queries can be expressed either through a low-level interface or via a DSL. The former accesses the schema directly with Rust iterators and methods. The DSL is a more compact way to implement common queries.

The first step for all queries is to construct a database instance. Since an instance wraps around a specific view of the datastore, constructing it requires specifying a path, a savepoint and substores. The following snippet constructs an instance for small projects available on December 1st, 2016:

```
let db = Djanco::new(PATH, timestamp!(December 2016), substore!(SmallProjects))?;
```

Alternatively, an instance for C, C++, and Python programs is constructed like this:

```
let db = Djanco::new(PATH, timestamp!(December 2016), substores!(C, C++, Python))?;
```

Parameters can be skipped; an instance from all substores at their most recent savepoint is constructed thus (values of defaults are recorded for reproducibility):

```
let db = Djanco::from(PATH)?;
```

Iterators offer access to entities. The snapshot iterator is lazy, the others eagerly load information from the datastore. Iterators are entry points to queries; they return objects that conform to the schema of Fig. 2. This snippet extracts a vector of all languages occurring in projects:

```
let all_languages = db.projects()
  .map(|project| project.language())
  .unique()
  .collect()::<Vec<Language>>;
```

While iterators suffice for just about any query, most queries can be expressed more concisely in our DSL. The DSL uses a pipeline paradigm, where an initial data structure is transformed by a series of methods (aka verbs) that do part of the processing in each step. We provide the following verbs: `group`, `filter`, `sort_by`, `sample`, and `map_into`. We also provide access to any attribute in the schema. In addition, objects and their attributes are composable into complex statements expressing comparisons (e.g. `AtLeast`, `AtMost`, `Matches`, `Contains`), basic statistical functions (`Count`, `Max`, `Median`), sampling methods (`Top`, `Random`), and many others. The code below showcases a few of these:

```
let selection = db.projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(project::Users), 5))
  .sort_by(project::Stars)
  .sample(Top(50));
```

Projects are grouped according to their language, then filtered so that only projects that have at least 5 users are kept, these are sorted by the number of stars in each project and, finally, a sample of top 50 projects is returned.

A useful feature is the ability to deduplicate projects while sampling them according to specific criteria. For example, in the following snippet projects will not be added to the result set unless 90% of their commits are unique with respect to any other project already within the result set:

```
selection.sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)))
```

The final step of a query is to output its results; here we show results written to a CSV file:

```
selection.into_csv(OUTPUT_PATH)?;
```

Each object serializes verbosely, including all information about itself. If only specific information is required, an appropriate format may be imposed by using the map verb to translate an object into its attributes. Here each project is translated into its ID and URL:

```
selection
  .map_into(Select!(project::Id, project::URL))
  .into_csv(OUTPUT_PATH)?;
```

We also provide a function that outputs all information related to a project, including commits, users, paths and snapshots. This creates multiple CSV files.

```
selection.dump_all_info_to(OUTPUT_DIR_PATH)?;
```

Crucially, end-users can do their own use-case–specific formatting by resorting to Rust:

```
selection.for_each(|project| println!("{}:␣{}", project.url(), project.has_wiki()))
```

Further details about our query facilities can be found in the Djanco GitHub repository.

A friend in need. We had an opportunity to test our system when posed a question that was difficult to answer with GitHub's REST API. The query had to retrieve popular C++ repositories that use custom allocators. Finding out whether a project is using a custom allocator requires checking if it imports a library called `memory_resource`. Therefore, we

```
let wanted: HashMap<SnapshotId> = db
 .snapshots()
 .filter(|snapshot|
   snapshot.contains(
    "#include␣<memory_resource>"))
 .map(|snapshot| snapshot.id())
 .collect();

let projects = db.projects()
 .filter(|project| {
  project.snapshots()
   .map_or(false, |snapshots| {
    snapshots.iter()
     .map(|snapshot| snapshot.id())
     .any(|snapshot_id| {
      wanted.contains(snapshot_id)
     })
    })
 .sorted_by_key(|project|
  project.star_count());
```

```
let wanted: HashSet<SnapshotId> = db
 .snapshots()
 .filter_by(
   Contains(snapshot::Contents,
      "#include␣<memory_resource>"))
 .map_into(snapshot::Id)
 .collect();

let projects = db.projects()
 .filter_by(
   AnyIn(project::SnapshotIds, wanted))
 .sort_by(project::Stars);
```

Figure 3 Emery query.

`grep` through source code for the string `"#include␣<memory_resource>"`. In a second step, we iterate over projects and find those, which contain one of the selected snapshots. At that point, we order them by popularity and retrieve some number of the most popular projects. For comparison we wrote the query in pure Rust and then in the DSL. Both implementations are in Fig. 3. As expected the DSL is more compact and more readable. We ran the query on a store with 3M projects and 429M snapshots. The first part of the query found 1724 snapshots in 12 hours. The second part of the query retrieved 1197 projects and their metadata in 24 hours. Then, an additional 6 hours was spent on preparing the project metadata for CSV export.

3.3.3 Data management

Djanco transparently manages the loading and pre-processing of data from the datastore. This involves two mechanisms: lazy loading and caching. Given the size of the data, loading it all into memory is not desirable. Most queries are interested with a small slice of the data, usually filtering out most projects and neglecting most attributes. Therefore, Djanco uses lazy loading to tailor the in-memory data to the needs of each specific query. Snapshots (source code files) are bulky and cannot be split into independent attributes. Only a single snapshot is held in memory at once. The database retrieves them from the datastore only when needed either by scanning the store sequentially or by using the datastore's ability to seek and access a specific snapshot. For the other objects (projects, commits, paths, and users), their attributes are loaded independently on request. Attributes are cached in the database as they can be needed several times.

Memory usage is not the only concern while loading data from the store. From our experiences in querying GitHub, we find that many similar queries are executed on the same datastore view, especially when a query is being developed. Loading attributes from the datastore can be costly, especially in places where the Djanco schema requires the values to be calculated, e.g. for mappings between entities. Therefore, we found it beneficial to avoid recalculating some attributes across queries by implementing on-disk attribute caching, thus improving performance of similar or repeated queries.

For each attribute occuring in a query, the database creates an in-memory map, mapping an entity ID to that entity's value for a given attribute. After an attribute has been loaded, the caching extension serializes it onto disk using the CBOR serialization format. The on-disk cache structure preserves information about which datastore, savepoint, and substore a particular attribute map was read from. Subsequent queries requesting this attribute for this particular datastore view then prefer loading data from the cache rather than the datastore. This process is transparent to the end-user, and can be turned off to save disk space.

Table 3 Caching performance.

	extracting from store	writing to cache	reading from cache	size on disk	cached?
`commit::Parents+commit::Users`	1h 21m 28s	35m 16s	7m 25s	2.3GB	Y
`user::Experience`	1h 10m 19s	1s	1s	5.7MB	Y
`user::CommitterExperience`	1h 9m 52s	1s	1s	5.6MB	Y
`user::AuthoredCommits`	1h 8m 47s	1m 1s	39s	213MB	Y
`project::Commits`	1h 8m 33s	5m 29s	3m 25s	1.1GB	Y
`commit::Changes`	52m 29s	2h 53m 53s	1h 21m 28s	20GB	N
`commit::CommitterTimestamp`	41m 49s	1m 55s	1m 21s	418MB	Y
`commit::Message`	41m 24s	3m 20s	1h 38m 3s	6GB	N

However, while the cache uses up disk space, reading an attribute from CBOR is potentially orders of magnitude faster than loading it from the store. On the other hand, when loading from the store is simple and the data is difficult to serialize (e.g. it consists of large string vectors) caching is not indicated. We have benchmarked and pre-tuned the database to cache only when it is clearly advantageous. Table 3 shows the performance impact of caching while extracting selected attributes on a dataset containing 130K projects and 44M commits. The table lists a few representative attributes in the first column. Columns two and three present what happens when the attribute is requested for the first time: how long it takes to extract it from the datastore and how long it takes to subsequently serialize it onto disk. The fourth and fifth columns show the impact of caching: how long it takes to read the argument from cache (e.g. when the query is re-executed or when another query requires the same attribute from the same datastore view) and how much disk space has to be devoted to the CBOR file. The final column shows our decision whether to cache this attribute or not.

3.3.4 Availability

While users can download their own datasets and run queries on them locally, doing so requires time and computational resources. Therefore, we also provide a procedure for running queries on our hardware using our incrementally updated dataset. A durable, publically available resource also fosters reproducibility.

The submission procedure plugs into the standard Rust toolkit. Queries are submitted as cargo crates. These crates include functions marked as individual queries via annotations which also specify the savepoint and subsets that the specific query expects. For convenience, we provide a template for query crates that works with the `cargo generate` command.[4] We also provide an accompanying `cargo djanco` command[5] which generates an execution harness around query functions. The harness is a small standalone Rust program that sets up the datastore and runs each query according to the specifications found in their annotations. The harness includes a commandline interface through which it can be executed with a specific dataset paths, output directory, and other parameters. We generate the harness for executing the query on our server, but it can be used to test queries locally as well.

As of this writing queries are scheduled manually by the authors. Users should contact us by email with a link to the repository. The query will undergo a manual inspection and will be executed on our hardware and dataset using the same generated harness as above. After the query is executed, a snapshot of the crate is created and stored it in the query archive. The snapshot contains the complete source code of all the queries, logs, the exact generated harness used for execution, and the results of all the queries – files generated to the designated output directory. Any result file exceeding 50MB is ignored (if a query produces large files we contact the user to advise on compaction or to negotiate different means of delivery).

In the future, we will extend our infrastructure to include a web API that will allow users to execute queries themselves. These queries will be expressed in a limited query language (to obviate security risks) and the volume of results will be limited. Queries and results will also be archived and accessible publicly with a receipt. Another extension we foresee is to extend the existing mechanism to allow automatic query execution. This would resemble our current process but it would remove the need for a manual check and emailing the authors

[4] `https://github.com/PRL-PRG/djanco-query-template#template`
[5] `https://github.com/PRL-PRG/cargo-djanco`

as submission could be automated. This option is contingent on our ability to create a static checker for incoming crates and sufficiently isolating them during execution.

Finally, storing user emails has privacy issues. we are considering whether it is appropriate to expose emails for external queries. If retaining emails becomes problematic, we may have to obfuscate the emails and replace them with numeric identifiers.

3.3.5 Reproducibility

To further support reproducibility, above and beyond the ability to deterministically run historical queries, every query executed by Djanco is stored in a public query archive. The query archive is a git repository hosted on GitHub.[6] Each query is hosted in a separate branch in the repository. We expect queries to undergo revisions. Each revision and execution results from that revision are archived as separate commits in a single branch. This produces a development history of the query.

Each query execution produces a *receipt* – a hash representing a specific commit in the archive repository representing the execution. The hash can be used to share queries (exactly as executed) and their results (exactly as produced). It can be used to retrieve the cargo crate and to re-execute the code (e.g. on a different dataset). Code re-execution is helped by the fact that queries are deterministic and the snapshot of the crate contains a list of all depedencies, a timestamp, a list of all subsets and all random seeds. The receipt for the queries in this paper is `da6ae7dd50565e84efbeac990f5788f383939014`.[7]

4 A Case Study: Of Bugs and Languages

The work's motivation is the claim that the *selection of inputs matters in empirical studies of software and that CodeDJ can assist researchers in that process.* We illustrate these points with a case study. We start from prior work, and show that input selection impacts scientific claims, and that CodeDJ allows rapid exploration of the input space.

The starting point is a Foundation of Software Engineering (FSE) paper published in 2014 [15].[8] One contribution of that work is to establish that some programming languages have a greater association with defects than others (RQ1 in [15]). Their methodology can be summarized as follows. For 17 popular languages, select 50 projects hosted on GitHub that have at least 28 commits. For each commit touching a file that contains code in one of the target languages, label the commit as bug-fixing if its message contains a bug-related keyword. Fit a Negative Binomial Regression (NBR) against the labeled data and obtain, for each language, a coefficient and a p-value. The coefficient indicates the strength of the association (positive means more bugs), and the p-value tells us about statistical significance (less than .05 means the coefficient is significant). The FSE paper concluded that TypeScript, Clojure, Haskell, Ruby and Scala were associated with *fewer* bugs, while C, C++, Objective-C, JavaScript, PHP and Python were associated with *more* bugs. The remaining languages did not have statistically significant coefficients.[9]

[6] `https://github.com/PRL-PRG/codedj-query-archive`
[7] `https://github.com/PRL-PRG/codedj-query-archive/tree/da6ae7dd50565e84efbeac990f5788f383939014`
[8] A revised version of the work appeared in the Communications of the ACM in 2017 with some issues fixed, notably the removal of TypeScript from the analyzed languages.
[9] These results were questioned, but the issues raised in [2] are orthogonal to the selection of inputs.

4.1 Corpus

For this experiment we created a datastore using stratified sampling of data available on GHTorrent. We started with 11,000 projects with at least 28 commits written in each of the 17 languages. For each language, we added 6,000 projects randomly selected from GitHub (including smaller projects). In total, our dataset had 172K projects with 28 or more commits and 230K projects in total. Only 3.8K large Erlang projects were available. The dataset has 47M unique commits (and 66M commits in total, suggesting a commit-duplication of 30%, high given forks were excluded). The datastore occupies 51GB on disk. Our goal was to have enough variety to represent the richness of GitHub. Unlike the FSE paper, which was written in 2013, our corpus goes all the way to 2020.

4.2 Random input selection

Our first experiment explores the distribution of possible analysis outcomes. For this, we repeatedly pick a random subset of 50 projects of each of the 17 languages and fit them with NBR. Fig. 4 shows the distribution of the coefficients obtained by 1000 such random selections compared to the results obtained in [15] (shown as a tick to the right of the distribution). Positive values indicate a higher association of the language with defects. The spread of each distribution is a measure of the sensitivity of the analysis to its inputs.

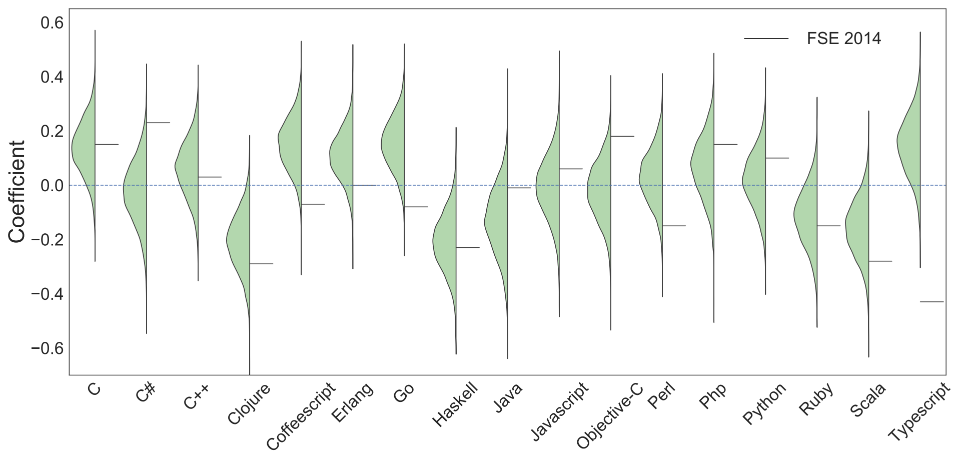

Figure 4 Random subsets.

Intuitively, consider the distribution of coefficients for Objective-C, it is roughly centered around 0. This means, that a random input is about equally likely to say that the language has a positive association with defects as a negative one. One could argue that picking close to the median of the distribution could give a representative answer. As we can see the FSE paper often picks subsets that are outliers; see the cases of CoffeeScript, Go, Perl, Scala and most strikingly TypeScript.

Discussion: As most distributions straddle the axis, random selection is likely to result in noisy conclusions. But, GitHub is noisy itself – for instance there is much code duplication, and the are many low quality projects. A random selection is not the appropriate choice for making conclusions about software developed by professionals. One could choose to mitigate selection bias by increasing the size of the sample; CodeDJ can be used to generate multiple random inputs, if the inputs agree, then our confidence in the results increases.

4.3 Observing change over time

As we have more data than was available in 2013, we can use CodeDJ to select inputs at various times. Here we create eight datasets, each containing data up to one of the years between 2013 and 2020. For simplicity, we only plot the distribution of coefficients for TypeScript. The original paper's coefficient was $-.43$ (shown as a red line). The graph clearly shows that the value was an outlier. The association with bugs shifted over time, increasing to a relatively stable position from 2016.

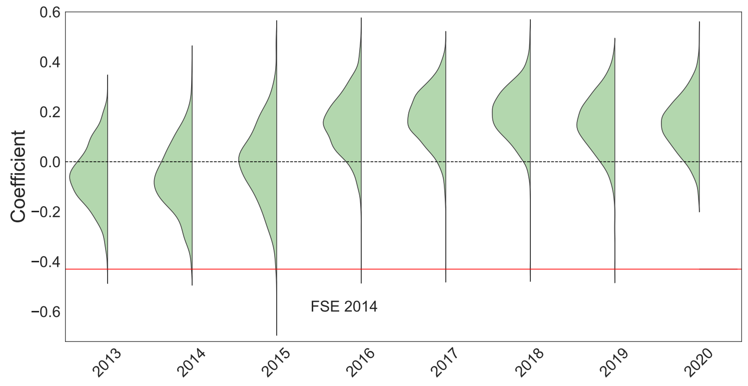

Figure 5 TypeScript over time.

While it is reasonable to expect variations from year to year, TypeScript experienced a rather large shift over a short period. The language was released in 2012, so there were few projects on GitHub in 2013. Furthermore, a number of human language translation files were misidentified as TypeScript; these files did not have bugs, biasing the result. The rising popularity of TypeScript quickly caused real code to crowd out the translation files, and the association with bugs settled to around 0.2.

Discussion: Using CodeDJ to prepare inputs at different time points can help researchers spot trends in the data. For some properties of interest one expects changes over time, for others changes may be an indication of bias that needs to be controlled for. For instance, one would expect the association with bugs of an established, popular, language to be stable.

4.4 Introducing domain knowledge

Choosing any subset of a larger population introduces bias, but this may be intentional, reflecting domain knowledge about the relative importance of observations. For instance, small projects with few commits may be less interesting as they correlate with student projects. These projects have fewer descriptive commit messages and their defects reflect beginner mistakes. It stands to reason to exclude such projects from consideration. Justifying the choice of any particular selection criterion is beyond the scope of our work. CodeDJ allows researchers to explore the impact of various subsets. Our next experiment looks at 6 different criteria for selecting projects and compares them to the original paper's criterion. The Djanco code for those queries is in Fig. 8 in the appendix.

- **Stars:** Pick projects with most stars. *Rationale:* starred projects are popular and thus likely to be well written and maintained. [**Used in FSE 2014**]
- **Touched Files:** compute #files changed by commits, pick projects that changed the most files. *Rationale:* indicative of projects where commits represent larger units of work.

- **Experienced Author:** experienced developers are those on GitHub for at least two years; pick a sample of projects with at least one experienced contributor. *Rationale:* less likely to be throw-away projects.
- **50% Experienced:** projects with two or more developers, half of which experienced. *Rationale:* focus on larger teams.
- **Message Size:** Compute size in bytes of commit messages; pick projects with the largest size. *Rationale:* empty or trivial commit messages indicate uninteresting projects.
- **Number of Commits:** Compute the number of commits; pick projects with the most commits. *Rationale:* larger projects are more mature.
- **Issues:** Pick projects with the most issues. *Rationale:* issues indicate a more structured development process.

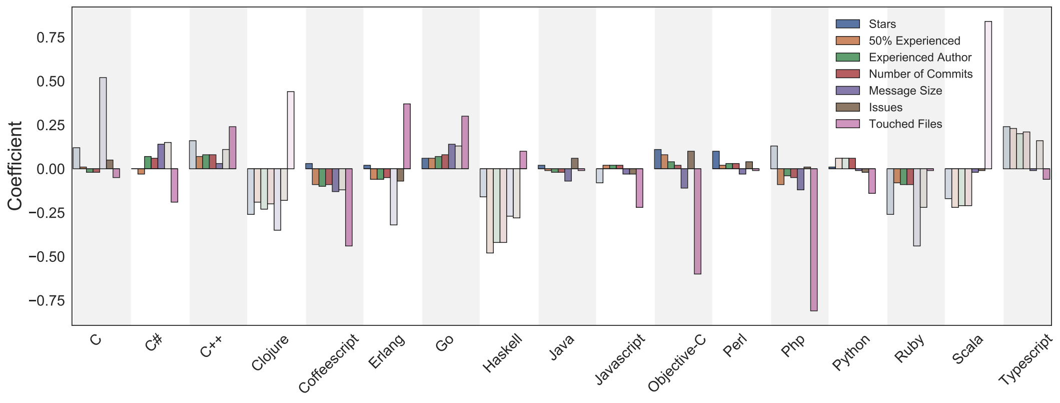

Figure 6 Domain knowledge.

Fig. 6 shows, for each language, the value of the coefficients (higher means more bugs); the queries returned 50 projects in each of the 17 target languages: Coefficients that are not statistically significant are shown in faded colors. If the input set did not matter for the model, one could expect the different queries to give roughly the same coefficients with the same significance. This is not the case. If we focus on how many languages have statistically significant coefficients: The touched files query is highly predictive, 14 of the languages are significant, but the coefficients are frequently opposite from those of other queries. Specifically, C is associated with slightly fewer bugs, so are C#, CoffeeScript, Java, JavaScript, Objective-C, Perl, PHP, Python, Ruby and TypeScript. On the other hand C++, Erlang, Go and Haskell are associated with more defects. This is striking as it goes against expectations. The stars query is the least informative. It only gives 7 statistically significant coefficients with remarkably low values.

Discussion: While some queries yield broadly similar conclusions, this is not the case for all. We stress the importance of understanding the selection criteria and its impact, as statistical significance should not be confused with validity. To help, CodeDJ provides distributions of various measures in the data, Fig. 7 visualizes the distribution of project sizes (left) and project age (right) for the entire dataset and for the various queries.

Looking at these distributions makes it clear that the queries return quite different projects. The experienced author and number of commits are remarkably similar and return projects that meet our expectations. The issues distribution is similar, which should raise red flags given that it frequently disagrees. The stars query returns many smaller projects. Finally, message sizes and touched files show distributions opposite to those expected. They favor

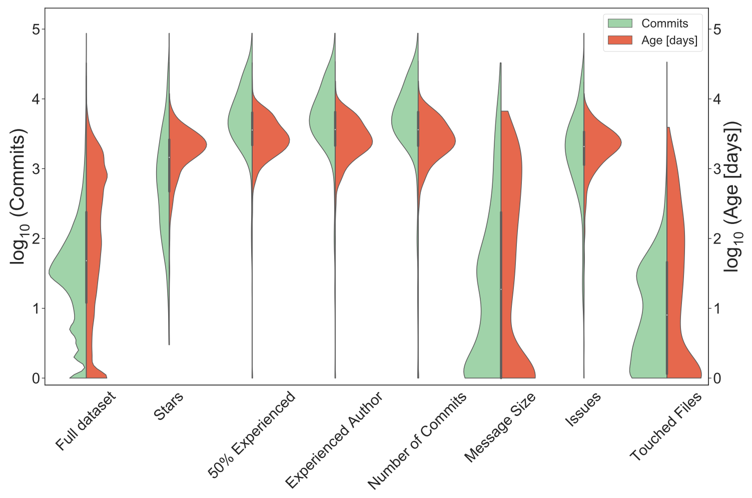

Figure 7 Project Size and Age Distributions.

degenerate young projects with few commits that are either verbose, or disproportionately large (touching over 100K files). This is reflected in the input sizes, ranging from 8M rows for the experienced author query to mere 79K rows of the touched files query. It is likely that these queries are “wrong” in the sense they do not return the population of interest. The figure also suggest that stars is a bad choice.

5 Conclusions

Finding projects on GitHub is akin to looking for the proverbial needle in a haystack. While having a wealth of data at our fingertips is an undeniable asset to empirical software engineering research, the sheer size of the code being hosted is a challenge to any data processing pipeline. Selecting manageable subsets of available projects can introduce subtle, but significant biases that, in turn, can influence or even invalidate the conclusion of the analysis being conducted. Our case study illustrates this problem – we demonstrate that by choosing various, apparently sensible, subsets of the data at hand, we can significantly change the observed association between programming languages and software defects.

This paper introduces CodeDJ, an infrastructure designed to support the reproducible specification of selection criteria for projects hosted on large-scale software repositories. Our implementation is geared towards GitHub. As GitHub is a living system undergoing constant change, ensuring reproducibility requires extra work. The same project downloaded today and last month may contain different code, different commit histories, or the project may disappear entirely. Our infrastructure mitigates this problem by building on a time-indexed, append-only datastore. Queries are expressed in a front-end database that can access a view of the data at a specific point in the history of the datastore.

For future work, three directions stand out: Expanding the datastore, improving the query evaluation performance, and extending accessibility of the our dataset. The dataset provided contains only a fraction of the data we expect to eventually need. As the data grows in volume, our downloading, storage, and processing capabilities will be put to the test and adjusted accordingly to ensure they scale up. We will explore how to ensure backwards compatibility and determinism of queries in the face of changes to the implementation, and to the data format (e.g. adding new information, such as issues, or new file kinds). In terms of performance, our implementation does not try any optimizations of the query evaluation. We intend to parallelize queries and explore ideas from the database community regarding query compilation strategies. Finally, we plan on extending our infrastructure. We will create a web API and a limited query language to make our dataset more generally accessible. We will also investigate an infrastructure for automatic security checking and execution scheduling for query crates which would allow for their automated submission.

References

1 Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2019. `doi:10.1145/3359591.3359735`.

2 Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.*, 41(4):21:1–21:24, 2019. `doi:10.1145/3340571`.

3 T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. Orion: A software project search engine with integrated diverse software artifacts. In *International Conference on Engineering of Complex Computer Systems*, 2013. `doi:10.1109/ICECCS.2013.42`.

4 Hudson Borges, André C. Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. *CoRR*, 2016. URL: `http://arxiv.org/abs/1606.04984`.

5 Andy Cockburn, Pierre Dragicevic, Lonni Besanc on, and Carl Gutwin. Threats of a replication crisis in empirical computer science. *Communications of the ACM*, 2020. `doi:10.1145/3360311`.

6 Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. *International Conference on Digital Preservation*, 2017. URL: `https://hal.archives-ouvertes.fr/hal-01590958`.

7 Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE)*, 2013. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486844`.

8 Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicable approach for project selection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017. `doi:10.1109/ESEM.2017.22`.

9 Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. Collecting data about FLOSS development: The FLOSSMetrics experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS)*, 2010. `doi:10.1145/1833272.1833278`.

10 Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In Michael W. Godfrey and Jim Whitehead, editors, *Working Conference on Mining Software Repositories (MSR)*, 2012. `doi:10.1109/MSR.2012.6224294`.

11 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Working Conference on Mining Software Repositories (MSR)*, 2014. `doi:10.1145/2597073.2597074`.

12 P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *European Conference on Software Maintenance and Reengineering*, 2013. `doi:10.1109/CSMR.2013.48`.

13 Crista Lopes, Petr Maj, Pedro Martins, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjà Vu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. `doi:10.1145/3133908`.

14 Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, 2013. doi:10.1145/2491411.2491415.

15 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *International Symposium on Foundations of Software Engineering (FSE)*, 2014. doi:10.1145/2635868.2635922.

16 Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *International Conference on Software Engineering (ICSE)*, 2016. doi:10.1145/2884781.2884877.

17 Gerald Schermann, Sali Zumberi, and Jürgen Cito. Structured information on state and evolution of dockerfiles on github. In *International Conference on Mining Software Repositories (MSR)*, 2018. doi:10.1145/3196398.3196456.

18 Christopher Vendome, Gabriele Bavota, Massimiliano Di Penta, Mario Linares-Vásquez, Daniel German, and Denys Poshyvanyk. License usage and changes: a large-scale study on GitHub. *Empirical Software Engineering*, 2016. doi:10.1007/s10664-016-9438-4.

19 Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain Franc ois, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019. doi:10.21105/joss.01686.

20 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010. doi:10.5555/1863103.1863113.

A Analysis with GitHub toolkits

Can users do without CodeDJ? Consider the case study queries: stars and touched files.

GitHub exposes a REST API that can return any object and its metadata. The API is limited. It allows filtering by language and sorting by stars, but not by touched files. Furthermore it only returns 1000 results. Therefore, we can't get directly the 17K projects of the case study. While repositories can be obtained by numeric IDs, given the rarity of some of languages such as Erlang means that a random sample would, in the worst case, end up sampling every project on GitHub.

Repository URLs can be retrieved with the `/repositories` query. Assuming 150M repositories, 1.5M queries are needed to find them all. The rate limit is 5K queries/user/hr, so this takes 12 days. We also need language and number of commits to perform stratified sampling. Getting languages is another 12 days. This can be done by getting a list of contributors and summing up their contributions. This only requires one query per repository, so another 12 days. Stratified sampling thus requires approximately a month.

The GitHub data is in JSON, which is not easy to query. One can convert it into a more useful format, such as a relational database. From there, one can retrieve top 50 most-starred projects in each language within that dataset with a query like:

```sql
select id from (
  select id, row_number() over(partition by language order by stars desc) as place
  from projects
) ranks
where place <= 50;
```

The second use case query requires ordering projects by average number of changes per commit. This requires information about all commits. The REST API can list commits, but not changes. To get those, the detailed metadata of each commit is need. This requires one query per commit. With 66M commits, that is 550 days. Deduplicating commits before retrieval shaves this down to 391 days. Having retrieved the data, one can select projects:

```
select id from (
  select id, row_number() over(partition by lang order by avg_touched desc) as place
    from (
      select id, language as lang, avg(touched) as avg_touched
      from project_commits
      join (
        select commit_id, count(path_id) as touched
        from commit_changes
        group by commit_id
      ) touched on project_commits.commit_id = touched.commit_id
      join projects on projects.id = project_commits.project_id
      group by project_id, language
    ) projects
) ranks
where place <= 50;
```

The query is complex. An alternative is to update the data with precomputed attributes.

As the reader may have gathered using GitHub is impractical. An alternative is to use multiple sources of information. Project URLSs, stars and commit counts can be obtained from GHTorrent, commits can be obtained by cloning repositories and analyzing their logs locally. However, these sources have their own shortcomings. GHTorrent does not contain all information, and it can be out of date. For instance, we found commit and star counts off by orders of magnitude. Cloning repositories requires significant bandwidth. In addition, care must be taken with large projects as they can take weeks to analyze if approached naïvely. Gathering data never goes smoothly. The code will likely run for weeks even if massively parallel and then fail on some unexpected corner case. If one then continuously and incrementally update the obtained dataset, then one has essentially reinvented CodeDJ.

B Domain queries

Fig. 8 gives the queries used to inject domain knowledge in the analysis discussed in Sec. 4.

Stars:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(project::Stars)
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Touched Files:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Median(FromEach(project::Commits, Count(commit::Paths))))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Experienced Author:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(FromEachIf(project::Users,
                                      AtLeast(user::Experience,
                                              Duration::from_years(2)))), 1))
  .sort_by(Count(project::Commits))
  .sample(Distinct(Random(50, Seed(42)), MinRatio(project::Commits, 0.9)));
```

50% Experienced:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(project::Users), 2))
  .filter_by(AtLeast(Ratio(FromEachIf(project::Users,
                                      AtLeast(user::Experience,
                                              Duration::from_years(2))),
                           project::Users),
                     Fraction::new(1,2)))
  .sample(Distinct(Random(50, Seed(42)), MinRatio(project::Commits, 0.9)));
```

Message Size:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Mean(FromEach(project::Commits, commit::MessageLength)))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Number of Commits:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Count(project::Commits))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Figure 8 Domain queries.