The Software Heritage Graph Dataset: Large-scale Analysis of Public Software Development History

Antoine Pietri
dante.pietri@inria.fr
Inria
Paris, France

Diomidis Spinellis
dds@aub.gr
Athens University of Economics and Business
Athens, Greece

Stefano Zacchiroli
zack@irif.fr
Université Paris Diderot and Inria
Paris, France

ABSTRACT
Software Heritage is the largest existing public archive of software source code and accompanying development history. It spans more than five billion unique source code files and one billion unique commits, coming from more than 80 million software projects. These software artifacts were retrieved from major collaborative development platforms (e.g., GitHub, GitLab) and package repositories (e.g., PyPI, Debian, NPM), and stored in a uniform representation linking together source code files, directories, commits, and full snapshots of version control systems (VCS) repositories as observed by Software Heritage during periodic crawls. This dataset is unique in terms of accessibility and scale, and allows to explore a number of research questions on the long tail of public software development, instead of solely focusing on "most starred" repositories as it often happens.

1 INTRODUCTION
Analyses of software development history have historically focused on crawling specific "forges" [17] such as GitHub or GitLab, or language specific package managers [5, 13], usually by retrieving a selection of popular repositories ("most starred") and analyzing them individually. This approach has limitations in scope: (1) it works on subsets of the complete corpus of publicly available software, (2) it makes cross-repository history analysis hard, (3) it makes cross-VCS history analysis hard by not being VCS-agnostic.

The Software Heritage project [6, 11] aims to collect, preserve and share all the publicly available software source code, together with the associated development history as captured by modern VCSs [16]. In 2019, we presented the Software Heritage Graph Dataset, a graph representation of all the source code artifacts archived by Software Heritage [15]. The graph is a fully deduplicated Merkle DAG [14] that contains the source code files, directories, commits and releases of all the repositories in the archive.

The dataset captures the state of the Software Heritage archive on September 25th 2018, spanning a full mirror of Github and GitLab.com, the Debian distribution, Gitorious, Google Code, and the PyPI repository. Quantitatively it corresponds to 5 billion unique file contents and 1.1 billion unique commits, harvested from more than 80 million software origins (see Section B for detailed figures).

We expect the dataset to significantly expand the scope of software analysis by lifting the restrictions outlined above: (1) it provides the best approximation of the entire corpus of publicly available software, (2) it blends together related development histories in a single data model, and (3) it abstracts over VCS and package differences, offering a canonical representation (see Section B) of source code artifacts.

2 AVAILABILITY
The dataset is available for download from Zenodo [15] in two different formats (~1 TB each):
- a Postgres [18] database dump in csv format (for the data) and ddl queries (for recreating the DB schema), suitable for local processing on a single server;
- a set of Apache Parquet files [2] suitable for loading into columnar storage and scale-out processing solutions, e.g., Apache Spark [19].

In addition, the dataset is ready to use on two different distributed cloud platforms for live usage:
- on Amazon Athena, [1] which uses PrestoDB to distribute SQL queries.
- on Azure Databricks, [3] which runs Apache Spark and can be queried in Python, Scala or Spark SQL.

3 RESEARCH QUESTIONS AND CHALLENGES
The dataset allows to tackle currently under-explored research questions, and presents interesting engineering challenges. There are different categories of research questions suited for this dataset.

- **Coverage**: Can known software mining experiments be replicated when taking the distribution tail into account? Is language detection possible on an unbounded number of languages, each file having potentially multiple extensions? Can generic tokenizers and code embedding analyzers [?] be built without knowing their language a priori?
- **Graph structure**: How tightly coupled are the different layers of the graph? What is the deduplication efficiency across different programming languages? When do contents or directories tend to be reused?
- **Cross-repository analysis**: How can forking and duplication patterns inform us on software health and risks? How can community forks be distinguished from personal-use forks? What are good predictors of the success of a community fork?
- **Cross-origin analysis**: Is software evolution consistent across different VCS? Are there VCS-specific development patterns? How does a migration from a VCS to another affect development patterns? Is there a relationship between development cycles and package manager releases?

The scale of the dataset makes tackling some questions also an engineering challenge: the sheer volume of data calls for distributed computation, while analyzing a graph of this size requires state of the art graph algorithms, being on the same order of magnitude as WebGraph [7, 8] in terms of edge and node count.
Appendix A FIGURES

The dataset contains more than 11B software artifacts, as shown in Figure 1.

<table>
<thead>
<tr>
<th>Table</th>
<th># of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>85 143 957</td>
</tr>
<tr>
<td>snapshot</td>
<td>57 144 153</td>
</tr>
<tr>
<td>revision</td>
<td>1 125 083 793</td>
</tr>
<tr>
<td>directory</td>
<td>4 422 303 776</td>
</tr>
<tr>
<td>content</td>
<td>5 082 263 206</td>
</tr>
</tbody>
</table>

Figure 1: Number of artifacts in the dataset

Appendix B DATA MODEL

The Software Heritage Graph Dataset exploits the fact that source code artifacts are massively duplicated across hosts and projects [11] to enable tracking of software artifacts across projects, and reduce the storage size of the graph. This is achieved by storing the graph as a Merkle directed acyclic graph (DAG) [14]. By using persistent, cryptographically-strong hashes as node identifiers [10], the graph is deduplicated by sharing all identical nodes.

As shown in Figure 2, the Software Heritage DAG is organized in five logical layers, which we describe from bottom to top.

Contents (or “blobs”) form the graph’s leaves, and contain the raw content of source code files, not including their filenames (which are context-dependent and stored only as part of directory entries). The dataset contains cryptographic checksums for all contents though, that can be used to retrieve the actual files from any Software Heritage mirror using a Web API1 and cross-reference files encountered in the wild, including other datasets.

Directories are lists of named directory entries. Each entry can point to content objects (“file entries”), revisions (“revision entries”), or other directories (“directory entries”).

Revisions (or “commits”) are point-in-time captures of the entire source tree of a development project. Each revision points to the root directory of the project source tree, and a list of its parent revisions.

Releases (or “tags”) are revisions that have been marked as noteworthy with a specific, usually mnemonic, name (e.g., a version number). Each release points to a revision and might include additional descriptive metadata.

Snapshots are point-in-time captures of the full state of a project development repository. As revisions capture the state of a single development line (or “branch”), snapshots capture the state of all branches in a repository and allow to deduplicate unmodified forks across the archive.

Deduplication happens implicitly, automatically tracking byte-identical clones. If a file or a directory is copied to another project, both projects will point to the same node in the graph. Similarly for revisions, if a project is forked on a different hosting platform, the past development history will be deduplicated as the same nodes in the graph. Likewise for snapshots, each “fork” that creates an identical copy of a repository on a code host, will point to the same snapshot node. By walking the graph bottom-up, it is hence possible to find all occurrences of a source code artifact in the archive (e.g., all projects that have ever shipped a specific file content).

The Merkle DAG is encoded in the dataset as a set of relational tables. In addition to the nodes and edges of the graph, the dataset also contains crawling information, as a set of triples capturing where (an origin URL) and when (a timestamp) a given snapshot has been encountered. A simplified view of the corresponding database schema is shown in Figure 3; the full schema is available as part of the dataset distribution.

Appendix C SAMPLE SQL QUERIES

To further illustrate the dataset’s affordances and as motivating examples regarding the research possibilities it opens, below are some sample SQL queries that can be executed with the dataset on AWS Athena.

Listing 1: Most frequent file name

```sql
SELECT FROM_UTF8(name , '') AS name ,
COUNT(DISTINCT target) AS cnt
FROM directory_entry_file
GROUP BY name ORDER BY cnt DESC LIMIT 1;
```

Listing 1 shows a simple query that finds the most frequent file name across all the revisions. The result, obtained by scanning 151GB in 3’40′″, is index.html, which occurs in the dataset 182 million times.

1https://archive.softwareheritage.org/api/
As an example of a query useful in software evolution research, consider the Listing 2. It is based on the convention dictating that commit messages should start with a summary expressed in the imperative mood [12, 3.3.2.1]. Based on that idea, the query uses a regular expression to extract the first word of each commit message and then tallies words by frequency. By scanning 37 GB in 30′′ it gives us that commits concern the following common actions ordered by descending order of frequency: add, fix, update, remove, merge, initial, create.

SQL queries can also be used to express more complex tasks. Consider the research hypothesis that weekend work on open source projects is decreasing over the years as evermore development work is done by companies rather than volunteers. The corresponding data can be obtained by finding the ratio between revisions committed on the weekends of each year and the total number of that year’s revisions (see Listing 3). The results, obtained by scanning 14 GB in 7′′ are inconclusive, and point to the need for further analysis.

The provided dataset forms a graph, which can be difficult query with SQL. Therefore, questions associated with the graph’s characteristics, such as closeness, distance, and centrality, will require the use of other methods, like Spark (see Section D). Yet, interesting metrics can be readily obtained by limiting scans to specific cases, such as merge commits. As an example, Listing 4 calculates the average number of parents of each revision (1.088, after scanning 23 GB in 22′′) by grouping revisions by their parent identifier. Such queries can be used to examine in depth the characteristics of merge operations.

Although the performance of Athena can be impressive, there are cases where the available memory resources will be exhausted causing an expensive query to fail. This typically happens when joining two equally large tables consisting of hundreds of millions of records. This restriction can be overcome by sampling the corresponding tables. Listing 5 demonstrates such a case. The objective here is to determine the modularity at the level of files among diverse programming languages, by examining the size...
of popular file types. The query joins two 5 billion row tables, the file names and the content metadata. To reduce the number of joined rows a 1% sample of the rows is processed, thus scanning 317 GB in 1’20”. The order of the resulting language files (JavaScript>C>C++>Python>C#>Ruby) hints that, with the exception of JavaScript, languages offering more abstraction facilities are associated with smaller source code files.

Appendix D SPARK USAGE

For a more fine-grained control of the computing resources, it is also possible to use the dataset on Spark, through a local install or using the public dataset on Azure Databricks.

Once the tables are loaded in Spark, the query in Listing 6 can be used to generate an outdegree distribution of the directories.

Listing 6: Outdegree distribution of directories

```sql
SELECT degree, count(*) as degree
FROM
ORDER BY count(*) DESC
```

To analyze the graph structure, the GraphFrames library [9] can also be used to perform common operations on the graph. Listing 7 demonstrates how one can load the edges and nodes of the revision tables as a GraphFrame object, then compute the distribution of the connected component sizes in this graph.

Listing 7: Connected components of the revision graph

```python
from graphframes import GraphFrame

revision_nodes = spark.sql("SELECT id FROM revision")
revision_edges = spark.sql("SELECT id as src, parent_id as dst "
"FROM revision_history")
revision_graph = GraphFrame(revision_nodes, revision_edges)

revision_cc = revision_graph.connectedComponents()

distribution = (revision_cc.groupby(["component"]).count()
.withColumnRenamed(\"count\", \"component_size\"))

display(distribution)
```

By allowing users to choose the amount of resources in the cluster, Spark lifts the constraints imposed by limits in Athena, such as timeouts and limited scale-out factor. This is important for computationally intensive experiments or very large join operations, which can only be achieved through sampling in Athena.

Spark is also more flexible in terms of the computations it can perform, thanks to User-Defined Functions [4] that can be used to specify arbitrary operations to be performed on the rows, which isn’t possible with Athena.

Appendix E DATA SAMPLE

A sample of the data as well as instructions to run live queries on the dataset using Amazon Athena can be found here: https://annex.softwareheritage.org/public/dataset/graph/2019-01-28/
REFERENCES