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

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

Diomidis Spinellis
dds@aueb.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	# of objects
origin	85 143 957
snapshot	57 144 153
revision	1 125 083 793
directory	4 422 303 776
content	5 082 263 206

Figure 1: Number of artifacts in the dataset

Appendix B DATA MODEL

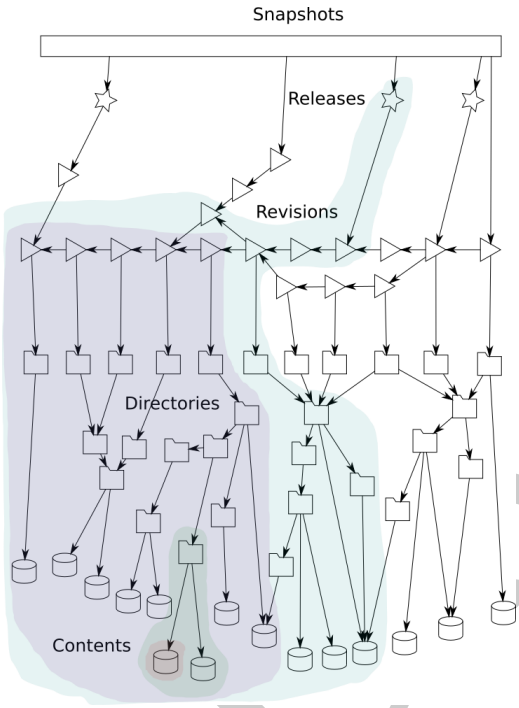

Figure 2: Data model: a uniform Merkle DAG containing source code artifacts and their development history

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 API[1] 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

```
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.

[1] https://archive.softwareheritage.org/api/

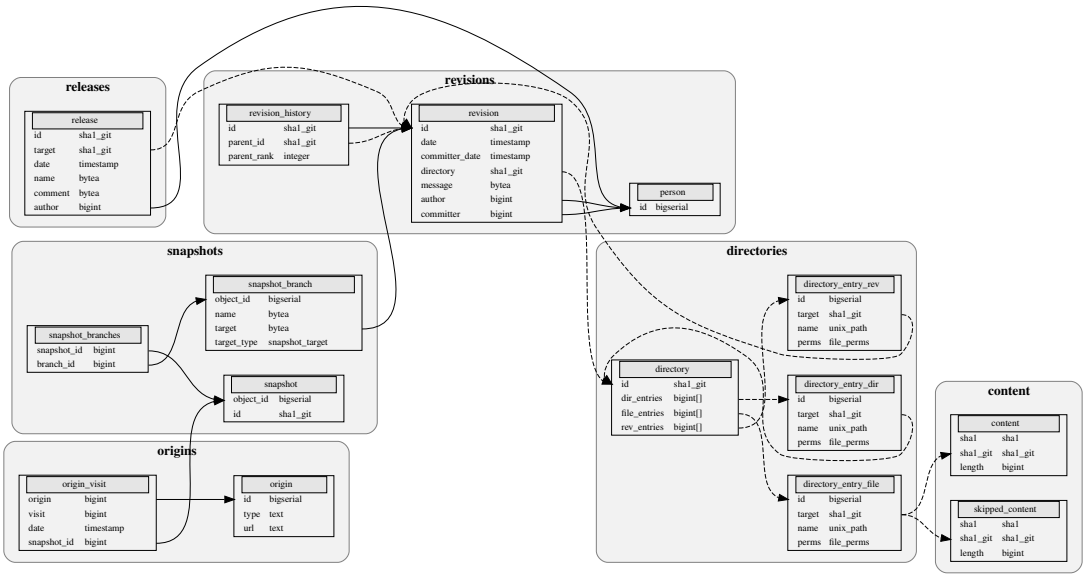

Figure 3: Simplified schema of the Software Heritage Graph Dataset and the number of artifacts in it

Listing 2: Most common commit operations

```
SELECT COUNT(*) AS c, word
FROM
  (SELECT LOWER(REGEXP_EXTRACT(FROM_UTF8(
          message), '^\w+')) AS word
  FROM revision )
WHERE word != ''
GROUP BY  word
ORDER BY  COUNT(*) DESC LIMIT 20;
```

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.

Listing 3: Ratio of commits performed during each year's weekends

```
WITH revision_date AS
  (SELECT FROM_UNIXTIME(date / 1000000) AS date
  FROM revision)
SELECT yearly_rev.year AS year,
  CAST(yearly_weekend_rev.number AS DOUBLE)
  / yearly_rev.number * 100.0 AS weekend_pc
FROM
  (SELECT YEAR(date) AS year, COUNT(*) AS number
  FROM revision_date
  WHERE YEAR(date) BETWEEN 1971 AND 2018
  GROUP BY  YEAR(date) ) AS yearly_rev
JOIN
  (SELECT YEAR(date) AS year, COUNT(*) AS number
  FROM revision_date
  WHERE DAY_OF_WEEK(date) >= 6
      AND YEAR(date) BETWEEN 1971 AND 2018
  GROUP BY  YEAR(date) ) AS yearly_weekend_rev
  ON yearly_rev.year = yearly_weekend_rev.year
ORDER BY  year DESC;
```

Listing 4: Average number of a revision's parents

```
SELECT AVG(fork_size)
FROM (SELECT COUNT(*) AS fork_size
        FROM revision_history
        GROUP BY  parent_id);
```

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

Listing 5: Average size of the most popular file types

```
SELECT suffix,
  ROUND(COUNT(*) * 100 / 1e6) AS Million_files,
  ROUND(AVG(length) / 1024) AS Average_k_length
FROM
  (SELECT length, suffix
  FROM
    -- File length in joinable form
    (SELECT TO_BASE64(sha1_git) AS sha1_git64, length
    FROM content ) AS content_length
  JOIN
  -- Sample of files with popular suffixes
  (SELECT target64, file_suffix_sample.suffix AS suffix
  FROM
    -- Popular suffixes
    (SELECT suffix FROM (
      SELECT REGEXP_EXTRACT(FROM_UTF8(name),
       '\.[^.]+$') AS suffix
    FROM directory_entry_file) AS file_suffix
    GROUP BY  suffix
    ORDER BY  COUNT(*) DESC LIMIT 20 ) AS pop_suffix
  JOIN
    -- Sample of files and suffixes
    (SELECT TO_BASE64(target) AS target64,
      REGEXP_EXTRACT(FROM_UTF8(name),
  '\.[^.]+$') AS suffix
    FROM directory_entry_file TABLESAMPLE BERNOULLI(1))
    AS file_suffix_sample
  ON file_suffix_sample.suffix = pop_suffix.suffix)
  AS pop_suffix_sample
  ON pop_suffix_sample.target64 = content_length.sha1_git64)
GROUP BY  suffix
ORDER BY  AVG(length) DESC;
```

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>PHP>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(*) from (
  select source, count(*) as degree from (
    select hex(source) as source,
           hex(target) as dest from (
      select id as source,
             explode(dir_entries) as dir_entry
      from directory)
    inner join directory_entry_file
      on directory_entry_file.id = dir_entry
  )
  group by source
)
group by degree
order by degree
```

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

```
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')
                .groupby(['component_size']).count())
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

[1] 2019. Amazon Athena. https://aws.amazon.com/athena/.
[2] 2019. Apache Parquet. https://parquet.apache.org.
[3] 2019. Azure Databricks. https://azure.microsoft.com/en-in/services/databricks/.
[4] 2019. User-Defined Functions (UDFs) · The Internals of Spark SQL. https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-udfs.html.
[5] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. 2015. Mining Component Repositories for Installability Issues. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, Massimiliano Di Penta, Martin Pinzger, and Romain Robbes (Eds.). IEEE Computer Society, 24–33. https://doi.org/10.1109/MSR.2015.10
[6] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. 2018. Building the Universal Archive of Source Code. *Commun. ACM* 61, 10 (October 2018), 29–31. https://doi.org/10.1145/3183558
[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.
[8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.
[9] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.
[10] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. 2018. Identifiers for Digital Objects: the Case of Software Source Code Preservation. In *iPRES 2018: 15th International Conference on Digital Preservation*.
[11] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES 2017: 14th International Conference on Digital Preservation*.
[12] Michael Freeman. 2019. *Programming Skills for Data Science: Start Writing Code to Wrangle, Analyze, and Visualize Data with R.* Addison-Wesley, Boston.
[13] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, Jesús M. González-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, 102–112. https://doi.org/10.1109/MSR.2017.55
[14] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science)*, Carl Pomerance (Ed.), Vol. 293. Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32
[15] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. 2019. The Software Heritage Graph Dataset: Public software development under one roof. In *MSR 2019: The 16th International Conference on Mining Software Repositories*. IEEE, 138–142. https://doi.org/10.1109/MSR.2019.00030
[16] Diomidis Spinellis. 2005. Version control systems. *IEEE Software* 22, 5 (2005), 108–109.
[17] Megan Squire. 2017. The Lives and Deaths of Open Source Code Forges. In *Proceedings of the 13th International Symposium on Open Collaboration, OpenSym 2017, Galway, Ireland, August 23-25, 2017*, Lorraine Morgan (Ed.). ACM, 15:1–15:8. https://doi.org/10.1145/3125433.3125468
[18] Michael Stonebraker and Greg Kemnitz. 1991. The POSTGRES next generation database management system. *Commun. ACM* 34, 10 (1991), 78–92.
[19] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.