



HAL
open science

Organizing the graph of public software development for large-scale mining

Antoine Pietri

► **To cite this version:**

Antoine Pietri. Organizing the graph of public software development for large-scale mining. Computer Science [cs]. Inria, 2021. English. NNT: . tel-03515795v1

HAL Id: tel-03515795

<https://hal.science/tel-03515795v1>

Submitted on 6 Jan 2022 (v1), last revised 13 Jan 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



École doctorale 386 : Sciences mathématiques de Paris centre
Université de Paris
Software Heritage, Inria

Thèse de doctorat en informatique

Organizing the graph of public software development for large-scale mining

Par **Antoine PIETRI**

Dirigée par

Stefano ZACCHIROLI

Professeur, Télécom Paris, Institut Polytechnique de Paris Directeur de thèse

Présentée et soutenue publiquement le 5 novembre 2021,
devant un jury composé de :

Jesus M. GONZALEZ-BARAHONA

Professor, Universidad Rey Juan Carlos Rapporteur

Marianne HUCHARD

Professeure des universités, Université de Montpellier Rapportrice

Daniel M. GERMÁN

Professor, University of Victoria Examineur

Hamida SEBA

Maître de conférences HDR, Université Claude Bernard Lyon 1 Examinatrice

Fabien DE MONTGOLFIER

Maître de conférences, Université de Paris Examineur

Ralf TREINEN

Professeur des universités, Université de Paris Examineur

Roberto DI COSMO

Professeur des universités, Inria Examineur

Abstract

The Software Heritage project is a software archive containing the largest public collection of source code files along with their development history, in the form of an immense graph of hundreds of billions of edges. In this thesis, we present architectural techniques to make this graph available for research. We first propose some utilities to access the data at a micro-level in a way that is convenient for smaller-scale research. To run analyses on the entire archive, we extract a property graph in a relational format and evaluate the different ways this data can be exploited using in-house and cloud processing services. We find that while this approach is well suited to process large amounts of flat data in parallel, it has inherent limitations for the highly recursive graph structure of the archive. We propose the use of graph compression as a way to considerably reduce the memory usage of the graph, allowing us to load it entirely in physical memory. We develop a library to run arbitrary algorithms on the compressed graph of public development, using various mapping techniques to access properties at the node and edge levels. Then, we leverage this infrastructure to study the topology of the entire graph, looking at both its local properties and the way software projects are organized in forks. The in-depth understanding of this structure then allows us to discuss different approaches for distributed graph analysis.

Keywords: empirical software engineering, source code, open source software, version control system, digital preservation, graph topology, graph compression

Résumé

Software Heritage est une archive de logiciels contenant la plus grande collection publique de fichiers de code source ainsi que l'historique de leur développement, sous la forme d'un immense graphe de centaines de milliards d'arêtes. Dans cette thèse, nous présentons des techniques architecturales pour rendre ce graphe disponible à des fins de recherche. Nous proposons d'abord quelques utilitaires pour accéder aux données à un niveau local d'une manière adaptée à la recherche à petite échelle. Pour effectuer des analyses sur l'ensemble de l'archive, nous extrayons un graphe de propriétés dans un format relationnel et évaluons différents systèmes de traitement pour exploiter ces données. Cette approche est adaptée au traitement de grandes quantités de données horizontales, mais elle présente des limites inhérentes à la structure fortement récursive du graphe. Nous proposons d'utiliser la compression de graphe comme moyen de réduire considérablement la taille du graphe, ce qui nous permet de le charger en mémoire vive. Nous développons une bibliothèque pour exécuter des algorithmes arbitraires sur le graphe compressé, en utilisant des techniques de mise en correspondance de ses propriétés au niveau des nœuds et des arêtes. Nous utilisons ensuite cette infrastructure pour étudier la topologie locale du graphe ainsi que son organisation en forks de projets logiciels. Comprendre cette structure nous permet ensuite de discuter de différentes approches pour l'analyse distribuée.

Mots-clés : génie logiciel empirique, code source, logiciel open-source, système de gestion de version, conservation numérique, topologie de graphe, compression de graphe

Résumé détaillé

Software Heritage est une initiative ambitieuse de préservation numérique qui amasse des quantités sans précédent de code source de logiciels d'origines diverses, en gardant la trace de toutes leurs évolutions telles que capturées par les systèmes de contrôle de version (VCS). L'archive Software Heritage est devenue le corpus le plus important et le plus complet d'artefacts logiciels publics, englobant les principaux hébergeurs de code source (par exemple, GitHub, GitLab, Bitbucket, Google Code), et supportant une variété de VCS (par exemple, Git, Mercurial, SVN) et de gestionnaires de paquets (Debian, NixOS, PyPI, NPM, ...).

Ces données sont stockées dans un format canonique et dédoublées à travers toute l'archive au niveau des artefacts logiciels, ce qui construit un immense graphe partagé du développement logiciel, où les nœuds sont les fichiers sources, les répertoires et les commits, et les arêtes contiennent les noms des fichiers et les liens entre les commits et les répertoires.

La mise à disposition de cette base de connaissances universelle sur le développement de logiciels offre des opportunités uniques pour ce que l'on appelle communément la recherche sur le « Big Code » : interroger, analyser et extraire des connaissances à partir du contenu des données et de la structure du graphe de l'historique du développement logiciel. Rendre ces données accessibles aux chercheurs pourrait débloquent de nouvelles capacités dans le domaine de la fouille de logiciels. La possibilité d'effectuer des expériences sur l'ensemble du graphe de développement des logiciels publics pourrait nous aider à mieux comprendre l'évolution des logiciels au niveau macro et les structures sociales sous-jacentes des projets logiciels. Cela réduirait également les barrières à l'entrée des études empiriques en diminuant les coûts de la collecte de données et en éliminant le besoin de récupérer et extraire manuellement les données, et faciliterait la reproduction des études à grande échelle en utilisant des échantillons de données complets.

L'exploitation d'une telle collection de codes sources sans précédent pose des défis importants en termes de disponibilité et de représentation des données, d'architecture système et de passage à l'échelle. Le graphe de l'archive Software Heritage contient plus de 20 milliards de nœuds et 200 milliards d'arêtes et croît de manière exponentielle au fil du temps. À cette échelle, il est généralement difficile d'utiliser des outils standard pour

l'analyse des données, et de nouvelles techniques doivent être élaborées sur la base de l'état de l'art du traitement des grands graphes.

Mise à disposition des artefacts logiciels à différentes échelles

Pour mieux comprendre les types de données qui doivent être mises à disposition, nous classons systématiquement les exigences fonctionnelles des chercheurs dans les études d'extraction de logiciels. Nous classons les données disponibles dans l'archive Software Heritage, y compris le contenu des fichiers de code source, l'historique de développement et d'autres types de métadonnées, mais aussi les données qui peuvent être dérivées du graphe : les différences de commit, les graphes de dépendance, les graphes de communauté, etc. Pour cela, nous faisons une revue de littérature de 54 articles scientifiques sur la fouille de logiciels, et agrégeons leurs besoins en analysant les types de données utilisées par chaque étude.

Notre première étape pour mettre à disposition une plateforme généraliste d'analyse de données consiste à rendre les données disponibles dans un format rudimentaire mais exploitable. Cela peut susciter un intérêt pour l'utilisation de l'archive à des fins de recherche et nous permettre de mieux comprendre les défis liés à son exploitation, en identifiant les principaux points sensibles des chercheurs et les limites des différents formats que nous mettons à leur disposition.

Nous fournissons quelques utilitaires pour récupérer les données à un niveau micro : le Vault, un outil simple pour assembler la fermeture transitive d'un artefact logiciel spécifique dans le graphe et le représenter dans un format ouvert, et SwhFS, un système de fichiers virtuel pour exécuter des programmes d'analyse ordinaires sur le code stocké dans l'archive sans avoir à le télécharger au préalable.

Pour permettre des macro-analyses sur l'ensemble du graphe, nous extrayons de l'archive un graphe de propriétés contenant toutes les métadonnées de développement du logiciel au niveau du système de fichiers, de son historique de développement et de l'endroit où il est hébergé. Nous représentons ce graphe dans deux formats relationnels différents : un format en colonnes pour un traitement à grande échelle sur des plateformes de cloud, et un format adapté à l'importation dans des bases de données locales pour une analyse en interne. Nous discutons de diverses considérations liées à ce jeu de données : les différents formats, les performances, la dénormalisation des données et la confidentialité des données. La mise en place de ces jeux de données sur des plateformes cloud nous permet de comprendre les cas d'utilisation pour lesquels ce format relationnel est adapté, ainsi que ses limites, notamment la difficulté d'exécuter des expériences qui nécessitent des algorithmes de parcours de graphe (par exemple, descendre une chaîne de commits).

Exploitation du graphe de développement logiciel en tant que structure réursive

Étant donné que le jeu de données contenant le graphe est une structure intrinsèquement réursive, les outils d'analyse des données volumineuses qui exploitent l'horizontalité des données pour mettre à l'échelle leur traitement ne peuvent pas être utilisés de manière similaire sur des problèmes qui n'ont pas une structure avec un parallélisme embarrassant. Une approche naïve consistant à utiliser des hachages d'objets pour affecter aléatoirement les artefacts à différentes shards n'est pas très efficace, car un algorithme de parcours nécessitera de passer en permanence d'une shard à l'autre. Nous examinons les solutions de passage à l'échelle existantes (GraphFrames, Neo4j, AIGraph) et discutons de leurs limites pour l'analyse de notre graphe.

Une autre approche est celle de la compression de graphes : en utilisant des techniques existantes pour la compression de très grands graphes, nous pouvons faire tenir le graphe Software Heritage sans ses propriétés en mémoire sur une seule machine. Cela nous permet d'exécuter des algorithmes de graphes standard sans avoir à trouver un moyen de les paralléliser. Le graphe compressé peut être utilisé à des fins de prototypage, mais aussi comme un service en production qui peut répondre à des requêtes de parcours simples. Nous discutons des différentes techniques de compression utilisées pour faire tenir la topologie entière du graphe dans seulement ~ 150 GiB de RAM, ainsi que de certains arbitrages de stockage pour les métadonnées des nœuds et des bords du graphe de propriétés.

Pour permettre aux chercheurs d'effectuer des expériences à plus petite échelle, nous présentons un moyen d'extraire des sous-ensembles de données de graphes représentatifs de n'importe quelle taille donnée à partir de l'archive. Nous discutons de diverses implémentations des moyens d'obtenir une « vue » d'une partie donnée du graphe, et nous mettons à disposition quelques sous-ensembles de données « accrocheurs » en utilisant ces techniques.

Structure en graphe du développement logiciel

La compréhension de la structure du graphe de développement est une étape fondamentale vers l'analyse à grande échelle, car elle peut nous aider à déterminer comment partitionner les données en groupes étroitement liés. De plus, une meilleure compréhension de la topologie de ses différentes couches ainsi que des différents cas limites et aberrants que l'on y trouve nous permet de mieux organiser les données pour certains algorithmes spécifiques.

Dans un premier temps, nous examinons les propriétés topologiques de bas niveau du graphe, en mesurant systématiquement quelques paramètres clés sur ses différentes couches : distributions des degrés entrants et degrés sortants, taille des composantes connexes, coefficients de clustering, longueur des plus courts chemins. Nous comparons

ces métriques avec d'autres graphes à grande échelle générés par des activités humaines (réseaux sociaux, graphe du Web, ...).

Nous observons une grande disparité topologique entre les différentes couches qui constituent le graphe, tant au niveau local que dans leur structure globale. En décomposant le graphe en trois couches en fonction des types des nœuds, nous constatons qu'elles ont des formes, des densités et des connectivités radicalement différentes. En particulier, la couche du système de fichiers domine largement les propriétés topologiques du graphe étant donné qu'elle contient 90% de ses nœuds et 97% de ses arêtes. Les deux couches principales du graphe ont des propriétés diamétralement opposées : la couche du système de fichiers a une forte connectivité, une profondeur moyenne caractéristique et un degré sortant moyen divergeant, tandis que la couche de l'historique de développement a une faible connectivité, une profondeur moyenne divergeante et un degré sortant moyen caractéristique.

Une autre observation importante est que, étant donné que les nœuds de la couche du système de fichier s'agrègent dans une composante géante, le graphe ne peut pas être partitionné facilement grâce à une séparation en composantes connexes strictes. Cependant, la plus grosse composante connexe de la couche de l'historique de développement ne contient que 3% des nœuds de cette dernière, ce qui rend possible leur séparation en multiples partitions localement connectées qui peuvent être traitées en parallèle de manière efficace.

Au niveau macro, nous cherchons à comprendre comment les artefacts logiciels sont organisés et partagés entre différents dépôts en étudiant les « forks » dans l'archive, en comparant les différentes définitions de ce qu'est un fork. En exploitant le fait que les projets qui sont développés à partir d'autres projets partagent une partie de leur histoire de développement, nous proposons un algorithme pour organiser la liste de tous les dépôts dans l'archive en « cliques de forks » (groupes de projets qui sont tous des forks les uns des autres) et en « réseaux de forks » (groupes de projets qui sont transitivement liés entre eux par des relations de forks). Nous calculons ces structures pour l'ensemble du graphe de Software Heritage et mesurons leurs distributions de taille.

Conclusion

Enfin, nous résumons les principales contributions académiques de cette thèse, et ses implications pour la recherche empirique en génie logiciel. Nous discutons des futures directions de recherche et des sujets ouverts : le support des requêtes arbitraires sur le graphe par le biais d'un langage de requête expressif, la mise à disposition d'outils pour faciliter les analyses à grande échelle, ainsi que la réduction du délai entre les jeux de données publiques et les données de l'archive par le biais d'exportations incrémentales du graphe.

Extended abstract

Software Heritage is an ambitious digital preservation initiative that is amassing unprecedented quantities of software source code from a variety of origins, keeping track of all their evolution histories as captured by version control systems (VCS). The Software Heritage archive has grown to be the largest and most comprehensive corpus of public software artifacts, encompassing major source code hosts (e.g., GitHub, GitLab, Bitbucket, Google Code), supporting a variety of VCS (e.g., Git, Mercurial, SVN) and package managers (Debian, NixOS, PyPI, NPM, ...).

This data is stored in a canonical format and deduplicated across the entire archive at the level of software artifacts, which constructs an immense shared graph of software development, where the nodes are the source files, directories and commits, and the edges carry the names of the files and the links between commits and directories.

The availability of this universal knowledge base on software development provides unique opportunities for what is now known as “Big Code” research: querying, analyzing and extracting knowledge both from the contents of the data and from the structure of the development history graph. Making this data accessible to researchers could unlock new capabilities in the software mining field. Being able to run experiments on the entire graph of public software development could help us gain a deeper understanding of the evolution of software at a macro level, and the underlying social structures of software projects. It would also reduce the barriers of entry to empirical studies by lowering the costs of data collection and eliminating the need for manual data scraping and retrieval, as well as facilitate replicating studies at a large scale using comprehensive data samples.

The exploitation of such an unprecedented source code collection poses significant challenges in terms of availability, data representation, system architecture and scalability. The graph in the Software Heritage archive contains more than 20 billion nodes and 200 billion edges and grows exponentially over time. At this scale, it is generally difficult to use off-the-shelf tools for general purpose data analysis, and new techniques must be built on top of the state of the art of large graph processing.

Making software artifacts available at different scales

To better understand the kinds of data that need to be made available, we systematically categorize the functional requirements of researchers in software mining studies. We classify the data available in the Software Heritage archive, including the content of the source code files, the development history and other kinds of metadata, but also data that can be derived from the graph: commit diffs, dependency graphs, community graphs, etc.

Our first step towards providing a general purpose platform for data analysis is to make the data available in a crude but still exploitable format. As this can gather interest towards using the archive for research purposes, it allows us to better understand the challenges of exploiting it, by identifying the main pain points of researchers and limitations of the different formats.

We provide a few utilities to fetch the data at a micro-level: the vault, a simple tool to gather the transitive closure of a specific software artifact in the graph and then representing it in an open format, and SwhFS, a virtual filesystem to run common programs on the code stored in the archive without having to pre-download it.

To make possible macro analyses on the entire graph, we extract from the archive a property graph containing all the software development metadata at the filesystem, history and hosting levels. We represent it in two different relational formats: a columnar format for scalable cloud processing, and a format suitable for import in local databases for in-house analysis. We discuss various considerations related to the dataset: the different formats, performance, data denormalization and data privacy. Putting these datasets on cloud platforms allows us to understand the use cases for which this relational format is suitable, as well as its limitations, notably the difficulty of running experiments that require graph traversal algorithms (e.g., walking down a commit chain).

Exploiting software development data as a recursive graph

As the graph dataset is an inherently recursive structure, big data analysis tools that exploit the “flatness” of data to scale-out their processing cannot be used in a similar fashion on problems that are not embarrassingly parallel. A naive scale-out approach of using object hashes to randomly assign the artifacts to different shards is not very efficient, as a traversal algorithm will require jumping between different shards all the time. We investigate existing scale-out solutions (GraphFrames, Neo4j, AIGraph) and discuss their limitations for scale-out graph analysis.

Another approach is that of graph compression: using existing techniques for compressing very large graphs, we can fit the Software Heritage graph without its properties in memory on a single machine. This allows us to run standard graph algorithms without having to find a way to parallelize them. The compressed graph can be used for prototyping purposes, but also as a production service that can answer simple traversal queries. We

discuss the different compression techniques used to fit the entire graph topology in only ~ 150 GiB of RAM, as well as some storage trade-offs for the node and edge metadata on the property graph.

To let researchers run experiments on a smaller scale, we introduce a way to extract representative graph subdatasets of any given size from the archive. We discuss various implementations of ways to obtain a “view” of a given part of the archive, and make some “teaser” datasets available using these techniques.

The graph structure of software development

Understanding the structure of the development graph is a fundamental step towards scale-out analysis, as it can help us determine how to partition the data into tightly-knit clusters. Moreover, getting a better sense of the topology of its different layers as well as the various edge cases and outliers found in it will help us better organize the data for some domain-specific algorithms.

As a first step, we look at the low-level topological properties of the graph, by systematically measuring a few key metrics on its different layers: in-degree and out-degree distributions, connected component sizes, clustering coefficients, average path lengths. We compare these metrics with other large-scale graphs generated by human activities (social networks, graph of the Web, ...).

On a macro-level, we seek to understand how the software artifacts are organized and shared across different repositories by studying the “forks” in the archive, comparing different definitions of what a fork is. By exploiting the fact that projects that are built on top of others will share some of their development history, we propose an algorithm to arrange the list of all the repositories in the archive in “fork cliques” (clusters of projects that are all forks of each other) and “fork networks” (clusters of projects that are transitively linked together by forking relationships). We compute these structures for the entire Software Heritage graph and measure their size distributions.

Conclusion

Finally, we summarize the main academic contributions of this thesis, and its implications for empirical software engineering research. We discuss future research directions and open subjects: supporting general-purpose graph queries through an expressive query language, providing tooling to streamline scale-out analyses, as well as reducing the lag between the public datasets and the live data through incremental graph exports.

Remerciements

Je tiens tout d'abord à remercier Stefano Zacchiroli pour avoir encadré cette thèse jusqu'à son aboutissement. *Zack* était pour moi le directeur de thèse idéal ; il a su être à la fois présent et encourageant, tout en me laissant suffisamment de liberté et d'autonomie dans ma recherche. Merci également à Roberto Di Cosmo,¹ directeur de Software Heritage, pour m'avoir proposé cette thèse et avoir toujours soutenu mon projet de recherche. Ces remerciements s'étendent naturellement aux autres chercheurs et ingénieurs de l'équipe de Software Heritage qui furent un important soutien à la fois technique et moral pendant plus de quatre ans. Merci à (par ordre d'apparition) Nicolas Dandrimont, Antoine Dumont, Guillaume Rousseau, Morane Gruenpeter, Antoine Lambert,² David Douard, Valentin Lorentz et Vincent Sellier. Merci également à Thibault Allançon qui, en l'espace de seulement deux stages, a implémenté une infrastructure solide sur laquelle repose une grande partie de ma thèse.

Je remercie également toute ma famille pour leurs encouragements, notamment mes parents qui ont rendu cette aventure possible et m'ont toujours soutenu dans mes choix d'orientation et dans mes projets. Merci à ma mère qui a été particulièrement attentive durant mes études, et mon père chez qui je partais tous les étés me changer les idées dans les montagnes de l'Alta Rocca.³ Merci à mes frères : Pierre-Louis, Mathieu et François, à mon cousin Victor, qui est sans doute la première personne à m'avoir initié à l'informatique,⁴ à mes cousines : Amélie, Pauline, Claire et Margot, à mes tantes et oncles : Marie-Pascale, Christine,⁵ Pierre,⁶ Chantal et Jean-Pierre, à mon grand-père Jean-Louis qui me fascinait avec ses points de Lagrange et ses programmes BASIC sur disquette, à Thibault Hanin et ses panneaux solaires, et à toute ma famille plus ou moins éloignée. Merci également à Marie-Antoinette et Geneviève de prendre régulièrement de mes nouvelles.

Cette thèse aurait sans doute été très différente sans l'aide de plusieurs personnes. Je remercie Jill-Jênn Vie⁷ qui, en plus d'avoir été un ami très proche pendant plus de dix ans, et avec qui je partage de nombreuses passions informatiques et culturelles, a également

¹Zack et Roberto sont de vraies *rockstars* du logiciel libre : Zack était Debian Project Leader, et Roberto a créé la première distribution Linux pour Live CD.

²Oui, ça fait trois Antoine, bientôt nous dominerons le monde.

³Je reste à ce jour persuadé que Stefano et Roberto pensaient recruter un autre Italien lorsqu'ils m'ont pris en thèse, et qu'ils ont été déçus d'apprendre que mon nom était en fait corse.

⁴Notre complicité n'a d'égal que notre bêtise : nous sommes mus depuis notre plus jeune âge par une inébranlable volonté de tout péter, ce à quoi nous nous appliquons avec créativité et détermination.

⁵Pour se motiver mutuellement pendant notre période de rédaction, on a fait la course. J'ai fini par rendre ma thèse avant qu'elle rende son HDR, mais à seulement dix jours près.

⁶Lui aussi a fait une thèse, mais elle fait 1240 pages et résume 42 ans de recherche. On ne joue pas dans la même cour.

⁷Monsieur le Président.

trouvé ma thèse,⁸ puis l'a intégralement relue.⁹ Thanks to Amie Harte for reviewing every single sentence of this thesis and being such a great friend.¹⁰ Merci à Théo Zimmermann d'avoir relu certains de mes articles et de m'avoir donné de précieux conseils. Merci à Mathieu Tarral qui m'a expliqué le langage Cypher.

Ces quatre années de thèse furent pour moi un véritable plongeon dans l'univers de la recherche académique, mais ma première initiation à celle-ci date de quelques années auparavant. Merci à Akim Demaille pour m'avoir donné goût à la recherche en m'encadrant pendant deux ans au LRDE sur VCSN, un projet qui fut pour moi très enrichissant tant d'un point de vue scientifique que technique.

Merci à Ambre Williams, Thibault Suzanne, Nicolas Jeannerod, Victor Lanvin, Athénaïs Vaginay, Adrien Fabre, Behrang Shirizadeh, Thomas Hegland et tous les doctorants et jeunes docteurs que j'ai eu le privilège de côtoyer pendant ma thèse.

Merci à mes plus ou moins lointains amis avec qui la distance n'est bien heureusement que géographique.¹¹ C'est toujours un immense plaisir de retrouver Rémi Audebert, Sophie Boissonneau, Alexandre Macabiès, Pierre Bourdon,¹² Valentin Tolmer et Sami Boukortt au détour d'un lac suisse, Paul Hervot dans un café strasbourgeois, Antoine Lecubin et Tomoko Kozu dans un *onsen* japonais, ou même Sarah Fachada-Dury et Mercedes Haiech dans les contrées perdues de Gwendalavir.

Regarder un mauvais film est, contre toute attente, une excellente façon de passer une bonne soirée quand on est bien entouré. Merci à Stéphane Lefebvre, Antoine Becquet, Nicolas Allain, Nicolas Salaün, Jennyfer Burlet, Alexandre Meunier, Lucie Mader, Maël Le Garrec, Marcelin Dupraz et Audrey Tinney pour tout ce bon temps passé à regarder des ninjas, requins mutants et autres reptiliens au centre de la Terre.

Organiser pendant dix ans le concours national d'informatique était une formidable aventure, et je tiens à remercier Cyril Amar, Aliona Dangla, Mélanie Godard, Garance Gourdel, Céline Baraban, Jérémie Marguerie, Sacha Delanoue, Nicolas Hureau, Marin Hannache, Maxime Audouin, Nicolas Baradakis, Julien Guertault, Sylvain Laurent, Enguerrand Decorne, Héloïse Vallerio, Victor Collod, Kaci Adjou, Arthur Remaud, Éloi Charpentier, Rémi Dupré ainsi que tous les autres membres de Prologin avec qui j'ai eu le plaisir de travailler, si nombreux que je ne pourrais en dresser une liste exhaustive.

Enfin, merci à tous mes amis de plusieurs époques avec qui mes relations furent aussi diverses qu'enrichissantes : Julian Farella, Romain Maurizot, Guillaume Sanchez, Guillaume Sanchez,¹³ Alexandra Kientz, Pierre-Olivier Di Giuseppe, Thomas Barousseau, Adrien Schildknecht, Juliette Tisseyre, Lucille Tachet, Anthony Seure, Solène Pichereau, Yoann Bourse, Sarah Erika Elvang, Laurent Xu, Anatole Moreau, Arnaud Agbo, Céline Chenu, Ange Daumal, Izia Pétilion, Sébastien Pétilion, Mathieu Stefani, Jean Dupouy, Alexis Cassaigne, Rémi Labeyrie, Julien Morais, Garance Coggins, Juliette Conte et bien d'autres.

⁸Il est allé voir le stand de Software Heritage à une convention Open Source, et m'a dégoté un entretien avec Roberto et Zack, qui m'ont proposé un sujet de doctorat deux jours plus tard.

⁹Il aime aussi les zeugmas et sa femme.

¹⁰She shares my obsession of always being right, and is almost too good at it for my taste.

¹¹Pour être honnête, tant que le lag sur IRC reste raisonnable, ça ne change pas grand chose.

¹²Qui, contrairement aux autres personnes nommées ici, a eu une influence extrêmement mauvaise sur moi, en me conseillant sans cesse d'arrêter ma thèse et de jouer à des jeux vidéo.

¹³L'autre.

Contents

I	Software development and software mining	21
1	Introduction	23
1.1	The rise of large-scale software mining	23
1.2	Universal software mining	24
1.3	Availability and accessibility	25
2	Software Heritage: the Great Library of Source Code	27
2.1	Project goals	27
2.2	Software collection	28
2.3	The world's largest repository of source code	32
3	Version Control Systems	35
3.1	A simple repository model	35
3.2	Artifact deduplication	39
4	The Software Heritage Graph	47
4.1	Canonical Software Artifacts	47
4.2	Consolidating software artifacts in a unified archive	52
4.3	The Software Heritage Merkle DAG	54
4.4	Implementation	58
5	Towards Universal Software Mining	61
5.1	Cataloging research needs	62
5.2	Functional requirements	66
5.3	Challenges	67
5.4	Roadmap	70

II	Making software artifacts available at different scales	73
6	The Vault: Assembling Related Source Code Artifacts	75
6.1	Design	76
6.2	Cooking process and bundle formats	79
6.3	Discussion	81
7	The Software Heritage Filesystem (SwhFS)	83
7.1	Related work	84
7.2	Design	84
7.3	Walkthrough	87
7.4	Discussion	91
8	The Software Heritage Property Graph Dataset	93
8.1	Introduction	93
8.2	Relational model	95
8.3	Dataset export pipeline	99
8.4	Export formats	102
8.5	Dataset coverage	105
8.6	Data analysis platforms	106
III	Exploiting software development data as a recursive graph	117
9	Graph Compression	119
9.1	Introduction	119
9.2	Background	121
9.3	Compressing the Graph of Public Software Development	124
9.4	Exploitation	130
9.5	Discussion	136
9.6	Related work	137
10	Property Data in the Compressed Graph	141
10.1	Introduction	141
10.2	SWHID mappings	142
10.3	Node properties	146
10.4	Edge properties	149
10.5	Memory considerations	155
10.6	Walkthrough	158
11	Graph Exploitation for Software Mining	161

11.1 Querying the compressed graph	161
11.2 Working with graph subsets	168
IV The graph structure of software development	175
12 Topological Properties	177
12.1 Introduction	177
12.2 Related work	180
12.3 Methodology	182
12.4 Topological properties	184
12.5 Discussion	194
12.6 Threats to validity	196
13 Identifying Software Forks	199
13.1 Introduction	199
13.2 What is a fork?	202
13.3 Methodology	206
13.4 Results	212
13.5 Threats to validity	217
13.6 Related work	218
13.7 Discussion	220
14 Conclusion	223
14.1 Summary of academic contributions	223
14.2 Empirical findings and impact on software mining research	225
14.3 Future work	227
14.4 Publications	229
14.5 Software	230
Bibliography	233

Thesis Structure

The first part of this thesis introduces the key concepts surrounding software development, software mining, and how the Software Heritage archive has the potential to enable research in these fields at a larger, almost universal scale. **Chapter 1** introduces the literature of software mining and how it evolved over time, from the study of small numbers of source code files to the systematic exploitation of millions of software projects. **Chapter 2** presents the Software Heritage archive, the largest and most diverse corpus of software development artifacts to date, and the ways in which it can be leveraged to achieve universal software mining. In **Chapter 3**, we construct an abstract model for software artifacts stored in deduplicated Version Control Systems (VCSs). This model is then used in **Chapter 4** to describe the data model underpinning the Software Heritage archive, a consolidated, deduplicated and canonicalized repository of all software artifacts in public software development. **Chapter 5** reviews recent pieces of software mining literature and categorizes their research needs to help narrow the contours of our roadmap towards enabling universal software mining.

The second part focuses on ways to make the data stored in the archive immediately available for software mining purposes under different formats, as a first step towards building a more comprehensive research platform. **Chapter 6** introduces the Vault, a server-side tool to assemble bundles of artifacts on request, from single source code directories to entire repositories. **Chapter 7** offers a client-side alternative to explore small quantities of source code: a virtual filesystem which presents source code directories stored on the remote archive as if they were on a local filesystem. For larger-scale research, **Chapter 8** introduces a property graph dataset which can be exploited in big data engines and on cloud analysis platforms.

The third part proposes graph compression as a way to exploit the graph of software development for resource-hungry recursive algorithms in a relatively inexpensive manner by fitting the graph on a single machine. **Chapter 9** introduces techniques for graph compression applied to the graph of software development, and benchmarks the compression ratio of the graph and its performance on a few graph traversal algorithms. **Chapter 10** explains how different types of graph property data can be attached to the nodes and edges of the compressed graph for exploitation. **Chapter 11** describes ways to make the

compressed graph data exploitable by researchers, by introducing a remote querying API as well as methods to extract small subsets of the full dataset.

The fourth part attempts, using the datasets and frameworks introduced in the two previous parts, to describe the intrinsic structure of the graph of software development as a way to understand and document its main properties and to derive research implications for scale-out analysis approaches. In **Chapter 12** we measure low-level topological properties of the graph: degree distributions, average path lengths, connected components and clustering coefficients. In **Chapter 13** we take a higher-level view and look at how software projects are organized in clusters of “forks” of varying sizes, largely due to collaborative development and code reuse patterns.

Finally, we conclude by summarizing our main academic contributions in this work, discussing the ways they contribute to enabling universal software mining and their implications for empirical software engineering research, and exploring potential future research directions.

PART I

**SOFTWARE DEVELOPMENT AND
SOFTWARE MINING**

Introduction

1.1 The rise of large-scale software mining

Software mining is a field of empirical software engineering which aims to study datasets of extant software to uncover patterns and knowledge that can help improve future software development [111]. By organizing the knowledge obtained from software mining studies, researchers can build models using statistics and machine learning techniques that can then be queried to get design insights and analytics [74], discover bugs [169, 96], or even obtain a high-level architectural view of how software components interact together [75]. Other major topics of research can range from the study of code clones [155, 142, 157, 132] to automated vulnerability detection and repair [90, 71, 97]; and from code recommenders [176, 177] to software license analysis and license compliance [170, 165].

Software mining particularly shines in the context of *software evolution* [99, 82], which studies the dynamic behavior of software as it is maintained and enhanced over its lifetime. The software engineering industry has a profound awareness of how pervasive long-lived software is in the world's digital infrastructure, and there is a clear need for research to be focusing not just on the software source code itself, but also on its dynamic evolution over time. Methodologically sound empirical studies have a particularly important role to play as the basis for improving software maintenance tools, methods and processes.

Historically, performing software evolution research was challenging and focused on small amounts of data, one software project at a time. In one seminal empirical work in the '70s, Belady and Lehman [19] studied 20 releases of the OS/360 operating system and drew some observations on the complexity growth of a large software project. This scale of study was increasing slowly up until the late '90s, with Basili et al. [17] studying 25 releases of

around 100 different software projects at NASA Goddard. In a 1999 paper, Kemerer and Slaughter [84] highlighted the inherent challenges of collecting empirical data to study software evolution: researchers had to collect data at a minimum of two different points in time, which required sustainable research efforts over a long period, or collaborating with organizations that retained useful software measurement data and development history.

These constraints drastically shifted in the past decades with the rise in popularity of Free/Open Source Software (FOSS) [156] and collaborative development platforms [42, 83]. Developers have started making publicly available a large wealth of *software artifacts*: the source code files and directories of the software projects, as well as their complete development history over time and all its associated metadata, which have in turn benefited empirical software engineering research fields such as software mining and software evolution. This was made possible especially thanks to the emergence of Version Control Systems (VCSs) [148], collaborative software development systems which track the history of development by retaining the successive states of the source code over time. They have been frequently analyzed [82] due to the rich view they provide on software evolution, and their ease of exploitation since the advent of Distributed Version Control Systems (DVCSs). The peer-to-peer approach to version control used by DVCSs makes it so that each user can retrieve the full development history locally, which allows complex development patterns [67, 69] like “branches” and “forks” to be directly embedded inside the change history.

1.2 Universal software mining

DVCSs hosting platforms have allowed researchers to easily retrieve and process software repositories for mining purposes and made the literature on software evolution abundant [77]. Yet, most present-day evolution studies still focus on the evolutive patterns of *individual* software projects, which is particularly interesting to ascertain which factors contribute to maximize software health [4].

Larger-scale studies can sometimes analyze hundreds or thousands of repositories at the same time, but their approach has generally remained limited in scope: researchers will tend to focus on some specific sampling rule, e.g., retrieving the top ten thousands repositories in a specific programming language from a particular hosting platform. This can be a methodological issue for empirical research, as the most popular repositories in the most popular platforms are not necessarily representative of development practices at large [159].

At an even larger scale, some studies have been performed on up to the entirety of a specific software ecosystem like app stores or package managers [65, 36], but those were limited to the granularity of software releases and did not study finer-grained history data

like commits.

Arguably, one of the next frontiers in the field would be to attain **universal software mining**: the methodical analysis of software at the largest scale possible down to a fine granularity of software artifacts. At present, the largest scale possible is that of the *software commons*, i.e., the body of all software which is available at little or no cost and which can be reused with few restrictions [18, 86].

Unlocking this capability would have profound implications on the field of software mining. First, it would considerably **reduce barriers to entry** for empirical studies, by removing the need for researchers to manually retrieve thousands of repositories from various sources to perform their analyses: having access to the entire body of the software commons in a single centralized location would allow them to simply request a subset of the data they are interested in to perform their analysis, and obtain it in a format that is convenient for data processing. It would also **enhance study replicability** [38, 87, 53] by providing an entry point to a static collection of already available data that can be used to rerun studies at will. In addition, it would significantly **reduce potential sources of selection bias** by offering a representative view of the software found in all different kinds of formats, on diverse more or less popular hosting platforms, tracked by a variety of VCSs. Finally, the exhaustiveness of the data would allow us to get a **detailed high-level view of the social processes and interactions** that govern software development, by looking at the global network of the programming community in its entirety rather than focusing on a particular subset.

Systematic initiatives [79, 60, 106] have been established to gather as much public VCS data as possible in a single centralized place. Two recent initiatives, World of Code [92] and Software Heritage [48, 3] go one step further and aim to build a sustainable and accessible archive of *all* the source code artifacts of the software commons, down to the level of the source code files. Even though full coverage w.r.t. the software commons is by nature a moving target for any archive, the far-reaching comprehensiveness and diversity of Software Heritage makes it a good approximation for an exhaustive corpus of public software development; notably, it covers more types of VCSs and a wider array of data sources than comparable corpuses [93].

In this thesis, I explore the ways the body of software commons stored in the Software Heritage archive can be organized and made accessible to researchers for the purpose of universal software mining.

1.3 Availability and accessibility

To understand the motivations and design decisions behind the work presented in this thesis, it is important to establish the difference between making data *available* and

accessible. The Software Heritage initiative aims to contribute to both of these aspects in different fashions.

Some of the data collected in the Software Heritage archive is already *available* inasmuch as it is at everyone's disposal by sole virtue of being public. However, this availability is often temporary, as software data sometimes get rewritten or deleted from their hosting places. Relevant knowledge is routinely lost in code hosting platforms because their main purpose is not to archive code, but rather host it. Because of this, other history rewriting patterns like `git rebase` or "force-pushes" are not observable on these platforms, but *could* be made observable by an archive which retains the past states of the repositories.

While making the data stored in the Software Heritage archive *available* is relevant to software mining research, it could be achieved simply by publishing periodical hard drive dumps of the storage, as anyone would be able to retrieve and exploit it. What we aim to achieve by building a research platform for universal software mining is to make the data not only available but also *accessible*, which means that the data should be presented in a way that is already suitable for analysis purposes. Using the data should be convenient and frictionless, as the platform only has value to researchers to the extent that its data can be easily leveraged for many kinds of experiments. Merely providing raw data dumps does not solve the accessibility problem, and what we need instead is a comprehensive understanding of the needs of researchers in order to develop reusable tools and infrastructures that can be built upon, instead of having those be constantly rebuilt for one-off purposes. Simply put, the goal is to build an *abstraction* that allows researchers to absolve themselves from the low-level technicalities of software mining to focus on their own experiments.

In the following chapter, we will describe the goals of the Software Heritage archive in more detail, as well as examine the variety of ways in which data accessibility for the sake of research fits the mission statement of the initiative.

Software Heritage: the Great Library of Source Code

2.1 Project goals

Launched in 2016, the Software Heritage initiative¹ [3] has the stated mission of collecting, preserving and sharing all the software that is available in source code form, together with its entire development history and associated metadata.

As software has become a key technology lying at the heart of our modern civilization, it has become clear that it embodies a growing part of our shared cultural, scientific and technical heritage, and as a result it has attracted growing attention from the digital preservation community. Recently, the focus of archival efforts have started to shift to preserve the software *source code*, which is where this technical knowledge is embedded, rather than executable binaries which are mostly unreadable for humans and stripped of all development insights. Developers sometimes tend to call this source code the “preferred form of modification”, emphasizing its importance as the format that human programmers directly interact with when writing software.



Figure 2.1: Logo of the Software Heritage initiative.

¹<https://www.softwareheritage.org/>

The advent of DVCSs as the state-of-the-art for source code management in software development has largely facilitated the exchange of source code, by ensuring that the full development history is replicated among each developer in a project. However, DVCSs *per se* do not guarantee *code availability*. Public repositories can disappear: they can be made private or moved; they can be targeted by (potentially bogus) copyright takedown notices; their hosting platform might shut down. Specific commits can also disappear from repositories, e.g., due to their history being rewritten. The commits will still be available from other copies of the repository, but there is no easy way to find them.

Large-scale source code archival is a crucial part of a solution to this problem, especially for the vast corpus of free/open source software. As a way to pursue its three-fold mission of collecting, preserving and sharing the software commons, the Software Heritage initiative aims at becoming three major things:

1. A **reference catalog** of all the software source code. By painstakingly finding and referencing the extant source code and all the different places it appears in, the archive aims to become a general entry point where historical and contemporary software can be found and retrieved.
2. A **universal archive** where the software is safely preserved in the long-term. There have been multiple occurrences of large troves of data being lost, be it for business-driven reasons (e.g., Gitorious, Google Code, Bitbucket), inconsiderate or malicious handling (e.g., Code Spaces) or gradual data degradation over time, which have highlighted the need for an infrastructure to protect these commons.
3. A **research platform** to enable analysis of all the source code in the archive. This implies making the data not only *available*, but also *accessible* to researchers and industrial practitioners, by providing a framework to perform experiments in ways that are convenient and suitable for large-scale data mining.

The work presented in this thesis is focused on the latter point and explores various directions to enable large-scale analysis of the software source code present in the archive. However, to understand *how* we intend to make this data accessible, we first have to look at the way the archive works: its scale, what it covers, how its data is crawled and ingested, how the archive grows over time, etc.

2.2 Software collection

2.2.1 Finding all the source code

While the Software Heritage archive aims to assemble and preserve *all* the software source code, all software projects are not equal in terms of ease of collection. Figure 2.2 shows how

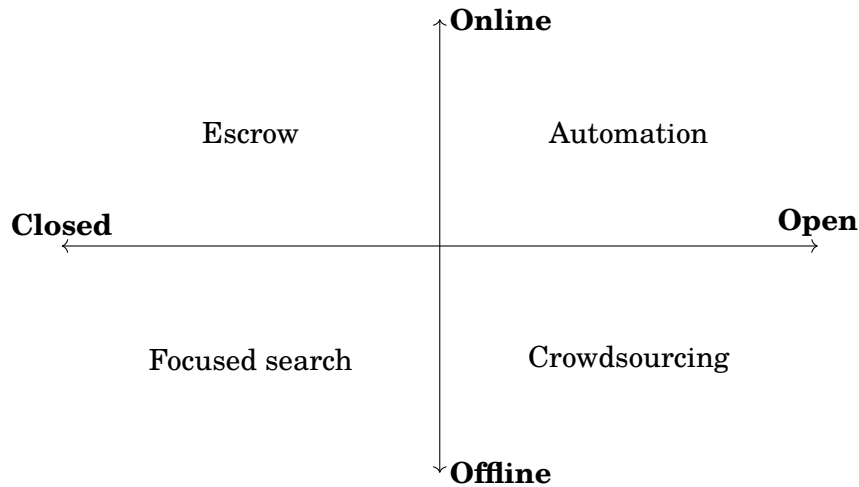


Figure 2.2: Two axes of software collection, with each quadrant its retrieval challenges

this complexity can be mapped on two axes. The Open \leftrightarrow Closed axis references the right to archive the source code, as obtaining the right to archive private repositories is more challenging than for ones already open to the public. The Online \leftrightarrow Offline axis indicates whether the software is already online in a digital format on a (potentially private) network, or whether it would require efforts to retrieve it from a physical medium, such as offline hard drives or even old source listings.

Each quadrant will entail different resources and techniques to be ingested in the archive. Naturally, the Software Heritage initiative has thus far focused its resources predominantly on the top-right quadrant, i.e., source code that is already publicly available online on code hosting platforms, due to how comparatively easy it is to automate the retrieval of large quantities of software source code. Throughout this thesis, we focus on what we call *public development*, which mainly refers to this quadrant of open and online software.

Note that this has implications for empirical research: knowledge gained from studies on the data amassed by Software Heritage will necessarily have a selection bias, as it will capture the patterns that are specifically found in public development. A general rule is that the more accessible a piece of source code already is (by being present on well-known hosting platforms), the more likely it is to be archived in Software Heritage.

2.2.2 Software origins

Software origins are abstract objects representing the places where we can find a given piece of software source code. In practice, they are often represented in the form of a URL, pointing to the web page where a given software was found.

Origins are conceptually different from *projects*. A specific software project can be found

in multiple origins, especially in the case of DVCSs where there is no central authority to be the main place of development of the software. As an example, Linus Torvalds' version of the Linux repository is hosted in at least three different origins:

```
https://github.com/torvalds/linux
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
https://gitlab.com/linux-kernel/linux
```

Additionally, developers using DVCSs can upload their own working copy of the repository they are working on, thereby creating more origins of the same software. Greg Kroah-Hartman has his own working copy of the Linux repository hosted at <https://github.com/gregkh/linux>. All in all, there can be thousands of origins hosted in different places, all pointing to the same software source code.

Origins are often found in code hosting platforms called *software forges* [150, 149]. These forges are hubs for software development, aggregating and organizing code repositories as well as their associated project metadata like issues, bug reports, continuous integration services, etc. These forges can be centralized services where any developer is free to upload their own repository. GitHub is by far the largest example of such a centralized platform, totaling 40 million users and more than 190 million repositories as of January 2020. Other forges like GitLab or Phabricator function on a more decentralized model, where the code of the forge itself is open source, and organizations can self-host their own instances of the forge to store and organize their projects.

Another source of origins is curated software ecosystems, like app stores or package manager repositories [85, 1]. For instance, Linux distributions like Debian generally maintain a set of software packages that are published online, and users can use the package manager of their distribution to automatically install or upgrade the software in their operating system. Programming language ecosystems also often have their own package managers as a way for developers to quickly install libraries and frameworks that their own software depends on, for instance the Python language uses the PyPI repository, and Javascript uses the NPM registry.

2.2.3 Archival process

Indexing every single software origin that exists in order to ingest them in the archive would be an incredibly tedious task. Fortunately, this process can largely be automated with the use of *listers*. Over the years, Software Heritage has been continuously establishing a list of software forges and package repositories where publicly available software can be retrieved and archived. Generally, these code hosting platforms provide APIs that can be used to get a list of all the projects hosted on the platform. The Software Heritage listers use these APIs to periodically get a list of all the repositories in the forge and schedule

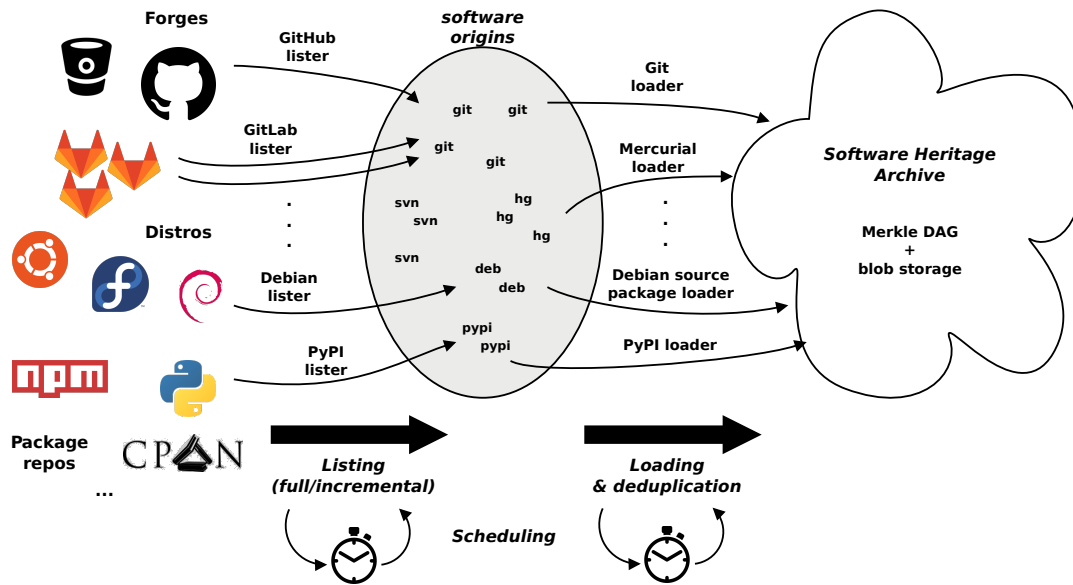


Figure 2.3: Data flow of the archival process.

them to be loaded in the archive. Some forges also support subscribing to event feeds when repositories get added or modified, which is used to schedule repositories at a finer granularity.

Of course, the software found when visiting an origin can be stored under a variety of formats: a source code repository can be versioned under one of the many VCSs in existence (Git, Mercurial, SVN, Bazaar, etc.), or even just released as a succession of tarball archives. Likewise, package managers all have specific packaging formats that they use to track package metadata and installation rules (e.g., deb and RPM files).

The role of understanding and processing the various formats of code repositories to archive is delegated to another component in the Software Heritage infrastructure, the *loaders*. Each loader is written for a specific type of VCS repository or source package, and thus all the formats supported by the Software Heritage archive have their own loader. To ingest a code repository in the archive, the loaders first retrieve it from the remote either by communicating in the protocol of the VCS, or using a simple downloading protocol like HTTP, FTP or rsync. Then, they extract all the different software artifacts contained in the repository and ingest them in the archive.

This entire archival process is shown in Figure 2.3. One important thing to note is that while the input projects are all retrieved in very heterogeneous formats, the loaders do not keep the data that way: they transform the artifacts in a *canonical format* that allows the entire archive to be a unified and deduplicated collection of software artifacts, thanks to a comprehensive data model described in Chapter 4.

2.2.4 Software notability

One of the stated goals of the Software Heritage initiative is exhaustiveness, which implies not making value judgments as to what constitutes “important” or “notable” software. At no point in this process does any filtering occur that would remove software artifacts considered unimportant or irrelevant: binary files, unpopular projects, source file in unknown languages etc. are all ingested as first-class citizens in the archive. The only limits enforced are technical limitations and those that help prevent abuse: software artifacts larger than a few hundred megabytes are considered to be outside of the scope of the archive and are not inserted in it.

A rationale for that design choice is that there is no way to know *a priori* which software is going to be notable in the future, as most software projects start small then gain popularity over time. By the time a project would have reached notability threshold, traces of its past history could have already been lost. This exhaustiveness principle allows us to follow in real time the evolution and growth of projects and the social processes that govern their communities.

One side effect of this design is that the archive contains a lot of data that is not typically considered to be software source code: homepages, datasets, configuration files, periodical database dumps, images and other assets, etc. While software forges are mostly used as a source code development platform, they can also be used as general hosting providers in some cases. Researchers should be aware of this fact when exploiting the data in the archive, and filter out these non-source code objects if desired.

2.3 The world’s largest repository of source code

2.3.1 Coverage

Software Heritage has a coverage from a wide array of sources thanks to its numerous listers and loaders and its extensible framework for adding new data sources. The loaders are capable of processing and ingesting the three most popular VCSs currently in use: Git, Mercurial and SVN. They also handle the packages of the Linux distributions Debian, Nix and Guix, as well as those from the repositories PyPI, CRAN, NPM and Packagist (respectively for the Python, R, Javascript and PHP programming languages).

The listers are configured to run on the major code hosting platforms, and the archive contains a full copy of GitHub, Bitbucket, GitLab, SourceForge and the GNU project. They also periodically crawl the repositories of hundreds of decentralized instances of Phabricator, GitLab, cgit, Launchpad and Gitea. Some projects preserved in the archive were also recovered from defunct hosting platforms such as Gitorious and Google Code, then loaded as a one-shot task, as those sources do not require recurrent listing. A more detailed breakdown of archived projects per source host is available in Section 8.5.

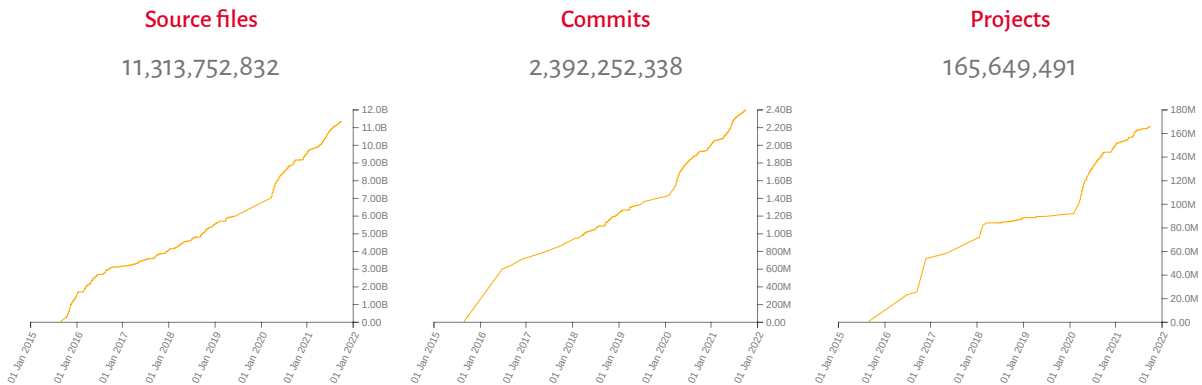


Figure 2.4: Size of the Software Heritage archive as of October 1, 2021.

This extensive and far-reaching coverage has implications for research, as it allows us to identify heterogeneous properties that appear when doing analysis across different VCSs used on different hosting platforms. Finding patterns and training models on the Software Heritage archive can be a way to ensure robustness and applicability to a diverse mosaic of collaboration and development practices.

2.3.2 Current scale and future growth

As of May 2021, the archive has ingested more than 10 billion source files from 156 million software origins. In the VCS history, we identified as many as 42 million different authors of more than 2.1 billion commits. Figure 2.4 shows the growth of the archive over time in terms of the number of source files, commits and projects stored.

Together, these source code files would weigh around 850 TiB in total. However, they are compressed in the main in-house storage, which reduces their on-disk size to around half of that. While the source code files themselves take a huge amount of space, the other software artifacts (directory trees, development history checkpoints, origins, etc.) weigh significantly less overall. They are stored in a PostgreSQL database which weighs around 7 TiB in total, including database indexes on some fields to allow for more efficient retrieval operations.

The archive is expected to grow over time as more and more listers, loaders and data sources are added to the crawling process, but maybe more importantly because of the natural growth rate of the quantity of code produced worldwide over the years. Rousseau et al. [137] project this growth using the commit timestamps over a period of more than 40 years, and find that the amount of commits in public source code doubles every ≈ 30 months, while the number of unique source code files doubles every ≈ 22 months. Zacchioli [172] further confirmed this exponential trend while studying historical gender differences in public software development, as shown in Figure 2.5. As of current projections, this exponential growth is still assumed to be sustainable at the scale of the archival project

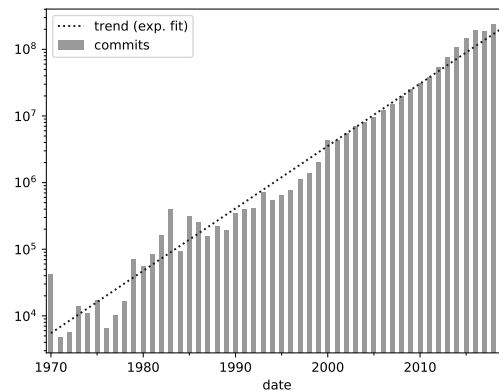


Figure 2.5: Total number of yearly commits, increasing exponentially [172]

due to the also exponentially decreasing cost of storage [137].

The massive scale of the archive as well as its projected growth in the foreseeable future highlights the architectural need for large-scale infrastructures to collect, store, analyze and make the data accessible to researchers. This has a key implication taken into account throughout this thesis, that designs of research platforms should precautionously take that growth into account and aim to be as scalable as possible to be sustainable in the long term.

Version Control Systems

The work described in this thesis is about organizing the *software artifacts* in the Software Heritage graph to make them accessible to researchers. These artifacts are abstract building blocks that represent the source code trees, development history and hosting data stored in the archive. In the next chapter, we will look at how the Software Heritage data model is a graph built on the relationships between software artifacts. However, to better grasp this model, it is good to first get a better understanding of how these objects are captured in the data model of traditional VCSs.

This chapter describes a generic model for how data is stored in most VCSs; it is very close to Git, the most popular of these systems, while being abstract enough to be independent of any specific implementation.

3.1 A simple repository model

3.1.1 Files and directories

At a fundamental level, the most basic elements in a source code repository are source code files. Developers tend to organize their code in different logical units that are then hierarchized in several levels of directories. At a low granularity, programmers separate their program in small logical blocks (like *functions* or *classes*) that perform a specific task or computation. A *source code file* is generally a collection of conceptually related functions, put together as a single logical unit. Source files that define the behavior of a logically distinct component of a program are also often logically grouped in the same directory, sometimes called a *module* or *package*.

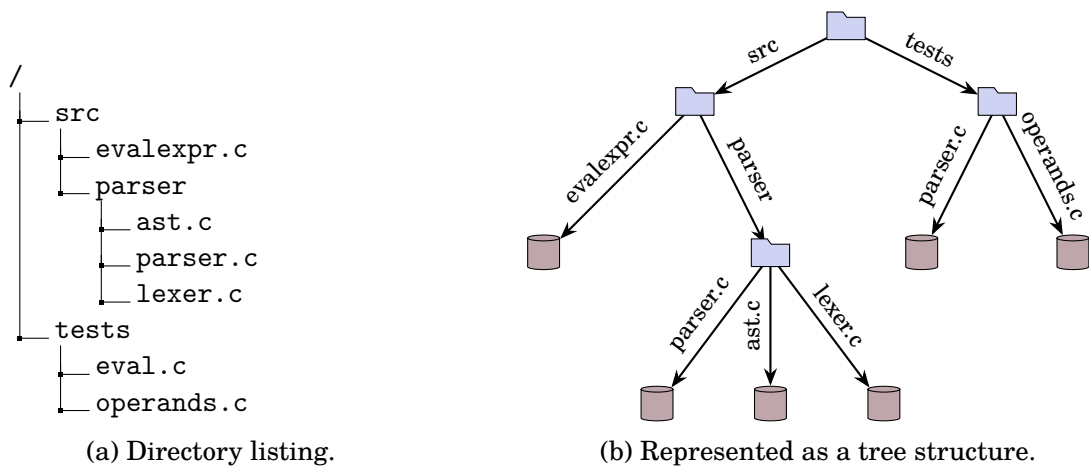


Figure 3.1: Example of a directory hierarchy for a code repository.

The example in Figure 3.1(a) illustrates a file hierarchy for a project meant to evaluate simple mathematical expressions. There is always one root directory containing the entire project, that we represent with a slash. At the root level, there is a directory containing C source code files, with some of them organized in a “parser” module, and a test directory containing tests also written in C.

The file hierarchy in this example project can naturally be visualized as a tree data structure, by representing the directories and the file contents as the vertices, with their respective names on the edges. The resulting tree is shown in Figure 3.1(b).

3.1.2 Revisions

The concept of Version Control System arose from a need to keep track of the changes happening in the code. This ability is critically important for software systems, as it allows potentially erroneous changes to be reverted without having to recall or reconstruct the previous behavior. Furthermore, the ability to retain institutional knowledge on a codebase by having the possibility to look up past source code changes is very valuable.

Traditionally, this “versioning” could be done by simply copying the software source tree to a new directory for each new version and keeping around the old versions as archives of the past state of the code. Generally, creating an entire new source tree is convoluted, making this manual process tedious and discouraging small incremental changes. In addition, it hinders collaboration between developers, who have to exchange work in progress source files and can easily lose track of which version they are developing on.

However, this basic concept of keeping track of “snapshots” of the state of the source tree at different points in time was a key design insight in the development of most modern VCSs. They introduce the notion of *revisions*, which can be conceptualized as a chain of

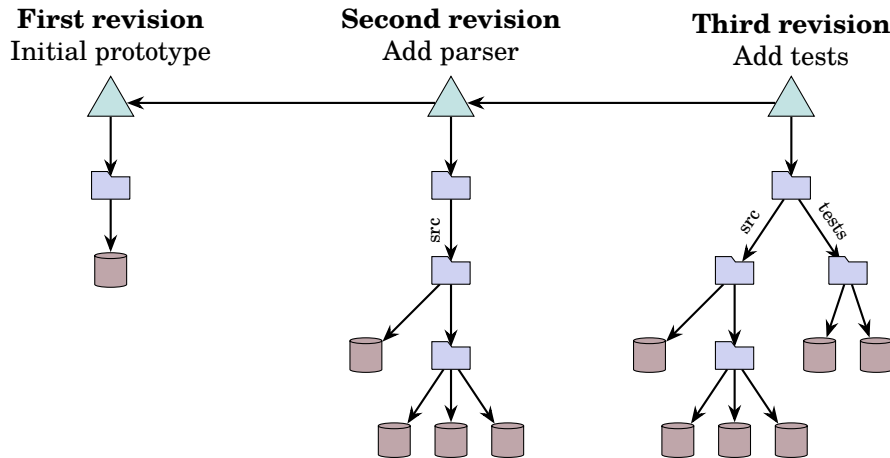


Figure 3.2: Three revisions in the example repository, forming a chain of past states of the source tree.

frozen past states of a source tree that were recorded at various points in time during development. In some systems, these successive states are known as *commits*, or sometimes *changesets*, but they all refer to the same abstract concept.

Figure 3.2 shows three successive states of the source tree from our example repository, recorded as three different revisions. The revisions all point to a full copy of the source tree, exactly as it was when its state was recorded. Each revision also contains a reference to the revision immediately preceding itself, as a way to keep track of the order in which the changes were made.¹ By iteratively walking the chain of parents from a given revision, it is thus possible to visit all the previous states of the repository on which this revision was based.

Along with its source tree and a reference to its parent, a revision also typically contains additional metadata: the *date and time* at which it was created, its *author*, and a *message* describing the changes it introduced since the last state of the code. Retaining this information directly in the VCS allows one to precisely track down when a given change was made, who authored the change and the rationale given for it. Most systems expose this information in two ways: as a *log*, where one can look at an ordered list of all the changes that were made to the source tree (or even a specific file or sub-directory), and through an *annotate* (or *blame*) command that shows the revision in which each line of a given file was modified for the last time.

¹Note that, somewhat confusingly, this implies that the arrows between commit nodes are in the opposite direction of time: each revision holds a reference to its *previous* revision, and thus points towards the *past*.

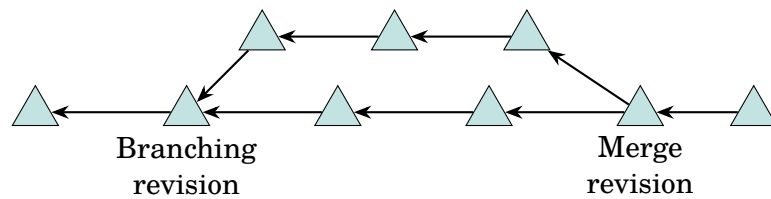


Figure 3.3: Branching and merging.

3.1.3 Branching and merging

A crucial challenge in collaborative software development is that the changes do not necessarily happen in a linear fashion. To efficiently share tasks amongst multiple people, development teams need the ability to work on the codebase simultaneously, which requires having several versions of the repository in parallel. In a VCS, this is typically achieved through *branching*: from a single revision, developers can create multiple states of the repository that can be modified separately and in parallel, which splits the revision chain. Generally, developers working on different branches will then attempt to integrate their respective changes to the main codebase by *merging* them. This merging operation combines the split chains back into a single state, called a *merge revision*.

Figure 3.3 shows an example of branching and merging in a revision history. As can be seen on the merge revision, this concept extends the simple model introduced above by allowing revisions to refer to multiple parent revisions as a way to support merging.

When working with multiple revision chains in parallel, it is useful to give them a name to keep track of their purpose. For that, we can use *branch* objects, which are simple references to the tip of a revision chain and which get automatically updated when new revisions are added to it.

In most simple development workflows, developers tend to always keep a main branch that is long-lived and reflects the current state of the project, and various kinds of additional “topic” branches which contain features that are currently being worked on in parallel. Figure 3.4 shows an example of three branches pointing to different revision chains. Note that since branches are just pointers, there is no obvious notion of a revision *belonging* to a

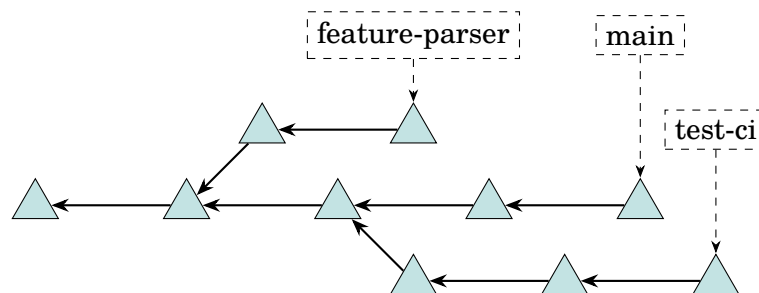


Figure 3.4: Feature branches keeping track of parallel revision chains.

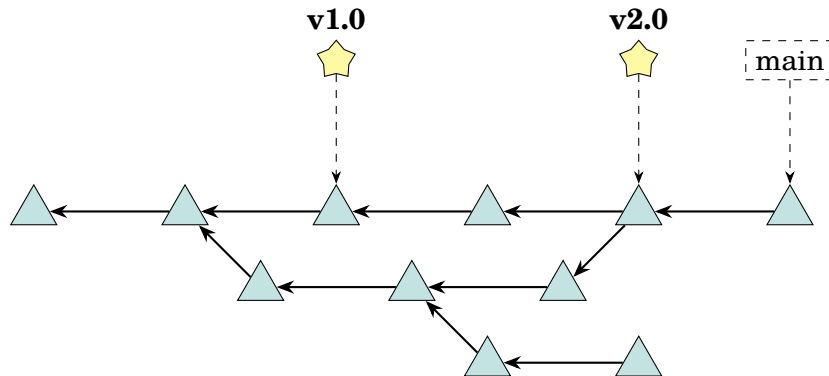


Figure 3.5: Example of a development history with two revisions tagged as *releases*, each corresponding to a new version of the software distributed to end-users.

specific branch, but rather revisions are *reachable* from one or even several branches.

3.1.4 Releases

Some revisions in the development history are particularly important because they represent specific milestones in the history of a software. This is often the case for *software releases*, where the software is distributed to its users as part of its release cycle, usually being given a numbered version like “v2.4.3”.

To be able to refer to these special revisions using mnemonic names like the software versions themselves, VCSs allow developers to add named markers, or *tags*, to some revisions. While branches already provide a way to refer to a specific revision by a mnemonic name, they are by nature dynamic and ever-changing, as adding more revisions to a branch will change the tip of the branch and thus the revision to which the branch points. In contrast, *releases* are static and always refer to one specific revision. Figure 3.5 shows an example of two releases as first-class objects in the development history that point to two specific revisions.

As with revisions, but unlike branch names which are just shallow named references, releases can also contain additional metadata: the *date* at which they were created, the *author* of the release, and a *message* describing the release.

3.2 Artifact deduplication

The simple repository model we described is general enough to represent the data stored in the majority of Version Control Systems, both at the level of the files and directories that constitute the source tree, and at the level of the development history by capturing the successive states of this source tree over time.

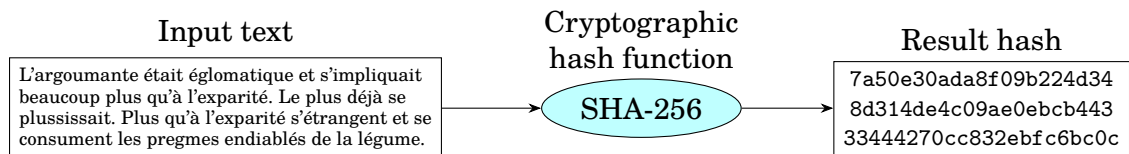


Figure 3.6: A cryptographic hash function deterministically converts data of arbitrary size to a fixed-length and virtually unique identifier.

One immediately apparent issue that arises when trying to implement this model in a naive way is the sheer amount of *duplication* of identical data across revisions: creating a new revision for a single line change duplicates the entire source tree and all the files it contains. This drawback goes opposite to one of the goals of modern VCSs, which is to encourage a high granularity of development history to isolate every logical change in a separate revision.

However, it is possible to integrate *deduplication* in the model as a way to reduce its storage footprint. This section details how we can leverage cryptographic hash functions to achieve deduplication at the level of single objects and entire subtrees.

3.2.1 Cryptographic hash functions

The basic primitive used for deduplication in many VCSs are *cryptographic hash functions*: mathematical algorithms that can map arbitrary data into a single fixed-length unique identifier. By computing the identifier, or *hash*, of each object, it is possible to check when two objects are identical very quickly by simply comparing their hashes. Because they have a fixed length, comparing two hashes is an operation with a time complexity of $O(1)$.

Due to the pigeonhole principle, the identifiers cannot be truly unique. If the resulting hash is 256 bits long, there must be at least one *collision* in any set of $2^{256} + 1$ elements, that is, at least two elements must have the same hash. For sufficiently large hash sizes however, assuming the hash function is resistant to cryptanalytic attacks, it is in practice impossible to generate a collision as it would require inordinate amounts of time and computing power, and we can thus be confident that no such collision exists. Finding a collision in a 256-bit collision resistant cryptographic hash function would take more than 2600 times the lifespan of the universe, rendering it impossible for all intents and purposes.

In some cases, the hash functions can have weaknesses that are exploited by cryptanalytic attacks. This is the case of SHA-1 [152], the hash function used by default as a deduplication method for Git, where a collision was found in 2017 [153]. Various methods can be deployed to reduce susceptibility to these collision attacks, including detecting and rejecting payloads designed to produce collisions, or migrating to more secure hash functions like SHA-256 [54] or BLAKE2 [14]. In this thesis, we always assume that cryptographic hash functions map to a single unique identifier and discard the possibility of

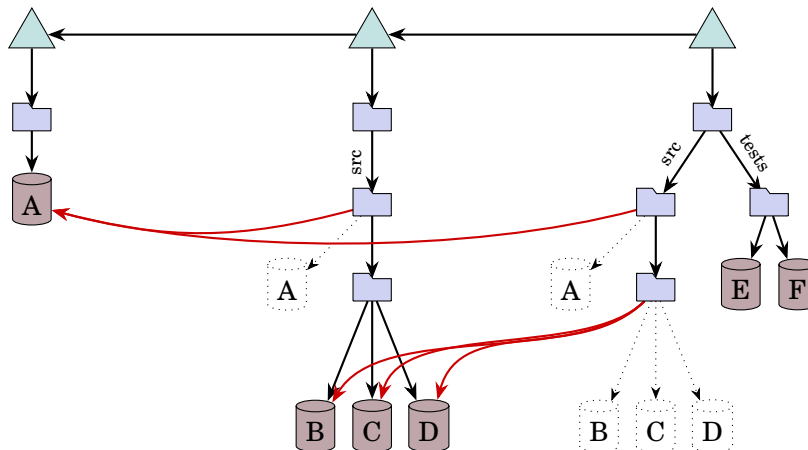


Figure 3.7: Deduplication of files with identical hashes. The transparent files are detected as duplicate objects, removed and replaced by a pointer to the already existing objects (red arrows).

collisions. These attacks do not have a big impact for our use case, as objects with colliding hashes are artificially generated for this purpose, reducing the interest of archiving and analyzing them.

3.2.2 Deduplicating single files

Armed with cryptographic hashes, it becomes relatively easy to deduplicate the identical files that get copied between each revision. We systematically compute the hash of each file and store them only once. If a new directory references a file with a hash that has already been stored, the directory simply references the file that is already in the storage rather than storing the file again.

Figure 3.7 shows an example of deduplication at the file level, based on the example repository shown in Figure 3.2. The VCS computes the hashes of all the files, which are represented as single letters for the purpose of clarity. As more revisions are added to the repository, the multiple states of the source tree will tend to reference files that have already been stored in the system. Because the hash of these files will match the hash of the previous files, they will get deduplicated and only stored once. Here, the file with hash A is present in three different directories and the files with hashes B, C and D are present in two different directories, but these directories all reference the same unique objects.

3.2.3 Deduplicating subtrees

While deduplicating the files that remain identical between successive revisions significantly reduces the storage impact of the model, each version still requires recreating and storing an entire file hierarchy. In a repository with n directories, modifying a single file

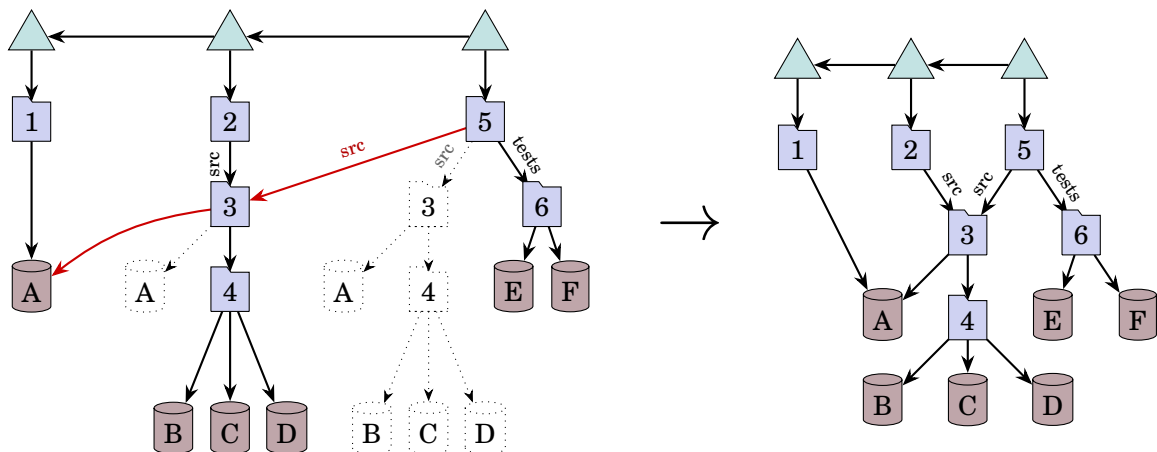


Figure 3.8: Deduplication of an entire subtree in the example repository. An intrinsic hash is recursively computed for each directory, allowing the storage to deduplicate the unmodified “src” directory between the second and third revisions.

creates $O(n)$ new directories. This is quite inefficient, considering that the majority of directories are unchanged between the two revisions.

Instead of solely deduplicating files, which are only the leaves of the source trees, a better approach could be to deduplicate *shared subtrees*: if a node and all its descendants are not modified between two revisions, they should not be copied and instead be stored as a single tree.

The challenge of doing so arises from the difficulty of checking whether two subtrees are identical, which is typically an operation with a time complexity of $O(n)$, as it requires recursively checking that all the descendants of the root of the subtree are identical. To improve this, we can instead *recursively compute a cryptographic hash* for each directory, that will be dependent on the hashes of all its children. Put simply, if a single element anywhere in the subtree changes, its hash will also change, which will then change the hash of its parent and recursively do so up to the top of the subtree.

Using this hashing scheme, if two subtrees have the same hash, it will mean that they are entirely identical, from their root down to their leaves. This again makes checking the equality of two subtrees a simple hash comparison, which has a time complexity of $O(1)$. Because these cryptographic hashes can perfectly identify the entire content of a subtree, we call them *intrinsic hashes*: in some sense, these hashes are an algorithmic way to capture the essential nature of a directory and all its contents.

Figure 3.8 shows this subtree deduplication in action. In the same vein as for file deduplication, we now recursively compute the intrinsic hash of each object in the source trees. Whenever a new directory has to be added, its hash is compared to the hashes of the directories already present in the main storage. If a matching hash is found, the new directory is deduplicated and replaced by a reference to the already existing directory.

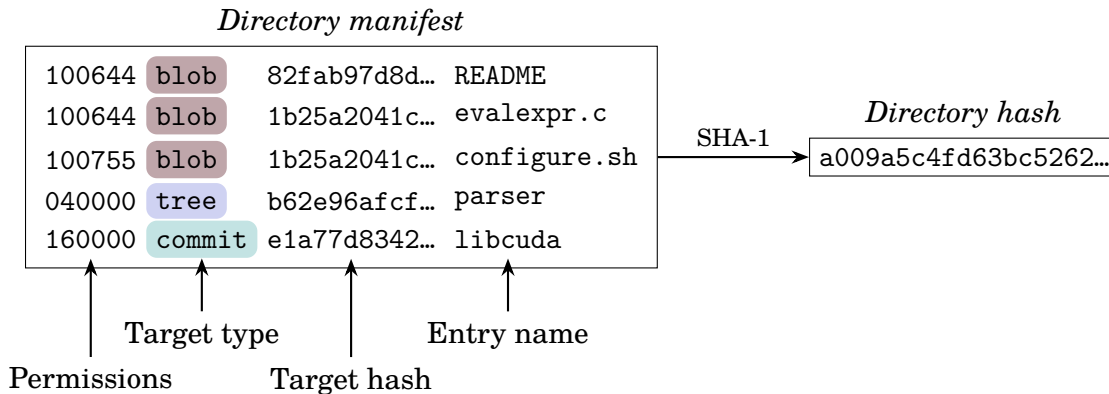


Figure 3.9: Data manifest of an example directory. Inner nodes in a Merkle DAG are identified by a cryptographic hash of a *data manifest* that contains the hash of all their children, which recursively makes their hash dependent of all their descendants.

3.2.4 Merkle trees and directed acyclic graphs

The hashing scheme we described is generalized and applied to all the objects in the graph of development history, like revisions and tags. All the software artifacts are attributed an intrinsic hash recursively computed from the objects it refers to (e.g., in the case of revisions, its parents and thus, recursively, its descendants). These intrinsic hashes will be the primary identifiers of the software artifacts, and guarantee their unicity in the graph of the VCS.

This technique of building trees identified by a cryptographic hash that is recursively dependent on its entire chain of descendants has first been described by Merkle [100]. We call *Merkle tree* or *hash tree* a tree in which every node is labelled with the cryptographic hash of a data manifest containing its key and the hashes of its child nodes. An example of such a manifest for a directory object is shown in Figure 3.9.

In our case, the structure that contains the development history is technically not a tree, but a *Directed Acyclic Graph (DAG)*: a tree can only have one parent node, which is not the case if a software artifact is referenced by two other artifacts. This can happen when merging (it creates a revision with two parents) or when sharing objects (a deduplicated directory or a blob is referenced by multiple objects). We thus call this structure a **Merkle DAG**. Note that it, by definition, cannot contain any cycles: because adding a node requires the hash of all its children to be already known to compute its own hash, a cycle would require computing two mutually dependent cryptographic hashes, which is at least as hard as finding a collision.

Like all Merkle structures, this data model exhibits interesting and useful algorithmic properties. In particular, it is worth noting that as long as two Merkle DAGs are *complete* (i.e., no node is missing), one can efficiently determine if they are identical or not by simply comparing the identifiers of all their root nodes, which in our case are snapshot nodes.

Also, in case they differ, one can efficiently identify the topmost differences by performing a parallel visit on the two graphs.

Furthermore, Merkle structures natively deduplicate all artifacts stored in it, as detailed in Section 3.2.3. As object identifiers are not assigned, but rather computed intrinsically on the content of the objects and their descendants, adding a node to a Merkle structure is an idempotent operation. Trying to add to our data model the same source code blob or tree multiple times (e.g., because it is found in multiple commits in the same repository) will result in adding it only once; the same applies to all types of objects, so a commit which occurs in multiple branches or repositories will be stored only once.

In addition to how they allow us to identify and deduplicate software artifacts, intrinsic hashes also have the side benefit of enabling data integrity checks at any time when using a VCS. If any piece of data is corrupted in the development history, rehashing the nodes in the graph would yield different hashes. This can be used to precisely track down the pieces of corrupted data and reject them, which is particularly useful in the case of DVCSs where artifacts can be broadcast through unreliable data links and from potentially untrusted sources.

3.2.5 Purely functional persistence

Because the goal of VCSs is to preserve the history of the software artifacts they track, the Merkle DAG on which they base their data model is designed to be *persistent*: updating a file or a directory in a source tree does not destroy the existing version of the subtree, but rather creates a new version that coexists with the old one. As we have seen in the previous sections, persistence is the basis upon which VCSs store successive states of the development history, and is achieved by *copying* the subgraphs that need to be modified, and *sharing* all the nodes unaffected by the update.

These kinds of data structures which are immutable, persistent and which use data sharing to minimize memory usage were systematically categorized in a seminal work by Okasaki [117], where he describes them as *purely functional data structures*.

Purely functional DAGs like the ones used in our model to store source code trees have interesting properties, notably in terms of memory complexity. As shown in Figure 3.10, creating a new revision with a single node changed only requires to recreate the nodes constituting the chain of parents from the changed node to the root of the tree. A revision changing a node in a source tree of size n and height h will therefore have a memory complexity of $O(h)$. If the tree is balanced, as well-hierarchized source code trees generally tend to be, this memory complexity will approach $O(\log(n))$.

The immutability of purely functional DAGs is also an interesting property for archival and empirical research purposes, as it allows us to preserve rewritten VCS history. When developers use history rewriting commands like `git rebase` or `git commit --amend`, instead

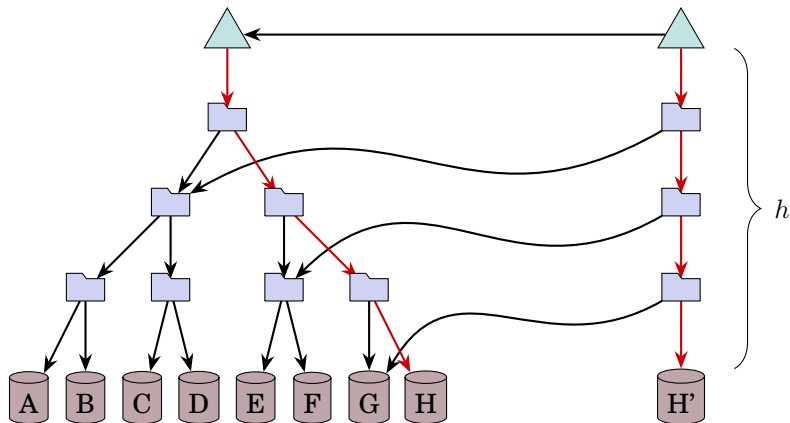


Figure 3.10: Modifying a single file between two revisions in a source tree of height h only requires $O(h)$ new nodes, thanks to data sharing in purely functional trees.

of modifying the nodes themselves, the VCS will create a new node and make the branch point to that modified node. This model makes it easy to store past states of rewritten history, simply by preserving all the immutable nodes that were added to the graph.

All in all, Merkle DAGs are an overarching keystone of the development of modern VCSs thanks to their ability to *deduplicate* common artifacts, to *ensure integrity* of the data they contain, and to their memory efficiency through *purely functional data sharing*.


The Software Heritage Graph

4.1 Canonical Software Artifacts

The Software Heritage archive is continuously ingesting software artifacts from a wide array of sources, including different VCS and package managers that each have their own internal data model. The main purpose of the Software Heritage data model is to provide a generic structure in which the individual object types specific to each system can be mapped to abstract concepts. For example, Git “commits”, SVN “revisions” and Mercurial “changesets” all correspond to the same idea of a frozen state of the source tree, and thus they are all *canonicalized* as a single type of artifact that we call “revisions”.

By stripping the implementation peculiarities of the individual data sources, the artifacts are boiled down to a purely abstract form. This unifies the representation of all the artifacts stored in the archive, which is particularly interesting for research: since all the artifacts are already stored in canonical form, we can provide a uniform interface for researchers to study artifacts coming from a variety of different sources. This isolates the complexity of handling artifacts sourced from different VCSs behind an abstraction layer; researchers can then run analyses on the abstract artifacts instead of having to deal with each specific system.

The following kinds of canonical software artifacts are supported in the data model:

 **Blobs** (or “file contents”) represent the raw content of source code files, as recorded in modern VCSs. A blob contains only the data stored in a file as a raw sequence of bytes. File names and other properties usually associated to the more abstract notion of “file” are not stored in blob nodes. Other types of nodes, and most

notably directories, attach such directory-dependent information to blobs.

Blobs are identified by a cryptographic hash computed from the full binary data they contain. An example of a blob object is depicted in Figure 4.1.

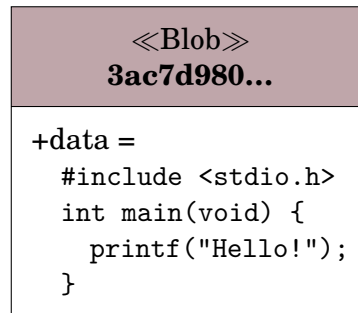


Figure 4.1: Example of a Blob object.

Directories represent source code trees. Each directory is a list of *named* directory entries, each entry pointing to either blob objects (“file entries”), directory objects (“directory entries”), or revision objects (“revision entries”). Each entry from a directory is associated to a local name (i.e., a *relative* path without any path separator) and permission metadata (i.e., a Unix permission mode such as 0o755 for an executable file). While file and directory entries are the most common to form nested source code trees, *revision entries* also exist and are used to represent sub-directories that reference specific revisions from external repositories, as it is permitted by VCS like Git (to reference so called “git submodules”) and SVN (with “subversion externals”). Permission metadata is also used to recognize symbolic links from regular files.

A directory node is identified by a cryptographic hash of a canonical textual representation of all its entries, that include the identifier of target objects. An example of a directory object is depicted in Figure 4.2.

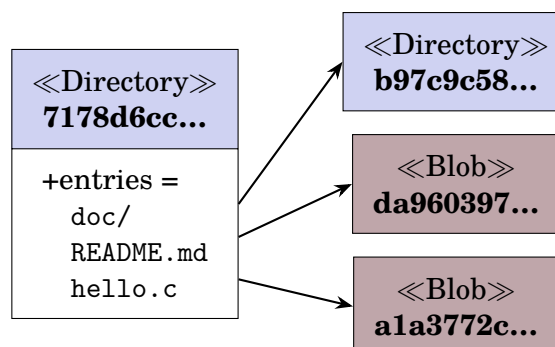


Figure 4.2: Example of a Directory object.

▲ **Revisions** (or “commits”) are point-in-time captures of the state of the entire source code tree of a project. Each revision points to the “root” directory of the project source tree at the time the commit is recorded. The following properties are associated to commit objects:

- *commit message*: a descriptive, human-targeted message explaining the reasons for the changes made since the previous revision.
- *author*: the name and e-mail of the person who authored the revision.
- *date*: the date at which the revision was authored, including timezone information.
- *committer/committer date*: two properties analogous to author/date, but capturing the person who actually committed the change (who is not always the person who *authored* it, in particular in development workflows that rely on code reviews) and when the commit happened.

Finally, each revision points to an ordered list of all its parent revisions, referenced using their own intrinsic identifier: zero parent revisions for the first revision in a given development history (e.g., first revision in a VCS repository), one parent for non-merge revisions, two or more parents for revisions that merge together several development branches. The order of the parents does not have a strict semantic meaning, it is mostly used as a guide by tools that show the difference between two revisions. The first parent is generally considered to be issued from the “main” branch that the revision is merged onto, and thus diffing tools will show the impact of this revision on the main branch by default.

Revisions are identified by an intrinsic hash of a canonical textual manifest containing all their metadata, their parent identifiers, and the identifier of the directory node denoting the root of the source tree at the time of the revision. An example of a revision object is depicted in Figure 4.3.

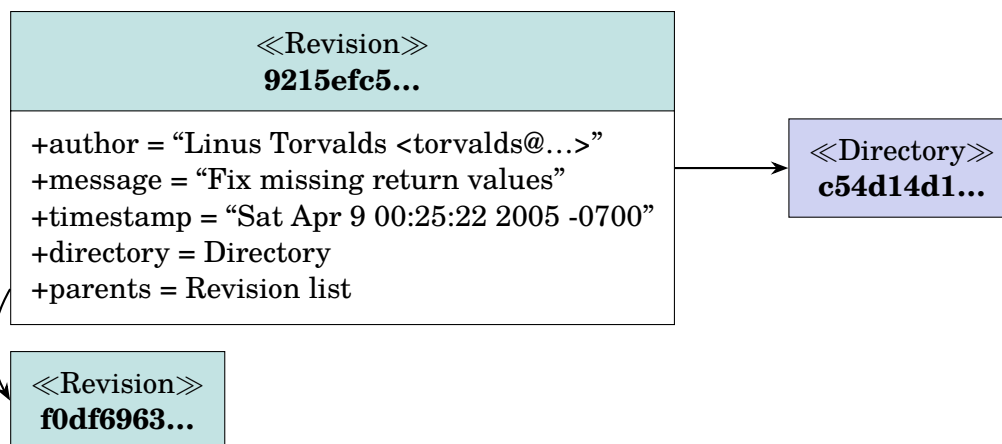


Figure 4.3: Example of a Revision object.

★ **Releases** (or “tags”) denote marker objects that label specific revisions as project milestones. These usually denote the revisions where the software is distributed to its user base, as well as the various steps of its release cycle (“alpha”, “beta”, “rc1”, etc.). These releases are marked with a specific and usually mnemonic short name (e.g., a version number such as v2.0). Aside from this name and a reference to their target revision, releases additionally contain some metadata:

- *message*: analogous to revision messages, an annotation describing the release, generally by including its full changelog.
- *author*: the name and e-mail of the person who authored the release.
- *date*: the date at which the release was created, including timezone information.

The data model also supports revisions pointing to other kinds of artifacts (blobs and directories), which is supported by some VCSs and can be found in real-world repositories.

Releases are identified by a cryptographic hash taken on a canonical text manifest containing release name, release properties, and the identifier of the revision node they reference. An example of a release object is depicted in Figure 4.4.

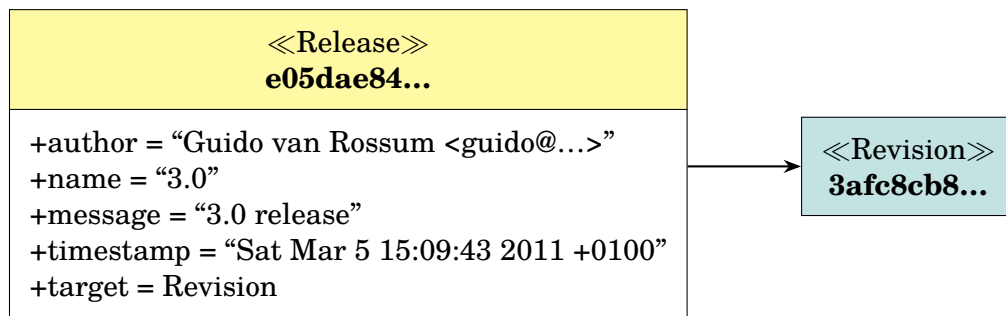


Figure 4.4: Example of a Release object.

■ **Snapshots** are point-in-time captures of the *full state* of a project development repository. Unlike revisions, which capture the state of a single development branch, snapshots capture the state of *all* the branches and releases in a repository. These artifacts are not typically part of VCSs, because they generally have no need to retain the information of the past states of the full repository, as the important historical information for developers is already tracked by revisions. However, in the case of a software archive where development history can be rewritten or even erased, we need to track the successive states of the repositories for each crawling *visit*, which includes retaining old information about which branches and tags were formerly present in the repository, and what they pointed to.

More specifically, each snapshot contains a list of references to other artifacts, which have an associated binary name (e.g., “refs/heads/main” or “refs/tags/v0.1.2”) and the intrinsic hash of the artifact they reference. Most references are either to revisions (in the case of branches) or release objects, but on rare occasions they can also point to directories and blobs, which is supported by some VCSs and also sometimes present in real-world repositories.

The data model also supports *branch aliasing*: some branches stored in snapshots do not point to a specific artifact, but rather reference another branch name in the same snapshot (e.g., HEAD→refs/heads/main). These are to be treated as symbolic links to the target of the branch they reference.

Snapshots are also deduplicated and identified by an intrinsic hash, computed from a textual manifest which associates each branch name and the intrinsic identifier of its target. Therefore, they also benefit from the advantageous space-complexity of purely functional persistent structures as described in Section 3.2.5: if two consecutive visits of the same repository show that nothing changed in the interval, the visits can both share the same deduplicated snapshot in the data model, instead of having to copy and store the set of branches and tags twice. An example of a snapshot object is depicted in Figure 4.5.

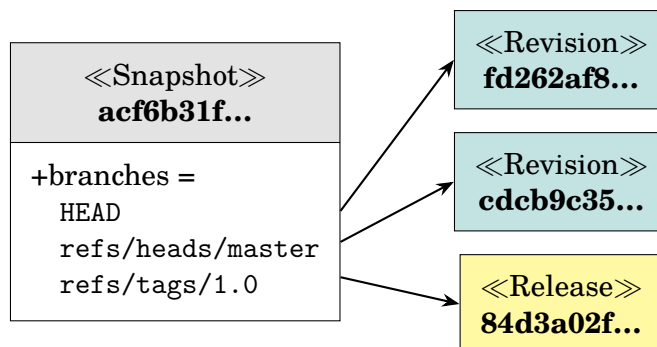


Figure 4.5: Example of a Snapshot object.



Origins are objects referencing the specific places from which source code artifacts have been retrieved to be ingested into the archive. They are represented by a URL (e.g., the address at which one can `git clone` a repository or `wget` a source code tarball.)

Origins only represent specific locations which host code, and are distinct from the more abstract notion of *projects*: a project is an entity that can relate together different development resources, including websites, issue trackers, mailing lists, and software origins. A software project can migrate its development from one origin to another for various reasons, and even migrate to a different VCS. Projects are ontologically complex notions that are not directly present in the archive; the data model is mostly concerned

about directly addressable and concrete artifacts, and *origins* are better suited for this purpose. In the rest of the thesis, we sometimes use the term “project” to refer to origins when the meaning is not ambiguous, as the latter are often our best approximation for the former.

4.2 Consolidating software artifacts in a unified archive

So far, we have covered the different abstract artifacts stored in VCSs in a way that allows us to represent an entire repository using these building blocks, notably by deduplicating them using their intrinsic identifiers. In fact, it is possible to go even further, and deduplicate these artifacts across the *entire archive*.

A key point of consideration is that software products are generally built by reusing components from other projects, rather than being mostly isolated and independent pieces of work. This organic code reuse happens through different means, either by simply copying source code files or modules between different projects, or by “forking” an existing project (i.e., starting an independent development on an existing project by building upon its development history). Modern VCSs also allow referencing external software projects to be fetched as dependencies (e.g., Git submodules or Mercurial subrepositories), which further entangles the relationships between different software projects.

VCSs and the abstract software artifacts used by Software Heritage already allow us to canonicalize and deduplicate objects inside a single software project. This principle can be generalized to deduplicate the artifacts found in several projects *across the entire archive*. The process of systematically gathering and storing all publicly available software artifacts in a deduplicated and canonical fashion is inherently a way to materialize an immense highly interconnected graph, linking together all derivative works and shared codebases.

This process is illustrated in Figure 4.6. The intrinsic cryptographic hashes of the software artifacts are used to enforce their full deduplication, consolidating them in a unified Merkle DAG with all the artifacts in the archive. In short, this means that whenever a directory, file or any artifact is referenced by multiple projects, it will get deduplicated as a single node in the graph, and will be directly referenced by all projects and artifacts that contain it. As an example, there is only one artifact for the empty file content (a blob of length zero) in the entire archive, and millions of directories reference it.

Merging all archived repositories into a single large collection of software artifacts takes all the benefits of Merkle DAG models for single repositories and leverages them for the entire archive: simple identification of unique artifacts, built-in data integrity checks, very high rate of deduplication and reduced storage costs.

In addition, storing canonical artifacts only once in the archive is also semantically useful for research. Essentially, it becomes easy to visualize and analyze code reuse: walking

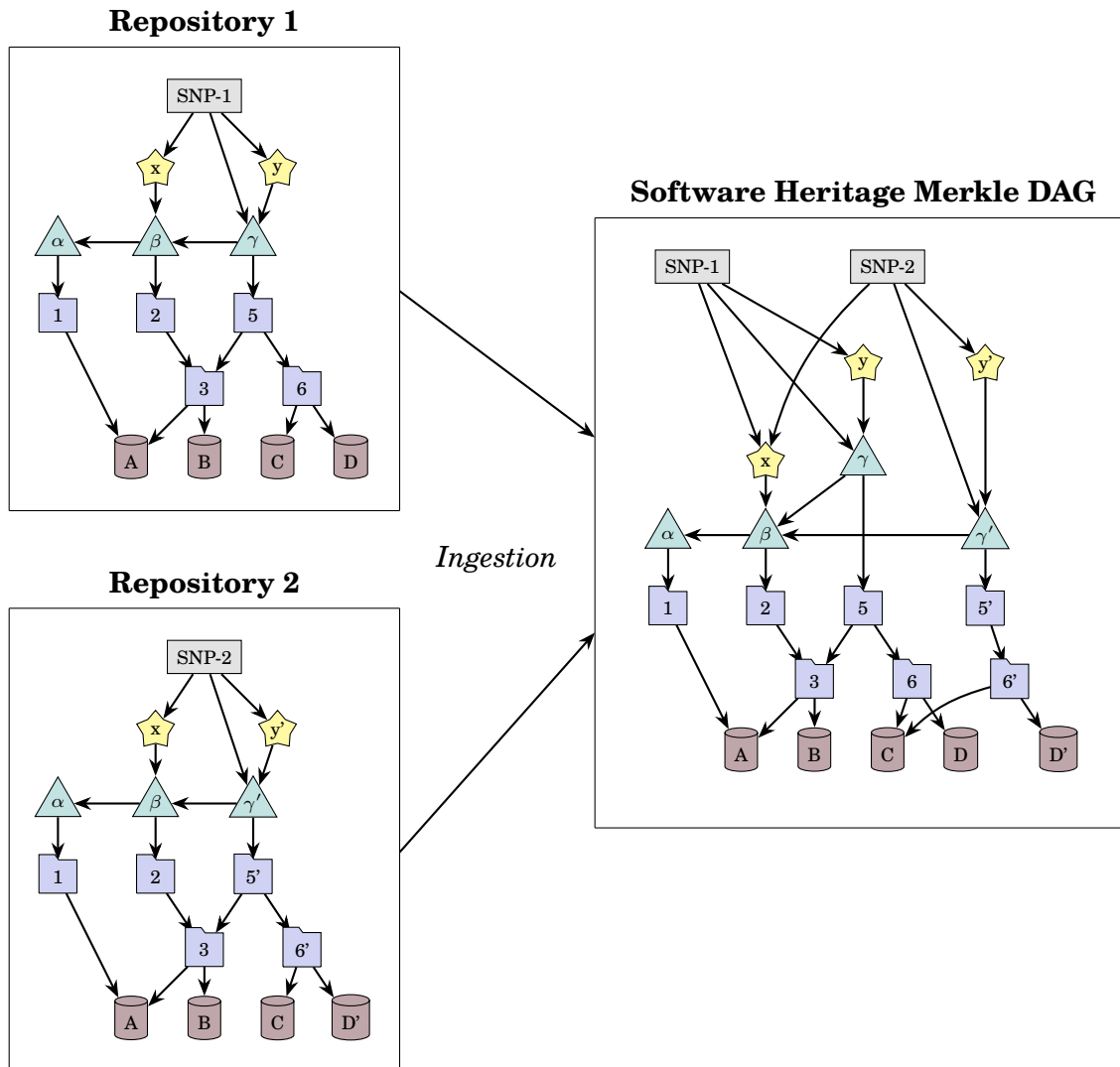


Figure 4.6: Consolidation of two different repositories in the archive by deduplicating and sharing all their common artifacts.

back the DAG can generate a list of all projects that contain a specific piece of code. We can easily draw research applications for software evolution from this capability: by having direct access to the history of changes of a specific fragment of code in the entire corpus of public software commons, it could be possible to make automatic linting suggestions based on likelihood that a given piece of code will be modified in the future. Overall, materializing these relationships between canonical software artifacts has the potential to widen the frontier of software mining by providing access to a unique corpus of provenance data and evolution history of the artifacts.

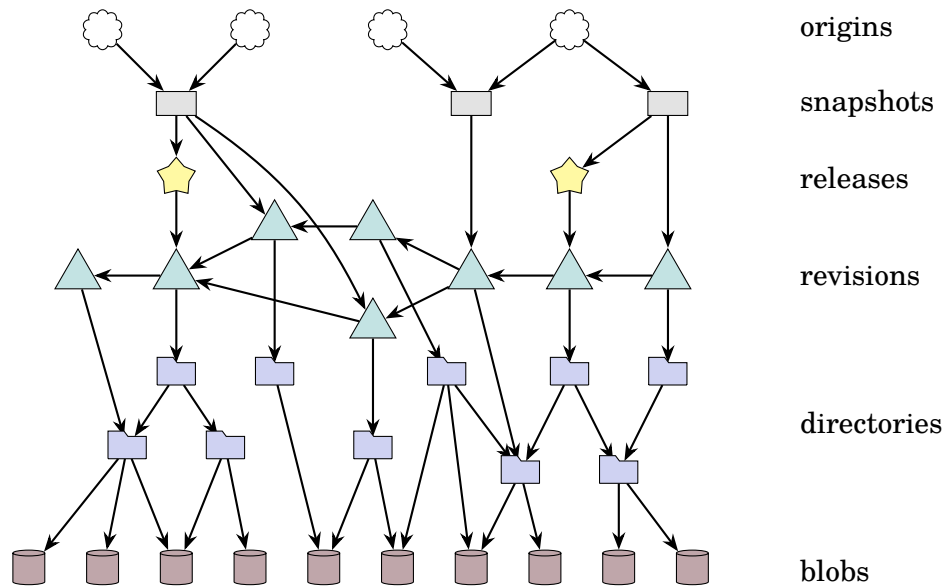


Figure 4.7: The Software Heritage Archive: a unified Merkle DAG of all the software artifacts in public development.

4.3 The Software Heritage Merkle DAG

4.3.1 Overview

After having walked through the key principles that underpin the design of the Software Heritage archive, this section describes its data model in more detail.

The archive is stored as a single large Merkle DAG, as depicted in Figure 4.7. In this graph, all software artifacts we have described correspond to the nodes, which can be of six different types: blobs, directories, revisions, releases, snapshots and origins¹. All the nodes are deduplicated and identified by a single intrinsic identifier, recursively computed from the identifiers of its descendants, which serves as the primary key in the Merkle DAG.

The edges between the nodes of the graph are derived from the relationships between the artifacts: directory entries point to other directories or file contents; revisions point to directories and previous revisions; releases point to revisions; snapshots point to revisions and releases; origins point to the snapshots that they contained in past visits. The cardinality diagram shown in Figure 4.8 formalizes the different sorts of edges in the graph and their numerical relationships: edges noted $* \rightarrow^1$ represent *many-to-one* relationships (e.g., a revision only points to a single directory, but a directory can be pointed by multiple revisions) and edges noted $* \rightarrow^*$ are *many-to-many* relationships (e.g., a directory can point to multiple blobs, and a single blob can be pointed by multiple directories).

A few observations can be drawn from this diagram:

¹Origins are technically not part of the Merkle DAG but rather only *point* to it, as their intrinsic identifiers are not computed from their list of descendants, which changes over time.

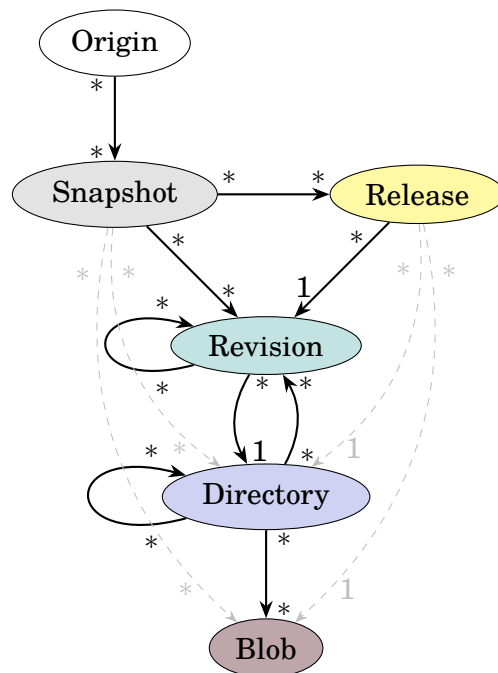


Figure 4.8: Cardinality diagram of the possible relationships between the node types in the archive, highlighting the difference between *many-to-one* and *many-to-many* edge types.

- Most relationships are many-to-many. Only release \rightarrow commit and commit \rightarrow directory relationships are many-to-one, as they can only point respectively to a single commit and a single directory.
- Commit and directory nodes are parts of recursive relationships. Commits point to their parent commits, while directories point to their children (sub)directories and files. These two node types are thus what gives the DAG an arbitrary depth, as all the other layers have a fixed maximum height. Note however that this cycle in the graph cardinality diagram does *not* induce cycles in the graph itself, because graph nodes are created bottom-up and need to know in advance the cryptographic identifiers of target nodes (which thus need to exist *before* their parents) in order to be identifiable.
- Commits and directories are also mutually recursive, as commits point to directories and directories can occasionally point to commits (for submodules/externals). This is the only case in which a node can point to a node from an upper layer in the cardinality diagram.

Note: in the wild one can rarely encounter VCS repositories that contain unusual relationships between source code artifacts, depicted as dashed arrows in Figure 4.8. For instance, Git allows you to tag blobs and directories as releases, or to use blobs as branch destinations. These are anomalies that in most cases result in non usable VCS data. We

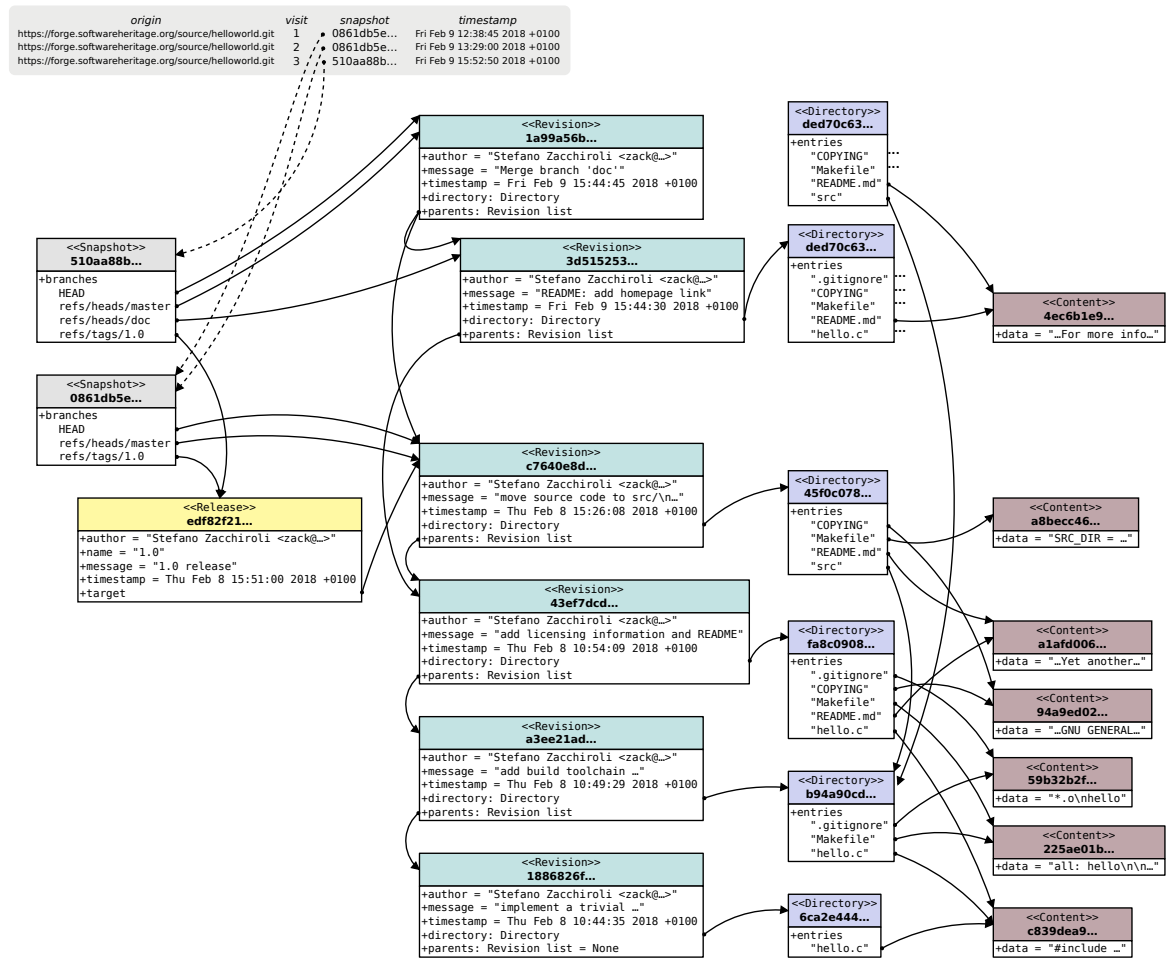


Figure 4.9: Detailed view of the Merkle DAG, with object contents, metadata fields and intrinsic hashes.

measured these occurrences in our corpus and verified that they are rare and unconventional (less than 0.0004% of the total number of edges from releases and snapshots) and thus, even if they *can* be represented in the Software Heritage data model, we generally exclude them from our discussion for the sake of simplicity.

4.3.2 Layers

While each relationship can be analyzed independently, it is useful to regroup the nodes and edges of the Merkle DAG by type into logical layers that are conceptually meaningful. To that end we define the following subgraphs of the Software Heritage graph:

- *Full corpus*: the entire graph of public software development
- *Filesystem layer*: subset of the full corpus consisting of blob and directory nodes only, and edges between them

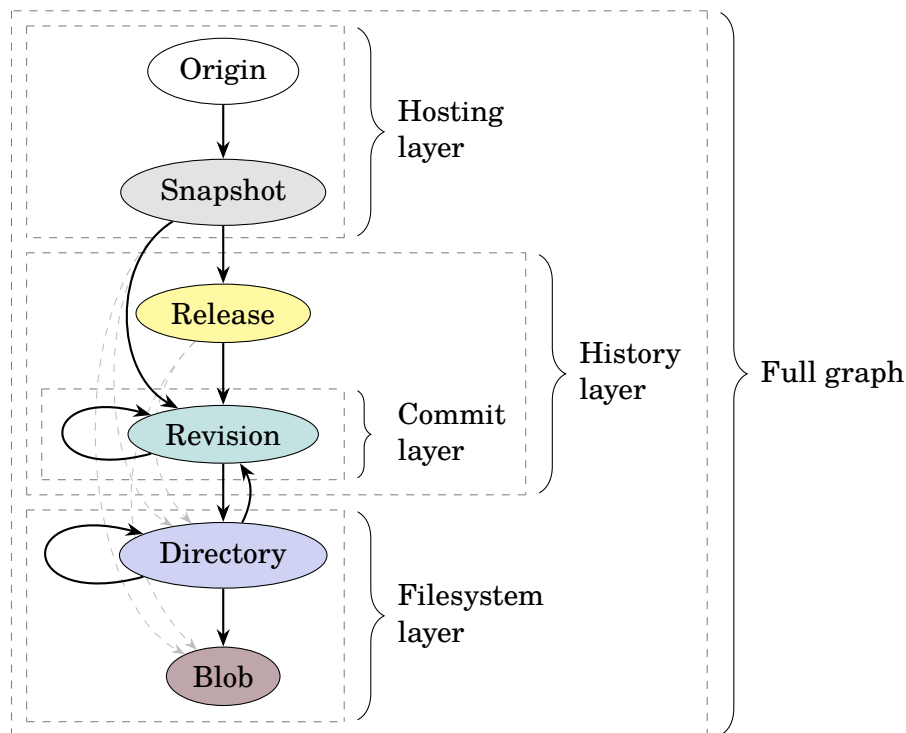


Figure 4.10: Logical layers used to refer to specific subsets of the graph.

- *History layer*: subset of the full corpus consisting of revision and release nodes only, and edges between them
- *Commit layer*: subset of the history layer consisting of revision nodes only, and edges between them
- *Hosting layer*: subset of the full corpus consisting of origins and snapshot nodes only, and edges between them

Figure 4.10 shows the various layers, which types of nodes belong to each of them, as well as how edges connect them. In later chapters, we will study properties of both the full graph of public software development and of the specific subgraphs named above.

4.3.3 Persistent identifiers

The intrinsic hashes used to identify the nodes can be used to provide a standardized and portable way of referring to the artifacts in the archive [46]. This is achieved by Software Heritage Persistent Identifiers (SWHIDs), short strings that identify a single object and which are guaranteed to remain stable over time. These identifiers were introduced by Di Cosmo et al. [47] and are documented in the Software Heritage development documentation². SWHIDs are represented in the following basic syntax:

²<https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html>

```
swh:<scheme version>:<object type>:<object id>
```

The current version of the identifier scheme, stored in the second field, is 1. The third field contains a three-letter code for the object type (snp for snapshots, rel for releases, rev for revisions, dir for directories and cnt for file contents). The last field contains an intrinsic hash computed with the SHA-1 algorithm on the content and metadata of the object and encoded in hexadecimal.

Here are a few examples of SWHIDs and the objects they point to:

- `swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2` points to the content of a file containing the full text of the GPL-3 license.
- `swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d` points to a revision in the development history of the Darktable photography application.
- `swh:1:snp:c7c108084bc0bf3d81436bf980b46e98bd338453` points to a snapshot of the entire Git repository of Darktable as of May 4, 2017.

SWHIDs have interesting uses for research replicability: studies citing software by URL can be subject to *link rot*, which could make it harder to run the software again as a way to independently verify the study results. Because SWHIDs are persistent, using these identifiers to cite scientific software is a way to ensure that the citation remains unambiguous and resilient to URL changes. SWHIDs can be computed independently and do not require a centralized authority for allocation, as they are solely computed from the intrinsic properties of the object they refer to.

4.4 Implementation

Although a full description of the physical infrastructure powering the Software Heritage archive is out of scope for this thesis, a cursory understanding of some of its core systems is needed for later chapters, as we present frameworks and tools built upon these foundations.

At a physical level, the archive is stored using a few different technologies, due to the differences in technical requirements for storing the different layers of the graph [48]. Most notably, the source file contents require orders of magnitude more space (≈ 850 TiB as of May 2021) than the rest of the archive and are thus stored on dedicated key-value object storage systems, keyed by their intrinsic hash. In the infrastructure hosted in-house by the Software Heritage initiative, the object hashes allow for efficient horizontal sharding, by assigning hash prefixes to specific servers. This technique is used to trivially implement a reasonably performant and redundant storage. Key-value object storage systems are an almost universal primitive in cloud offerings, which allows Software Heritage to store an entire copy of the blob layer in two different clouds: the Microsoft Azure Blob Storage and Amazon AWS S3.

On the other hand, the upper layers of the archive require significantly less storage space (≈ 10 TiB) but have different functional requirements, as they have more associated metadata (author names, revision messages, children nodes, ...) that need to be made searchable and joinable. This part of the graph is stored in a Relational Database Management System (RDBMS), where nodes and edges have a few associated relational tables that describe it. These systems can be leveraged to build indexes, e.g., on the hashes stored in the different fields, which can then be used to randomly access specific nodes and edges, and to efficiently *join* tables together to combine data from multiple node types by following their links.

This approach can have scalability issues, as inserting data in traditional RDBMSs gets exponentially slower as the tables grow. Other storage systems for the graph are under study to circumvent this problem, notably the use of key-value document storage systems like Cassandra or ScyllaDB.

One last relevant component is the *journal*, which serves as a synchronization pipeline across the infrastructure. In order to keep the variety of storage backends (object storage, database, document stores...) consistent together, the journal acts as a persistent log of the archive, where all the ingested objects are pushed in order. All the backends and their replicas can then subscribe to the journal and read its messages to keep themselves up-to-date with the current state of the archive.

Towards Universal Software Mining

As the field of software mining is constantly growing, notably for the purposes of assessing and understanding the dynamic evolution of software projects, there is a clear interest on the part of Empirical Software Engineering (ESE) researchers and industrials in accessing the data stored in the Software Heritage archive.

The main direction of the work presented in this thesis is to make large-scale analysis of this immense body of data possible in an efficient manner. We limit our scope to the software artifacts themselves, as studying external project management metadata (issues, CI, mailing list discussions, etc.) is a largely different problem, although relevant to the field in its own way.

For that purpose, we first cataloged different use cases by looking at software mining studies and qualitatively assessing which types of data they were extracting for their own purposes. This gives us a general idea of the queries researchers would want to run on the archive, given the opportunity to do so.

Of course, we cannot limit ourselves to use cases we find in the literature, as the scopes of past studies are endogenous to the infrastructure researchers had at their disposal while conducting it. Part of our goal in making Software Heritage a platform for universal software mining is to make new research opportunities available by allowing researchers to run studies that were not possible before. Therefore, in addition to the use cases we find in the literature, we also want to understand general interests in future research directions as a way to expand this horizon of possibilities.

This chapter is based on an article [127] accepted at the 17th Belgium-Netherlands Software Evolution Workshop (BENEVOL 2018).

5.1 Cataloging research needs

5.1.1 Literature Review

After various preliminary conversations with researchers working in the field and having a familiarity with the software mining literature, we have preconceived notions of how studies on software repositories are designed, which generally happens in a two-fold process. The first step is to precisely *select* the software artifacts that are relevant for the study using a variety of criteria: studies on Java source code will select files ending with the `.java` extension, studies on large projects will require a given number of revisions in the project, etc. Once the artifacts have been selected, researchers can *analyze* this new corpus by running custom-made tools and processing algorithms.

To validate this understanding of the literature more rigorously, we systematically reviewed 54 papers published in the *16th International Conference on Mining Software Repositories (MSR 2019)* [154]. In each paper, we sought two important pieces of information: (1) what were the *selection criteria* used to narrow the study to work on a specific set of software artifacts, and (2) which types of *analyzed data* were required for their analyses.

Not all articles in our corpus were relevant to improve our understanding of software mining studies and had to be removed from the review. In particular, we excluded:

- 6 papers which were not actually mining software repositories (tool papers, polling study, meta-analyses, etc.);
- 4 papers which used software artifacts corpuses that cannot be reconstructed from a software archive (proprietary datasets, custom-built datasets, manual example generation, etc.);
- 3 papers where no experimental methodology was described;

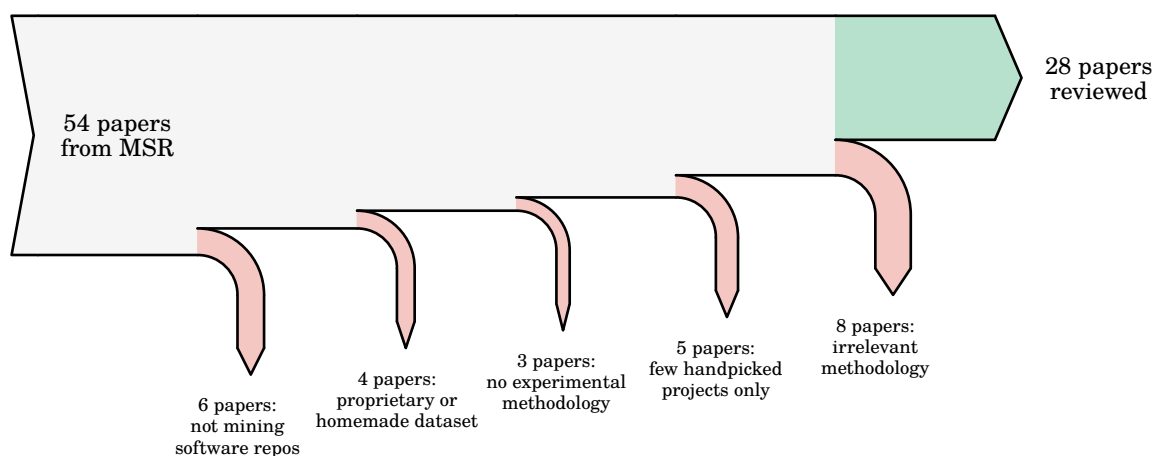


Figure 5.1: Sankey diagram of the selection process in our literature review.

Table 5.1: Selection criteria found in MSR literature review.

Criteria	Number of studies
a. Number of repository stars	12
b. Programming language of repository	10
c. Matching string or pattern in file contents	5
d. Pre-established list of URLs	4
e. File names and extensions	4
f. Natural language used in the project	2
g. Number of commits or releases	2
h. Number of forks	2
i. Random sampling	2
j. Commit dates	1
k. Licenses	1
l. Number of issues or pull-requests	1
m. Number of contributors	1
n. File sizes	1
o. Lines of code in source blobs	1

- 5 papers which were analyzing a specific set of handpicked repositories, and in which the analysis was only relevant for these particular repositories;
- 8 papers which had an irrelevant methodology (demonstrations of tools for building software corpuses, extensive reliance on context such as code snippets from development chats or web pages, etc.).

In total, we removed 26 papers as shown in the Sankey diagram in Figure 5.1, and included the 28 remaining papers in our review.

5.1.2 Selection criteria

This review allowed confirmation of the two steps previously identified as the general research pattern followed by most empirical studies: criteria-based selection of a subset of artifacts, followed by analysis of selected artifacts. Table 5.1 shows a detailed tally of the different criteria we observed in the study designs.

While some of these criteria would be easy to implement in a research platform based on the Software Heritage archive as a way to select a subset of artifacts for a given study design, others would require more work. They can be categorized in several groups, ordered by difficulty of implementation:

- *Direct Addressing* (crit. d): when the study methodology includes a pre-established list of artifact identifiers (e.g., repository URLs or revision hashes). As long as a mapping between the unique identifiers and the objects in the graph is properly maintained, selecting objects based on their identifiers should be trivial.

- *Artifact Properties* (crit. e, g, j, n, m): selection is often done on properties that are directly stored as part of the artifacts themselves: commit authors or dates, file names etc. This data is already present in the database and should only require indexes that allow to return all objects matching a given property, assuming these indexes are sufficiently performant.
- *External Metadata* (crit. a, l): where the selection uses metadata that is not part of the software data tracked in the archive itself (contextual popularity information such as number of GitHub stars or pull requests). While external metadata is out of scope for this thesis, we should acknowledge the frequent need for researchers to filter on external and contextual metrics, and leave open the possibility of linking the objects in the archive with a metadata store.
- *Derived Metadata* (crit. b, f, h, o, k): for filtering on properties that are not directly part of the data, but which can be derived from them. By running a programming language recognition tool on the file contents, we can index blobs by their detected programming language and allow users to filter objects on this criterion. This requires additional indexes as well as processing pipelines to generate this derived data. While some derived data can be easily generated (e.g., counting the lines of code in a file), the process can be more involved for others: filtering on the number of file changes introduced in a given commit requires computing the difference between the directory subtrees of all successive commit pairs.
- *Code Search* (crit. c): where the selection happens on the contents of the files themselves. Finding all files containing a specific string or pattern is significantly harder than filtering on other artifact properties. As we noted in Section 4.4, the total size of the file contents is around two orders of magnitude larger than the rest of the database, dramatically increasing the infrastructure requirements of indexing. In addition, the problem of full-text search in source code is a notably hard one in this context, as it requires a high level of expressiveness to be able to match on specific tokens or API uses, and the source code files are written in arbitrary programming languages and cannot necessarily be properly parsed to an AST form.

Each selection criteria described here defines one specific filter on the objects, but researchers generally combine these filters together to narrow down their corpus of study. Therefore, it is not sufficient to simply provide a way to select the data based on one of these filters; the partial filters must be able to be joined together relationally to ensure enough expressiveness in the selection phase.

Table 5.2: Types of analyzed data found in MSR literature review.

Data type	Number of studies
a. File contents	15
b. File names	10
c. Commit graph and commit properties	9
d. Commit diffs	7
e. Authors and community graph	5
f. Dependency data	2

5.1.3 Analyzed Data

After having selected the relevant objects of study using specific filters and heuristics, researchers perform the analysis that constitutes the core of their experimental design. In order to make this possible in Software Heritage, it is necessary to understand which categories of data need to be made accessible.

In our review of MSR studies, we identified 6 categories of data that were frequently analyzed in mining studies, shown in Table 5.2. By far the most common type of analyzed artifacts are file contents, highlighting the clear need to make the source code files easily accessible. While there are some studies which only rely on the upper layers of the artifact graph like community detection algorithms, research on the source files themselves is evidently of foremost interest in software mining.

Studies generally also require basic artifact properties like file names, commit messages, dates and authors, etc. Those fields should be relatively easy to provide from the Merkle DAG of the archive. Some studies also depend on the topology of the subgraphs themselves: commit chains when analyzing software evolution, or source subtrees when looking at file hierarchies. This is important to keep in mind, as it means we cannot always simply provide an unorganized flat stream of artifacts, and need to leave open the possibility of running algorithms on the graph which links them together.

Here again, studies will tend to use derived data which can be computed from the data in the archive. The most prominent example is commit diffs, as a lot of studies look at the *changes* that were introduced by each commit.

Some types of derived data can pose additional challenges. For instance, computing dependency graphs will involve a different process for each programming language, or even each package manager. For complex cases such as this one, our focus should be to make sure all objects required to compute the derived data can be provided, so that researchers can generate the corpus themselves.

5.2 Functional requirements

A cursory review of the literature gives us a good understanding of how *current* software mining studies are designed and the main categories of data they exploit, but it should not necessarily be our sole guide for the capabilities we want to provide in a research platform. Current research directions are said to be *endogenous* to capabilities offered by mining platforms: one cannot evaluate the needs of researchers solely by looking at what researchers *do*, as some data mining questions might be left unexplored not because of a lack of interest but due to the lack of an exploitable dataset or platform to study them, which is precisely what we aim to provide. In consequence, while the current literature reflects the state of what is currently achievable for researchers, one of our goals is to open new possibilities by leveraging the unique properties of the archive: notably its canonical format, diversity, and universality.

To that end, the efforts in building a mining platform in Software Heritage should primarily focus on capabilities that would be uniquely impactful on the state of the art when applied to this particular software development corpus. Mining studies on handcrafted datasets, small number of repositories or using metadata more easily accessible in other platforms (e.g., dependency graphs, call graphs), while important to the field, might not be best suited for Software Heritage, at least in its first iterations. As a general rule of thumb, a major strength of running experiments on the archive is that the universality of the corpus can allow researchers to analyze public source code exhaustively. This benefit is mostly present when the processing step can largely be automated; studies requiring a lot of manual curation can leverage the archive only to a minor extent and will not be our main focus.

As an initial target for the research platform, we identified five main categories of research requirements for capabilities that we want the platform to have:

Content access. One of the most common requests is to obtain a set of file contents stored in the archive based on some specific criteria. Those requests are usually made for the purpose of analyzing the code itself: code patterns recognition, language detection, static analysis, malware detection, and so on. Occasionally, those requests also require some data preprocessing to be applied to the file contents before the analysis (comment removal, data or binary strings removal, etc.).

Filtering on metadata. It is generally useful to filter the query results depending on some criteria on the file metadata. This metadata can either already be present in the archived repositories (file extensions, file names, file sizes, directory depth...), or derived from the data (MIME types, detected programming language, license...). This metadata has to be precomputed and indexed along with the files.

Content search. The ability to perform full-text search for specific code fragments or patterns is very useful to focus computations on the relevant parts of the code, and it

requires an up-to-date full-text search index.

History graph: The ability to access the revision history graph along with its associated metadata (authors, commit messages, etc.) and being able to examine the relationships between the different objects in the revision graphs is imperative to analyze not only the software itself but its evolution through time. In the context of Software Heritage, these relationships are also captured across different repositories: forks point to the same ancestor nodes, directories that were moved from one repository to another point to the same object, etc.

Provenance indexing: While the software DAG works top-down (the nodes only point to their children i.e., their content, but never their parents), it is also sometimes necessary to be able to list the different parents that point to a specific object. There is a growing body of literature in empirical software engineering regarding the ability to track the lineage of a software artifact at various granularities [6, 64, 62, 137]. There are multiple applications for this: find the possible file names of a file, the different repositories that contain a code fragment, a directory or a revision, etc. These traversals on the transposed version of the DAG are known as “provenance” queries, as they allow us to find out the sources of specific objects.

Of course, the platform should be flexible enough for complex queries that combine multiple of these capabilities together. For instance, it should be possible to find all file contents referenced from a file with a specific extension and which contain some specific function name, or to search nodes in the history graph of a repository while filtering on their associated metadata.

5.3 Challenges

5.3.1 Data volume

Most of the challenges which stand in the way of providing some form of access to the entire corpus directly stem from the sheer size of the software archive. In most cases, allowing people to locally retrieve a large chunk of the archive to perform local computations is very impractical at the Software Heritage scale, both from a network transfer perspective and for the undue burden it would cause on local storage requirements.

Handling the file contents of the archive requires a lot of resources and expertise. The total size of the blobs (currently ≈ 850 TiB) require large amounts of storage capacity, and the blobs cannot easily be stored on a single machine using consumer-grade storage. The unusual size distribution of the blobs, whose median size is approximately 3 KiB, also makes it challenging to use industry-grade storage solutions, as they generally are not designed to store a large quantity of very small files. On conventional systems, some limitations on inode management may apply. Other distributed storage solutions like

Ceph cannot easily handle a large amount of small files (because of the per-file overhead needed for replication [43]), and require some form of custom content packing to take place beforehand.

Compressing the blobs works well to reduce the size with a compression ratio of ≈ 2 (estimated on a random sample of about 1% of the archive). Further techniques based on “packing heuristics” to compress similar blobs together should be investigated to reduce their size even more.

The size of the Merkle DAG itself is more reasonable (around 6 TiB when stored in the relational database format), but using it efficiently often requires various indexes, which significantly increases its size on-disk. Moreover, some intensive processing on the graph itself could require having it stored directly in main memory, which is difficult to achieve on standard machines that have orders of magnitude less RAM.

Even if the recipient of the dataset already has the storage capacity and expertise to handle such a vast amount of data, transferring it through the network is impractical and expensive. Sending the whole dataset through a connection with a speed matching the common industry standard of 1 Gbps would take more than 60 days, assuming the absence of sequential overhead between fetches.

Of course, some researchers do not want to analyze the entire corpus but rather a subset of specific repositories, as we observed in Section 5.1.1. Even so, this poses another challenge: the price to pay for deduplication is that all artifacts, even from relatively small repositories, are scattered around in the archive. Collecting small and sparsely arranged files from a distributed storage is generally less efficient than mining a locally available repository that is stored in a very compact and efficient way.

5.3.2 Representation mismatch

Researchers and data scientists usually try their experiments by prototyping on small sample sizes, before reaching out to experiment on larger datasets. Doing so, they generally use tools that are fit for specific data representations, and thus they often expect the data to be presented in specific formats. One of the challenges of making software analysis accessible to them is to help them transform the data from a format well-suited for *archival* to a format suited for large-scale *analysis*.

Files and directories contained in a specific revision are usually expected to be represented as on-disk filesystem trees, so the children of the directories can be directly accessed through the primitives of the filesystem. In the Software Heritage archive, the deduplication requires this to go through an additional index of the hashes of the directories. The interface therefore has to provide a utility to “flatten” the compact representation into a more classical directory structure, although doing so systematically would invalidate the benefits of deduplication.

For the revision graph itself, there is no current standard of representation, so most of the research experiments thus far have worked on tool-specific representations (often depending on the version control system used). While there is value in providing a universal representation for commit-level software evolution from different sources, it is still important to provide a data representation that does not stray too far from what those domain-specific analysis tools usually expect.

5.3.3 Provenance mappings

Providing a way to look up the different places where an object can be found (i.e., its *provenance*) is a hard problem, because of the combinatorial explosion of ancestry mappings. A single file content can be found in thousands if not millions of origins. There is a difficult balance to find between reducing the size of the mappings using intermediate objects in the relationship as layers to compress the volume of edges, and reducing as much as possible the amount of indirections that require index hits for performance reasons.

Moreover, maintaining this (bottom-up) provenance index is harder than its top-down counterparts, since there is no way to know all the objects of the relationship a priori, and thus represent them with an intrinsic hash for indexing. These mappings will therefore have to be dynamic, meaning a provenance query for a given software artifact will give different responses as more snapshots get archived.

5.3.4 Repeatability and reproducibility

An important part of scientific experiments is reproducibility, which is something to keep in mind when making a very large and constantly-evolving dataset available for research applications. While intrinsic hashes guarantee full consistency of the data at the snapshot level, it might be useful to provide a way to describe the state of the *entire archive* at some point in time. If we are able to reconstruct a previous state of the archive from a timestamp, including this timestamp along with the experiment methodology will allow it to be repeated on the exact same dataset as when it was performed for the first time. That way, an experiment can be *repeated* by performing it on the timestamped state of the archive, and *reproduced* by performing it on a different dataset.

The usual way to get an intrinsic identifier of a Merkle DAG is simply to hash the list of its roots. However, it does not work in this case because the graph is incomplete: the Software Heritage DAG can have a lot of holes (missing revisions, files, etc.) that can be added or removed without changing the intrinsic identifier of the nodes, relying solely on those hashes is not sufficient to reliably obtain a hash of the archive as a whole.

A better way to achieve this would be to use the journal described in Section 4.4 by adding timestamps to each inserted object, then read all the objects from the journal up to the archive timestamp.

5.3.5 Expressiveness

Researchers who want to run analyses on the Software Heritage dataset will perform *queries* on the archive to describe the computations and the part of the archive they will be run on. Running these queries on the archive will require an API that can express these different use cases.

The expressive power of the query language determines how easy it is to use the different data selection features, computation primitives and result aggregations when running data selection queries on the dataset. The semantics of the language have to provide ways of representing and combining those different operations so that the breadth of computations that queries are able to represent is as wide and generic as possible for the use cases we identified.

5.4 Roadmap

A preliminary step of this work is to make the dataset available in some format suitable for scale-out analysis, so that Software Heritage and other researchers can try out a few experiments and document their own use cases. Some public cloud computing providers like Amazon Web Services or Google Cloud have public dataset programs, on which we can make the Software Heritage dataset publicly available without bearing the cost of the storage.

While allowing people to run queries directly on a public cloud instance is well-suited for one-off experiments, it does not always work well for more intensive use cases. Researchers having access to hardware resources and software engineering skills might find it more cost-efficient to run their experiments on a local copy of the archive. As we build a pipeline to keep the Software Heritage public datasets up-to-date, we need to provide a way for researchers to have their own local copy of the dataset for in-house processing.

Once the dataset is available in some format for people to run queries on it, we will be able to collect more real-world use cases as a way to improve our understanding of researchers' needs, which in turn guides the platform design: what are the types of queries

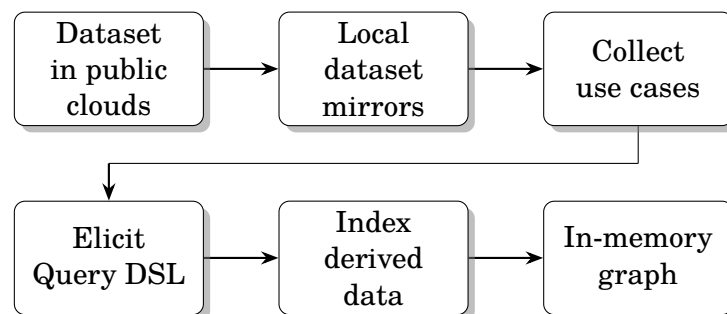


Figure 5.2: Steps towards creating a research platform in Software Heritage.

that scientists want to run? What are the data and metadata filters that they need? What is the sort of information that is the most often retrieved from the data model? Answers to these questions can deepen our understanding of the use cases of researchers, and eventually work out the best ways for them to query the archive in a sufficiently expressive way, whether through low-level APIs or more abstract domain specific languages.

Real-world use cases also exhibit patterns of access to data derived from the dataset: diffs between revisions, branching and merging histories, etc. Isolating this “derived data” to index it in the dataset would also be useful to the platform, as it would significantly improve the performance of computations on these common use cases.

While the cost of disk access for the file contents cannot be avoided, another interesting research direction performance-wise will be to store as much of the history graph as possible directly in memory. This should enable efficient querying of the complex structures that constitute the archive graph and allow us to analyze it in more detail.

These different steps, summarized as a roadmap in Figure 5.2, together form a useful overview of the work required to implement a large-scale research platform for universal software mining. As this thesis inscribes itself in this long-term plan, these steps will be discussed in more detail in the following chapters.

PART II

**MAKING SOFTWARE ARTIFACTS
AVAILABLE AT DIFFERENT SCALES**

The Vault: Assembling Related Source Code Artifacts

In the previous chapters, we have outlined the different challenges in making Software Heritage a universal software mining platform. By studying the needs of researchers in the fields of empirical software engineering who perform studies on software repositories, it is possible to get a good sense of the best ways the archive could become useful as a research platform.

However, building the universal platform constitutes a huge endeavor that has to be done incrementally. As we cannot aim to cover most use cases overnight, we should instead prioritize features that are immediately useful to researchers so that they can start using the building blocks we provide in their work. Aside from driving up engagement with the research community, this also helps identify the main pain points they encounter, which in turn can plot the roadmap in the steps we take to alleviate them.

The very first step that has to be taken to make the archive usable as a bare-bones platform is to provide a simple way to retrieve single artifacts or logical groups of artifacts from our own storage. While this cannot necessarily make large-scale analysis possible, this could at least be used for some smaller-scale experiments and prototypes.

For single artifacts, the Software Heritage API¹ already somewhat covers this use case: using the SWHID of an object, it is possible to retrieve its associated data as stored in the archive. This use case is relevant for researchers because it provides a first form of availability: many studies can be conducted using only VCS object identifiers; the archive is able to complement these studies by allowing readers to retrieve the actual objects behind

¹<https://archive.softwareheritage.org/api/>

these identifiers instead of putting on researchers the burden to provide the full artifacts themselves. As an example, the following HTTP GET request can be used to get the revision `swh:1:rev:aafb16d69fd30ff58afdd69036a26047f3aebdc6`:

```
% curl 'https://archive.softwareheritage.org/api/1/revision/
      aafb16d69fd30ff58afdd69036a26047f3aebdc6/'

{
  "id": "aafb16d69fd30ff58afdd69036a26047f3aebdc6",
  "message": "Merge branch 'master' into pr/584\n",
  "author": {
    "email": "nicolas.dandrimont@crans.org",
    "fullname": "Nicolas Dandrimont <nicolas.dandrimont@crans.org>",
    "name": "Nicolas Dandrimont"
  },
  "date": "2014-08-18T18:18:25+02:00",
  "directory": "9f2e5898e00a66e6ac11033959d7e05b1593353b",
  "metadata": {},
  "parents": [ ... ]
  [ ... ]
}
```

While this works well to check the status of a single artifact, it is tedious to use for more complex use cases. Even for relatively small-scale experiments on a single repository, researchers will generally want to fetch at least an entire source code tree. In this chapter, we introduce a new component in the Software Heritage infrastructure to fetch entire groups of software artifacts at once: the Vault.

6.1 Design

A core design choice in the Software Heritage archive is to deduplicate every single software artifact and consolidate them in a shared storage. While this has a lot of interesting properties for archival (outlined in Section 4.2), like traceability and reduced storage space, it comes with an important trade-off: artifacts contained in a single repository are spread out all over the archive during the ingestion process, and the retrieved VCS bundles and packages are then thrown away. This means that retrieving a repository from the archive requires reassembling all its components and putting them back in a compact, downloadable format.

The general goal of the Vault is to provide a way to assemble logical groups of related software artifacts into a single *bundle*. This assembling process is called “cooking”. It takes a single SWHID as an input and traverses the Merkle DAG to assemble all the artifacts that are *reachable* in the subgraph rooted at that artifact.

Concretely, this means that cooking a directory will assemble all its subdirectories and

blobs, whereas for a revision it will walk down the entire chain of parent revisions and assemble all their associated root directories (and, recursively, their own subdirectories and blobs). Once the objects are cooked, they are assembled in a suitable format that is specific to the type of object that was cooked, respectively:

- For **directories**, the entire file hierarchy contained in the directory is cooked as a *tarball*, a file format which combines and compresses multiple files and directories.
- For **revisions**, the entire commit chain of the revision’s ancestors is put in an empty *Git repository*, along with all the files and directories they contain.
- For **snapshots**, each branch and tag in the snapshot is followed to get the entire commit graph of the snapshotted repository, and all objects are similarly written to an empty Git repository.

In essence, cooking a snapshot allows one to “clone” an entire repository stored in the archive as a Git repository. Because this cooking process can take some time, the Vault is designed as a two-step cache. First, the cooking step asynchronously prepares the bundle of artifacts and writes it as a single file in its internal storage, then notifies the requester that the object is ready for retrieval. After doing so, the Vault can efficiently serve the artifact from its internal storage as a static file via HTTP, in effect acting as a cache for the bundle cooking process.

The cooking process is deterministic: its input is a SWHID, which, because of the properties of the Merkle DAG, always refers to the same sub-DAG of software artifacts². This means that the cooking process can be deduplicated: if some artifact experiences a sudden surge in popularity (because it disappeared from its hosting place or was publicized through other means), many users will try to request it at once. There is no need to restart the cooking process for each user; instead, by keying each bundle with the SWHID it assembles, the process can be shared so that the cooking of each bundle happens only once.

Figure 6.1 shows the architecture of the service as a C4 diagram [35]. It is centered around the Vault backend, which directly communicates with a database where the status of the bundles is kept up-to-date, as well as an internal storage where the bundles can be written and from which they can be later efficiently retrieved. It can also distribute tasks to a worker pool of “cookers” which retrieve and assemble artifacts from the main Software Heritage database. This design ensures a low coupling between the backend and the cookers, which allows the cooking capacity to be horizontally scaled in a straightforward manner by having new resources simply register themselves in the task queue.

²In theory this is not always true because the data model can contain holes of artifacts that were not retrieved at first, but then found in a later crawl. This edge case is largely ignored for now in the archive because of the added complexity handling it would incur.

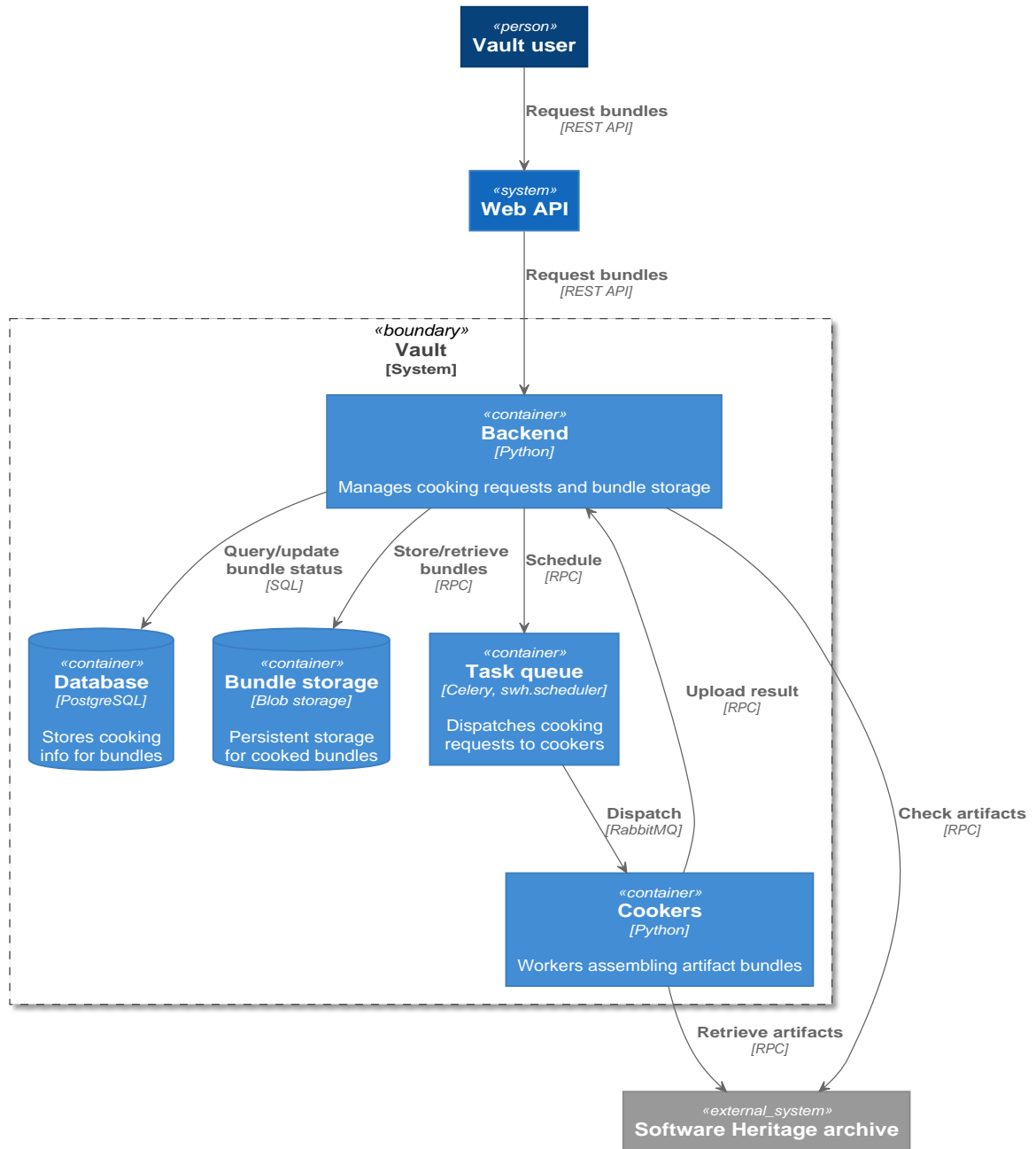


Figure 6.1: C4 architecture diagram of the Software Heritage Vault.

In a typical workflow, a user requests the cooking of a bundle from the web API,³ which forwards the cooking request to the Vault backend. The backend then tries to add this cooking request to the database; if no cooking request for this artifact is already present as pending or completed, it asynchronously dispatches the task to one of the “cooker” servers through a task queue. It then immediately returns a task identifier to the user. While the cooking process is pending, the cookers periodically send requests to the backend with the current progress of the cooking; this progress is written to the database and can be requested at any time by the user. Once the cooking process is complete, the cookers stream the resulting bundle to the backend which writes it in its storage. The backend then updates the database to mark the bundle as ready for retrieval, notifying the users who can then send a download request to efficiently retrieve it.

6.2 Cooking process and bundle formats

The cooking process itself has many design considerations that impact ease of retrieval and ease of use. Being able to cook bundles efficiently is key for this process to be practical for large-scale analysis, as researchers can sometimes request thousands of repositories to use for a single study. In this section we look at the ways to efficiently assemble the two main kinds of bundles: directories and commit graphs (for revisions and snapshots).

6.2.1 Cooking directories

Cooking directories from the main archive storage is a relatively simple endeavor: the children of each directory can be recursively retrieved by making several directory listing queries to the storage. As the directory traversal happens in a depth-first fashion, the directory names are pushed onto a stack; during the traversal, the full path of the current object is obtained by reading the contents of the stack.

As fetching blobs individually is generally expensive, the blob retrieval requests to the storage are *batched*: instead of fetching the blobs during the recursive directory traversal, they are added to an asynchronous fetching queue, which retrieves the files by batches of a thousand items. These files are then written to the tarball output, as archives in the tar format can be built incrementally.

Algorithm 1 shows the recursive directory cooking algorithm. In this algorithm, fetching blobs is *embarrassingly parallel* and can be scaled up at will. However, building the queue of blobs to fetch is done in a sequential fashion with several directory listing requests sent to the archive storage, each request being directly dependent on the SWHID retrieved by the previous requests.

³<https://docs.softwareheritage.org/devel/swh-vault/api.html>

Algorithm 1 Recursively “cook” a directory in a tarball.

```

function FETCHDIRREC( $Q$ ,  $swid$ ,  $path$ )
  for all  $c \in \text{STORAGE.GETCHILDREN}(swid)$  do
     $n \leftarrow \text{NAME}(c)$ 
     $p \leftarrow path + "/" + n$ 
    if  $\text{TYPE}(c) = \text{DIRECTORY}$  then
      FETCHDIRREC( $Q$ ,  $c$ ,  $p$ )
    else
      add  $(c, p)$  to  $Q$ 
    end if
  end for
end function
function COOKDIR( $swid$ )
   $Q \leftarrow$  empty queue ▷ Queue of blobs to fetch
   $T \leftarrow$  empty tarball ▷ Output tarball
  FETCHDIRREC( $Q$ ,  $swid$ , "")
   $C \leftarrow \text{FETCHCONTENTS BATCHED}(Q, 1000)$ 
  for all  $(c, p) \in C$  do
    write  $c$  in  $T$  at path  $p$ 
  end for
  return  $T$ 
end function

```

This bottleneck massively increases the number of round-trips between the cooker and the storage. This behavior could be improved in two ways. The first would be to replace the depth-first traversal by a parallel breadth-first search, reducing the number of sequential round-trips from $O(n)$ to $O(h)$ (the height of the directory tree). While this improves throughput by reducing total latency, it still requires a large number of round-trips. Another option would be to compute the entire subtree of the directory recursively directly in the storage side, then return the file hierarchy in a single response. However, our current storage system backed by PostgreSQL cannot efficiently compute these requests and still has to do round-trips with the database itself. Later in the thesis (Chapter 9), we will explore ways based on graph compression to efficiently reply to those recursive “vault queries”, i.e., being able to quickly compute the entire subgraph reachable from a node.

6.2.2 Cooking revisions and snapshots

The standard format used to cook directories is relatively straightforward: a source code tree can easily be exploited for software mining when distributed as a compressed tarball. When the subgraph reachable from an artifact includes revision chains however, as is the case when cooking revisions and snapshots, there is no natural format in which the commit graph can be distributed; the best format to use requires further evaluation.

One simple format would be to “flatten” the entire chain: a directory at the root would

contain one subdirectory per revision, then each of these would contain the state of the source tree for the given revision. While cooking bundles in this format is easy to implement using the algorithm described in the previous section, the output bundle will be of considerable size: flattening the revisions removes all the Merkle DAG deduplication, as each new commit copies the entire source tree. While hard links and symbolic links could be valid ways to implement deduplication for these objects, the former are not portable, and the latter get in the way of studies which rely on trustworthy file modes and types.

A preferable option is to export these bundles directly as Git repositories. Software mining researchers are already accustomed to working with these repository formats and thus generally have tools already available to extract relevant data from them. Because Git is the most popular VCS, it makes sense to provide a way to export revisions and snapshot bundles as Git bundles.

To export archived artifacts as a Git repository, we first investigated the format used by `git fast-import`,⁴ designed to be an interchange format between VCSs and which can efficiently import data into a Git repository. This format contains a stream of “commands” which describe all the Git objects to import (tags, commits, trees and blobs), and is used by many VCS converters. Implementing this cooker exposed a few limitations of the format itself: poor documentation, unfriendly interface due to its relatively uncommon usage, lossiness (no tagged trees and incomplete handling of signed tags) and large output (the format was not meant for long term storage and has no built in delta-compression of blobs). Most importantly, it is very expensive to export from the archive, because the commits are represented as a list of changes between one state to the other. Generating revisions under this format requires computing diffs between consecutive commits, which is expensive and hard to parallelize.

Yet a better approach is to create a cooker for Git *bare* repositories⁵, i.e., a tarball of the internal data store of a Git repository which can be easily cloned with the Git command line. Implementing this cooker is relatively straightforward: it involves listing all the objects reachable from the revision or snapshot artifact, then directly writing each object in the Git object storage. These objects can then be delta-compressed together using `git repack`.

6.3 Discussion

The Vault is generally an efficient way to retrieve substantial lists of repositories or source code trees from the archive. A moderately sized repository with around ≈ 390 commits takes around 7 minutes, while its source code tree of ≈ 60 files takes 19 seconds. By extrapolating these benchmarks and from our own experiments, the Vault is suitable to retrieve up to tens of thousands of directories and thousands of repositories from the archive in a reasonable

⁴<https://git-scm.com/docs/git-fast-import>

⁵<https://git-scm.com/book/en/v2/Git-on-the-Server-Getting-Git-on-a-Server>

time (between a few hours and a few days). This service offers a first rudimentary way of performing software mining studies on data extracted from the archive.

One possible future direction for this work would be to “pre-cook” a large number of objects according to certain heuristics, as a way to decrease cooking latency. While we cannot anticipate which objects would be the most relevant to study for researchers, it is likely that some particular objects such as revisions pointed by releases or the latest available source tree of each repository would be of particular interest to them, and thus a reasonable heuristic for pre-cooking.

The main advantage of retrieving repositories using the Vault is that their format is common (simple directories and Git repositories), allowing researchers to use their own day-to-day tools for analysis, without having to write custom programs specifically for running experiments on the archive. This advantage comes at the loss of the deduplication that was present in the original data model: while the Vault can be used to analyze repositories one by one, all the data sharing links between the repositories is lost. Aside from having implications on space usage, it also makes it challenging for researchers to study the duplication relationships between the different artifacts. More generally, this approach is not suitable for exhaustive or macro studies on the entire software commons.

A final caveat is that using the Vault to retrieve repositories is frictional and requires some upfront work to retrieve repositories that can be harder than to simply clone the original repositories themselves. The Vault is still valuable for these use cases, as the original repositories are not guaranteed to still be available or even stable over time due to history rewrites, however it leaves significant room for improvement. The next chapter tries to bridge the gap between simplicity and low friction for small-scale mining studies by exploring another way of making repositories available for use with day-to-day CLI utilities while requiring less setup.

The Software Heritage Filesystem (SwhFS)

While the Software Heritage Vault provides a simple way to retrieve repositories in a format suitable for existing analysis tools and workflows, the process to do so can be tedious. This is especially true for use cases which only need to look at specific files in the repository, in which case users have to pay the cost of bundling, downloading and extracting the entire repository (or directory) when only a fraction of this data is necessary. Providing a way to work on these repositories by fetching their resources in a *lazy* way would allow researchers to quickly explore repositories and prototype data mining tools. This kind of lazy-loading of resources over the network can be provided at the filesystem level by *virtual filesystems*.

In this chapter, we introduce the *Software Heritage Filesystem (SwhFS)*,¹ a tool that gives access to the Software Heritage archive by exposing a user-space POSIX filesystem as its interface. SwhFS integrates with research and development workflows by presenting the data in the archive as regular directory trees. Provided that a reference to the desired code can be obtained, the corresponding file, directory, or commit can be used as if it were locally available. More generally, SwhFS offers a convenient way to explore huge code bases [104, 105] that require significant time to be fully retrieved over the network, large amounts of disk space, and high I/O costs when switching branches.

This chapter is based on an article [7] accepted at the 43rd International Conference on Software Engineering (ICSE 2021).

¹<https://docs.softwareheritage.org/devel/swh-fuse/>

7.1 Related work

To the best of our knowledge SwhFS is the first attempt to integrate large-scale source code archival into development workflows via the filesystem interface, although other VCS filesystems have been proposed in the past.

RepoFS [139] exposes a Git repository as a FUSE [131, 163] filesystem, making different trade-offs than SwhFS. RepoFS needs a *local* Git repository, while SwhFS loads it lazily over the network. This makes SwhFS more suitable for quick exploration, and RepoFS more suitable for compute-intensive repository mining.

GitOD [141] and VFS for Git [105] expose remote Git repositories as local filesystems, loading them lazily in order to reduce disk and bandwidth usage for large code bases, retaining compatibility with standard Git tools. Scalar [104] is the successor of VFS for Git and no longer provides a virtual filesystem UX.

GitFS [128] and FigFS [70] also offer virtual filesystem interfaces on top of Git. FigFS is now abandoned. GitFS is not, but focuses on the use case of authoring new commits.

All related works reviewed thus far are Git-specific and have a single-repository scope. SwhFS is VCS-agnostic and spans the entire Software Heritage archive, giving access to hundreds of millions of code repositories in a unified view.

Other non-filesystem based interfaces on top of VCS repositories exist. Gitana [40] and gitbase [146] are two such examples, providing SQL-based views onto Git repositories. While more useful for querying and analysis, non-filesystem interfaces are less amenable to integration with development tools.

7.2 Design

Figure 7.1 shows the SwhFS architecture as a C4 diagram [35].

7.2.1 Front-end: POSIX filesystem and FUSE

Users control SwhFS by starting/stopping its user-space daemon via a convenient CLI interface. While running, SwhFS provides a filesystem view of all the source code artifacts archived by Software Heritage. This view is exposed as a POSIX filesystem that can be browsed using file navigation tools and accessed by programming tools like text editors, IDEs, compilers, debuggers, custom shell scripts, etc.

According to the FUSE design [163], the POSIX filesystem interface seen by SwhFS clients is implemented by the OS kernel, which delegates decisions about the content of virtual files and directories to a user-space daemon, communicating with it via the FUSE API [131]. The SwhFS daemon is implemented in the `swh.fuse` Python module, using the `pyfuse3` bindings to the C FUSE API.

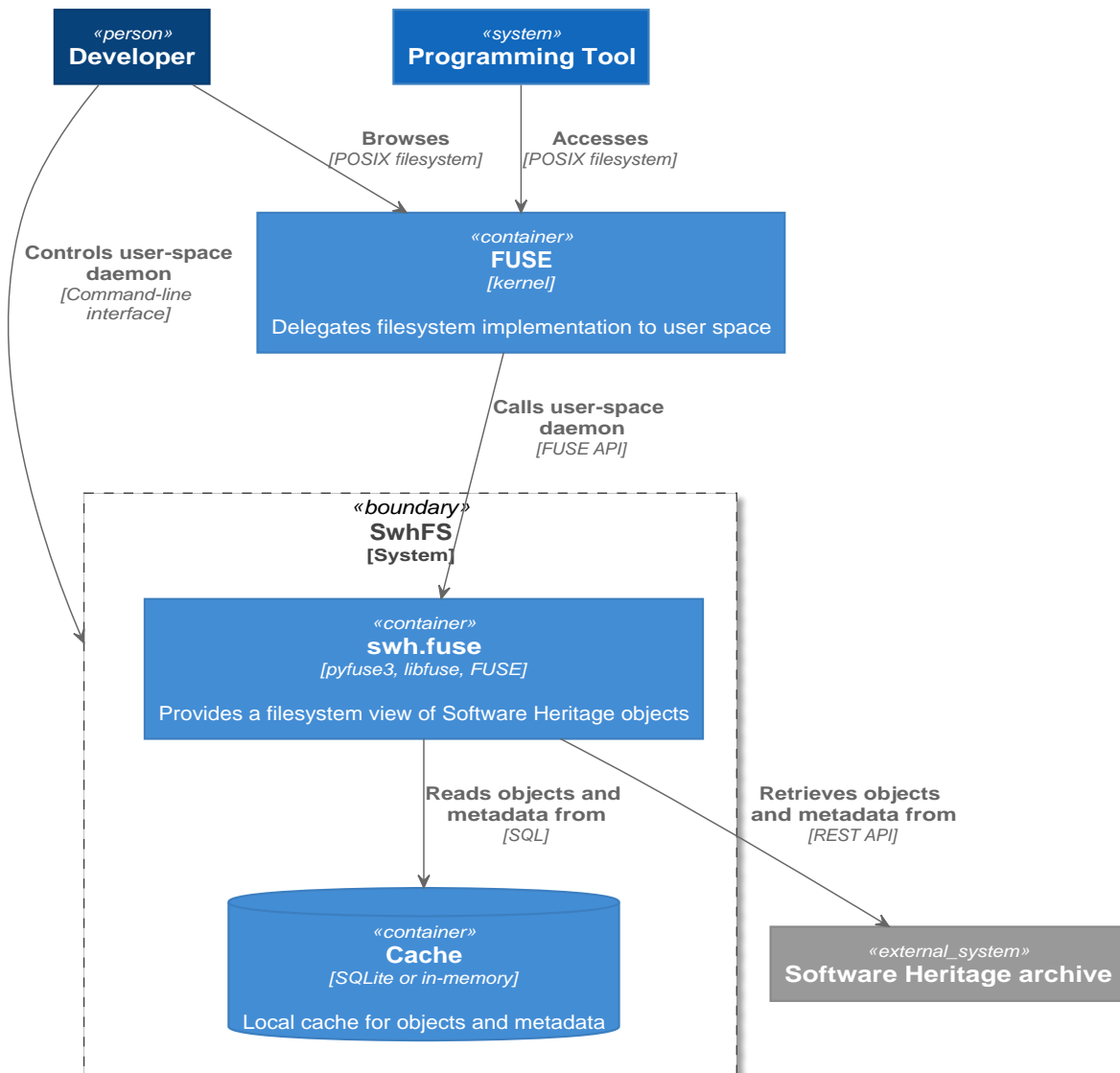


Figure 7.1: Architecture of the Software Heritage virtual user-space filesystem (SwhFS)

7.2.2 Filesystem layout

The filesystem layout implemented by `swh.fuse` consists of (1) a set of entry points and (2) filesystem representations of Software Heritage objects. *Entry points* are located just below the SwhFS mount point; most notably `archive/` allows browsing the Software Heritage archive by known SWHIDs while `origin/` allows doing so by the URLs of archived projects (see Section 7.3 for examples).

Filesystem representations of Software Heritage objects depend on the object type:

- *Source code files* (SWHID type: `cnt`) are represented as regular POSIX files; everything else is mapped to directories.
- *Source code directories* (`dir` type) as directories whose content matches what was archived.
- *Commits and releases* (`rev` and `rel`) as directories containing a root sub-directory pointing to the source code tree plus auxiliary files containing metadata such as commit message and ancestry, timestamps, author information, etc.
- *Repository snapshots* (`snp`) as directories containing one entry for each repository branch (`master`, `v1.0`, `bug-6531`, etc.), each pointing to the filesystem representation of the corresponding commit or release.

Relationships between accessed archived objects are represented using symbolic links. For instance, the source tree of a commit object `archive/swh:1:rev:.../root` is a symbolic link to a directory object located under `archive/`, e.g., `..-> ../../swh:1:dir:...'`. This approach avoids duplications in the virtual filesystem, reifying the sharing of Merkle structures on disk.

The walkthrough of Section 7.3 presents additional details about the SwhFS filesystem layout. A full layout specification is included with SwhFS documentation.

7.2.3 Backend: a Software Heritage ↔ FUSE adapter

`swh.fuse` is an adapter between the Software Heritage REST API² (SWH API) and the FUSE API. When filesystem entities that represent archived objects are accessed, the `swh.fuse` backend uses the SWH API to retrieve data and metadata about them. Obtained results are then used to assemble the expected filesystem layout and return it to the kernel via the FUSE API.

For example, when a source tree object is listed using `readdir()`, SwhFS invokes the `/directory` endpoint of the Software Heritage API and return to the kernel its directory

²<https://archive.softwareheritage.org/api/>

entries using a lazy iterator; when a source code file is `open()`-ed, `/content/raw` is called and byte chunks of the retrieved blob are returned to the kernel piecemeal.

As part of adaptation, `swh.fuse` takes care and abstracts over details such as pagination, encoding issues, inode and file description allocation, etc.

7.2.4 Performance optimizations

Remote filesystems can be notoriously painful to use, on account of network overhead. To make things worse for SwhFS, the backend technology stack (REST APIs and long-term archival storage) was not designed with filesystem-level performances in mind. In order to make it fast enough for practical use, SwhFS implements several optimizations.

Caching Several *on-disk caches* are stored in SQLite databases to avoid repeating remote API calls. The *blob cache* stores raw source code file contents. The *metadata cache* stores the metadata of any kind of object which has already been looked up. Using these two caches it is possible to navigate any part of the Software Heritage archive that has been accessed in the past, even while disconnected from the network.

Merkle properties and the read-only nature of SwhFS make cache invalidation unnecessary, as no cached object can ever change. The option of purging objects from these caches also remains available.

In-memory caching is used for directories, as is typical in most filesystems. The *direntry cache* maps directories to their entries, so that frequently accessed directories can be listed without even incurring SQLite query costs.

Asynchronicity In order to maximize I/O throughput, SwhFS is implemented in asynchronous style: all blocking operations yield control to other coroutines instead of blocking. Additionally, SWH API calls are delegated to a shared thread pool that can keep persistent HTTP connections to the remote API backend, rather than establishing new ones at each call.

7.3 Walkthrough

This section shows how to use SwhFS via concrete examples. A screencast video is available online at <https://dx.doi.org/10.5281/zenodo.4531411>.

Installation SwhFS is implemented in Python, released under the GPL3 license, and distributed via PyPI. It can be installed from there running `pip install swh.fuse`.

The development of SwhFS happens on the Software Heritage forge,³ where issues and patches can be submitted.

Setup and teardown Like with any filesystems, SwhFS must be “mounted” before use and “unmounted” afterwards. Users should first mount the Software Heritage archive as a whole and then browse archived objects looking up their SWHIDs below the archive/entry-point. To mount the Software Heritage archive, use the `swh fs mount` command:

```
$ mkdir swtrfs
$ swtrfs mount swtrfs/ # mount the archive
$ ls -F swtrfs/ # list entry points
archive/ cache/ origin/ README
```

By default, SwhFS daemonizes in background and logs to syslog; it can be kept in foreground, logging to the console, by passing `-f/--foreground` to `mount`.

To unmount SwhFS use `swh fs umount PATH`. Note that, since SwhFS is a *user-space* filesystem, (un)mounting are not privileged operations, any user can do it.

The configuration file `~/.swh/config/global.yml` is read if present. Its main use case is inserting a per-user authentication token for the SWH API, which might be needed in case of heavy use to bypass the default API rate limit.

Source code browsing Here is a SwhFS Hello World:

```
$ cd swtrfs/
$ cat archive/swh:1:cnt:c839dea9e8e6f0528b468214348fee8669b305b2

#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
}
```

Given the SWHID of a source code file, we can directly access it via the filesystem. We can do the same with entire source code directories. Here is the historical Apollo 11 source code, where we can find interesting comments about the antenna during landing:

```
$ cd archive/swh:1:dir:1fee702c7e6d14395bbf5ac3598e73bcbf97b030
$ ls | wc -l
127
```

³<https://forge.softwareheritage.org/source/swh-fuse/>

```
$ grep -i antenna THE_LUNAR_LANDING.s | cut -f 5
# IS THE LR ANTENNA IN POSITION 1 YET
# BRANCH IF ANTENNA ALREADY IN POSITION 1
```

We can checkout the commit of a more modern codebase, like jQuery, and count its JavaScript lines of code (SLOC):

```
$ cd archive/swh:1:rev:9d76c0b163675505d1a901e5fe5249a2c55609bc
$ ls -F
history/ meta.json@ parent@ parents/ root@
$ find root/src/ -type f -name '*.js' | xargs cat | wc -l
10136
```

Commit history browsing meta.json contains complete commit metadata, e.g.:

```
$ jq .author.name,.date,.message meta.json
"Michal Golebiowski-Owczarek"
"2020-03-02T23:02:42+01:00"
"Prevent collision with Object.prototype ..."
```

Commit history can be browsed commit-by-commit digging into parent(s)/ directories or, more efficiently, using the history summaries located under history/:

```
$ ls -f history/by-page/000/ | wc -l
6469
$ ls -f history/by-page/000/ | head -n 2
swh:1:rev:358b769a00c3a09a...
swh:1:rev:4a7fc8544e2020c7...
```

The jQuery commit at hand is preceded by 6469 commits, which can be listed in “git log” order via the by-page view. The by-hash and by-date views, inspired by RepoFS [139], list commits sharded by commit identifier and timestamp:

```
$ ls history/by-hash/00/ | head -n 1
swh:1:rev:0018f7700bf8004d...
$ ls -F history/by-date/
2006/ 2007/ 2008/ ... 2018/ 2019/ 2020/
$ ls -f history/by-date/2020/01/08/
swh:1:rev:437f389a24a6bef...
$ jq .date history/by-date/2020/01/08/*/meta.json
"2020-01-08T00:35:55+01:00"
```

Note that to populate the `by-date` view, metadata about all commits in the history are needed. To avoid blocking, metadata is retrieved asynchronously in the background, populating the view incrementally. The hidden `by-date/.status` file provides a progress report and is removed upon completion.

Repository snapshots and branches Snapshot objects record where each branch and release (or “tag”) pointed to at archival time. Here is an example with the Unix history repository [147], which uses historical Unix releases as nested branch names:

```
$ cd archive/swh:1:snp:2ca5d6eff8f04a671c0d5b13646cede522c64b7d/refs/heads
$ ls -f | wc -l ; ls -f | grep Bell
40
Bell-32V-Snapshot-Development
Bell-Release
$ cd Bell-Release
$ jq .message,.date meta.json
"Bell 32V release ..."
"1979-05-02T23:26:55-05:00"
$ grep core root/usr/src/games/fortune.c
    printf("Memory fault -- core dumped\n");
```

Two of the 40 top-level branches correspond to Bell Labs releases. We can dig into the UNIX/32V fortune implementation instantly, without having to clone a 1.6 GiB repository.

Software origins Software can also be explored by the URL it was archived from, using the `origin/` entry point:

```
$ cd origin/
$ cd https%3A%2F%2Fgithub.com%2Ftorvalds%2Flinux
$ ls
2015-07-09/ 2016-02-23/ 2016-03-28/ ...
$ ls -F 2015-07-09/
meta.json snapshot@
```

we can see a list of all archival crawls of the Linux kernel repository made by Software Heritage, and then navigate to the state of the repository as it was in 2015 (as a snapshot object). Note that one needs to use the exact origin URL and percent-encode it. To help with that, the companion `swh web search` CLI tool is available:

```
$ swh web search "torvalds linux" --limit 1 --url-encode | cut -f1
https%3A%2F%2Fgithub.com%2Ftorvalds%2Flinux
```

7.4 Discussion

SwhFS provides numerous improvements in archive data accessibility. It considerably reduces the friction of accessing various software artifacts as it does not require any up-front setup to navigate any software artifact once the virtual filesystem has been mounted. This is very useful for prototyping experiments, as it allows researchers to use software mining tools as if the source code files were on their local computers. When using a filesystem interface, existing POSIX tools and coreutils can be used to easily select files on which to run analyses (`find`, `ls **/*.c`, etc.).

Because each file is fetched lazily when its contents are accessed, this reduces the total size of the objects retrieved from the archive, in contrast to the Vault which always bundles an entire subgraph and all the objects it contains, even when they are not necessary for the analysis. This makes SwhFS an efficient way to run experiments that only access a small number of specific files in a repository, e.g., license or README files. Moreover, the use of symbolic links to implement sharing relationships between the software artifacts materializes the DAG in a way that can be studied practically: it is easy to check whether two artifacts are the same by checking whether their symbolic links resolve to the same filesystem node. This makes it possible to implement graph traversal algorithms directly at the filesystem level.

However, SwhFS has a few important limitations that makes it generally unsuitable for large-scale software mining. First, it is not optimized for heavy workloads, as its primary goal was to target small experiments and localized analyses. The software components of SwhFS (the SQLite cache, the HTTP library, etc.) were not designed to handle large-scale analyses, high parallelism and data intensive tasks. In general, while user-space filesystems based on FUSE can sometimes reach performance reasonably close to kernel-level filesystems, they require a particular attention to optimization and some workloads are unfriendly even when optimized [163].

While there is always room for performance improvements, another important issue that makes SwhFS often unsuitable for large-scale mining is the lack of entry-points to find the relevant artifacts. The virtual filesystem makes it easy to find all the Python files in a single repository, for instance, but there is no easy way to find all the files or revisions matching a certain pattern in the entire archive. As described in Section 5.1.2, software mining studies are often designed to be run on arbitrary artifacts matching certain criteria. SwhFS does not provide a way to iterate on these artifacts, as its primary function is to list the descendants of a given node in the software graph.

The Vault and SwhFS provide two different ways to perform studies on individual artifacts and their descendants, and thus make the data corpus available for small to medium-scale studies (\approx tens of thousands of repositories) assuming researchers have pre-

established a list of repositories of interest for their study. The next logical step is to explore ways to efficiently select massive quantities of software artifacts on more fine-grained criteria, and run analyses for larger workloads, up to the size of the entire archive.

The Software Heritage Property Graph Dataset

This chapter is based on an article [126] accepted at the 16th International Conference on Mining Software Repositories (MSR 2019), and the presentation [125] for its subsequent use as the “Mining Challenge” of the 17th International Conference on Mining Software Repositories (MSR 2020).

8.1 Introduction

Up to this point, we have proposed ways to make the Software Heritage corpus available to software mining studies of medium scale, up to tens of thousands of repositories, either by retrieving bundles of related software artifacts (Chapter 6) or by accessing the content remotely through a virtual filesystem (Chapter 7). Ultimately, our goal is to enable *universal software mining*, i.e., making it feasible for researchers to study the *entire corpus of software commons* for all the reasons described in Section 1.2. Notably, exhaustive approaches are virtually the only way to get a realistic high-level picture of the complex social networks and processes governing software development.

The main object of interest to study these strongly entangled and dynamic relationships is the *property graph* [9, 31] of public software development, i.e., the graph containing all the software artifacts in our corpus, from the development history down to the source code trees and individual blobs, along with all their associated properties.

Exhaustive studies of the source code files themselves are very expensive, both in terms of required storage space and bandwidth to fetch the entirety of more than 10 billion blobs

totaling around 850 TiB of storage space as of May 2021, and in terms of computing power to process this vast amount of data. However, the property graph of software development *excluding the blobs themselves* is in itself a valuable trove of data, while only being a fraction of that size. It notably contains all the directory and file hierarchies of the source code, its entire development history and all associated metadata, including authorship graphs.

By making the property graph easy to analyze in an exhaustive manner, we can significantly advance towards our goal of universal software mining on several fronts. The immediate benefit is to enable studies that aim to analyze the underlying development processes and social communities in the software commons, as they generally do not need access to the blobs themselves since the network relationships are directly present in the graph. More broadly, it also helps mining studies that do analyze source code by allowing researchers to precisely narrow the set of source code files they need prior to actually fetching them, in order to minimize bandwidth and storage requirements.

In this chapter we introduce the *Software Heritage Property Graph Dataset*, a corpus of all development artifacts archived by Software Heritage made freely available for research in both a graph representation and a relational format suitable for scale-out analysis. While this dataset excludes source code blobs, it does include their cryptographic checksums, which 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 in other datasets.

8.1.1 Related Work

To the best of our knowledge, no existing public dataset contains a comparable amount and diversity of data about source code artifacts extracted from public software development.

The Debsources dataset [36] contains a wealth of information about Debian releases over several decades, but stops at the release granularity (rather than commits) and is at least three orders of magnitude smaller than the present dataset.

GHTorrent [68] covers GitHub and includes non-source-code artifacts (e.g., issues, pull requests, etc.), but lacks software hosted in other locations (e.g., GitLab, Debian, PyPI). Also, no deduplication applies to GHTorrent, making cloning tracking more challenging. The GitHub Activity Data [78] dataset is similar to GHTorrent, but contains less metadata; it can be directly queried in the Google BigQuery cloud platform using relational query languages. CodeFeedr [164] is a natural evolution of GHTorrent, but it is oriented toward expanding coverage for additional sources of non-source-code artifacts.

World of Code (WoC) [92] is a recent attempt at providing a corpus for large-scale VCS analyses, with a target scale similar to ours (18 billion objects as of October 2020). However, the sourcing diversity of the dataset is more limited, only covering Git and GitHub and excluding other VCSs and hosting locations.

¹<https://archive.softwareheritage.org/api/>

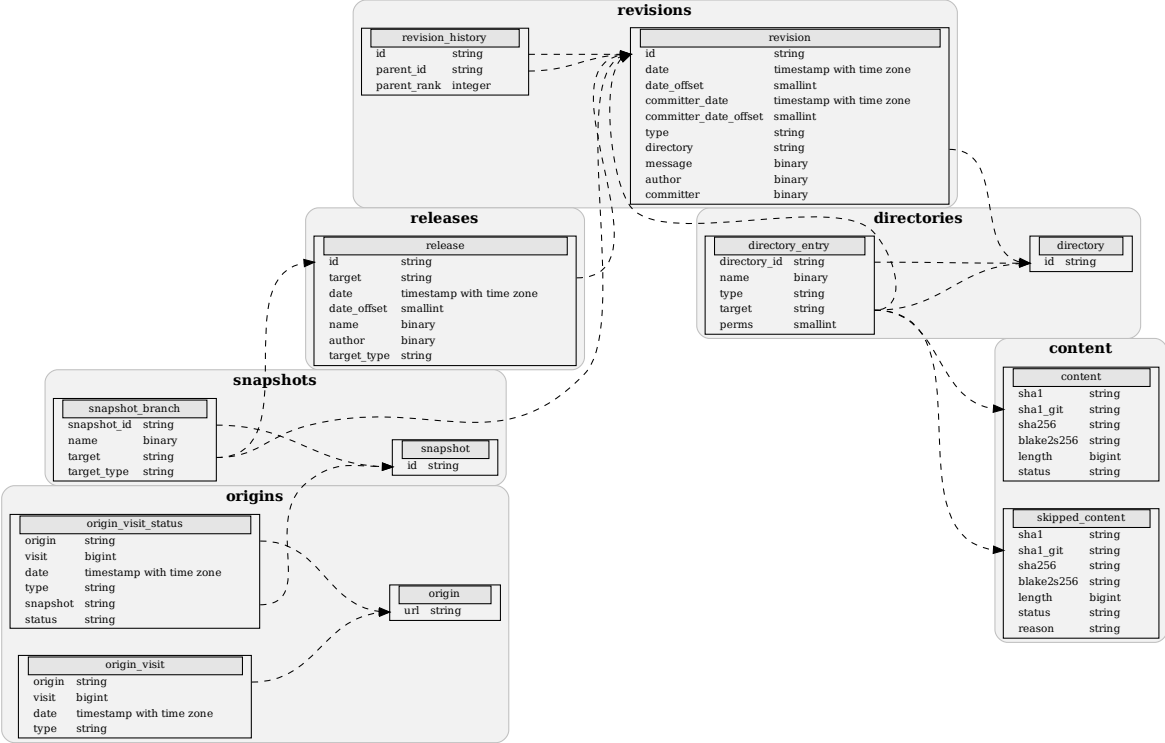


Figure 8.1: Relational schema of the Software Heritage Property Graph Dataset.

8.2 Relational model

The most common pattern in distributed big data analysis is to query datasets provided in a *relational format*, i.e., as a set of data tables linked together by predefined relationships. The relational *schema* describes these tables, notably the types of their individual columns and the way they relate to each other. These data tables can then be loaded in distributed Big Data engines and queried using relational query languages like SQL.

The data in the Software Heritage archive is already internally stored in a relational format inside a PostgreSQL database. However, this format is optimized for different constraints, notably shrinking disk space usage at the expense of speed by *normalizing* tables (i.e., reducing data redundancy as much as possible). While this trade-off is acceptable for long-term storage, it requires more expensive “join” operations to combine tables together, which is generally undesirable for scale-out processing as it adds more synchronization points between the different shards. Instead, when using distributed Big Data engines it is more efficient to *denormalize* the data: using larger tables which increases redundancy while reducing the need for join operations.

Figure 8.1 presents the relational schema of the Software Heritage Property Graph Dataset. It significantly differs from the original schema of the PostgreSQL storage² in

²<https://docs.softwareheritage.org/devel/swh-storage/sql-storage.html>

a few ways. In the latter, the directory table is normalized to avoid storing redundant copies of hashes by using an array of directory entries. After some experimentation on various cloud services, we determined that arrays of arbitrary length in distributed Big Data engines are a performance killer, as they hinder the ability of the query planner to properly shard computations among a right-sized pool of nodes by making the size of each row hard to predict. Some cloud services like Google BigQuery have size limits on arrays, which makes it impossible to store directories of arbitrary length. We denormalized that table, as well as two of the snapshot tables. Another minor change is the use of hexadecimal string representation of intrinsic hashes instead of binary strings, as developers tend to be more comfortable using this hexadecimal representation directly. Finally, some columns which we considered to be mostly implementation details that would not be useful for data analysis were removed from the schema.

Anonymization and ethical considerations. Because the dataset is intended to be distributed to a wide audience, particular attention needs to be paid to the Personally Identifiable Information (PII) contained in the graph, namely the names and e-mail addresses of revision authors. Because this personal data has been shared by subjects publicly, storing and sharing this data on an opt-out basis complies with data protection laws. However, this poses some ethical concerns, as making this data easy to access and process also makes these commit authors more susceptible to potential abuse, like e-mail spam. This is not a new problem, and has notably been encountered by Gousios in the GHTorrent project [66].

Our approach to deal with these ethical considerations is two-fold. First, we anonymize all the data provided in the dataset by running a cryptographic hash on the names and e-mail addresses; if needed, they can be individually requested from the archive API using the commit identifiers, although this access is subject to the usual API quotas. Further, we make access to the dataset conditional on the acceptance of an Ethical Charter³ and specific terms of use for bulk access⁴.

This anonymization process does not hinder the possibility of relating together commits which have the same author, which is an important need in many research use cases. Because cryptographic hashing is both one-way and deterministic, it is possible to check whether two commits have the same author by comparing the resulting hashes, without being able to reverse the original author name.

Schema. The tables in the relational schema of the Software Heritage Graph Dataset are defined as follows:

- **content:** contains information on the blobs (file contents) stored in the archive.

³<https://www.softwareheritage.org/legal/users-ethical-charter/>

⁴<https://www.softwareheritage.org/legal/bulk-access-terms-of-use/>

- sha1 (string): the SHA-1 of the content (hexadecimal)
 - sha1_git (string): the Git SHA-1 of the content (hexadecimal)
 - sha256 (string): the SHA-256 of the content (hexadecimal)
 - blake2s256 (bytes): the BLAKE2s-256 of the content (hexadecimal)
 - length (integer): the length of the content
 - status (string): the visibility status of the content
- **skipped_content**: contains information on the contents that were encountered in the wild but not archived for various reasons.
 - sha1 (string): the SHA-1 of the skipped content (hexadecimal)
 - sha1_git (string): the Git SHA-1 of the skipped content (hexadecimal)
 - sha256 (string): the SHA-256 of the skipped content (hexadecimal)
 - blake2s256 (bytes): the BLAKE2s-256 of the skipped content (hexadecimal)
 - length (integer): the length of the skipped content
 - status (string): the visibility status of the skipped content
 - reason (string): the reason why the content was skipped
- **directory**: contains the directories stored in the archive.
 - id (string): the intrinsic hash of the directory (hexadecimal), recursively computed with the Git SHA-1 algorithm
- **directory_entry**: contains the entries in directories.
 - directory_id (string): the Git SHA-1 of the directory containing the entry (hexadecimal).
 - name (bytes): the name of the file (basename of its path)
 - type (string): the type of object the branch points to (either revision, directory or content).
 - target (string): the Git SHA-1 of the object this entry points to (hexadecimal).
 - perms (integer): the permissions of the object
- **revision**: contains the revisions stored in the archive.
 - id (string): the intrinsic hash of the revision (hexadecimal), recursively computed with the Git SHA-1 algorithm. For Git repositories, this corresponds to the commit hash.
 - message (bytes): the revision message
 - author (string): an anonymized hash of the author of the revision.
 - date (timestamp): the date the revision was authored

- `date_offset` (integer): the offset of the timezone of `date`
 - `committer` (string): an anonymized hash of the committer of the revision.
 - `committer_date` (timestamp): the date the revision was committed
 - `committer_date_offset` (integer): the offset of the timezone of `committer_date`
 - `directory` (string): the Git SHA-1 of the directory the revision points to (hexadecimal). Every revision points to the root directory of the project source tree to which it corresponds.
- **revision_history**: contains the ordered set of parents of each revision. Each revision has an ordered set of parents (0 for the initial commit of a repository, 1 for a regular commit, 2 for a regular merge commit and 3 or more for octopus-style merge commits).
 - `id` (string): the Git SHA-1 identifier of the revision (hexadecimal)
 - `parent_id` (string): the Git SHA-1 identifier of the parent (hexadecimal)
 - `parent_rank` (integer): the rank of the parent, which defines the ordering between the parents of the revision
- **release**: contains the releases stored in the archive.
 - `id` (string): the intrinsic hash of the release (hexadecimal), recursively computed with the Git SHA-1 algorithm
 - `target` (string): the Git SHA-1 of the object the release points to (hexadecimal)
 - `date` (timestamp): the date the release was created
 - `author` (integer): the author of the revision
 - `name` (bytes): the release name
 - `message` (bytes): the release message
- **snapshot**: contains the list of VCS snapshots stored in the archive.
 - `id` (string): the intrinsic hash of the snapshot (hexadecimal), recursively computed with the Git SHA-1 algorithm.
- **snapshot_branch**: contains the list of branches associated with each snapshot.
 - `snapshot_id` (string): the intrinsic hash of the snapshot (hexadecimal)
 - `name` (bytes): the name of the branch
 - `target` (string): the intrinsic hash of the object the branch points to (hexadecimal)
 - `target_type` (string): the type of object the branch points to (either `release`, `revision`, `directory` or `content`).
- **origin**: the software origins from which the projects in the dataset were archived.

- url (bytes): the URL of the origin
- **origin_visit**: the different visits of each origin. Since Software Heritage archives software continuously, software origins are crawled more than once. Each of these “visits” is an entry in this table.
 - origin: (string) the URL of the origin visited
 - visit: (integer) an integer identifier of the visit
 - date: (timestamp) the date at which the origin was visited
 - type (string): the type of origin visited (e.g., git, pypi, hg, svn, git, ftp, deb, ...)
- **origin_visit_status**: the status of each visit.
 - origin: (string) the URL of the origin visited
 - visit: (integer) an integer identifier of the visit
 - date: (timestamp) the date at which the origin was visited
 - type (string): the type of origin visited (e.g., git, pypi, hg, svn, git, ftp, deb, ...)
 - snapshot_id (string): the intrinsic hash of the snapshot archived in this visit (hexadecimal).
 - status (string): the integer identifier of the snapshot archived in this visit, either partial for partial visits or full for full visits.

8.3 Dataset export pipeline

8.3.1 Point-in-time exports

The dataset is exported from the `swh-journal` component of the archive, which provides a log of all objects inserted in the archive, and is backed by Apache Kafka as described in Section 4.4. While the journal exposes a continuous stream of data, the property graph dataset is intended to be a periodic dump in which the entire state of the graph is exported at a given point in time.

Making periodic exports is generally easier for researchers, as they can simply download the latest export without having to continuously synchronize their own data with the archive using a mirroring protocol. This kind of analysis on “offline” data is often the preferred way to run experiments on one’s own infrastructure. This is also useful for study repeatability, as the date of the export can uniquely identify its content, whereas it would be harder to reproduce the exact state of a continuously updating dataset at a given date.

The journal is divided in “topics”, one for each object type described in Section 4.1. The process exports each object type one by one, in a sequential fashion. These topics are ordered so that objects at the top of the graph are always exported first and those at the

bottom exported last (in order: origins, snapshots, releases, revisions, directories, contents). This order uses the directionality of the graph to reduce the number of “dangling” objects (i.e., objects that refer to objects not present in the graph) at the cost of having more “loose” objects (i.e., objects that are not referred to by any other objects). If the upper layers were exported last, they would refer to newer objects in the journal that would have not been exported before; exporting them first ensures that they refer to objects that will be exported in the next steps.

8.3.2 Parallelism

Kafka achieves streaming scalability by dividing each topic into a certain number of partitions that can be processed independently on the client side. When new data is sent to the journal, each object is assigned to a partition deterministically using its intrinsic hash.

Before exporting a topic, the current *offsets* of each partition of that topic are retrieved; they constitute the “boundaries” from and to which the journal will be read. This ensures that the stream reader can terminate at a fixed point, instead of trying to catch up with the live state of the journal.

A parallelism factor can be given on the command line of the exporter as follows:

```
$ swH dataset graph export --processes 64 output_dir/
```

The partitions are split equally among each subprocess, which then start to process them until they reach the high offset. Periodically, these subprocesses report their current progress in the stream to a single process which aggregates them and displays in real time the total progress in the queue.

8.3.3 Resumability

A dataset export can be attributed a unique export identifier that is passed to the command line:

```
$ swH dataset graph export --export-id dataset-2021-03-23 output_dir/
```

This identifier is stored on the server side to associate the journal clients with a given export. When a batch of objects has been read from the journal, the client automatically acknowledges to the server that they have correctly been processed. The server maintains the last offset that has been acknowledged, or “committed”, for each partition. If an export is interrupted because of an error or a server failure, resuming it can be achieved by reusing the same export identifier. The journal client will then start reading the log after the last committed offset.

8.3.4 Object unicity

To guarantee that each object is only present once in the final export, there needs to be a way to check whether a given object has already been exported in case it appears multiple times in the log. The message delivery semantics⁵ in `swh-journal` are “at least once”, which means that objects are guaranteed to be present in the stream but might be delivered multiple times. Moreover, in the event that the export is suddenly interrupted (e.g., due to a power failure), export clients must be able to resume the export even as they might not have acknowledged their current progress in the stream to the journal server.

To guarantee unicity on the client side, the exporter uses a local on-disk database which contains the set of all object IDs which have already been inserted. Because objects are deterministically assigned to their partitions, this database can be sharded directly using the object intrinsic identifier. Two different local on-disk databases can be used, SQLite [119] and LevelDB [63]. The latter is preferred in production exports, as the performance of inserts degrades less quickly than SQLite after a few hundred million insertions.

While the exports themselves are massive (around 10 TiB, depending on the export format) and are generally expected to be output on a large Hard Disk Drive (HDD), the on-disk sets are significantly smaller (in the order of magnitude of hundreds of gigabytes); their location is configurable, and it is recommended to store them on a Solid State Drive (SSD), as reading and writing performance of this set will usually be the bottleneck of the export.

8.3.5 Removing pull requests from snapshots

During the archival process, when Git repositories are cloned by the loaders to be ingested in the archive, the `--mirror` option of `git clone` is used to retrieve all the remote refs, including some that are generally kept on the server only. These are used to implement application-specific behavior by keeping references to hidden branches that will not be cloned by the client by default. One such instance is the way “pull requests” (or “merge requests”) are implemented in software forges like GitHub or GitLab: they are stored as repository branches on the server that do not show up as part of the repositories.

Because those branches are present in the archive, but would lead to unexpected results when performing analyses if they were considered as part of repositories, they need to be filtered from the dataset export. We use the following heuristic to find branches that should be removed: all the branches that are prefixed by `refs/` but are neither prefixed by `refs/heads` nor by `refs/tags`. This matches the default behavior of Git clients, which only

⁵<https://kafka.apache.org/documentation/#semantics>

fetch branches matching those two prefixes when `--mirror` is not specified on the command line.

8.4 Export formats

The dataset export pipeline is written in a generic way, such that it can output the dataset in multiple different formats. The list of formats to export can be specified on the command line, as such:

```
$ swl dataset graph export --formats orc,edges output_dir/
```

In general, there are three different categories of formats that we want to export:

- **Database dumps** (e.g., CSV) which can be imported in a local RDBMS like PostgreSQL.
- **Columnar data storage** (e.g., ORC, Avro, Parquet) which can be used in data lakes and big data processing ecosystems like the Hadoop environment.
- **Edge files** (e.g., Gremlin CSV files) which can be loaded in graph database services like Apache TinkerPop, Amazon Neptune or Neo4j.

8.4.1 Relational formats

The first two formats are *relational*, which means they are exported as a set of tables with columns and rows. We use a common routine to transform document-style objects into relational table rows. For instance, the directory object described by the following JSON document:

```
{
  "id": "87b339104f7dc2a8163dec988445e3987995545f",
  "entries": [
    {
      "name": "main.c",
      "type": "file",
      "perms": 33188,
      "target": "4b825dc642cb6eb9a060e54bf8d69288fbee4904",
    },
    {
      "name": "tests",
      "type": "dir",
      "perms": 33261,
      "target": "ee4d20e80af850cc0f417d25dc5073792c5010d2",
    }
  ]
}
```

will be transformed into data rows in two different relational tables:

directory		directory_entry				
<i>directory_id</i>		<i>directory_id</i>	<i>name</i>	<i>type</i>	<i>perms</i>	<i>target</i>
87b339104f...		87b339104f...	main.c	file	33188	4b825dc642...
		87b339104f...	tests	dir	33261	ee4d20e80a...

8.4.2 Columnar formats

For the columnar exports we provide, we settled on the Apache ORC column-oriented format [80], which is highly efficient and supported by most Big Data engines (Hadoop, Spark [173]) and cloud platforms (Google BigQuery⁶, AWS Athena⁷). Earlier versions of the dataset used the Apache Parquet [23, 11] format, but the lack of stream writing support in Python libraries made it a less advantageous option.

Columnar datasets store their data by column rather than by row. This method of data storage is particularly efficient for large-scale processing and especially in the case of full-table scans, because it reduces the amount of data that has to be scanned to the precise subset of columns queried. This comes at the cost of no longer ensuring data locality of single rows, which is generally useful for Online Transactional Processing (OLTP) operations but less so for Online Analytical Processing (OLAP) workflows. Columnar storage also has useful properties for parallelism, notably ease of partitioning on single columns, which can be used to efficiently scale-out computations.

8.4.3 Edges dataset

The edge format is stripped of all the metadata except the intrinsic identifiers and types of the nodes (i.e., whether an artifact is a revision, a directory, etc.), and is instead constituted of a list of graph edges in CSV format, one source/destination pair of labels per line:

```
<source node> <destination node><CR>
```

Because nodes are identified by their SWHIDs (as seen in Section 4.3.3), these identifiers already embed the typing information of the nodes (swh:1:cnt:4b825...).

Some edge types also have data attached to them: `snapshot` → * edges contain the name of the corresponding branch or tag, and `directory` → * edges contain the name of the entry and its permissions. In a typical relational format, additional data on many-to-many relationships are generally represented using intermediate tables. For the edges dataset, this data is included as part of the edges CSV directly after the edges themselves. Binary strings (branch, tag and directory entry names) are encoded in base64, while integers are

⁶<https://cloud.google.com/bigquery/docs/loading-data-cloud-storage-orc>

⁷<https://docs.aws.amazon.com/athena/latest/ug/columnar-storage.html>

written in decimal representation. For example, this represents a snapshot branch named `refs/heads/master` and a file entry named `test.c` with permissions `0o644`:

```
swh:1:snp:4548a5... swh:1:rev:0d6834... cmVmcy9oZWFkcy9tYXNOZXI=  
swh:1:dir:05faa1... swh:1:cnt:a35136... dGVzdC5j 33188
```

Generally, the edges CSV is not sufficient for graph processing services, as they also require a list of all the unique nodes in the graph. While it could seem that these node IDs would be easy to export from the database used to guarantee node unicity (described in Section 8.3.4), this is in fact not the case due to the presence of loose objects. For instance, if a directory contains a revision entry pointing to a revision that has never been archived before, the destination of the edge will not be a proper object in the graph. However, trying to import this edge in a graph processing system will error out if the object has never been declared in the graph before.

Instead, we get the list of all nodes that are *referred to* as destinations in the edges file, and add them to the existing node files. For that purpose, we use a Unix pipeline to cut the edge files in half, then use GNU `sort(1)` with a large memory and disk buffer (hundreds of gigabytes of RAM and around 10 TiB of disk space) to get a unique sorted list of all the nodes in the graph.

We also use this pipeline to aggregate statistics on the number of each edge type using a hash-table in Awk. Finally, we compress all the output using the Zstandard algorithm [39], as it offers drastic improvements in terms of on-disk size and decompression time, which was observed to be a consequent bottleneck when using other compression algorithms in previous iterations. This is done in a single pass, which minimizes disk I/O. The full pipeline is shown here:

```
counter_command="awk '{ t[$0]++ } END { for (i in t) print i,t[i] }'"  
  
# concatenate the edge subfiles and reports read progress  
pv */*.edges.csv.zst |  
# write the output edges file  
tee graph.edges.csv.zst |  
# decompress the edges CSV with zst  
zstdcat |  
# count the number of edges  
tee >( wc -l > graph.edges.count.txt ) |  
# count the number of each type of edge  
tee >( cut -d: -f3,6 | eval "$counter_command" | sort \  
    > graph.edges.stats.txt ) |  
# cut the CSV to get the destination nodes only
```

```

cut -d' ' -f2 |
# concatenate them to the existing list of nodes
cat - <( zstdcat */*.nodes.csv.zst ) |
# sort and filter duplicates to get a unique list of nodes
sort -u -S"$ram_buffer_size" -T"$disk_buffer_path" |
# count the number of nodes
tee >( wc -l > graph.nodes.count.txt ) |
# count the number of each type of node
tee >( cut -d: -f3 | eval "$counter_command" | sort \
    > graph.nodes.stats.txt ) |
# compress and write the output in the nodes file
zstdmt > graph.nodes.csv.zst

```

This pipeline only requires $O(|E| \log(|E|))$ operations where $|E|$ is the number of edges in the graph, with a small constant factor as the source data is read only once from disk. The log term is due to the sorting algorithm required to get a unique list of nodes. In practice, this entire sorting process on the entire graph takes around 10 days with a sufficiently large memory buffer as of 2021.

8.5 Dataset coverage

The first version of the dataset we published captured the state of the Software Heritage archive as of September 25th 2018, spanning a full mirror of GitHub and GitLab.com, the Debian distribution, Gitorious, Google Code, and the PyPI repository. Quantitatively, it corresponded to 5 billion unique file contents and 1.1 billion unique commits, harvested from more than 85 million software origins. It was subject of a publication [126] accepted at the 16th International Conference on Mining Software Repositories (MSR 2019), and later used [125] as the “Mining Challenge” of the 17th International Conference on Mining Software Repositories (MSR 2020). This first iteration is available in two formats (Parquet and CSV, ≈ 1 TiB each) and downloadable from Zenodo at <https://zenodo.org/record/2583978>, (doi:10.5281/zenodo.2583978). It used a different schema derived from a less efficient exporting pipeline.

Since then, multiple versions of this dataset have been exported and made available to researchers, each new version improving the export pipeline. These versions are shown in Table 8.1. They can be downloaded from the Software Heritage annex at <https://annex.softwareheritage.org/public/dataset/graph/>.

A recent export of the Software Heritage Property Graph Dataset, dated 2020-12-15, contains 19 billion nodes and 221 billion edges in total. Table 8.2 gives a detailed breakdown of each node and edge types. At first glance we can see that the filesystem layer of the

Table 8.1: List of dataset exports as of July 2021.

Export date	# of nodes	# of edges	Formats
2018-09-25	11.1 billion	160.0 billion	Edges, Parquet, CSV
2020-05-20	17.1 billion	203.4 billion	Edges
2020-12-15	19.3 billion	221.5 billion	Edges
2021-03-23	20.7 billion	232.9 billion	Edges, ORC

graph contains most of the nodes (67%) and edges (90%) in the graph: new versions of source code files and directories in public code are produced in higher volumes than other source code artifacts such as commits and releases. The number of visits and origins on the other hand depend only on the crawling throughput of the Software Heritage archive and its coverage of real-world collaborative development platforms.

Table 8.3 and Table 8.4 give an overview of the coverage of the corpus, broken down along various dimensions: the mechanism used to retrieve the source code artifacts (for the most part a VCS or a source package format), the domain of the forge or package repository used to host them, as well as the number of origins and visits of them present in the archive. We can notice that Git dominates the corpus as a source code distribution mechanism, and that GitHub is the dominant forge. Nonetheless, there is a long tail of both source code distribution mechanisms (other popular and historical VCSs; Debian, NPM, PyPI, CRAN, Nix, and Guix source packages) and hosting platforms (several popular and historical forges, various self-hosted instances of GitLab and other forges, as well as GNU/Linux distribution and package manager repositories) that are also present in the corpus. They account for a very diverse corpus, even if one that is *quantitatively* dominated by the popular technological choices of the day among developers.

8.6 Data analysis platforms

While the dataset can be freely downloaded and imported on many large-scale analysis platforms, often researchers do not have easy access to in-house infrastructure and resources where they can analyze such a large amount of data. As a way to make this dataset more accessible for general use, we developed partnerships with **OLAP cloud platforms** to make it available as a public dataset. Sometimes, these cloud platforms have *open dataset programs*, in which they offer storage space for the dataset itself, then let researchers run analyses on it while paying for the cost of processing only. This is a sustainable solution as it does not incur any running costs on the Software Heritage project itself, and benefits to both software mining researchers and cloud platforms.

The Software Heritage Property Graph Dataset became an open dataset in two cloud services, each covering different use cases that are detailed in the sections below: AWS Athena [8] and Azure Databricks [103].

Table 8.2: Node and edge statistics for the 2020-12-15 dataset.

Node type	Nodes	%
origins	147,453,557	0.76%
snapshots	139,832,772	0.72%
releases	16,539,537	0.09%
commits	1,976,476,233	10.22%
directories	7,897,590,134	40.86%
contents	9,152,847,293	47.35%
Total nodes	19,330,739,526	100%

Edge type	Edges	%
origin → snapshot	776,112,709	0.35%
snapshot → release	700,823,546	0.32%
snapshot → commit	1,358,538,567	0.61%
release → commit	16,492,908	0.01%
commit → commit	2,021,009,703	0.91%
commit → directory	1,971,187,167	0.89%
directory → commit	792,196,260	0.36%
directory → directory	64,584,351,336	29.16%
directory → blob	149,267,317,723	67.39%
Total edges	221,488,073,659	100%

Table 8.3: Crawling statistics: number of origins and visits by origin type for the 2021-03-23 dataset.

Origin type	No. of origins	%	No. of visits	%
git	136,684,905	88.8%	545,124,995	53.9%
unknown	14,330,675	9.3%	14,330,675	1.41%
npm	1,533,346	0.9%	300,806,714	29.7%
svn	575,952	0.3%	735,135	0.07%
hg	381,058	0.2%	6,105,706	0.60%
pypi	239,522	0.1%	147,102,654	14.5%
deb	72,303	< ϵ	10,894,679	1.07%
cran	18,019	< ϵ	29,596	< ϵ
ftp	1205	< ϵ	1205	< ϵ
deposit	900	< ϵ	1277	< ϵ
tar	385	< ϵ	955	< ϵ
nix/guix	2	< ϵ	445	< ϵ
Total	153,838,272	100%	1,010,810,868	100%

Table 8.4: Crawling statistics: number of origins and visits by forge domain for the 2021-03-23 dataset (domains with > 1000 origins).

Forge domain	No. of origins	%	No. of visits	%
github.com	147,881,630	96.1%	546,877,021	54.1%
bitbucket.org	2,058,279	1.33%	10,871,128	1.07%
www.npmjs.com	1,534,976	0.99%	300,808,344	29.7%
gitlab.com	990,334	0.64%	5,022,358	0.49%
pypi.org	239,620	0.15%	147,102,752	14.5%
gitorious.org	120,380	0.07%	120,392	0.01%
Debian	38,414	0.02%	10,661,918	1.05%
salsa.debian.org	33,617	0.02%	105,690	0.01%
snapshot.debian.org	33,044	0.02%	33,044	$< \epsilon$
git.launchpad.net	19,571	0.01%	21,198	$< \epsilon$
framagit.org	18,433	0.01%	132,803	0.01%
cran.r-project.org	18,019	0.01%	29,596	$< \epsilon$
hdiff.luite.com	13,861	$< \epsilon$	191,570	0.01%
gitlab.gnome.org	8016	$< \epsilon$	21,837	$< \epsilon$
gitlab.freedesktop.org	4752	$< \epsilon$	1,172,842	0.11%
gitlab.inria.fr	3628	$< \epsilon$	9921	$< \epsilon$
codeberg.org	3623	$< \epsilon$	3733	$< \epsilon$
git.savannah.gnu.org	2959	$< \epsilon$	7008	$< \epsilon$
git.baserock.org	2912	$< \epsilon$	4687	$< \epsilon$
anongit.kde.org	2488	$< \epsilon$	7389	$< \epsilon$
code.google.com	2240	$< \epsilon$	2240	$< \epsilon$
phabricator.wikimedia.org	2224	$< \epsilon$	631,835	0.06%
git.kernel.org	2083	$< \epsilon$	4224	$< \epsilon$
fedorapeople.org	1691	$< \epsilon$	4173	$< \epsilon$
ftp.gnu.org	1590	$< \epsilon$	2160	$< \epsilon$
gitlab.ow2.org	1119	$< \epsilon$	3111	$< \epsilon$
phabricator.kde.org	1030	$< \epsilon$	6269	$< \epsilon$
Debian-Security	1028	$< \epsilon$	199,900	0.02%
git.torproject.org	1014	$< \epsilon$	2503	$< \epsilon$
Total	153,838,272	100%	1,010,810,868	100%

8.6.1 Presto on AWS Athena

AWS Athena is an interactive OLAP service which can analyze data stored in the Amazon S3 storage service using standard SQL queries. Athena uses a hosted version of the Presto engine [143] to distribute queries on large clusters. One particularly interesting property of Presto is that it does not act as a database, but rather a tool to efficiently query vast amounts of data by doing automatic scale-out on pre-existing clusters. This has the advantage of not requiring any up-front setup cost, as the cluster of machines is not managed by the users but by Amazon itself. Instead, users are billed on a pay-as-you-go basis, depending on the total amount of data queried, at a base rate of around 5 USD per terabyte⁸.

After a successful graph export, the resulting ORC files are uploaded in a public S3 bucket that can be queried by anyone using Amazon Athena. The dataset is indexed in the Registry of Open Data on AWS,⁹ and contains information on the S3 bucket and how to query it using Athena. While Athena is offered as the default solution for analysis because of its low setup cost, researchers can also import this S3 bucket in other potentially more costly AWS services like AWS Redshift.¹⁰

To read the ORC files with the appropriate schema, Athena uses an approach known as *schema-on-read*, which means a predefined schema is projected on the data stored in S3 at the time of query. This technique avoids having to load, preprocess or convert the S3 data before querying it. However, as a preliminary step, it does require defining the schema on one's own Athena instance, as well as the location of the data it refers to. We provide a toolchain, `swl.dataset.athena`, which can automatically define this schema on a user provided Athena database. This command defines the schema of the latest graph export (2021-03-23) on the user's "swl" Athena database and sets its external location to the public dataset S3 bucket:

```
$ swl dataset athena create -d swl -l s3://softwareheritage/graph/2021-03-23
```

The database can then be queried directly using this same toolchain, outputting the resulting rows of the SQL query in CSV format:

```
$ swl dataset athena query -d swl <<<"select count(*) from content;"
"_col0"
"9935510171"
```

Alternatively, the queries can be tried interactively using the Athena web console, which provides more advanced features such as query history, syntax checking, detailed error messages, schema previews, etc.

⁸<https://aws.amazon.com/athena/pricing/>

⁹<https://registry.opendata.aws/software-heritage/>

¹⁰<https://aws.amazon.com/redshift/>

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 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;
```

To further illustrate the research possibilities it opens on the dataset, below are some sample SQL queries that can be executed with the dataset on AWS Athena.

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

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 [58, p. 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. After scanning 37 GiB in 30'' the engine results show that commit messages reference the following common actions, ordered by descending order of frequency: *add*, *fix*, *update*, *remove*, *merge*, *initial*, *create*. (We had to manually stem some verbs, because the most recent version of the Presto query engine, which provides a built-in stemming function, is not yet available on Athena.)

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 ever-more 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 GiB in 7'', are shown in Figure 8.2. A linear regression shows that this ratio generally remained constantly around 22% over time. These results are inconclusive, and points to the need for further analysis to validate the hypothesis, which is left as an

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;

```

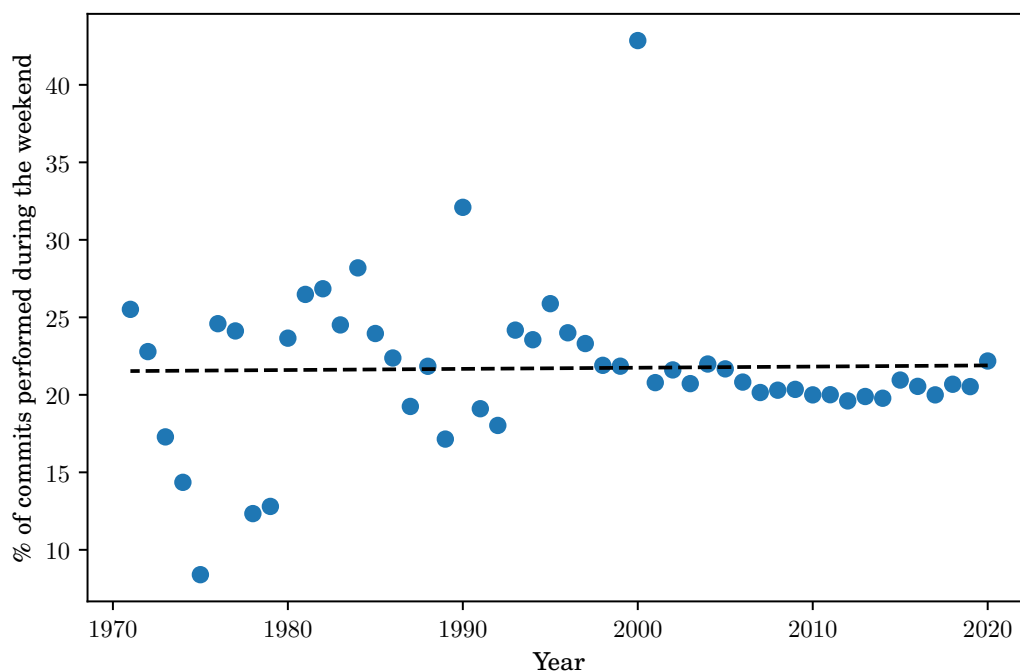


Figure 8.2: Ratio of commits performed during the weekend each year plotted over time. A linear regression shows a flat trend over a period of fifty years.

Listing 4 Average number of parents in a revision.

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

exercise to the reader.

The provided dataset forms a graph, which can be difficult to query with SQL. Therefore, questions associated with the graph's characteristics, such as closeness, distance, and centrality, will require the use of other query mechanisms. 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 GiB 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 GiB in 1'20''. The order of the resulting language files (JavaScript > C > C++ > Python > PHP > C# > Ruby) indicates that, with the exception of JavaScript,¹¹ languages offering more abstraction facilities are associated with smaller source code files.

8.6.2 Spark on Azure Databricks

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. Spark lifts the constraints imposed by limits in Athena by letting users have a direct control on the number of machines in a cluster dedicated to their analysis. This offers more flexibility by letting users choose their own scale factor and the computing power of the nodes in the cluster. This is important for computationally intensive experiments or very large *join* operations, which can only be achieved through sampling in Athena. One downside of this approach is that it can quickly become rather expensive, as the cost is no longer solely dependent on the amount of data analyzed but the real computing resources

¹¹This is likely because Javascript is often used as a target format for transcompilation, minification and source code bundling.

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;

```

Listing 6 Load ORC tables in Azure Databricks.

```
dataset_path = ('wasbs://swgraph@swhopensdataset.blob.core.windows.net'
               '/2021-03-23/orc')
tables = ['content', 'directory', 'directory_entry', 'origin', 'origin_visit',
         'origin_visit_status', 'release', 'revision', 'revision_history',
         'skipped_content', 'snapshot', 'snapshot_branch']
for table in tables:
    df = spark.read.parquet(dataset_path + '/' + table)
    df.createOrReplaceTempView(table)
```

Listing 7 Out-degree 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
)
```

used while the cluster is running. It also requires some upfront cost to start the cluster before running any queries on it.

The dataset is available in a public Azure Data Lake Storage Gen2 container, and can be loaded directly as temporary views from a Python notebook on Azure Databricks, as shown in Listing 6. Once the tables are loaded in Spark, the query in Listing 7 can be used to generate an out-degree distribution of the directories.

Spark is flexible in terms of the computations it can perform, thanks to User-Defined Functions (UDFs) [13] that can be used to specify arbitrary operations to be performed on the rows.

To analyze the graph structure of the dataset, the GraphFrames library [44] can also be used to perform common operations on the graph. Listing 8 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. However, running this experiment on the entire graph can be extremely expensive, as most distributed graph

Listing 8 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)
```

algorithms require a lot of data synchronization points between the nodes, and are thus significantly slower than equivalent algorithms that can run on a single machine. From our own experiments, one should expect a cost of around 5000 USD to compute the connected components of the graph.

PART III

**EXPLOITING SOFTWARE DEVELOPMENT
DATA AS A RECURSIVE GRAPH**

Graph Compression

This chapter is based on an article [27] accepted at the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering.

9.1 Introduction

Most state-of-the-art approaches for analyzing large amounts of development artifacts predominantly rely on classic “big data” approaches, partitioning the corpus over several machines and applying distributed algorithms. Alternatively, but less satisfactorily, sampling can be used, incurring the risks of selection bias and over-generalized findings. In Chapter 8, these scale-out approaches are explored on the full graph of public software development retrieved from the Software Heritage archive, by exporting the graph dataset in a format suitable for distributed processing.

Experimenting with these approaches highlights a major limitation of traditional distributed analysis systems when dealing with **recursive graphs**. The Merkle DAG of public software development is organized in an inherently hierarchical structure, which often renders ordinary queries recursive. Typically, a common primitive for most queries on directed graphs is to compute the *transitive closure* of a given node, i.e., the set of all nodes that are reachable by recursively following edges starting from said node. In relational database models, including distributed Big Data engines, this operation can be performed with *recursive joins*, which involves repeating a join operation to accumulate more nodes until reaching a fixed point when the entire transitive closure has been accumulated.

While this approach generally works efficiently for graphs with a limited depth (e.g., code indentation levels), it is often less suited for arbitrarily nested hierarchies like the

graph of software development. This notion can be understood quite intuitively when thinking about depth as the approximate number of synchronization points required for computing a transitive closure. Getting all neighbors of a set of nodes is embarrassingly parallel, but the resulting sets must be aggregated together at each step; this makes the quantity of recursive iterations the most expensive factor when performing recursive queries. Commit chains are a pathological case of arbitrary nestedness, as they tend to be very long, with some well-known repositories like the Linux kernel exceeding a million commits, but with a very low average degree, thus they cannot effectively be exploited in a distributed setting.

As a result, simple recursive algorithms on the graph of software development are very costly, as they require large Big Data clusters running for a relatively long time. In our experiments, computing the connected components of the undirected version of the graph, while being a linear algorithm with a theoretical time complexity of $O(|V| + |E|)$, takes around six hours when running on a cluster of more than 80 machines, each with 64 vCPUs and 500 GiB of RAM. On the Azure Cloud, the estimated total cost of running this linear algorithm for the entire graph is ≈ 5000 USD using the GraphFrames library on Azure Databricks.

In this chapter, we evaluate the feasibility of a less resource-hungry approach to the analysis of the development history of very large software collections. Specifically, we will answer the following research question:

Research question 9.1. Is it possible to efficiently perform software development history analyses at an ultra large scale, on a single, relatively cheap machine?

As the question is partly quantitative and some terms in it are still vague, we further narrow it down as follows:

- with *development history* we mean the information usually captured by state-of-the-art DVCS [148], with commits as the finest available granularity;
- with *ultra large scale* we mean a scale similar to the known extent of all publicly developed software, using the Software Heritage Graph Dataset described in Chapter 8 as our main benchmark;
- with *cheap machine* we mean commodity hardware, either desktop- or server-grade, that can be easily acquired by researchers with a moderate investment of a few thousand USD.

In the following we will answer this research question *in the affirmative*, by applying lossless graph compression techniques to the underlying Merkle DAG structure of the graph of public software development. As a concrete use case, we compress the full development

history of all publicly developed source code as captured by the 2018-09-25 version¹ of the Software Heritage Graph Dataset (see Chapter 8), consisting of 5 billion unique source code files and 1 billion unique commits, harvested from more than 90 million software projects.

As a size benchmark, we show that the resulting compressed VCS graph, containing the development history of the entire corpus, can be loaded in ≈ 94 GiB of RAM, for an impressive compression ratio of 4.9 bits/arc—as opposed to 8 bytes per arc that a naive in-memory representation of the graph using adjacency lists would require. At current market rates the graph can thus be fit in RAM on commodity hardware with an investment of less than 300 USD for main memory.

As a speed benchmark, we measure the time required to visit the entire graph, obtaining a visit time of less than 2 hours and a processing throughput of almost 2 million nodes per second using a single thread. We also measure the average time required to lookup successors of a given node, obtaining timings of 80 nanoseconds per arc, close to current DRAM random access times (50–60 ns).

To show the applicability of the proposed approach to repository analysis, we use the compressed graph to conduct two classic experiments in software clone detection: we measure (1) how often identical file contents are found in different commits, and (2) how often identical commits are found in different repositories. Our experiences show the advantages of having the development history of the entire corpus in main memory as opposed to secondary memory. In spite of the naive algorithmic approaches chosen—with time complexities of $O(|V| \cdot |E|)$ on the full graph—we manage to run the experiments on representative corpus subsets in just a few days of analysis.

Replication package A replication package for the experiments realized in this chapter is available on Zenodo [26].

9.2 Background

9.2.1 Graph compression

Many datasets are shaped into a graph structure that contains a wealth of information about the data itself, and many data mining tasks can be accomplished from this information alone (e.g., detecting outliers, identifying interest groups, estimating measures of importance and so on). Often, such tasks can be solved through suitable graph algorithms which typically assume that the graph is stored in main memory. However, this assumption is far from trivial to realize in many real-world cases, including the case of interest for the present

¹The graph used for this chapter is older than in the rest of the thesis, as it was frozen at the time the original paper was written.

thesis, where many billions of nodes and arcs might exist. Finding effective techniques to store and access large graphs that can be applied fruitfully to these situations is one of the central algorithmic issues in the field of modern data mining.

A (lossless) *compressed data structure* for a graph must provide very fast access to the graph (let us say, slower but comparable to the access time required by its uncompressed representation in main memory) *without* decompressing it. While this definition is not formal, it excludes methods in which the successors of a node are not accessible unless, for instance, a large part of the graph is decompressed.

Different compressed data structures for graphs offer different trade-offs between *compression time* (the time required to produce a compressed representation from an uncompressed one) and *compression ratio* (the ratio between the size of the compressed data structure and its uncompressed counterpart, typically measured in bits per arc). One should also decide whether the data structure should be *static* or *dynamic* (whether it allows for changes), whether it is aimed at *directed* or *undirected* graphs (or both), and which *access primitives* the structure allows for. In most cases, these aspects can only be evaluated experimentally on a certain number of datasets, although in some rare circumstances it is possible to provide worst-case lower bounds under some assumption on the network structure (e.g., assuming some probabilistic or deterministic graph model, and evaluating the compression performances on that model).

A common large, real-world graph that has been studied and compressed in the past is the directed graph of the Web (or *web graph*), consisting of one node per page and one arc per hyperlink between pages. A pioneering attempt at compressing the web graph is the LINK Database [129]. Suppose that nodes are ordered lexicographically by URL (i.e., node i is the node representing the i th URL in lexicographic order); then the following two properties are true:

- *Locality*: Most arcs are between nodes that are close to each other in the order, because most links are intra-site, and URLs from the same site share a long prefix, which makes them close in lexicographic order. Locality can be exploited using *gap compression*: if node x has successors y_1, y_2, \dots, y_k , gap compression stores for x the compressed successor list $y_1 - x, y_2 - y_1 - 1, \dots, y_k - y_{k-1} - 1$; by locality, most values in this list will be small, and can be stored efficiently using variable-length encodings.
- *Similarity*: nodes that are close to each other in the order tend to have similar sets of neighbors. Similarity can be exploited by using *reference compression*: some successor lists are represented as a difference with respect to the successor list of a previous nearby node.

In Table 9.1 we show a sample of lists of successors of a graph, and in Table 9.2 we show the same lists in which reference compression (in the form of a bit mask) copies successors

Table 9.1: Naive graph representation using adjacency lists.

Node	Out-degree	Successors
...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...

Table 9.2: Compact graph representation using gaps and copy lists.

Node	Outd.	Ref.	Copy list	Extra nodes
...
15	11	0		3,1,0,0,0,0,3,0,178,111,718
16	10	-1	01110011010	6, 293, 0, 2723
17	0			
18	5	-3	11110000000	32
...

from a previous list (identified by the “Ref.” column) following the information contained in a *copy list*, and then the remaining nodes (possibly all successors, if no reference compression is used) are gap-compressed.

9.2.2 The WebGraph framework

Some years later, building on the same approach, the WebGraph framework [29] attained a web graph compression of less than 3 bits/arc by using the *BV scheme* (and even less than 2 for the transposed graph) with a random-access successor enumeration time of a few hundreds of nanoseconds per arc (and much faster than that for sequential access).

The techniques described above are strongly sensitive to node order: this is not a big issue when applied to web graphs, because the lexicographic ordering of URLs is available, but makes it difficult to apply the same techniques to networks (e.g., social networks and, as we will see, version control system graphs) that do not have similarly meaningful canonical node identifiers.

A surprisingly effective ordering is simply that of enumerating nodes following a *visit*: in particular, a breadth-first visit [12] numbers nodes in such a way that might enable gap and reference compression to provide excellent results.

A set of different approaches is based on *clustering: layered label propagation* [28] is a reordering technique which combines the information from a number of clusterings to reorder the nodes.

In the rest of this chapter we will use the WebGraph² open source implementation as

²<https://webgraph.di.unimi.it/>

the technology to perform graph compression on a large corpus of VCS histories. We will also show that Breadth-First Search (BFS) visits and Layered Label Propagation (LLP) are indeed effective reordering strategies to achieve high compression on this corpus.

9.3 Compressing the Graph of Public Software Development

We set out to establish whether graph compression is a suitable approach for enabling ultra-large-scale repository analysis on modest hardware resources. To that end we conduct a case study by (1) retrieving a suitably large graph of development history, (2) compressing it using graph compression techniques, and (3) using the compressed result to conduct repository analysis. In this section we describe the compression pipeline and compression results; in the next we will exploit the obtained compressed representation.

As a dataset we use the 2018-09-25 version of the Software Heritage graph dataset described in Chapter 8, which is to the best of our knowledge the largest publicly available corpus of software development history. This version of the dataset contains the development history of more than 90 million software projects, encompassing full mirrors of GitHub and GitLab.com, historical archives of Google Code and Gitorious, as well as repositories of popular package managers such as NPM, PyPI, and Debian. In total, the graph contains more than 12 billion nodes and 165 billion edges.

9.3.1 Compression scope and metadata

Different analyses will need to access different information stored in VCS. A study of commit messages will not care about file contents, whereas one on code merges will need the revision graph. When considering keeping the entire dataset in main memory for performance reasons, there is an inherent trade-off between access time and RAM requirements. A line has to be drawn to separate the data that benefits the most from fast RAM access, from the metadata that can be left on-disk without becoming a bottleneck.

The major bottleneck when performing development history analyses with on-disk data is generally the access to the neighbors of a given node during graph traversal. Since knowledge of node neighbors is necessary to advance in the iteration steps, these disk accesses cannot be deferred to a later processing stage. It is therefore very beneficial to keep as much *graph structure* and neighboring data as possible in memory to speed up the visits.

On the other hand, once a visit has been performed, the metadata of the visited nodes can be retrieved in a post-processing phase for subsequent analysis. As this metadata does not need to be sent back to the graph traversal routine for it to proceed, access latency of node metadata does not matter as much as it does for the graph topology. Node metadata

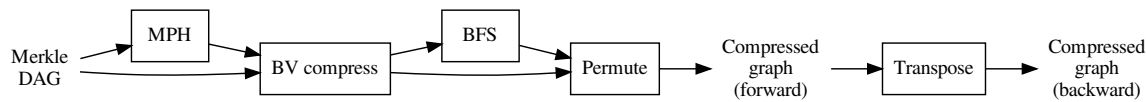


Figure 9.1: Graph compression pipeline. Individual compression steps are denoted with boxes; notable compression input/output artifacts as free text. The last transposition step is optional and only needed to visit the graph backwards.

can thus be retrieved in an asynchronous fashion without significantly impacting analysis time.

The scope of our compression experiment will therefore primarily focus on compressing and storing the *graph structure* into main memory, with the expectation that usage patterns will match the above scenarios—in-memory visits, then asynchronous retrieval and analysis of node metadata.

As a sole exception we will also keep *node types* in memory, i.e., whether a node is a blob, directory, revision, etc. That can be done very efficiently using a *type map* implemented as a bit array indexed by node identifiers and requiring only 3 bits per node (as there are 6 node types in total), or 4 GiB of RAM for the entire graph. The reason to make this exception is that node type filtering is useful in many use cases to determine at runtime what kind of objects graph traversals should be looking at.

For some workloads, it is useful to store other types of metadata associated to the nodes and edges in a way that is directly accessible from the compressed graph. In Chapter 10, we will see how these other types of metadata can be made accessible, and the different speed and memory trade-offs this entails.

9.3.2 Compression pipeline

To compress the dataset we use the WebGraph framework to realize the compression pipeline shown in Figure 9.1. The pipeline input is a simple graph representation (*Merkle DAG* in the figure) as a pair of textual nodes and edges file. These files use the “edges” format described in Section 8.4.3: the nodes file consists of one node label per line; the arcs file consists of one source/destination pair of node labels per line; each label is a SWHID. Then, the following steps are executed in order:³

MPH Generate a *Minimal Perfect Hash (MPH) function* [61] which maps input node labels to the set $\{0, \dots, N - 1\}$ consecutive integers, where N is the number of input nodes. The resulting MPH function will be used in the following to quickly associate *an* integer to node labels without incurring the risk of collisions.

³We refer to the WebGraph documentation on how to practically run them: <http://webgraph.di.unimi.it/>

BV compress Compress the adjacency matrix of the graph using gap compression and the other techniques described in Section 9.2, but without relying on a sensible ordering of nodes yet. The output of this step is a *BV graph* [29].

BFS As discussed in Section 9.2, finding a node ordering that, by permuting rows, maximizes various locality properties on the graph adjacency matrix is key to achieving good compression. Unlike web graphs, version control system graphs do not sport ready-to-use ordering heuristics (such as the URL of each page) that are compression friendly. In fact, given that native VCS node identifiers are generally based on cryptographic checksums (e.g., SHA-1), the links from one node to the next will tend to jump *randomly* from one identifier to another in the space of all possible identifiers.

Experimentally, we have verified that a breadth-first visit of the corpus graph, starting from graph roots (origin nodes pointing to snapshot nodes in the graph) and traversing down towards leaves (file contents) achieves good compression results compared to other orderings. The *BFS* step of the compression pipeline thus performs such a visit on the entire BV graph.

Permute Once the BFS ordering of nodes is known, this step will reorder nodes (as well as rows in the compressed adjacency matrix, performing all needed adaptations) according to BFS order. The result of this step is another compressed graph, in the same format as the original BV graph but with a better compression ratio.

Transpose Strictly speaking, the graph structure of the input dataset can be traversed only in one direction—from origins roots towards file content leaves. It is not uncommon for repository mining use cases to need to traverse the graph in the opposite direction though. For instance, looking up where a given file (or directory, or commit, etc.) has been found requires such *backward* visits.

As backward visits correspond to forward visits on the *transposed* input graph, one can optionally generate a compressed representation of the transposed input graph and use it in addition (or as an alternative) to the compressed graph obtained thus far. WebGraph supports generating the compressed representation of the transposed graph directly from the compressed (forward) graph. If desired, the *Transpose* step in the compression pipeline will take care of this.

9.3.3 Compression results

Compressing the full corpus is a resource-intensive endeavor. The wall time breakdown to perform the various steps on the initial dataset is given in Table 9.3, totaling less than 6 days of compression time.

Table 9.3: Compression time breakdown.

Step	Wall time (hours)
MPH	2
BV Compress	84
BFS	19
Permute	18
Transpose	15
<i>Total</i>	138 (\approx 6 days)

Table 9.4: Compression results. Compression ratios are w.r.t. the information-theoretical lower bound for graphs with the same density.

Forward (original) graph		Backward (transposed) graph	
total size	91 GiB	total size	83 GiB
bits per arc	4.91	bits per arc	4.49
compression ratio	15.8%	compression ratio	14.4%

Timings have been taken on a server equipped with 24 CPUs and 750 GiB of RAM. Note however, that such huge amounts of RAM are not actually *needed* for compression; minimum RAM requirements correspond to the resources needed to load the final compressed graph in memory. The only step that used more than 100 GiB of RAM was the BFS visit, which used a memory mapping to access the BV graph on disk using RAM as cache. Recent results on BFS memory efficiency [72] further confirm that the BFS step is not an impediment on the general applicability of the approach.

We also stress that even *if* significantly more resources were needed for compression than for exploitation of the compressed result, that would be an acceptable trade-off as: (1) compression can be done once and reused many times (possibly by other research groups), and (2) compression resources can be rented for one-time use, e.g., on public clouds.

Compression results are shown in Table 9.4. Besides the raw datum of less than 5 bits per arc, which shows that the compression is very effective in practice, the compression ratio ($\approx 15\%$) with respect to the information-theoretical lower bound of $\log \binom{n}{m}$ for a graph with n nodes and m arcs is about three times the one typical for web graphs, which are highly redundant, but significantly better than the typical values for social networks, which are above 50%. Moreover, as it often happens in networks generated by human activity, the transposed graph shows better compression performance because the in-degree distribution (shown in Figure 9.2(a)) has a fatter tail than the out-degree distribution, (Figure 9.2(b)) resulting in nodes with very dense predecessor lists.

Practically speaking, either direction of the input corpus can be fit in less than 100 GiB of RAM, even including the 4 GiB type map discussed above. Such an amount of memory can be easily installed on either powerful workstations or cheap server-grade hardware—at

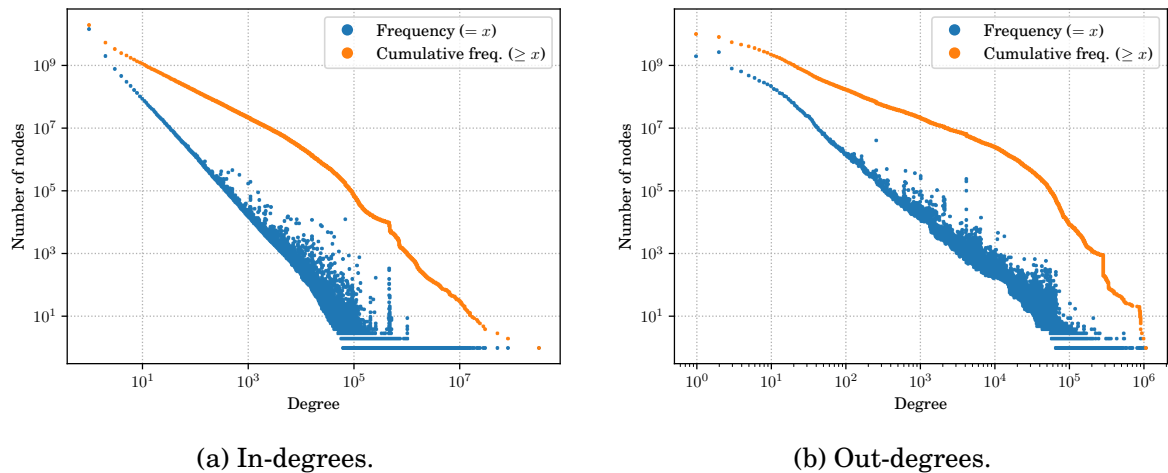


Figure 9.2: In-degree and out-degree distributions for the entire corpus as a graph. Note how the in-degree distribution has a fatter tail than the out-degree one leading, as we will show experimentally, to better compression (but worse traversal) performances for the transposed graph.

current market rates,⁴ 100 GiB of main memory costs less than 300 USD. Fitting *both* graph directions on workstation deployments might be more challenging, but it is not always needed (e.g., one can choose to load only one graph direction depending on experimental needs) and still fit cheap server-grade deployments by current standards.

9.3.4 Compression with Layered Label Propagation

Further compression improvements can be achieved by the Layered Label Propagation (LLP) algorithm [28] presented in Section 9.2 to reorder nodes. The LLP algorithm finds locality-preserving orders by clustering together nodes in close proximity. Similar to the BFS, this algorithm is particularly interesting for our use case as it is unsupervised, and does not rely on prior information on the clusters present in the graph. The idea behind the clustering algorithm is to randomly distribute communities to the nodes in the graph, then iteratively assign to each node the community most represented in its neighbors.

LLP is more costly than simple BFS-based compression in both time and memory. Even though the algorithm has a linear time complexity, it does multiple iterations on the graph and is significantly slower than the BFS which is just one single traversal. Moreover, keeping track of the communities requires a total of 33 bytes per node, which increases the RAM requirements to more than 600 GiB.

Because of these constraints, it is unrealistic to run the LLP algorithm on the uncompressed version of the graph which already takes around 800 GiB of RAM. Thus, instead of replacing BFS reordering compression, the LLP reordering is computed *from* the previ-

⁴<https://jcm.it.net/memoryprice.htm>, accessed 2019-10-18

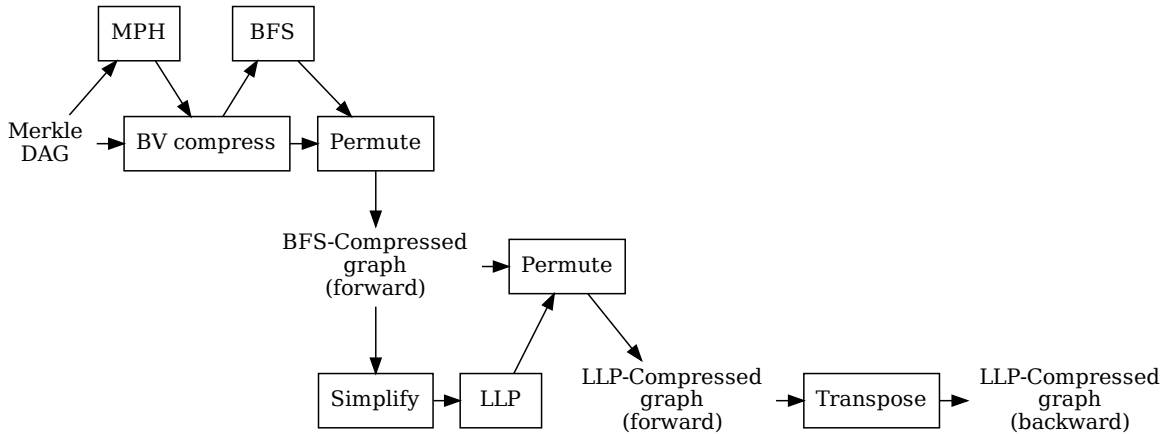


Figure 9.3: Graph compression pipeline using LLP ordering.

ously compressed graph, and therefore comes after the full compression pipeline shown in Section 9.3.2.

Figure 9.3 shows the full compression pipeline augmented with LLP reordering. Because the algorithm tries to find clusters of nodes which share a common neighborhood, it operates on the *undirected* version of the graph, i.e., a graph in which all the edges are transitive and can be used in both directions. This “symmetric” graph can be obtained by running the **Simplify** step, which returns a loopless and symmetric graph from the BFS-compressed graph. This symmetric graph is then fed to the **LLP** step, which computes the reordering.

The LLP algorithm is parameterized by a list of γ values, a “resolution” parameter which defines the shapes of the clustering it produces: either small, but denser pieces, or larger, but unavoidably sparser pieces. The algorithm then combines the different clusterings together to generate the output reordering. γ values are given to the algorithm in the form $\frac{j}{2^k}$; the default γ values used in the LLP implementation of the WebGraph framework are:

$$\left\{ 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}, \frac{1}{128}, \frac{1}{256}, \frac{1}{512}, \frac{1}{1024}, 0 \right\}.$$

However, the combination procedure is very slow, and giving that many γ values could take several months in our case. We thus need to narrow down a smaller set of values that give good compression results. In order to find an optimal combination of γ values for compressing the graph of software development, we performed a hyperparameter sweep with different lists of values. The results are shown in Table 9.5.

As can be seen, the compression ratio is significantly improved using the LLP algorithm. In the best case, the size of the compressed graph is reduced by 35% from the BFS baseline. If the graph is to be loaded fully in RAM, such differences are very significant to determine the practicality of the approach on some given hardware. The additional complexity of adding LLP in the compression pipeline can therefore be easily justified.

Table 9.5: LLP compression results for different γ values, compared to the BFS-compressed baseline.

γ values	Forward graph			Transposed graph		
	size	bits/arc	ratio	size	bits/arc	ratio
BFS baseline	118 GiB	5.2	16.2%	94 GiB	4.11	12.9%
0, 1, $\frac{1}{64}$	91 GiB	3.98	12.5%	82 GiB	3.58	11.2%
$\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$	77 GiB	3.37	10.6%	62 GiB	2.71	8.5%

An interesting result is that smaller values of γ seem to generate better compression ratios. The effect of a given γ is that the density of the sparsest cluster is at least $\frac{\gamma}{\gamma+1}$, so large γ values imply small, more dense clusters. It is reasonable to assume that since the graph is very sparse to start with, such clusters are not that useful.

The main takeaway of this section is that, from a size perspective, graph compression *can be used* to fit the structure of immense VCS datasets in memory on a single machine with limited hardware resources. Using a simple graph traversal for reordering, the graph size can be brought down to under around 150 GiB. Depending on the resources available for compression, the graph can be further compressed by applying more expensive reordering algorithms like LLP.

9.4 Exploitation

We now move to the speed perspective to experimentally assess how effectively the obtained compressed representation of ultra-large-scale repository collections can be leveraged to perform repository mining experiments. We first perform a few domain-agnostic benchmarks (e.g., graph visits, arc traversal time) and then perform a domain-specific experiment.

9.4.1 Graph traversal

In the worst case of any given mining experiment, one will have to traverse the entire corpus to obtain some insights. Hence, it is important to know the baseline of how long a single traversal takes. Table 9.6 shows the results of benchmarking complete graph visits in breadth-first order (BFS), with no parallelism (single thread) for both the original and transposed graphs. Timings have been measured on a server equipped with 3 GHz Intel Xeon Gold 6154 CPUs (only one of which has been used for the visits), with enough RAM to load either graph direction in memory without swapping to disk.

Results show that the in-memory approach delivers impressive performances. A full visit of the forward graph takes less than 2 hours with a throughput nearing 2 million

Table 9.6: Full graph visit benchmarks for a single-threaded BFS visit.

Forward (original) graph		Backward (transposed) graph	
wall time	1h48m	wall time	3h17m
throughput	1.81 M nodes/s (553 ns/node)	throughput	988 M nodes/s (1.01 μ s/node)

Table 9.7: Arc lookup benchmarks for 1 billion random nodes.

Forward (original) graph		Backward (transposed) graph	
visited arcs	13,644,656,586	visited arcs	13,625,228,259
throughput	12,018,223 arcs/s (83 ns/arc)	throughput	9,453,613 arcs/s (106 ns/arc)

nodes per second, or about 500 nanoseconds per node. Visiting the transposed graph is slower due to the already discussed differences in in-degree vs. out-degree distributions, but still very fast in absolute terms: the full transposed graph can be visited in little more than 3 hours with a throughput nearing 1 million nodes/second (1 μ s/node).

Table 9.7 shows the results of benchmarking random access to nodes and edges in the graph. A random sample of 1 billion nodes (8.3% of the entire graph) has been taken, enumerating for every node all of its successors. Results show that having the graph in memory gives impressive results also in terms of random lookup time, with minimal overhead due to the compressed representation. On the original graph, looking up the successor of a node takes 83 nanoseconds on average, close to the 50–60 ns estimates for current DRAM random access memory. Arc lookup on the transposed graph is slower, as already observed for full graph visits.

9.4.2 Source code artifact multiplication

As the goal is to benchmark the potential of the proposed approach for ultra-large-scale repository analysis, we focus on a domain-specific experiment which needs to intensely crawl VCS histories in the studied corpus. Specifically, we will replicate the experiments of [137] to quantitatively assess the *multiplication factor* of source code artifacts in the corpus. We will measure:

1. The multiplication of source code file contents across commits (*blob* \rightarrow *revision multiplication* in the following), i.e., how much the same unmodified file content reappears in different commits, regardless of which origins they were found in. This measure correlates with and is a requirement for Type 1 (exact) clone detection [21], both within and across repositories.
2. The multiplication of commits across origins (*revision* \rightarrow *origin multiplication*), i.e., the frequency with which the same commit reappears in different repositories. The fact

that the same commit reappears at different origins is partly the result of collaborative social coding, but can also hint at the migration of development from one platform to another.

In the given data model, blob \rightarrow revision multiplication can be measured by iterating on all nodes of type “blob”, then performing a visit on the transposed graph that only follows arcs leading to revision nodes—i.e., excluding (the transposed of) arcs snapshot \rightarrow revision, release \rightarrow revision, and origin \rightarrow snapshot—and finally counting the number of revision leaves reached with the visit. Results can then be visualized as a distribution of the “popularity” of blobs across commits.

Note that, even if the compressed graph representation does not natively store type information for arcs, we can use the type map discussed in Section 9.3.1 to stop the visit when it is no longer possible to reach additional revision nodes.

The approach for determining revision \rightarrow origin multiplication is similar, with the difference that visits will start from commit nodes and stop at origin nodes (graph roots), which can then be counted and visualized as before.

The chosen approaches are naive from an algorithmic point of view—the same edges will be traversed repeatedly for different input nodes, resulting in a time complexity of $O(|V| \cdot |E|)$, which is generally considered impractical on graphs of this scale. Having already established the overall efficiency of full graph visits, we chose these approaches for the sake of simplicity and explainability.

More time-efficient approaches are possible and would be equally well supported by the compressed graph framework. For instance, one could propagate ancestry information during the visit, obtaining linear-time algorithms at the price of extra memory requirements (or extra memory mapping). Our main point is that the entire corpus *itself* can be fit in relatively little memory and accessed with excellent performances; other algorithmic considerations will vary according to the needs of the planned mining task, as they always do.

9.4.3 Results

Figure 9.4 shows the multiplication factor distribution for blob \rightarrow revision, measured on a random sample of 953 M blobs. Looking at the cumulative distribution (top line) the average multiplication factor appears to be very high. There are more than 100 million blobs ($\approx 20\%$ of the sample) that are duplicated more than one thousand times in different commits; 10 million blobs (1%) reoccur in more than ten thousand commits; and 1 million blobs (0.1%) in more than a hundred thousand commits.

As we did not partition by origin, these results do not say whether this multiplication is due to the long life of unmodified source code files in long-lived code bases, or instead

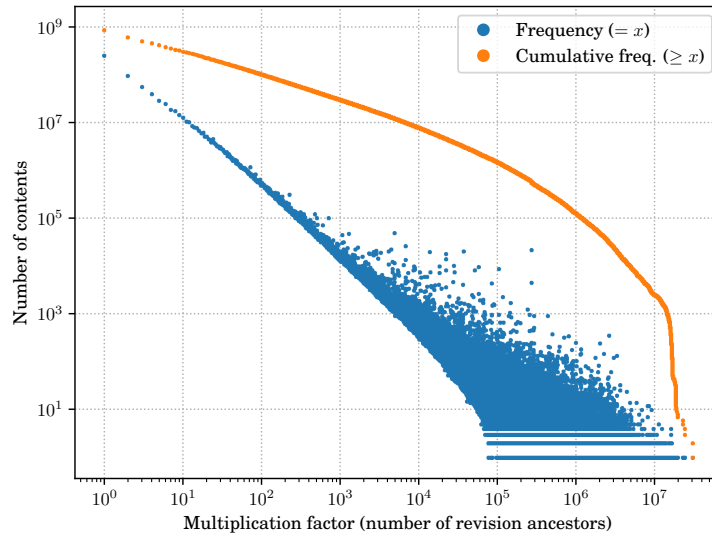


Figure 9.4: Content \rightarrow revision multiplication, i.e., how often file contents (Y axis) reoccur unmodified in different commits (X axis), based on a random sample of 953 M blobs (17% of all blobs).

due to reuse of the same unmodified files across different repositories. It would be easy to modify the experiment to follow edges up to origin nodes to determine that.

Figure 9.5 shows analogous results for the revision \rightarrow origin layer, on a random sample of 8.5 M revisions. To interpret them, it is important to realize how it happens that the same commit (i.e., with an identical SHA-1 identifier) is found in different repositories. The main reason is the distributed nature of modern version control systems, whose repositories are massively represented in the corpus under analysis. Developers that work together using Git will have individual repositories that communicate by exchanging SHA-1-identical commits. Furthermore, the pull request development model [67] popularized by GitHub natively creates new repositories (hosted at different origin URLs) that initially contain all commits (unmodified) of the originally “forked” repository.

The cumulative distribution of Figure 9.5 measures the amount of redistribution of the same commits via different repositories in our corpus. 5 million commits (60% of the sample) can be found in a single repository only, but multiplication grows quickly from there: 100 thousand commits (1%) can be found in 1000 repositories or more and 10 thousand commits (0.01%) can be found in 10 thousand repositories or more. We observe that software development is nowadays very distributed, to a point of strong resilience to the disappearance of a single point of source code distribution.

To better appreciate the graph-intensive nature of realizing the above two experiments in practice, we have also measured visit sizes. Figure 9.6(a) and Figure 9.6(b) show the

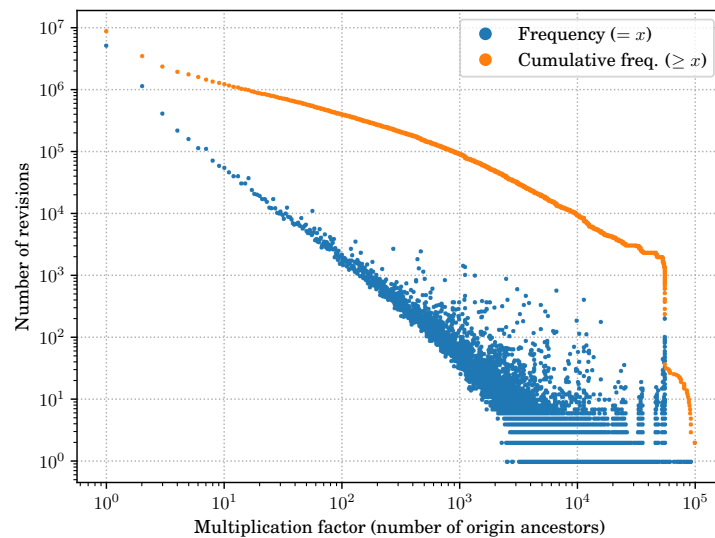


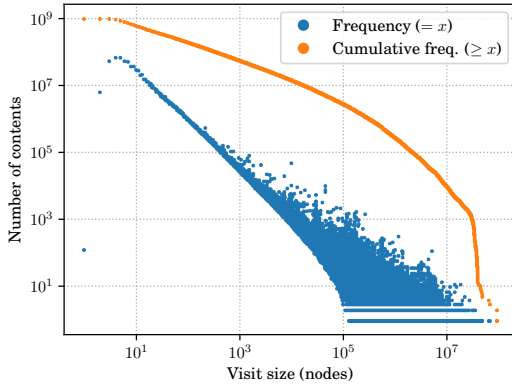
Figure 9.5: Revision \rightarrow origin multiplication, i.e., how often commits (Y axis) reoccur in different repositories (X axis), based on a random sample of 8.5 M revision (0.77% of all revisions).

results in terms of visited nodes, while Figure 9.6(c) and Figure 9.6(d) show them in terms of visited edges.

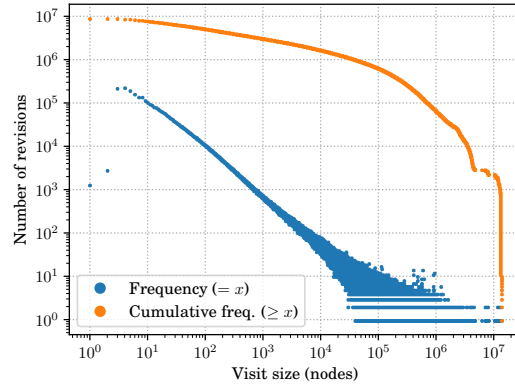
Content \rightarrow revision visits traverse significantly more nodes (and edges) than revision \rightarrow origin visits. This is expected due to the respective sizes of the traversed subgraphs and it is a dominant factor over the fact that, on average, the filesystem layer of the graph (blobs \cup directory nodes) has shorter graph paths than the revision layer (VCS repositories of large software projects such as the Linux kernel can have commit chains nearing 1 M commits in length).

In terms of timings, the presented results have been obtained on multi-core machines with, respectively, 20×2.40 GHz CPUs (for blob \rightarrow revision) and 36×3.00 GHz CPUs (for revision \rightarrow origin), letting the experiments run for about 2.5 days in total. In spite of the naive $O(|V| \cdot |E|)$ algorithmic approaches chosen, such short running times have enabled the processing of very large subgraphs.

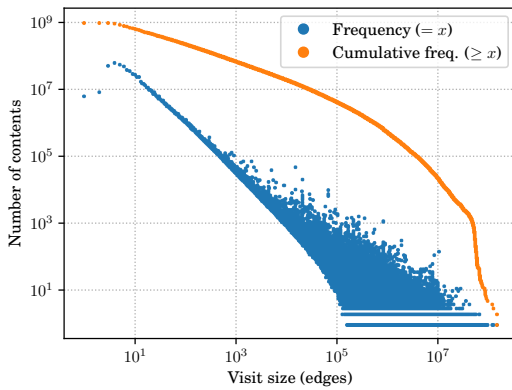
The main conclusions we can draw from these experiments are that: (1) graphs are suitable data models for conducting version control system analyses, including code duplication experiments; and (2) *compressed* graphs allow such analyses to be performed at ultra-large-scale with impressive graph traversal performances.



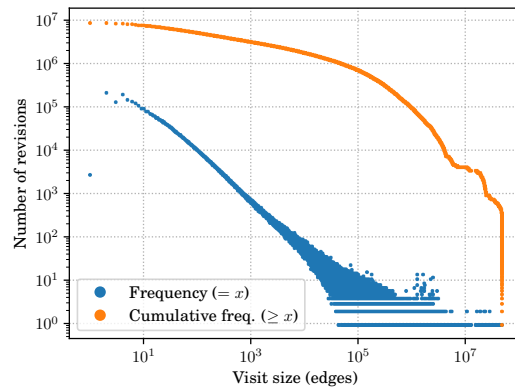
(a) Visit size, as the number of visited *nodes*, for measuring blobs \rightarrow revision multiplication on the same sample of Figure 9.4.



(b) Visit size, as the number of visited *nodes*, for measuring revision \rightarrow origin multiplication on the same sample of Figure 9.5.



(c) Visit size, as the number of visited *edges*, for measuring blobs \rightarrow revision multiplication on the same sample of Figure 9.4.



(d) Visit size, as the number of visited *edges*, for measuring revision \rightarrow origin multiplication on the same sample of Figure 9.5.

Figure 9.6: Distribution of the number of nodes and edges visited for the multiplication factor experiments.

9.5 Discussion

Both size-wise and performance-wise the results of applying graph compression to VCS graphs to support their ultra-large-scale analysis appear to be more than satisfactory. VCS graphs compress well both in absolute terms and in comparison with other large graphs compressed in the past (e.g., web graphs). Graph visit performances are consistent with main memory access time and can support visit-intense VCS analysis needs well on limited resources. That notwithstanding we are not claiming that graph compression is a silver bullet for VCS analysis. We discuss in this section limitations and trade-offs that apply to the proposed approach.

9.5.1 Graph design

As mentioned in Section 9.3.1 there exists a clear *space/time trade-off* between what fits in main memory and what should be left in secondary storage. Our choice—graph structure + node types in memory, everything else in secondary storage—will enable speeding up many use cases, under the assumption that VCS traversal is a common performance bottleneck of large-scale analyses.

Other choices are possible, valid, and can still be supported by graph compression, allowing *more* information to fit in RAM than what would be possible otherwise. We discuss a few possible scenarios below.

One might decorate graph nodes with *additional metadata* to be used during traversals and should hence be looked up efficiently. For instance, commit nodes might be equipped with in-RAM timestamps (either absolute, or relative, as in logical clocks) to direct visits to choose the earliest occurrence of what is being sought. Graph compression will be useful nonetheless, and the induced re-labeling of nodes to integers often enables storing additional metadata in memory in very compact ways, as we did for the type map. Attaching metadata to graph *arcs* is more tricky, but limited support for doing so is offered by WebGraph⁵ and other state-of-the-art frameworks. This is further discussed in Chapter 10.

Graph semantics is another variable that can be played with. In the presented case study graph leaves are unmodified file contents, enabling the tracking of bit-identical clones. One might instead use checksums of *normalized* source code files (e.g., removing spaces), obtaining the canonical definition of Type 1 clones, or parse source code files to ASTs and associate intrinsic identifiers to them (for Type 2 clones).

Graph granularity can also be increased, e.g., by adding nodes corresponding to individual lines of codes (normalized or otherwise). Doing so would enable tracking SLOC cloning and migrations at an unprecedented scale. It will also significantly increase the graph size, by a factor close to the average length of source code files (in SLOCs). The resulting graph

⁵via the `webgraph.labelling` package

will be huge, but as graph compression techniques are used in production on web graphs (with nodes in the trillions), we are confident they can scale up to SLOC analysis needs.

9.5.2 Limitations

A limitation of using static graph compression over classic information systems to store the data to be analyzed is that *compression is not incremental* with regard to the arrival of new artifacts (commits, files, etc.), due to the need of reordering nodes and updating the compressed representations of adjacency lists. Considering that popular VCS repositories receive hundreds of new commits a day, the already mentioned exponential growth of original publicly available code, and the observed multi-day compression times—this means that analyzed data will be chronically out-of-date w.r.t. reality.

This limitation is not problematic for research use cases, where datasets are generally frozen before conducting an experiment; but it might be problematic for other needs, like live-monitoring of interconnected private/public code bases. This is not a novel problem for other fields in which graph compression is applied, such as web search and social network analysis. Some compression techniques (e.g., k^2 -trees [32]) lend themselves more or less naturally to be adapted to the dynamic case, but with a definite degradation in compression performances. A common mitigation technique that can be applied to all static compressors is to use dynamic, updatable *overlay graphs* on top of the compressed one. Overlays will be more memory-hungry, but they are ephemeral: periodically the underlying compressed graphs will be recompressed to regain storage-efficiency.

It is also theoretically possible to exploit knowledge of the graph topology to leave “gaps” in the node ordering that can be used as room for adding new nodes of a given type dynamically to the compressed graph representation. We intend to explore this possibility in future work.

9.6 Related work

The main novelty presented in this chapter is to adapt and exploit known graph compression techniques for the purpose of large-scale mining. This new method builds on several extant works in the literature of these two different fields, that we summarize below.

9.6.1 Large-scale mining approaches

Other large-scale repository mining approaches have been proposed in the past. Boa [50] has pioneered the idea of a mutualized infrastructure hosting both data and compute resources to perform large-scale analyses on source code artifacts such as those discussed in this chapter. Our notion of “ultra-large-scale” is different, with a case study 100x larger by several metrics (projects, files, commits, etc.). The computing approach is also different,

with Boa relying on distributed clusters (Hadoop) and our approach relying on a single machine. Direct performance comparisons are not possible as we have not replicated their experiments, but our results hint at a very significant speedup when the main bottleneck is history traversal. It would be interesting—and it seems entirely possible—to realize an infrastructure like Boa, based on a compressed VCS representation like the one proposed in this chapter.

World of Code (WoC) [92] is a recent attempt at a mutualized infrastructure for large-scale VCS analyses. While limited to GitHub—contrary to our case study that also encompasses GitLab and major package repositories—the target scale of WoC is similar to ours. Again, the computing approach is different, with WoC relying on distributed databases running and ours on a single machine. The advantage of WoC is that it maintains pre-computed mappings, e.g., from files and directories to the places they come from, choosing a different spot than ours in the classic space/time trade-off. The approach proposed here looks more appealing in terms of cost and ease of deployment. But the two approaches are complementary: WoC might benefit from an *additional* compressed graph representation that would shine when users need to explore on the fly (and as quickly as possible) artifact relationships that are not available as pre-computed mappings.

Large-scale VCS analyses have been conducted in the past, usually at much smaller scale than ours. An exception is Rousseau et al. [137], which developed a compressed representation for software provenance tracking and used it to conduct the multiplication experiments that we have replicated in Section 9.4.2. Both approaches can be applied to conduct analyses on commodity hardware, but the compression trade-offs are different: specialized, lossy, and incremental in [137]; general purpose, lossless, but not incremental here.

LISA [6] is a framework for reducing artifact redundancy when analyzing VCS-stored source code. It is more fine-grained than our case study, reaching down to abstract syntax tree nodes. As such it could deduplicate more, but at the price of requiring a proper parser, which is not always available and might fail on syntactically incorrect files that one might still want to analyze. Also, we deal with all kinds of VCS artifacts while LISA is specific to source code files.

Large-scale experiments on development activities *not* captured by VCS histories (e.g., pull requests, code reviews, bug tracking, etc.) have also been conducted. They generally rely on dedicated activity databases, such as GHTorrent or GitHub Archive [68, 133]. Differences from the proposed approach are significant both in terms of scope (we focus on VCS histories, them on other activities) and needed resources.

9.6.2 Graph compression techniques

The problem of finding compression-friendly node orderings was studied from a theoretical viewpoint in Chierichetti et al. [37], where the authors show that the problem of determining the optimal renumbering of nodes is NP-hard, but propose a heuristic (called “shingle ordering”) for the problem, based on a fingerprint of the out-neighborhoods.

More recently, Dhulipala et al. [45] extended the theoretical model of [37] and designed a novel ordering technique, called *recursive graph bisection*, that yields the most competitive compression ratios in several cases. The core idea is to recursively divide the graph into two clusters of the same size so as to minimize an objective function that estimates the compressibility of the overall graph when nodes in the first cluster have smaller identifiers than nodes in the second cluster.

To the best of our knowledge, the first technique would not scale to our use case. Recursive bisection might have some margin of benefit over a BFS or LLP reordering, but at the price of a significantly slower computation. We plan to study this problem in the future.

A complementary approach to storing adjacency lists is that of k^2 -trees [33]. In this case, one aims at compactly representing the adjacency *matrix* of the graph (as opposed to its adjacency *lists*), exploiting its sparseness. However, k^2 -trees do not scale beyond a few dozen million vertices. They are hence inapplicable to the target scale of this thesis.

Since successor lists are increasing integers, one can use *succinct data structures* [112] to store them. A succinct data structure does not *compress* data, it represents it using the minimum possible number of bits instead. More precisely, if an object requires z bits to be represented, a succinct data structure uses $z + o(z)$ bits.

A simple, practical data structure of this kind is the *Elias-Fano representation of monotone sequences* [51], which are technically not fully succinct but *quasi-succinct*. An advantage of the Elias-Fano representation is the possibility of finding in constant time both the k th successor and the first successor greater than or equal a given bound b . In particular, the last property makes it possible to verify adjacency in constant time, and neighborhood intersections in sublinear time.

Partitioned Elias-Fano [118] is an improvement over the basic Elias-Fano representation: given a number k of parts, an algorithm splits the successor lists in k blocks in such a way that the space used is minimized. In this way, dense blocks as those generated by similarity can be compressed efficiently, with a constant additional cost for the access operations.

We remark that using a separate succinct encoding for each successor list can be more efficient than succinctly encoding the entire graph (for example, using an Elias-Fano list to represent the positions of the ones in the adjacency matrix, or using truly succinct representations [52]). This happens because by Jensen’s inequality the average cost *per*

successor is necessarily lower. However, if one also takes into account the constant costs associated with storing each successor list, there is a trade-off that has to be evaluated on a case-by-case basis. The Elias-Fano representation has been recently made dynamic [122] when the universe is polynomial in the list size, which makes it possible to use it in the case of dynamic graph representation.

Succinct approaches are particularly useful when the graph has no evident structure (as in that case reference compression does not really help) and when reordering the nodes is not possible (as the compression obtained is agnostic with respect to the distribution of gaps and to similarity). We therefore did not consider these approaches relevant to the case at hand.

Property Data in the Compressed Graph

10.1 Introduction

In Chapter 8 we introduced the Software Heritage Property Graph Dataset, a corpus made available in relational formats suitable for large-scale analysis through scale-out processing on large Big Data clusters. In Chapter 9, we proposed a method to compress the graph to fit a few hundred gigabytes to efficiently run shared-memory algorithms on a single machine. These two approaches are designed to be complementary: scale-out processing is particularly suited for embarrassingly parallel workflows and can process terabytes of data in a few minutes with the appropriate scale factor; in-memory graph compression can run more complex algorithms exploiting the recursive structure of the graph (traversals, connected components, etc.) in a relatively cheap way, albeit with longer runtimes of a few hours to a few days.

However, this latter approach is more limited in the kinds of data it can exploit. The compressed graph only contains the topological structure (nodes and arcs) of the graph, whereas the graph dataset in relational format can be used to study the entire *property graph* of development history, i.e., not only the graph structure of the dataset but also the properties associated with each node and edge (e.g., commit messages, timestamps, file names, etc.). Because of this, recursive graph queries cannot use graph properties during their traversals. This can be worked around by performing graph queries without using the properties themselves, then *joining* the results with data from the relational property graph, but this can be significantly more expensive and time-consuming. As an example, let us consider the following query:

Query 1: *Given a blob object, find the earliest revision in which this blob can be found*

anywhere in its source code tree.

If the commit timestamp property can be accessed “on the fly” during graph traversal, this query can be answered in a rather straightforward manner: for each revision in the transitive closure of the blob in the transposed graph, find the one with the lowest timestamp. However, if these properties are only available externally, the process is more involved: it requires returning the entire transitive closure from the blob, then joining it with the timestamps, which can require transferring very large amounts of nodes to join with the timestamps (up to significant fractions of the entire graph, e.g., for the empty file), before finally filtering on the lowest timestamp found.

The added complexity gets even worse when the traversed edges are conditional on the properties themselves. Consider the following query:

Query 2: *Given a snapshot, find all unique blobs which were present at the path `src/test/README.txt` at any point in its development history.*

With direct access to the properties of the graph, the algorithm is again straightforward: (1) traverse the revision history (2) for each revision, follow the given path in its source tree (3) add the destination blob to the resulting set if not present. This only requires following a single path of fixed depth for each source code tree, for a time complexity of $O(|R|)$ where R is the set of revisions in the transitive closure of the snapshot. In contrast, when the file and directory names of the graph edges are not directly accessible, one needs to return the *entire transitive closure* of the snapshot in $O(|N|)$ to later join this data with the graph properties, and use these to filter the nodes matching the condition in the query.

Whether the compressed graph is used only for selecting artifacts for the purpose of performing subsequent scale-out analyses on the extracted data, or directly used for the analyses themselves, having direct access to the graph properties from the compressed format would significantly improve the efficiency of more complex graph queries. In this chapter, we look at ways to bridge this potency gap between the two formats by enabling access to node and edge properties from the compressed graph.

10.2 SWHID mappings

The most basic property of the graph nodes is their Software Heritage Persistent Identifier (SWHID), their unique identifier in the archive. They are of prime importance to be able to relate the nodes being manipulated in the compressed representation with the archived artifacts themselves.

The input of the graph compression pipeline is a list of nodes and edges referenced by their SWHIDs in a format described in Section 8.4.3. As detailed in Section 9.3.2, this pipeline works by mapping each unique SWHID to a set of consecutive integers $\{0, \dots, N - 1\}$, where N is the number of nodes in the graph by computing a *minimal*

perfect hash function [61]. After this hashing step, the SWHID information is lost, and all the nodes are subsequently referred to as their integer identifier from the contiguous set. To restore the ability to associate the graph nodes with the objects they refer to, we need to construct mapping functions that allow us to translate between integer node IDs and SWHIDs in both directions.

10.2.1 SWHID \rightarrow Node ID

The natural way to map SWHIDs to the compressed node IDs is to reuse the minimal perfect hash function which was used to map the nodes in the first place. This hash function is stored as a `graph.mph` file (which weighs around 5 GiB for the 2020-12-15 version of the graph) and can translate the input SWHIDs to node IDs in this set. However, during the compression pipeline the graph is *reordered* to achieve better compression results, as described in Section 9.3.2. When compressing with the LLP algorithm, the graph is even reordered twice: once for the initial BFS-based compression, and then once for the LLP-based compression. As a consequence of these reorderings, the node IDs obtained from feeding SWHIDs to the MPH function no longer correspond to their matching output nodes.

A byproduct of the compression pipeline are “order files” which define the *permutations* of each reordering step. Their on-disk format is a binary array of integers in which the integer at position x is $p(x)$ where p is the permutation of the reordering step. As such, the SWHID \rightarrow node ID mapping can be obtained by composing the MPH with the subsequent reorderings. As a preliminary step, if the graph was permuted multiple times (as is the case for LLP), we compose the `.order` files to keep a single permutation p_G , which corresponds to the resulting permutation of successively applying all the compression permutations of the graph:

- For BFS-based compression: $p_G(i) = p_{\text{BFS}}(i)$
- For LLP-based compression: $p_G(i) = (p_{\text{LLP}} \circ p_{\text{BFS}})(i)$

This lowers memory usage by reducing the number of order files necessary for the translation to node IDs. For reference, an order file representing a permutation for 19.3 billion nodes (the size of the 2020-12-15 dataset) weighs around 145 GiB ($\approx 19.3 \times 10^9 \times 8 \times \frac{1}{2^{30}}$ for 64-bit node IDs). This composed permutation (or “graph permutation”, as it represents how the graph is permuted from the original order of the nodes given by the hashing function) is stored as a `graph.order` file in the same format as the input permutations.

Then, the mapping function for the SWHID \rightarrow node ID direction can be defined from the MPH and the composed permutation:

$$swhid2node(s) = p_G(mph(s))$$

In other words, getting the node ID associated with a given input SWHID s can be done through the following two-step process:

1. Hash the SWHID s using the MPH function loaded from the `graph.mph` file to obtain k_0 , the node ID from the initial graph order.
2. Take the image of k_0 from the graph permutation stored in the `graph.order` file to obtain the node ID k .

The storage space taken by the entire mapping is the on-disk size of the permutation plus the on-disk size of the hash function, which is ≈ 150 GiB in the 2020-12-15 dataset. This is comparable to the size of the graph itself (134 GiB for the forward graph only). This size could be reduced by compressing the `.order` file, which could be achieved by reducing the number of bits used for each entry in the permutation to the bare minimum necessary to represent an integer in the set of node IDs. Doing so would come at the cost of some inefficiency due to the overhead of correcting word misalignment, which would potentially be negligible. For the same example, the estimated size would then be:

$$n \times \left\lceil \frac{\log_2(n)}{8} \right\rceil \text{ bytes} = 19.3 \times 10^9 \times \left\lceil \frac{\log_2(19.3 \times 10^9)}{8} \right\rceil \times \frac{1}{2^{30}} \approx 90 \text{ GiB.}$$

10.2.2 Node ID \rightarrow SWHID

Now that we are able to easily convert SWHIDs to their corresponding node IDs in the compressed graph, we can build a *reverse mapping* which allows us to retrieve the SWHID preimage of any node ID found in the compressed graph. This reverse mapping is the inverse function $node2swhid$ ($= swhid2node^{-1}$) of the mapping described in the previous section.

This function is relatively easy to represent on disk for two reasons. First, its input domain is a contiguous set of integers $\{0, \dots, N - 1\}$, which means the function does not require any hashing scheme to associate images to its inputs; they can simply be stored as a contiguous array. Second, because the images are SWHIDs they all have a fixed size, which facilitates random access in the array.

This reverse mapping is stored in a file called `graph.node2swhid.bin`, a binary file containing a contiguous binary sequence of records, each record representing a SWHID. The binary format used to represent a SWHID as a 22-byte long byte sequence is constituted of:

- 1 byte for the namespace version, represented as a *C unsigned char*
- 1 byte for the object type as the integer value of a software artifact type enum (0 = content, 1 = directory, 2 = origin, 3 = release, 4 = revision, 5 = snapshot), represented as a *C unsigned char*

- 20 bytes for the SHA-1 digest represented as a byte sequence.

This mapping can be generated relatively straightforwardly by (1) allocating a binary file of $22 \times n$ bytes, (2) iterating on the list of all the SWHIDs used to compress the graph, (3) for each SWHID, getting k , its image node ID, using the mapping described in Section 10.2.1, (4) writing the 22 bytes binary record representing the SWHIDs at the offset $22 \times k$ in the binary file.

Once this mapping is written to disk, the SWHID of a node k can be retrieved by simply reading the 22-bytes binary record at index $22 \times k$. A `graph.node2swhid.bin` file mapping 19.3 billion node IDs (the size of the 2020-12-15 dataset) to their corresponding SWHIDs weighs around 395 GiB ($\approx 19.3 \times 10^9 \times 22 \times \frac{1}{230}$).

10.2.3 Domain checking

One caveat of the approach described in Section 10.2.1 to map SWHID \rightarrow node ID is that it does not do any kind of checking on its input domain; an unknown/invalid SWHID will be accepted by the hash function, which will return an arbitrary integer in its image $\{0, \dots, N - 1\}$ instead of throwing an error. This is problematic for some use cases: if the compressed graph is exposed as an API, it needs to be able to reject queries using unknown SWHIDs, instead of silently computing a garbage result on the wrong input node without any way for the user to notice.

While it is possible to externally check whether a SWHID is present in the Software Heritage archive by querying its API, this is not sufficient for this use case. The compressed graph is not built incrementally but is instead a *static* export of the state of the archive at a given point in time. This means that all nodes added to the archive during the time period between the graph compression and the present are unknown to the compressed graph.

One option to add checks to the input domain would be to add **signing** to the MPH function. By associating a signature of w bits to each input key of the MPH function, it is able to detect input strings which were not in the original key set. This approach is probabilistic, and false positives are possible with a probability of $\frac{1}{2^w}$. While this is useful to catch bugs, a probabilistic approach does not allow us to systematically fall back when encountering an unknown SWHID unless using very large signatures. These can also dramatically increase the size of the hashing function: for the 2020-12-15 dataset, a 32-bit signature increases its size from 5 GiB to 72 GiB while still allowing a false positive every 4 billion SWHID on average, which is less than a quarter of the number of objects in the graph.

Instead, we consider another option based on **round-trips** on the mapping from Section 10.2.1 and reverse mapping from Section 10.2.2. Because both mappings are bijections which are the inverse of one another, it follows that for every SWHID s in the input key set of the MPH, $node2swhid(swhid2node(s)) = s$. In contrast, if s is *not* in the input domain,

$swhid2node(s)$ will return an arbitrary node, and $node2swhid(swhid2node(s))$ will then return a random SWHID *from the input domain*, which thus cannot be equal to s .

Given this, it is possible to create a function which maps SWHID \rightarrow node IDs *with domain checking* by performing this “round-trip” to verify that the node is in the input domain, and returning an error otherwise:

$$swhid2nodeCheck(s) = \begin{cases} swhid2node(s) & \text{if } node2swhid(swhid2node(s)) = s, \\ -1 & \text{otherwise.} \end{cases}$$

10.3 Node properties

After having restored the association between the nodes in the graph and their archive identifiers, we now move to storing node properties *other* than SWHIDs. For the reasons outlined in Section 10.2.2, mapping nodes to properties in external storage is generally a relatively simple feat. Because the node IDs in the compressed graph are consecutive integers in the range $\{0, \dots, N - 1\}$, mapping each node to a given property is equivalent to creating a random-access list of properties, in which the property found at index k is the one associated with the node k .

The best way to achieve this depends on the type and characteristics of each property, detailed in Section 8.2. We can use these to organize the properties in different categories, for each of which we describe an implementation for compact on-disk storage in this section.

10.3.1 Node types

In Section 9.3.1 we already mentioned keeping *node types* in memory, i.e., whether a node is a blob, directory, revision, etc. Node types have a special status because they are necessary for almost all kinds of analyses on the graph as they allow determining at runtime which kinds of objects are being traversed. They are also already present as part of the textual identifiers of the nodes themselves, being one of the fields in SWHID. For that reason, we systematically load this node type mapping in memory along with the graph, contrary to the other mappings which have to be loaded on demand. This mapping is implemented very efficiently as a bit array indexed by integer node identifiers, with records of only 3 bits per node (as there are 6 node types in total), which amounts to around 7 GiB of RAM for the 2020-12-15 graph.

10.3.2 Integer properties

The following properties are simple integers, either 16, 32 or 64 bits:

- `content.length`: size of a blob in bytes

- `revision.date`: timestamp of when a revision was authored
- `revision.date_offset`: timezone offset of when a revision was authored
- `revision.committer_date`: timestamp of when a revision was committed
- `revision.committer_date_offset`: timezone offset of when a revision was committed
- `release.date`: timestamp of when a release was authored

All the dates can be represented as a 64-bit timestamp storing the number of nanoseconds since the Unix epoch. The timezone information is stored as a 16-bit integer representing the timezone offset with UTC.

Storing these integer properties in property files is done in a similar way to storing permutations in order files, which we described in Section 10.2.1. Because integers have a fixed width, these properties can be stored in a relatively straightforward manner as binary array of integers in which the k th integer corresponds to the value of the property for node k . The resulting property files have a size of $n \times \frac{w}{8}$, where n is the number of nodes in the graph, and w the number of bits of the integer field (16, 32 or 64). For the 2020-12-15 export, these bit sizes correspond respectively to property files of size 35 GiB, 71 GiB and 145 GiB.

This could further be improved if the nodes of each type were stored contiguously, as it would allow us to store integers in a contiguous array without having holes for nodes of irrelevant types. Using such a typed graph would have various advantages in terms of property storage and locality-based compression, and is left as a future work.

10.3.3 Persons

The following properties represent “persons”, which are stored as records containing a full name and an email address.¹

- `revision.author`: author of a revision
- `revision.committer`: committer of a revision
- `release.author`: author of a release

Those fields are natively represented as byte sequences in the relational format. However, two factors come into play to figure out the best way to store these properties for the compressed graph. First, there are considerably fewer unique persons than there are objects

¹Like software projects, persons are ontologically complex objects and cannot be solely described by a name and an email: people have multiple emails, change names, can misspell their own name, etc. Past works have attempted to map authorship information to a canonical concept of “persons” by merging duplicate information to single identities [168, 175]. For the purposes of this thesis, we call “person” a unique pair of full name and email address, and leave identity deduplication in the hands of the users of the platform.

referring to persons: only 42 million persons in the 2021-03-23 graph export, whereas there are 2 billion objects containing person fields (revisions and releases), which highlights the usefulness of having some deduplication of unique authors. Second, as discussed in Section 8.2, the research use cases we want to support are not those exploiting the names and emails of the authors themselves (which we anonymize anyway), but rather those trying to relate identical authors, e.g., for the purpose of analyzing the structure of social coding networks.

In consequence, the only data we actually want to provide for this property is a unique identifier of a given person, which will then be used in place of the full name and email and can be compared across different objects. Because the graph is static, persons can be mapped to integers in the range $\{0, \dots, N - 1\}$ where N is the total number of persons referenced in the graph artifacts. This is once again achieved with the use of an MPH: we first extract the list of unique persons in the graph, then compute a minimal perfect hash on the full names and emails to associate each person to a unique integer. The integer array is then stored in binary format using the method described above for regular integer fields. There is no need to build and provide a reverse mapping for the anonymization reasons explained above.

10.3.4 Text

Finally, the most complex type of node fields in the property graph is the textual type, or rather, arbitrary sequences of bytes. The following properties use this byte sequence type:

- `revision.message`: descriptive message of a revision
- `release.message`: descriptive message of a release

Because these fields need to be properly readable (i.e., one should actually be able to get the entire byte sequence from a node, contrary to persons), there is not much optimization to be done for this case. We can use an arbitrary on-disk string list which maps each node ID to its property value, as long as it allows random access. For this purpose we use the `FrontCodedStringBigList` class² from the `dsiutils` library³ used in the `WebGraph` framework. It contains an index table to efficiently jump to the k th bytestring in the array. For the 2021-03-23 export, the total size of all the byte strings we need to store is 159 GiB, which is a manageable amount. However, `FrontCodedStringBigList` objects have to be loaded entirely in memory, which can be a significant memory burden. Improving this by making it possible to memory-map this data structure from disk is left open as a future implementation direction.

²<https://dsiutils.di.unimi.it/docs/it/unimi/dsi/big/util/FrontCodedStringBigList.html>

³<https://dsiutils.di.unimi.it/>

10.4 Edge properties

While most properties in the graph are associated with the objects themselves, some properties are associated with the relationships between them, which we call “edge properties”. While they are relatively few, we expect them to be of crucial importance for most analysis workflows, as they include file and directory names which are a key property used in software mining; file extensions in particular are one of the most relevant properties that researchers use to find artifacts of interest. The complete list of edge properties in the relational schema which we want to make available from the compressed graph is as follows (a more detailed description of these properties are available in Section 8.2):

- `snapshot_branch.name` (binary): name of a snapshot branch (snapshot \rightarrow * edges)
- `directory_entry.name` (binary): name of a directory entry (directory \rightarrow * edges)
- `directory_entry.perms` (int): permissions of a directory entry (directory \rightarrow * edges)

10.4.1 Data structure

As we have seen, attaching properties to graph nodes using an external on-disk mapping is relatively straightforward: because the nodes are mapped to a contiguous integer range, in all cases the data structure only needs to map this range to a set of properties, which can be done efficiently in most cases. However, for properties attached to the graph *edges*, doing so gets technically harder: the input domain of the mapping is the sparse adjacency matrix of the entire graph, which cannot be compactly represented in a naive data structure.

To solve this issue, the WebGraph framework provides an experimental set of libraries⁴ under the `webgraph.labelling` namespace which implements arc labeling, i.e., associating arbitrary properties to the graph edges. The basic principle for the serialization of labels (the objects storing the edge properties) using this system is twofold.

First, all labels are written to a `graph.labels` file in the order obtained by iterating on all edges of the graph. As a reminder, WebGraph stores the outgoing arcs of each node as an ordered adjacency list, and all the nodes in the graph are ordered from 0 to $N - 1$. Enumerating all nodes and all outgoing arcs of each enumerated node in the compressed graph yields a natural ordering for all edges stored in the graph, which can then be followed to store the labels on disk.

Second, a `graph.labeloffsets` file is used to index the offsets of the arc labels. This file can be seen as an integer array which associates a node k with the position in the `graph.labels` file from which the labels of its outgoing arcs can be read. This offset file allows random access to the labels on the outgoing arcs of a given source node.

⁴<https://webgraph.di.unimi.it/docs/it/unimi/dsi/webgraph/labelling/package-summary.html>

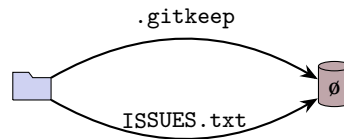


Figure 10.1: Two directory entries with different names pointing to the same blob (the empty file). This is stored as a single edge in the compressed graph, and thus has to be represented with multiple labels per arc.

In other words, given a node k , it is possible to lookup in constant time a position p_k from the offset file, then iterate on all outgoing edges of k while reading the arc labels file starting from p_k to successively retrieve all labels of the outgoing arcs of k . It is however not possible to get the specific arc label of a $k_1 \rightarrow k_2$ edge in constant time, as WebGraph does not allow access to arbitrary edges in constant time; instead, all outgoing edges of k_1 need to be iterated along with their labels until k_2 is encountered in the adjacency list, which has an average complexity of $O(d_{\text{out}})$, where d_{out} is the average out-degree of the nodes in the graph.

The labels themselves can be in any arbitrary self-contained format, as they are read sequentially from the label file and deserialized using a custom function. The simplest format for an arc label is to associate a record of a fixed number of bits to each arc. This poses an important challenge for our use case. Consider a directory with two files having different names but an identical content, for example two empty files, as shown in Figure 10.1. In our data model, this would be represented as two nodes (the directory and the blob corresponding to the empty file) and two edges from the first to the second node (one for each file entry). In the compressed graph however, due to the copy-list model of compression, these nodes would only be linked together by a single arc, to which we cannot associate a single file name.

To represent this possibility of having multiple file names for a single arc, in the case of multiple relationships between two identical nodes, we cannot rely on having one fixed-size label for each arc. Instead, each arc label is stored as a *list* of records, each record representing one relationship between two nodes, which allows us to circumvent the single arc constraint. A version of this design which can store lists of fixed-size integers as the arc labels is implemented as the `FixedWidthIntListLabel` class in the WebGraph framework; we extended this class to allow for lists of larger (up to 64 bits) fixed width labels in a new class, `FixedWidthLongListLabel`.

10.4.2 Label format

Each record in the label list described above represents one relationship between two nodes. Because there are $O(|E|)$ labels to store, it is important to find a compact representation

for these labels so as to avoid requiring inordinate amounts of storage space.

For the sake of simplicity, and to avoid keeping multiple label files around, we can find a uniform representation for both snapshot branches and directory entries. Directory entries are constituted of a binary name and an integer for permissions, whereas snapshot branch labels only contain a binary name; we can thus represent all labels as a pair of a binary sequence field and an integer field, which happens to always be an unused value for snapshot branches.

Moreover, we do not actually need to store permissions as arbitrary integers: file permissions are normalized when extracted from the archive and can only take one of five values (plus a special “none” value for the special case of snapshot branches). Normalized permissions can therefore be stored in a 3 bits enumeration:

Value	Bits	Unix permission	Entry type
0	000	—	None (used for snapshot branches)
1	001	0o100644	Regular file
2	010	0o100755	Executable file (+x)
3	011	0o120000	Symbolic link
4	100	0o040000	Directory
5	101	0o160000	Revision (e.g., git submodules)

Regarding file names and branch names, the naive implementation would be to write the full byte sequence representing the entry name in each label. As an order of magnitude, the total size of all binary entry names in the 2021-03-23 graph export is 4.0 TiB, which is impractical to store on disk. However, this can be drastically improved by compressing identical file names together, as (1) the nature of our VCS data model duplicates a lot of relationships between objects which only changed their contents, but not their names, and (2) entry names are massively reused in general (e.g., README.md, refs/heads/master, main.c). While there are 225 billion different directory *entries* in the latest export, there are only 2.3 billion different *entry names*. By only storing each unique entry name once, we can cut down the 4 TiB estimation of the naive approach down to only 62 GiB, which is far more manageable (especially for fast access as it can be stored in a single SSD).

For that purpose, we can once again use a MPH function to associate each label with a unique integer in the range $\{0, \dots, N_L - 1\}$ where N_L is the total number of unique names used in arc labels. Then, we can use a reverse mapping to retrieve a given byte sequence from its associated integer using a `FrontCodedStringList` as detailed in Section 10.3. Here, a new trade-off appears: front-coded string lists are particularly efficient at compressing similar strings when they are *ordered*, i.e., when successive strings in the list share a common prefix, as the front-coding can be used to compress the prefix across different values. However, standard MPH functions map their input to integers in an arbitrary order, which nullifies the advantages of the front-coding. Some MPH functions, called

monotone minimal perfect hashing functions, can be used to map their input in an order-preserving fashion [20], although the resulting functions are generally larger and have a slower hashing throughput. An implementation of such a function is implemented as the `LcpMonotoneMinimalPerfectHashFunction` in the `Sux4J` framework⁵.

We empirically benchmarked the storage performance and hashing speed of three different techniques for label name mapping on the 2020-12-15 graph:

1. **Standard MPH with unsorted front-coded string list:** this approach uses a standard MPH which does not preserve the order of its input keys, negating the compression benefits of front-coded string lists.
2. **Standard MPH with permuted sorted front-coded string list:** this approach uses a standard MPH which does not preserve the order. However, the front-coded string list is stored by lexicographical order, which allows it to benefit from front-coding compression. To map the results of the hash function to the order in the list, a *permutation array* is used, as described in Section 10.2.1.
3. **Monotone MPH with sorted front-coded string list:** this approach uses a monotone MPH which maps its input in an order-preserving fashion (but has a larger size and lower hashing rate), and a sorted front-coded string list.

The results of this benchmark are shown in Table 10.1. As can be seen in the results, the front-coded compression on sorted lists drastically reduces the size of the reverse mapping: 172 GiB for the unsorted list, 108 GiB for the sorted list, for a total compression ratio of 1.59. Among the two options with a sorted front-coded string list, option 2 adds to that amount a 16 GiB permutation array for a total of 124 GiB, while option 2 increases the size of the hash function from 0.6 GiB to 5.7 GiB for a total of 114 GiB. Because the hashing rate is practically identical between the two different hashing functions, and using a monotone MPH makes the label writing pipeline simpler, we pragmatically oriented ourselves towards that latter option.

Armed with efficient encoding mechanisms for the two fields of arc labels (permission enumeration and monotonically hashed entry name identifiers), we can specify the format

Table 10.1: Benchmark results of three different techniques to hash and build a reverse mapping of the set of snapshot branch names and directory entry names.

Option	Implementation	MPH size	Total size	Hashing rate
1	Standard MPH + unsorted FCL	0.6 GiB	172 GiB	521 k lines/s
2	Standard MPH + permuted FCL	0.6 GiB	124 GiB	521 k lines/s
3	Monotone MPH + sorted FCL	5.7 GiB	114 GiB	511 k lines/s

⁵<https://sux.di.unimi.it/>

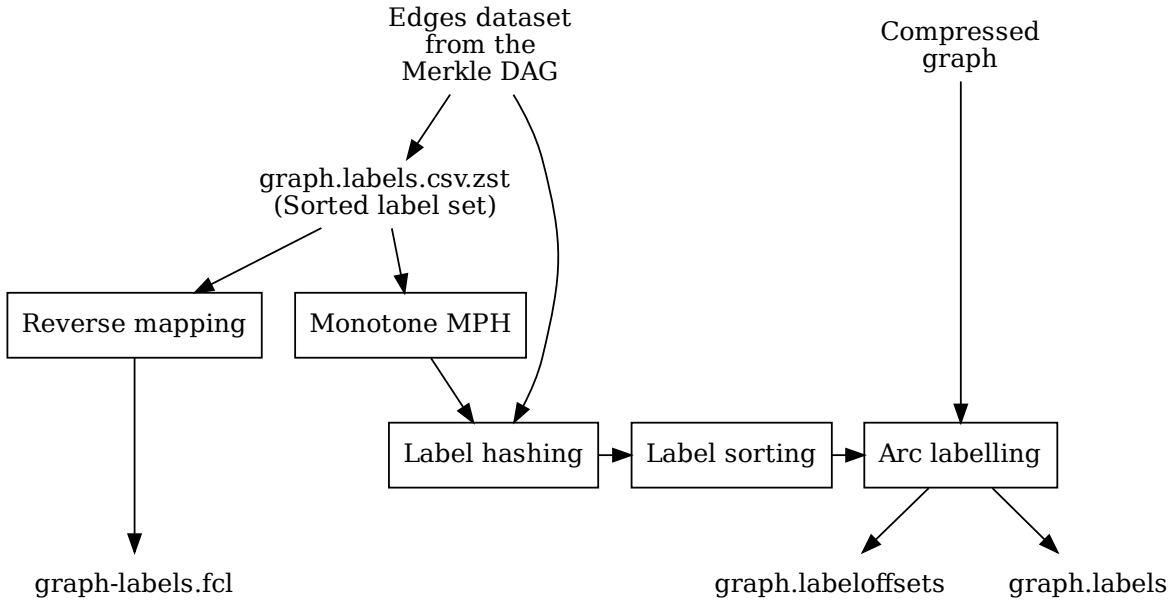


Figure 10.2: Label compression pipeline

for each record: a $3 + \lceil \log_2(N_L) \rceil$ bits long integer (= 35 bits for the 2021-03-23 graph) encoding the permission value in its three lower bits and the index of the entry name in the front-coded string list in its upper bits.

10.4.3 Label compression pipeline.

Having settled the format for arc labels, we can now describe a compression pipeline to generate the on-disk labels from the input dataset. As a reminder, in Section 8.4.3 we described the input edges format for graph compression, a CSV file of edges in the format:

```
<source node> <destination node> [<entry name> [<permissions>]]<CR>
```

Label set extraction The first step of the pipeline is to extract the set of unique label names (directory entry and snapshot branch names) from the edges dataset. We iterate on the entire set of edges and extract the third field (entry name) when it is present, then use this feed as an input to the GNU `sort(1)` command with the `-u` option, to only keep unique entries. We then compress the output using the Zstandard algorithm [39], and obtain a compressed and sorted set of all the unique label names in the graph.

Monotone MPH Using the set of label names, we can generate the hashing function to convert entry names to and from the set of integers $\{0, \dots, N_L - 1\}$ using a Monotone MPH as described in Section 10.4.2. The `LcpMonotoneMinimalPerfectHashFunction` class

takes the sorted set of labels L as an input, and generates an h_L function which converts any given entry name from the set L to its rank in the sorted list of label names.

Reverse mapping In order to access the value of any given label name from its integer rank in $O(1)$ time, we then build an efficient on-disk map using `FrontCodedStringBigList` by giving it the sorted list of labels as an input, which stores them efficiently using front-coding compression while allowing fast random access. This map m_L is the inverse function of h_L over its input domain: $\forall l \in L, m_L(h_L(l)) = l$.

Label hashing As explained in Section 10.4.1, the labels have to be stored in the natural order of edges in the compressed graph. This compressed order is different from the order observed in the edges dataset, which is arbitrary and depends on the export process. We thus need to hash the node SWHID using the `swhid2node` function on both the source and the destination, and the h_L function on the label byte strings in order to reorder them in the final compressed order.

To illustrate this process, given the following edges dataset (in the format described in Section 8.4.3):

```
<src> <dst> <label_name> <perms>
swh:1:snp:4548a5... swh:1:rev:0d6834... cmVmcy9oZWFkcy9tYXNOZXI=
swh:1:dir:05faa1... swh:1:cnt:a35136... dGVzdC5j 33188
swh:1:dir:05faa1... swh:1:dir:d0ff82... dGVzdA== 16384
```

We (1) hash and permute the source and destination nodes using $p_G(\text{mph}(s))$, and (2) monotonically hash the labels using $h_L(l)$, to obtain the identifiers which will be used in the compressed graph:

```
pG(mph(<src>)) pG(mph(<dst>)) hL(<label_name>) <perms>
4421 14773 154
1877 21441 134 33188
1877 14143 141 16384
```

Label sorting Then, we can reorder the list of edges to match the order of edges in the compressed graph. We use `GNU sort(1)` with a large in-memory buffer and temporary space on disk to obtain the sorted edges. Sort is called with the `-n -k1,1 -k2,2` options to numerically sort on each field:

```
1877 14143 141 16384
1877 21441 134 33188
4421 14773 154
```

This sorting step is extremely intensive in time and memory: because there are more than 220 billion edges to sort in the latest dataset and because the sorting algorithm has a complexity of $O(|E| \times \log(|E|))$, the process can take entire weeks, terabytes of disk space and hundreds of gigabytes of RAM.

Since the lines that need to be sorted are fixed-width numbers, other sorting methods can be used to reduce memory usage. The `bsort`⁶ program implements an in-place radix sort, which avoids storing terabytes of temporary buffers on-disk. To use it, we store the result of the hashing step in binary format, one record per line. To avoid spending time sorting on the leading zeros, the source and destination node IDs are written in a fixed size byte array of size $w = \lceil \frac{\log_2(N_L)}{8} \rceil$ bytes. The total record is of size $2 \times w + 8 + 4$, to store respectively the source and destination nodes, the 64-bit label identifier and the 32-bit permission integer. The `bsort` program is then called with the options `-k <w*2> -r <w*2+8+4>` to specify respectively the record size and the number of leading bytes used for sorting.

We benchmarked the label sorting pipeline using `sort` and `bsort` on a smaller graph of around 350 million edges, and obtained similar results (12'30" for `sort`, 11'30" for `bsort`). We have not yet been able to conclusively determine whether one sorting method would be significantly faster than the other on larger input files. Both sorting methods are thus made available, configurable by the user as a command line option.

Arc labeling Finally, with the ordered list of edges in numerical format, we can write the labels to the `graph.labels` file and its offset to the `graph.labeloffsets` file. For this, we need to traverse the graph edges in order, and synchronize the traversed edges with the stream of labelled edges, skipping unlabeled edges in the graph. This process is shown in Algorithm 2 and has a complexity of $O(|E|)$.

10.5 Memory considerations

We export graph properties as a set of files which live next to the compressed graph, one file per property. The specific formats used for these property files have been detailed in the previous sections. In this section we discuss time and memory considerations surrounding access to these properties from the compressed graph.

10.5.1 Direct loading and memory-mapping

As one might expect, in the graph of software development the vast majority of storage space is occupied by the properties themselves; the structure of nodes and arcs is just a small portion of the total size. This highlights an important speed/memory trade-off: it is

⁶<https://github.com/pelotoncycle/bsort>

Algorithm 2 Write the labels from the sorted edge streams to a compressed representation. This algorithm works by synchronizing two traversals: the sorted edges of the graph G and the sorted edges from the numerical stream of labelled edges S_E .

```
function WRITELABELS( $G, S_E$ )  
   $l \leftarrow \{-1, -1, -1, -1\}$  ▷ Placeholder edge line  
  for all  $n \in G$  do ▷ For all the nodes in the graph  
     $w \leftarrow 0$  ▷ Number of bits taken by the labels of the outgoing edges  
    for all  $e \in \text{OUT}(n)$  do ▷ For all outgoing edges from  $n$   
       $A \leftarrow$  empty array ▷ Array of labels for this outgoing edge  
      while  $(l_{src}, l_{dst}) < (e_{src}, e_{dst})$  do ▷ Synchronize traversal and label stream  
        if  $(l_{src}, l_{dst}) = (e_{src}, e_{dst})$  then ▷ Traversed edge is in sync with label stream  
          add  $(l_{entry}, l_{perms})$  to  $A$   
        end if  
        if  $S_E$  at EOF then ▷ Exit case for end of stream  
          break  
        else  
           $l \leftarrow \text{NEXTLINE}(S_E)$   
        end if  
      end while  
       $b \leftarrow \text{SERIALIZELABELARRAY}(A)$   
       $w \leftarrow w + \text{LENGTH}(B)$   
      Append  $b$  to  $\text{graph.labels}$   
    end for  
    Append  $w$  to  $\text{graph.labeloffsets}$   
  end for  
end function
```

highly unlikely that the entire property graph can be realistically stored in memory in the same way as the graph structure itself. Therefore, it makes sense for the properties to be stored offline by default (presumably on disk) and then loaded on demand when the analysis requires it.

There are different “levels” of loading external data on-demand, each with its own memory requirements:

1. Loading the entire mapping from disk to main memory. This immediately increases the main memory usage by the size of the mapping, which guarantees that property access is bounded by the speed of the physical RAM with no disk I/O involved.
2. Mapping the file in virtual memory with `mmap(1)`. This delegates the possibility to cache large sections of the file in main memory to the kernel when enough physical RAM is available; however, when a segment of the file is accessed without having been cached, accessing the mapping will trigger a page fault and require disk I/O, slowing down access times. Here, performance becomes dependent on (a) the amount of available RAM and (b) how well the kernel is able to predict access patterns (to the

extent that they are actually predictable and not random seeks). The main advantage of this method is to avoid reserving large amounts of physical RAM for the mapping.

3. Seeking to a given position in an offline on-disk file. This makes property access fully I/O bound, with each read requiring a system call. The I/O operation need not translate to a physical disk read, as the kernel can still cache some sections of the file in main memory. This method has minimal advantages compared to memory-mapping for large files, so we generally do not provide it as an option for property access.

From the level of the graph compression framework, it makes sense to put the direct loading and memory-mapping methods at the disposal of the end-user, so that they can tune on-demand property loading to their specific workloads and hardware. We thus try to provide, for the mappings that support it, two loading methods: `load` and `loadMapped` to implement options (1) and (2), respectively. Each mapping can be loaded at any point during runtime. It is generally more efficient to directly load smaller mappings which are often used in RAM (e.g., the node type mapping should generally always be loaded directly), whereas larger files like integer arrays should be memory-mapped if the amount of physical RAM is a binding constraint on the hardware.

10.5.2 Sharing mapped data across processes

Often, multiple processes can be working on the same data (mappings or the graph itself), for instance when running different experiments on the same graph. This is problematic in terms of RAM usage when using direct memory loading, as the same data of potentially hundreds of gigabytes is loaded in memory twice. Memory-mapping can be used to avoid storing redundant data in RAM, but comes at the cost of potentially slower I/O as the data is no longer guaranteed to be loaded in main memory and is reliant on kernel heuristics.

We propose another approach to share data across two different compressed graph processes while loading it *directly in RAM*, through the use of the `swk graph cachemount` command. By copying graph data to a `tmpfs` [145] not backed by a disk swap, we can force the kernel to load it entirely in RAM. Subsequent memory-mappings of the files stored in the `tmpfs` will simply map the data stored in RAM to virtual memory pages, and return a pointer to the in-memory structure.

From a user perspective, using the `cachemount` command can be done in just a few steps: (1) choose the graph data and mappings which need to always be loaded in main memory to be shared by the compressed graph processes; (2) call the `cachemount` command with the list of relevant files, which copies them in the `tmpfs` located at `/dev/shm`; (3) memory-map the files from the graph processes using the `loadMapped` method, which returns a virtual mapping to the in-RAM structure.

10.6 Walkthrough

We conclude this chapter by briefly demonstrating how to use graph properties from the Java API of the `swh.graph` framework using the mapping techniques described in the sections above.

The graph itself can be loaded either with memory-mapping or direct loading; it is however recommended to always load it using memory-mapping while using the `cachemount` command described in Section 10.5.2 to share data between several `swh.graph` processes. A third option is to load the graph in “offline mode”, without loading the edges themselves in memory; this disables random node access and only allows the graph to be iterated sequentially.

```
SwhGraph g1 = SwhGraph.load(graphPath); // direct loading
SwhGraph g2 = SwhGraph.loadMapped(graphPath); // memory mapping
SwhGraph g3 = SwhGraph.loadOffline(graphPath); // offline mode
```

The `SwhGraph` object automatically loads the node type maps described in Section 10.3.1 with direct loading, as this mapping is comparatively small. The type of each node can then be looked-up using `getType`:

```
// Printing all node types
NodeIterator it = graph.nodeIterator();
while (it.hasNext()) {
    long k = it.nextLong();
    // returns the node type (among {ORI, SNP, REL, REV, DIR, CNT})
    SwhNode.Type t = graph.getType(k);
    System.out.println(t);
}
```

The node ID \leftrightarrow SWHID mappings described in Section 10.2.1 and Section 10.2.1 are provided through `NodeIdMap`, a wrapper of the MPH function, the `.order` files for the permutation, and the binary file containing the reverse mapping. By default, the MPH function is loaded with direct loading, while the other bigger files are memory-mapped.

```
// Loading SWHID <-> node ID maps
NodeIdMap m = new NodeIdMap(graphPath, graph.numNodes());

// Printing all node SWHIDs
NodeIterator it = graph.nodeIterator();
while (it.hasNext()) {
    long k = it.nextLong();
```

```

    // returns the SWHID of the given node
    SwhNode.Type swhid = m.getSWHID(k);
    System.out.println(swhid);
}

```

Other node properties like integer timestamp arrays can be either loaded directly or using memory-mapping.

```

// Direct loading
long[][] revTimestamps = (long[][][]) BinIO.loadLongsBig(
    graphPath + "-rev_timestamps.bin");

// Memory-mapping (alternatively)
LongBigList revTimestamps = ByteBufferLongBigList.map(
    new FileInputStream(graphPath + "-rev_timestamps.bin").getChannel());

// Printing all commit timestamps
NodeIterator it = graph.nodeIterator();
while (it.hasNext()) {
    long k = it.nextLong();
    if (graph.getType(k) == Node.Type.REV) {
        long timestamp = BigArrays.get(revTimestamps, k)
        System.out.println(graph.getSWHID(k) + ": " + timestamp);
    }
}
}

```

Finally, properties on the graph edges described in Section 10.4 require the use of a graph wrapper class, `ArcLabelledImmutableGraph`, which provides an API to seamlessly read arc labels from the `graph.labels` file while iterating on the graph edges. From there, the on-disk front-coded string array can be loaded and used to retrieve textual label names from their hashed identifier:

```

// Load graph, node mapping and label name reverse mapping
ArcLabelledImmutableGraph graph = BitStreamArcLabelledImmutableGraph.loadMapped(
    graphPath + "-labelled");
NodeIdMap nodeMap = new NodeIdMap(graphPath, graph.numNodes());
FrontCodedStringBigList labelNameMap = (FrontCodedStringBigList) BinIO.loadObject(
    graphPath + "-labels.fcl");

```



```
// Print all edges and their labels
ArcLabelledNodeIterator it = graph.nodeIterator();
while (it.hasNext()) {
    long srcNode = it.nextLong();
    ArcLabelledNodeIterator.LabelledArcIterator s = it.successors();
    long dstNode;
    while ((dstNode = s.nextLong()) >= 0) {
        SwhLabel[] labels = (SwhLabel[]) s.label().get();
        if (labels.length > 0) {
            for (SwhLabel label : labels) {
                System.out.format(
                    "%s %s %s %d\n",
                    nodeMap.getSWHID(srcNode),
                    nodeMap.getSWHID(dstNode),
                    labelNameMap.get(label.filenameId),
                    label.permission);
            }
        }
    }
}
```

Graph Exploitation for Software Mining

In Chapters 9 and 10, we have addressed the main technical challenges in making it possible to run recursive algorithms on the graph of software development history as well as its property data. In this chapter we focus on how to make this data *accessible* to researchers and allow them to exploit it in a practical manner. To do so, we discuss ways to query the graph as a remote service, as a way to run traversal algorithms without having to write low-level code. We also introduce methods to extract consistent subsets of the main dataset, to ease prototyping and enable use cases on a restricted view of the data.

11.1 Querying the compressed graph

Using the graph compression framework presented in Chapter 9, it is possible to analyze the graph of software development by writing traversal algorithms using a low-level API. The main primitive for graph algorithms is a `successors()` function, which returns the successors of any given node as an adjacency list. Using this primitive, it is possible to write more complex algorithms working on the graph, such as BFS and Depth-First Search (DFS) traversals, connected components, centrality analysis, etc.

As an example, the following Java code performs a DFS traversal on the graph starting from a given node and returns all the descendant nodes in its transitive closure:

```
public void visitNodes(long srcNodeId, NodeOutputStream stream) {
    Stack<Long> stack = new Stack<>();

    stack.push(srcNodeId);
    visited.add(srcNodeId);
```

```
while (!stack.isEmpty()) {
    long currentNodeId = stack.pop();
    stream.write(currentNodeId);

    LazyLongIterator it = graph.successors(currentNodeId, edges);
    for (long neighborNodeId; (neighborNodeId = it.nextLong()) != -1;) {
        if (!visited.contains(neighborNodeId)) {
            stack.push(neighborNodeId);
            visited.add(neighborNodeId);
        }
    }
}
```

While this is useful for complex experiments, the complexity of writing custom low-level traversal algorithms is too burdensome for most use-cases. It requires sending compiled Java code to a remote server with the compressed graph to run the algorithm, then manually returning the results. Instead, a better approach is to provide a generic interface to *remotely query* the graph using more high-level primitives.

11.1.1 HTTP API

Expressing arbitrary graph queries requires a query language with a high level of expressiveness. However, the vast majority of queries researchers will tend to perform on the graph of software development can be restricted to a subset of graph traversal queries that can be expressed with a relatively simple API. To design such an interface, we identified the following common use cases for graph queries using the literature review from Chapter 5:

- **Listing directory entries** (“ls”): given a directory node, list (non-recursively) all the linked nodes of type directory and content in the forward graph.
- **Recursively browsing directories** (“ls -R”): given a directory node, recursively list all the linked nodes of type directory and content in the forward graph.
- **Revision log** (“git log”): given a revision node, recursively list all the linked nodes of type revision in the forward graph.
- **Vault**: given any node, recursively list all the linked nodes of any kind in the forward graph. This query is useful to implement more efficient node retrieval for the Vault (see Chapter 6).

- **Revision provenance:** given a content or directory node, return all the revisions, or any one revision, whose root directory (recursively) contains it in the transposed graph.
- **Origin provenance:** given a content, directory, revision or release node, return all the origins, or any one origin, which has at least one snapshot that (recursively) contains it.
- **Content popularity across commits:** given a content, count the number of commits that link to a root directory that (recursively) includes it.
- **Commit popularity across origins:** given a commit, count the number of origins that link to a snapshot that (recursively) includes it.

All of these queries can be expressed as relatively straightforward graph traversals, either on the forward or the transposed graph. Specifically, each of these queries can be expressed as a look-up of either (1) the successors of a node, (2) the transitive closure of a node, or (3) the leaves of the transitive closure of a node, either in the forward or the transposed graph. The traversal also has to be constrained on specific edge types, e.g., to avoid recursively listing directories when looking up a revision log. Finally, the result must either be returned directly as a list of nodes or edges, or aggregated with a counter (for popularity measures).

We translate these needs in a generic graph traversal API, backed by the HTTP protocol.

Terminology The API uses the following notions in its specification:

- **Node:** a node in the Software Heritage graph, represented by a SWHID.
- **Node type:** the 3-letter specifier from the node SWHID (cnt, dir, rel, rev, snp, ori), or * for all node types.
- **Edge type:** a pair `src:dst` where `src` and `dst` are either node types, or * to denote any node type.
- **Edge restriction:** Edge restrictions: a textual specification of which edges can be followed during graph traversal. Either * to denote that all edges can be followed or a comma separated list of edge types to allow following only those edges.

Note that when traversing the backward (i.e., transposed) graph, edge types are reversed as well. As an example, `ori:snp` makes sense when traversing the forward graph, but useless when traversing the backward graph, as no edges match this specification; conversely `snp:ori` is useful when traversing the backward graph, but not in the forward one. For the same reason `dir:dir` allows following edges from

parent directories to subdirectories when traversing the forward graph, and the same restriction allows following edges from sub-directories to parent directories.

Here are some examples of edge restrictions:

- “`dir:dir,dir:cnt`”: only allows traversing edges from directory nodes to directory nodes, or directory nodes to blob nodes.
- “`rev:rev,dir:*`”: only allows traversing edges from revision nodes to revision nodes, or from directory nodes to any other type of node.
- “`*:rel`”: only allows traversing edges with a release as its destination node.

Endpoints Five main endpoints serve as primitives which can be used to express common queries relatively straightforwardly. Each of these endpoints accepts two additional parameters: `?direction=(forward|backward)` determines whether the graph is traversed in the forward or transposed direction, and `?edges=<allowed edges>` sets the allowed edges of the traversal.

- **Neighbors** (GET `/neighbors/<src>`): returns the direct neighbors of a given node in the graph.
- **Leaves** (GET `/leaves/<src>`): returns the leaves of the subgraph rooted at a given source node in the graph. Edge constraints are used to determine whether a node is a leaf (e.g., in a traversal where only `rev:rev` edges are allowed, revisions with no children nodes will be considered leaves even if they do have directory children).
- **Visit nodes** (GET `/visit/nodes/<src>`): returns all the nodes explored during a BFS traversal from a given node.
- **Visit edges** (GET `/visit/edges/<src>`): returns all the edges explored during a BFS traversal from a given node.

Additionally, each of these endpoints is complemented with a “counting” endpoint, where the aggregation is done on the server side so as to avoid transferring large amounts of data that the client will have to aggregate regardless. These endpoints all follow a similar URL pattern, but the `/count` fragment is appended to their path prefix (e.g., GET `/leaves/count/<src>` returns the number of nodes returned by a GET `/leaves/<src>` query).

Implementation Under the hood, these HTTP endpoints are served by a Python service based on the `aiohttp` library¹. This service is able to do Inter-process communication (IPC) with the compressed graph framework (written in Java) using the `Py4J` library².

¹<https://docs.aiohttp.org/>

²<https://www.py4j.org/>

Whenever an endpoint which requires a graph traversal is queried, the service establishes a Unix pipeline between itself and the compressed graph framework. Low-level traversal algorithms then use this pipeline to stream the resulting nodes and edges back to the Python service while the graph is being traversed. The resulting stream of nodes is then forwarded to the user agent using `aiohttp.web.StreamResponse`. This streaming pipeline avoids buffering the entire result in memory before sending it to the user, which reduces memory usage for large responses (e.g., provenance queries for the empty file).

To limit abuse from untrusted clients and avoid Denial-of-Service attacks, a hard limit is set on the number of edges which can be traversed in a single query for unprivileged clients. For instance, the query `/leaves/<x>?direction=backward&edges=*`, where `<x>` is the SWHID of the empty file, would explore millions of edges if left unchecked; with a hard limit, the query will simply be aborted after reaching the limit of traversed edges.

Examples The Graph HTTP API can be used to express the most common use cases for queries on the graph of software development. Below are a few examples of how the endpoints and their parameters (direction and edge restrictions) can be leveraged to do so.

A non-recursive directory listing can be expressed as looking at the neighbors of a directory node in the forward graph:

```
> GET /neighbors/swh:1:dir:b5d2aa0746b70300ebbca82a8132af386cc5986d
swh:1:cnt:89c411b5ce6bb081976d7efb48c2158bb4b2bb86
swh:1:dir:0e0560d2cbbfaf890620b824586b17a82ca076fb
swh:1:cnt:00530420548225a8b26a36f504d9aa00468ddb42
...
```

To recursively list all the objects found in a given directory, the `visit/nodes` endpoint can be used on the forward graph starting from the directory:

```
> GET /visit/nodes/swh:1:dir:b5d2aa0746b70300ebbca82a8132af386cc5986d
swh:1:dir:5a36c7fe6704758ee33627173dae9921ed83d030
swh:1:dir:f6e483c62f06d922db557bdddce512d76f5a0d88
swh:1:cnt:2468b431bb22038cd051d0002983445f907cd364
...
```

The files and directories inside a given directory can be counted by using the `/count` aggregator on the previous query:

```
> GET /visit/nodes/count/swh:1:dir:b5d2[...]986d
66268
```

To get the list of origins where a given object can be found, we can use the `/leaves` endpoint (as we are not interested in intermediate objects), as well as the `direction` parameter to query the transposed graph:

```
> GET /leaves/swh:1:rev:f39d[...]2a35?direction=backward
swh:1:ori:634a2b699d442aa9abd5008f379847816f54ab85
swh:1:ori:571a86b198c6c66ef33025249f7e455b529aae65
swh:1:ori:c15194d6cb59a6d32777ca3b287ea6664d540df3
...
```

Edge restrictions can be used to define the types of edges that the traversals will follow. To get a revision log from a given release, we only want to traverse the `release → revision` and `revision → revision` edges:

```
> GET /visit/nodes/swh:1:rev:c6df[...]fc28?edges=rel:rev,rev:rev
swh:1:rel:c6df0a7ef73ca90825f1472b8a3c5f7a2ce3fc28
swh:1:rev:c8448ff2f9234332f0bc25dc3a13031f8ab3c73c
swh:1:rev:4b63dbd4e782e74bdc050c4579381d29b4bd41c0
...
```

11.1.2 Graph query languages

While the HTTP API is expressive enough to cover most common use cases, it cannot represent more advanced traversals and graph algorithms. Alternatively, it is possible to use the Java API to write those algorithms directly at a low level, but this is a tedious and involved process and requires direct access to the graph server.

Another option to express these complex queries and execute them on the remote graph would be to add support for *graph query languages*, expressive high-level languages which can be used to query property graphs. This has not been implemented and is left open as a future research direction; in this section we look at graph query languages and demonstrate their usefulness in this context.

Graph query languages are the equivalent of SQL for property graphs. They can be used to manipulate graph properties and express complex algorithms based on the structure of the graph and the relationships between its nodes. Over the past two decades, various attempts have been made to specify and standardize graph query languages. One of the earliest and most widely used language is Gremlin [136], used in the Apache TinkerPop graph computing framework. Another well-known language is Cypher [56], used in the Neo4j graph management system. More recently, attempts have been made at standardizing a single unified property graph query language named GQL [102], fusing together Cypher,

PGQL [134] and G-CORE [10]. Finally, SPARQL [121] has imposed itself as the standard for querying Resource Description Framework (RDF) graphs.

To illustrate the usefulness of expressive graph query languages for complex queries, let us consider the following example:

Query 1: *Given two arbitrary revisions, return the shortest path between them in the undirected graph if it exists.*

This query can help us understand how revisions belonging to the same connected components are linked together, and the length of the path which makes them connected. This query cannot be done efficiently using the HTTP API, as it would require computing the entire transitive closure of the two revisions, then computing the intersection of the resulting set. In contrast, a low-level algorithm implementing a bidirectional search would only have to explore nodes up to a depth of the shortest distance between the two nodes.

Using a graph query language, it is possible to properly express this entire query, and leave the possibility of executing it with an efficient bidirectional search to the graph engine. In the Cypher language, this query would be written as:

```
MATCH (a:Revision) WHERE a.swhid = "swh:1:abc123..."
WITH a
MATCH (b:Revision) WHERE b.swhid = "swh:1:def456..."
WITH a, b
MATCH p = shortestPath((a)-[*]-(b))
RETURN nodes(p)
```

The query is declarative and does not specify how the traversal should be executed. We simply define criteria for our two objects of interest, and define the shortest path we expect as a result, using the recursive graph traversal operators `[*]`. As another example, consider the following query:

Query 2: *Given an origin, return all the objects reachable from it, but not reachable from any other origin.*

This query is useful to find objects that are unique to a given origin. One use case is the ability to address legal take-down requests: if a given repository has to be removed from the archive, the only objects that have to be removed from the graph are those that are unique to this particular repository. Attempting to execute this query using the HTTP API is extremely inefficient, as it requires returning the entire transitive closure of the origin in the forward graph, then for each object in the result, finding and counting its origin leaves in the transposed graph. In the Cypher language, this query is simply expressed as:

```
MATCH (repo:Origin) WHERE repo.url = "github.com/..."
WITH repo
MATCH (allother:Origin) WHERE allother.url <> "github.com/..."
```



```
WITH repo, allother
MATCH (repo)-[*]->(x) WHERE NOT (x)<-[*]-(allother)
RETURN x
```

One way to allow the compressed graph to be queried with a graph query language is to wrap the WebGraph framework as an Apache TinkerPop Provider³. By implementing a few primitives which define how to access the nodes and edges of the graph, as well as the node metadata, it becomes possible to leverage the TinkerPop framework to query the compressed graph using the Gremlin language.

Another advantage of graph query languages is that they would provide a uniform interface for various graph processing backends, either using graph compression or more scale-out approaches: Amazon Neptune,⁴ a cloud-based distributed graph database service, can be queried with Gremlin, Cypher and SPARQL. It should be relatively feasible to use the Software Heritage Property Graph Dataset already available on Amazon S3 (see Chapter 8) to provide a drop-in replacement for the compressed graph, for queries more suited for large-scale distributed processing. This is left as future work; having a compressed graph that can be queried even with a simple API was a prerequisite that laid groundwork for a more elaborate future system and is therefore the main focus in this thesis.

11.2 Working with graph subsets

So far, we have worked on compressed graph representations as a way to make running analyses on the entire graph of development history manageable for researchers using commodity hardware. While the compression ratio achieved is impressive, the entire graph is often too computationally expensive to analyze to be practical for many research use cases. This is especially true for prototyping, when the research is still at an exploratory stage and analysis code is quickly iterated upon.

This problem can be addressed by providing less cumbersome *graph subsets*: smaller yet coherent collections of software artifacts which can be used to perform analysis at a more reasonable scale. A tangentially related problem of focusing analysis on a pre-narrowed logical subset of data, i.e., only analyzing repositories matching some specific criteria, can also be tackled by exporting “subdatasets” which only contain the relevant data to process.

This section presents ways to provide subsets of the graph data in a way that can be properly exploited for software mining, by leveraging the compressed graph to select and export relevant software artifacts.

³<https://tinkerpop.apache.org/docs/current/dev/provider/>

⁴<https://aws.amazon.com/neptune/>

11.2.1 Selecting artifacts of interest

To provide an exploitable subdataset of software development data, it is not sufficient to uniformly extract random nodes from the graph. Doing so would not preserve the logical structure linking software artifacts together and would make the graph extremely disconnected, rendering most of the analysis results non-generalizable and of relatively little value. A better approach to generate logically coherent subdatasets is to always export entire *repositories* at once, i.e., the entire transitive closure of a given set of origins. This minimizes the number of dangling links and loose objects, and is generally an intuitive and expected way to construct logical subsets of software mining data, as it closely matches the way the entire dataset was built in the first place.

This reduces the artifact selection problem to selecting the list of *origins* to be included in the subdataset. Often, researchers will have a predetermined set of repositories of interest (see Section 5.1.2), which can be used to compute the transitive closure of relevant artifacts. In other cases, the list of URLs can be obtained externally using specific criteria, such as taking all the repositories from a given hosting place (e.g., GitLab.com), or all those with a minimum number of stars. Finally, if the objective is to get a representative subset of origins present in the archive, this can be achieved with uniform random sampling on the list of origins in the archive.

To choose the appropriate number of origins to include in the subdataset, we need a heuristic to estimate the size of the resulting graph given the number of origins used to compute the transitive closure. Because of deduplication, the size of the resulting graph does not scale linearly with the number of origins: the more origins included in the subgraph, the more likely it is that the nodes in its transitive closure were already present from the closure of another origin.

To attempt to measure this effect, we run an experiment on the entire compressed graph: we (1) shuffle all origins of the graph in a uniform random order, (2) for each origin, lookup its transitive closure (3) once every 10k origins, tally the number of unique objects visited so far. The resulting data is then averaged over five runs, each run having a runtime of around 6 hours. The averaged data is shown plotted in Figure 11.1. We observed a low average standard deviation between each of the five experiments ($\sigma = 0.10\%$ for the node data, $\sigma = 1.08\%$ for the edge data), comforting our idea that this heuristic is a robust way to approximate relative proportions.

As expected, the proportion of objects present in the resulting subdataset is not proportional to the number of origins due to sharing effects. Instead, the number of included objects has a sharp initial growth rate when few origins have been visited, and this growth tapers off as a larger share of the graph has already been visited.

This data can be fitted in a logarithmic model of the form:

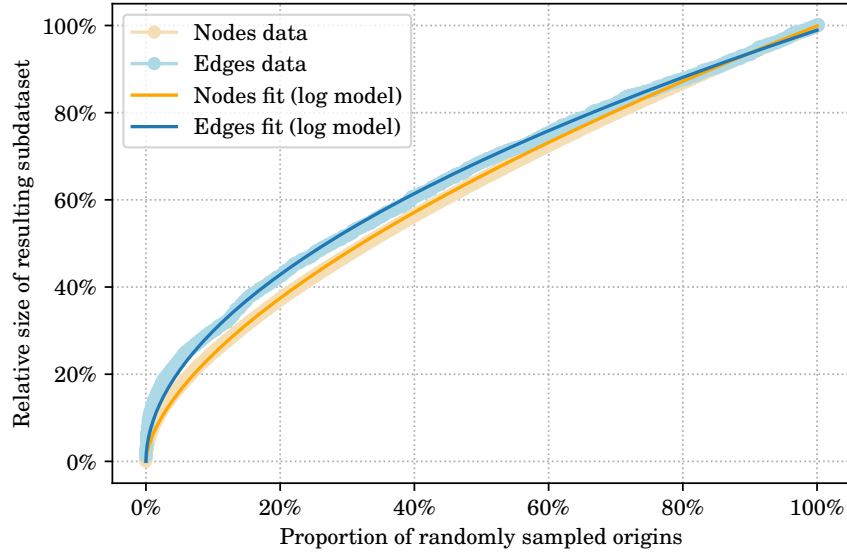


Figure 11.1: Proportion of nodes and edges obtained after sub-sampling the graph with a given number of origins. The x-axis represents the percentage of origins included in the subdataset; the y-axis represents the proportion of objects from the full dataset obtained in the resulting subdataset.

$$S(x) = a \log(1 + bx^c)$$

A curve fitting algorithm minimizing the sum of squared residuals yields the following coefficients for the subsampling heuristic, respectively for the number of nodes and edges in the resulting subdataset:

$$\begin{cases} S_V(x) = 500 \times \log(1 + 0.0020 \times x^{0.61}) \\ S_E(x) = 660 \times \log(1 + 0.0015 \times x^{0.52}) \end{cases}$$

This model can be used to predict the size of the resulting dataset when selecting a given sample size of origins, expressed as a proportion of the total number of origins in the original dataset. As an example, if one wanted to predict the number of nodes and edges in a subdataset containing a sample of 20% of the total number of origins, replacing 20% in the above model yields the following values:

$$\begin{cases} S_V(20\%) = 500 \times \log(1 + 0.0020 \times 0.2^{0.61}) = 37\% \\ S_E(20\%) = 660 \times \log(1 + 0.0015 \times 0.2^{0.52}) = 43\% \end{cases}$$

Therefore, a subdataset generated from a random sample of 20% of the origins will contain 37% of the nodes and 43% of the edges of the entire graph. In order to target a

specific amount of data in the subdataset (e.g., limiting the size of the subdataset to less than 100,000 nodes), the inverse of the model can be used:

$$\begin{cases} S_V^{-1}(x) = \left(\frac{1}{0.0020} e^{\frac{1}{500}x} - 1 \right)^{\frac{1}{0.61}} \\ S_E^{-1}(x) = \left(\frac{1}{0.0015} e^{\frac{1}{660}x} - 1 \right)^{\frac{1}{0.52}} \end{cases}$$

Replacing x in the model above with the target number of nodes or edges will return a sample size of origins which is predicted to generate a subdataset containing the given number of nodes or edges.

11.2.2 Exporting subdatasets

After having selected a subset of origins of interest (either with random sampling or manual selection using various criteria), the next logical step is to materialize the subdataset in a way that is suitable for a subsequent analysis. For this, we first need to compute the transitive closure of the set of all selected origins using the compressed graph. Once the subset of nodes has been narrowed down, it becomes possible to export the subdataset in the various formats presented in Chapter 8.

By reading node and edge properties stored in the graph as described in Chapter 10, it is relatively straightforward to export the subdataset in the input edges format (Section 8.4.3), then recompress it as a new graph using the graph compression pipeline (Chapter 9). The newly exported subgraph is then usable for small-scale experiments.

Afterwards, the list of SWHIDs obtained from the compressed graph traversal can be used to export the subdatasets in columnar format. Scale-out processing tools like Amazon Athena can perform large JOINS to compute the intersection between the full dataset and the set of SWHIDs of the subdataset, and write the result in columnar format compatible with the original graph dataset.

Using these techniques, we exported and made available a few “teaser” subdatasets containing a small set of origins selected using different criteria, which can be used for prototyping with minimal hardware requirements. These subdatasets illustrate different ways the Software Heritage Property Graph Dataset can be used to build exploitable datasets focused on narrowed data that is particularly relevant for some specific research. They have also been useful in the context of the MSR 2020 “Mining Challenge” [125], as they provided an entry point for researchers for whom handling the entire dataset can be quite challenging at first.

popular-4k a subset of 4000 popular repositories from GitHub, GitLab, PyPI and Debian. The selection criteria to pick the software origins was the following:

- The 1000 most popular GitHub projects (by number of stars)
- The 1000 most popular GitLab projects (by number of stars)
- The 1000 most popular PyPI projects (by usage statistics, according to the Top PyPI Packages database⁵),
- The 1000 most popular Debian packages (by “votes” according to the Debian Popularity Contest database⁶)

The resulting dataset is made available in the Apache Parquet and CSV formats, with respective sizes of 23 GiB and 27 GiB, as well as on the Amazon Athena query engine.

popular-python-3k a subset of 3052 popular repositories tagged as being written in the Python language from GitHub, GitLab, PyPI and Debian. The selection criteria to pick the software origins was the following, similar to popular-4k:

- The 1000 most popular GitHub projects written in Python (by number of stars)
- The 131 GitLab projects written in Python which have 2 stars or more
- The 1000 most popular PyPI projects (by usage statistics, according to the Top PyPI Packages database)
- The 1000 most popular Debian packages with the `debtag implemented-in::python` (by “votes” according to the Debian Popularity Contest database)

The resulting dataset is made available in the Apache Parquet and CSV formats, with respective sizes of 4.7 GiB and 5.3 GiB, as well as on the Amazon Athena query engine.

gitlab-100k a subset of 100,000 random repositories from GitLab, hosted at the main `gitlab.com` instance, sampled using a uniform random distribution. This dataset is made available as a compressed graph, containing 304 million nodes and 9.5 billion edges, for a total size of 6.8 GiB (3.6 GiB for the forward graph and 3.2 GiB for the transposed graph).

gitlab-all a subset of every repository from GitLab, hosted at the main `gitlab.com` instance. This dataset is made available as a compressed graph, containing 1.0 billion nodes and 27.9 billion edges, for a total size of 20.6 GiB (9.6 GiB for the forward graph and 11 GiB for the transposed graph).

⁵<https://hugovk.github.io/top-pypi-packages/>

⁶<https://popcon.debian.org/>

11.2.3 Subgraph overlays

While exporting entire subdatasets is extremely useful for a lot of research use cases, as it reduces the size of the datasets so that they only contain the relevant data for these needs, the process is quite slow. Running the full compression pipeline, including all the graph metadata, for even 10% of the graph can take more than a week.

A lot of research use cases involve working on a subset of the data, but not necessarily as a way to reduce the memory usage but simply to filter out irrelevant nodes from the analysis. As an example, computing the connected components of the subgraph containing only revision nodes, while doable by recompressing a revision subgraph, can also be done by simply “masking” non-revision nodes to the algorithm. Providing *views* on subsets of the graph which can mask irrelevant nodes is an effective alternative to re-exporting subdatasets for some types of workflows.

We propose a few different ways to wrap the compressed graph with a subgraph overlay which can be used to mask nodes not present in the subgraph. The overlay always has the same API as the original graph and can be used in its place in any graph analysis algorithm.

Subgraph This subgraph overlay is already present as an experimental library in the WebGraph framework (`ImmutableSubgraph`). It requires one to provide a set of node IDs to include in the subgraph. In order to keep its node IDs in a continuous range, the class remaps all the node IDs between the original graph and the subgraph, as seen in Figure 11.2. Two methods can be used to convert the node IDs back and forth using this mapping: `toSupergraphNode` and `fromSupergraphNode`. These methods are particularly useful to retrieve node properties from the subgraph, as these properties are indexed with the IDs of the supergraph.

LazySubgraph In some use cases, whether a node is included or not in a subgraph can be expressed as a simple function (e.g., “Is the node a revision?” to create a subgraph containing revision nodes only). Instead of writing the set of nodes included in the subgraph, the `LazySubgraph` class determines on the fly during iteration whether a node is part of the subgraph or not. This class does not remap any of the node IDs, and leaves holes for masked nodes, as seen in Figure 11.3. Programs using this subgraph overlay must be aware that node IDs are not contiguous; however, this simplifies access to graph properties by preserving node IDs between the subgraph and the supergraph.

EliasFanoSubgraph Lastly, another option to store the bijection between subgraph and supergraph nodes would be to use a succinct data structure [112] such as an *Elias-Fano* [51] integer list (see Section 9.6.2). This representation is particularly suited for this purpose,

as both sets of nodes are monotone lists of increasing integers. This approach was not implemented yet and is left open as a future research direction.

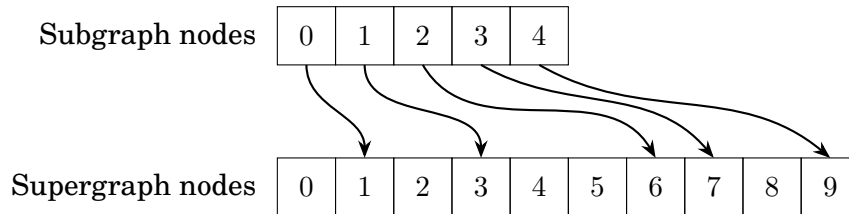


Figure 11.2: The `ImmutableSubgraph` overlay maps node IDs from the set of nodes in the subgraph to nodes in the supergraph.

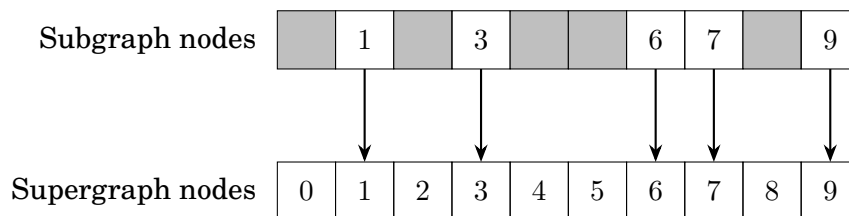


Figure 11.3: The `LazySubgraph` overlay conserves node IDs between the subgraph and the supergraph, leaving gaps in the node list.

PART IV

**THE GRAPH STRUCTURE OF SOFTWARE
DEVELOPMENT**

Topological Properties

In Chapter 4 we detailed how the wealth of software source code artifacts produced by collaborative software development, due to code reuse and the fork-based development model, form the *graph of public software development*: a globally interconnected graph that constitutes a shared software commons of a size comparable to the graph of the Web. Little is known yet about the network structure of this graph, yet such knowledge is needed to determine the best practical approach to efficiently analyze very large subsets of it (if not *all* of it) and to avoid methodological pitfalls (e.g., when extrapolating results from small samples) when doing so.

In this chapter, we determine the most salient network topology properties of the global public software development history as captured by VCSs. We leverage the graph compression framework described in Chapter 9 to analyze the structure of the Software Heritage graph. We explore topology characteristics such as: degree distributions, determining whether they are scale-free or not; distribution of connected component sizes; distribution of shortest path lengths. We characterize these topology aspects for both the entire graph and relevant subgraphs.

This chapter is implementing a preregistered study protocol [123] accepted at the 17th International Conference on Mining Software Repositories (MSR 2020).

12.1 Introduction

In this thesis, we have discussed the possibilities that the Software Heritage offers as a large trove of software artifacts, which makes accessible an approximation of the entire *software commons* as a corpus that can be exploited by researchers, and discussed various ways to

retrieve and analyze the data it contains for software mining purposes. One important property of the software commons is that they are not a collection of distant islands of software artifacts, but rather a single very entangled object. This is because modern software products are built by reusing components from other projects, rather than being completely independent pieces of work. This organic code reuse happens through different means, either by simply copying source code across projects (also known as “software vendoring”), or by explicitly forking [135, 124] an existing project and then building upon its pre-existing development history. Modern VCSs also allow one to explicitly reference external repositories (e.g., via Git submodules or Subversion external) to be fetched as dependencies, which further binds separate software projects together.

As we discussed in Chapter 4, gathering as much publicly available source code artifacts together and consolidating them in a canonical and deduplicated data model is a way to materialize the *global graph of public software development*: an immense interconnected network of artifacts of various kinds (files, directories, commits, repositories), linking together all derivative works, shared code bases and common development histories that form the software commons. As of today, little is known yet about the *network structure* of this entangled graph. This is in contrast with what is known about other large graphs that are obtained as byproducts of technology-related human activities, such as the graph of the Web [101] or social network graphs [161, 109].

To fill this gap, in this chapter we conduct the first systematic exploratory study on the intrinsic structure of the software commons and its most salient topological properties [5], using the Software Heritage Graph Dataset as our best approximation for the graph of public software development.

12.1.1 Motivations and relevance

Understanding the graph structure of public software development is important for a number of reasons.

Determine the most appropriate approach for large-scale analysis. Most “large-scale” studies of VCS generally fall short of the full body of publicly available source code artifacts and either resort to random sampling from existing collections or focus on popular repositories. This is a potential source of bias, but is understandable for practical reasons.

To enable studies on larger samples of the software commons (up to its fullest extent), in addition to suitable archival and compute platforms [50, 3, 92], we need an understanding of its intrinsic structure, to choose the most appropriate large-scale analysis approach depending on the study needs. For instance, if the graph turns out to be easy to partition into disconnected or loosely-connected components, then a *scale-out* approach with several compute nodes each holding graph (quasi-)partitions in memory would be best. Conversely,

if the graph is highly connected then a *scale-up* approach based on large-scale graph database or relying on graph compression [27] would be preferable.

Avoid methodological pitfalls. The same understanding is needed to avoid making overly strong assumptions on what constitutes “typical” VCS data. These pitfalls have been warned against since the early days of GitHub [83], but as of yet they have been neither quantified nor described at the scale we consider in this study.

The extent to which repositories on popular forges correspond to “well-behaved” development repositories, as opposed to being outliers that are not used for software development or are built just to test the limits of hosting platforms or VCS technology, remains unknown. In our experience GitHub alone contains repositories with very weird artifacts: commits with one million parents or artificially built to mimic bitcoin mining in their identifiers, the longest possible paths, bogus timestamps, etc. How many statistically relevant outliers of this kind exist is unknown and needs to be documented as reference knowledge to help researchers in the interpretation of their empirical findings.

Improve our understanding of daily objects of study. More generally, the field of empirical software engineering is collectively studying artifacts that naturally emerge from the human activity of collaborative software development. As it is commonplace in other sciences (and most notably physics), we want to study the intrinsic network properties of the development history of our software commons just because the corpus exists, it is available, and it is challenging to do so. The resulting findings will *also* be practically useful, but in spite of that we will obtain a deeper understanding of the nature of objects we study daily than what is known today.

12.1.2 Research questions and study protocol

In this chapter, we conduct the *first comprehensive exploratory study on the intrinsic structure of source code artifacts stored in publicly available version control systems*. To avoid methodological pitfalls such as publication bias, hypothesizing after the results, and data dredging, we have preregistered a study protocol [123] (phase 1) that the study described in this chapter is implementing (phase 2).¹ We detail below the research questions and other main aspects of the study protocol.

As our main corpus, we analyze the 2020-12-15 version of the Software Heritage Property Graph Dataset, described in Chapter 8, consisting of 9 billion unique source code files and 2 billion unique commits archived from about 150 million projects. We assess the most salient network topology properties [5] of the Software Heritage corpus represented

¹Only minor deviations have happened between phase 1 and phase 2, which we enumerate and discuss in detail in Section 12.6.

as a graph consisting of 19 billion nodes of different types (source code files and directories, commits, releases and repository snapshots) and 221 billion edges connecting them.

On this corpus we will perform an exploratory study, with no predetermined hypotheses, and answer the following research questions:

Research question 12.1. What is the distribution of in-degrees, out-degrees and local clustering of the public VCS history graph? Which laws do they fit? How do such distributions vary across the different graph layers described in Section 4.3.2 — file system layer (source code files and directories) vs. development history layer (commits + releases) vs. software origin layer?

Research question 12.2. What is the distribution of connected component sizes for the public VCS history graph? How does it vary across graph layers?

Research question 12.3. What is the distribution of shortest path lengths from roots to leaves in the recursive layers (file system and development history layers) of the public VCS history graph?

12.2 Related work

There is first an extensive body of work in empirical software engineering and mining software repositories about large-scale analyses of source code artifacts, which we already referenced extensively throughout this thesis (see Chapter 1, Chapter 5, Section 8.1.1 and Section 9.6). While the study described in this chapter is, to our knowledge, the first attempt at characterizing the structure of the public graph of software development, previous work has analyzed the structure of specific layers of this graph or other graphs derived from it. In particular, techniques to track provenance of source code artifacts [137] or identify software “forks” without relying on forge metadata have been proposed using either the history layer [124], or community subgraphs derived from the metadata in the history layer [107].

A recent work by Trujillo et al. [159] highlighted representativeness issues when only using popular platforms like GitHub as a convenience sample, by showing large disparities in the characteristics of projects stored in these centralized platforms and those existing outside them. The representativeness problem outlined in this study is one of the main motivations for characterizing topological structure at the scale of the entire graph.

In addition to this existing literature from software mining, there are comparable studies of the structure of large graphs and complex networks both in computer science and other fields, which give us important tools and methodology considerations that are applicable to the domain of source code artifacts.

The study of complex networks [5, 16] is a specific research field at the intersection of computer science, mathematics, and physics dedicated to the analysis and characterization of large graphs that exhibit non-trivial topological and/or evolutive properties. Large graphs that emerge naturally as byproducts of human activities have been common objects of study in the field for several decades now. One of the best examples and the most studied complex network is the graph of the World Wide Web. The understanding of it we obtained from complex network studies has given valuable practical insights to design crawling engines, searching and ranking methods, compact representation techniques, as well as sociological insights into its growth, social structure, and communities.

Studying the topology of the graph of the Web using large corpuses has been done as early as 2000 in Broder et al. [34] at a large scale: 200 million nodes (Web pages), 1.5 billion edges (hyperlinks between them). In this study we apply a similar analysis approach, including the study of in-degree and out-degree distributions, as well as the distribution of connected component sizes.

Further analyses of the graph of the Web have been pursued more recently in Meusel et al. [101], which extends previous work characterizing different aggregation levels (pages, hosts, and domains). The size of the analyzed corpus is comparable to ours (3.5 billion nodes, 128 billion edges), and their analysis methodology is very close to what we propose here. The graph compression framework that we use to conduct the present study (WebGraph [29, 30]) is also the same. One particularly notable finding is that this related work overturned the validity of prior power law-based characterizations of the various distributions, emphasizing the importance of empirically challenging preconceived notions about the graph properties at a large scale.

Other computer-related graphs that have been widely studied as complex networks are social network graphs, whose nodes correspond to users and whose edges to “friendship” or “follower” relationships. All major commercial social networks have been studied under these optics, including the social graphs of Facebook [161] and Twitter [109].

In the realm of interconnected software-related artifacts, software dependency graphs have also been studied and characterized as complex networks. LaBelle and Wallingford [89] looked at the basic properties of the dependency graphs of Debian and BSD packages, again with a methodology similar to the web graph studies, albeit on a much smaller graph. Abate et al. [2] is a different take on characterizing the graph of software dependencies in Debian, considering semantic rather than merely semantic dependencies. On the same graph, Maillart et al. [95] adds the time dimension to the analysis by studying its dynamic growth and trying to fit it to a Zipf law model. We do not look into dynamic growth of our graph in the present study; it hence remains a relevant research direction for future work.

The designs of specific software systems have also been studied as complex networks, mainly by looking at the relationships between related software modules and/or classes in

object-oriented systems [108, 162, 167]. Collaboration between software developers [144] and software engineering researchers [76] has also been analyzed using similar techniques. In this chapter we do not look at collaboration graphs (which would be graphs *derivable* from, rather than natively represented in, our data model) and neither dig archived software projects to establish their dependencies. Both remain interesting areas for further exploration. This study provides a fundamental stepping stone for those further analyses, providing a complex network characterization of the largest available approximation of the global VCS graph of public code.

12.3 Methodology

In this section we present our methodology for analyzing the global graph of public software development as captured by the Software Heritage graph dataset (see Chapter 8).

12.3.1 Dataset and graph processing

We use the latest export of the edge dataset that was available at the time we started our experiments, dated 2020-12-15. In terms of size, this export contains 19 billion nodes and 221 billion edges in total.

Some of the analyses we require could easily be performed in a scale-out / distributed fashion (e.g., degree measurements, for RQ 12.1), others benefit more from a scale-up / centralized approach (e.g., connected components for RQ 12.2 and path lengths for RQ 12.3) to avoid incurring the price of costly synchronization points. For uniformity, and also because there is no real drawback in doing *all* analyses with a scale-up approach once we have to do at least some of them in that fashion, we adopted a centralized approach for all analyses. In particular, using the work described in Chapter 9, we leverage the `swh-graph` pipeline for manipulating and analyzing the Software Heritage graph dataset in a compressed format.

To run the analyses on the different layers we described in Section 4.3.2 (hosting layer, history layer, commit layer and filesystem layer), we also need to export “subgraphs” containing only the relevant parts of the graph that belong to each layer. We use the `LazySubgraph` overlay described in Section 11.2, as it offers reasonable performance and flexibility and reduces the complexity of the pipeline. In practice, this means that only the main graph is stored in memory, and the “layers” are implemented as virtual wrappers which reimplement the graph access primitives by checking whether the node types belong to the current layer in use.

12.3.2 Topology analysis

In this section we discuss algorithmic approaches and methodological considerations to produce the raw data, as well as the analysis protocol followed to process it.

12.3.2.1 Graph directionality

The graph of software development is a DAG. While its directionality has an important semantic meaning (e.g., parentality relationships between revisions or directories do not commute), taking it into account is not appropriate for all the network analysis metrics. For instance, all strongly connected components in a DAG have a size of one.

As such, it is sometimes necessary to view the graph as being undirected by “symmetrizing” it (i.e., computing the union of the directed graph and its transposition). Conceptually, doing so corresponds to studying how the nodes are linked together by the underlying relationships, rather than the relationships themselves. Symmetrization can always be performed lazily and in constant memory using the directed forward graph and its directed transposition. Each primitive is computed on both graphs, then the results are added together.

12.3.2.2 Degree distributions

We measure the in-degree and out-degree distributions of each layer of the graph. That is, for each node, the number of edges pointing to it (in-degree) and the number of edges starting from it (out-degree). This requires a single pass on the graph in $O(|V| + |E|)$, as well as maintaining one frequency mapping of negligible size (a few megabytes) per layer.

A single pass is sufficient to compute the distributions of all the layers. As we iterate on the edges adjacent to each node, we check the type of their source and destination, and only increment the frequency counter of a specific layer if these types belong to the layer.

12.3.2.3 Clustering statistics

To get an overall sense of how the nodes in the different layers tend to cluster together in high-density groups, we estimate the global undirected clustering coefficient [166] of the entire graph. As getting an exact value of the clustering coefficient has a complexity of $O(|V|^3)$, which is impractical at this scale, we use approximation heuristics described by Schank and Wagner [140] based on uniform random sampling.

At a finer-grained level, we also represent the local undirected clustering distribution: for each node, the number of edges between nodes pointing to or from it. This is equivalent to the local clustering coefficient without dividing it by the number of possible triangles in the undirected graph. For the same reasons, this distribution has to be estimated from a uniform sample of nodes for each layer.

12.3.2.4 Connected Components

The size distribution of connected components is a crucial metric to understand whether the graph can be divided in reasonably-sized clusters as a way to perform large-scale analysis. Using a breadth-first traversal of each layer, we compute the sizes of all the connected components in the undirected graph. The traversal has a linear complexity of $O(|E|)$ in time and $O(|V|)$ in memory, which allows us to get an exact result for the entire graph. The frontier is stored as an on-disk queue to lower the RAM usage.

12.3.2.5 Path length distribution

We compute the distribution of the length of the shortest paths between all the root (in-degree = 0) and all the leaf (out-degree = 0) nodes in the filesystem and the commit layers. This is a common measure of network topology, and gives an idea of the difficulty of finding the *provenance* of an artifact. Computing this metric requires one breadth-first search per root node, making its complexity superlinear in time, which would require computing this metric on a sample. However, as the degree distribution will show in section Section 12.4.1, commit chains are generally long and degenerate, allowing us to run an exact algorithm on the entire commit layer.

12.4 Topological properties

12.4.1 In-degrees and out-degrees

The first indicator that can be used to understand the density of each layer in the graph is the *average degree*, which is the ratio between the number of arcs and the number of nodes in a graph. In the entire Software Heritage graph, there is an average of 11.0 edges per node, which means it is relatively sparse compared to other large graphs studied in the literature: around 5 times more sparse than the page graph of the web, and 15 times more sparse than the social graph of Facebook.

Table 12.1 shows a breakdown of the average degree of each layer of the SWH graph, as well as a comparison with various other networks generated by human activities, which confirms the graph density lies in the lower range of other real-world networks.

Because most commits generally only have one parent, with the exception of occasional merge commits which can have more, it is not surprising that the commit layer has a density slightly above, but very close to 1. The commit chains conform to the general structure of *degenerate trees*. The history layer adds a relatively low number of releases to the commit layer, and each release always adds one node and one edge. The hosting layer still has a low density, although around three times denser than the commit layer, because

Table 12.1: Average degree of various large graphs.

Dataset	Average degree
swh-2020-history	1.021
swh-2020-commit	1.022
swh-2020-hosting	3.39
bitcoin-2013 [94]	6.4
dblp-2011	6.8
swh-2020	11.0
swh-2020-filesystem	12.1
twitter-2010 [88]	35.2
clueweb12	43.1
uk-2014 (Web) [25]	60.4
fb-2011 (Facebook) [15]	169.0

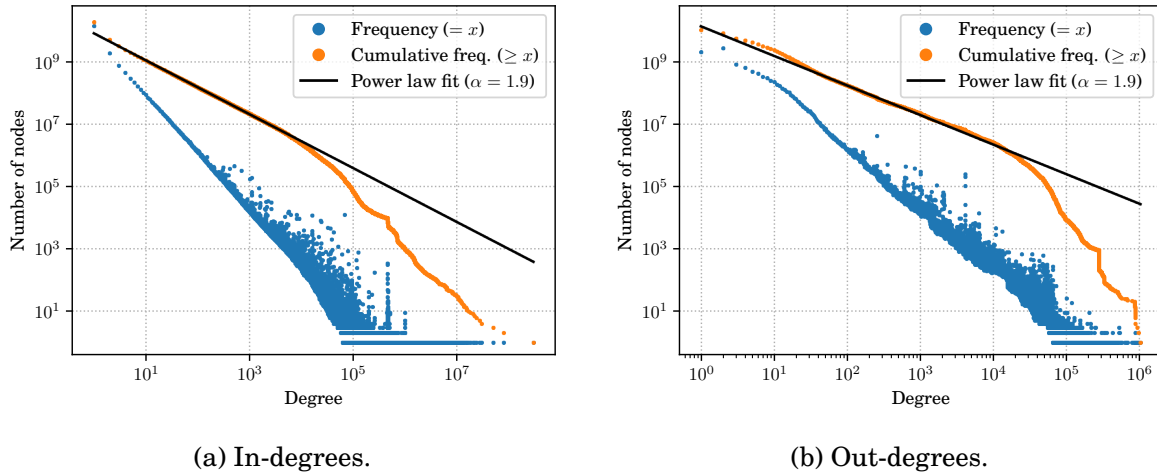


Figure 12.1: Degree distributions: Full graph.

its contents are arranged as a *bipartite graph*: each vertex connects one origin and one snapshot, which limits the density by construction.

The filesystem layer has a higher density, with an average degree of 12.1. Being the dominant component of the graph, it drives the average of the entire graph up to 11.0. The filesystem layer can be seen as a forest of directory trees, with a deduplication of identical subtrees. When a new commit changes a file at a depth h in the tree, a new tree will be created with h new nodes, corresponding to the path from the root of the original tree to the modified file. All the other nodes of the tree are shared with the previous tree thanks to deduplication, which increases the density of the graph by adding new edges to the shared nodes without creating new nodes. As a general rule, the more deduplication there is in the filesystem layer, the higher we can expect its density to be. While the filesystem layer is the densest part of the graph, because it is still a directed acyclic graph, it is by nature more sparse than cyclic or undirected graphs (e.g., social networks).

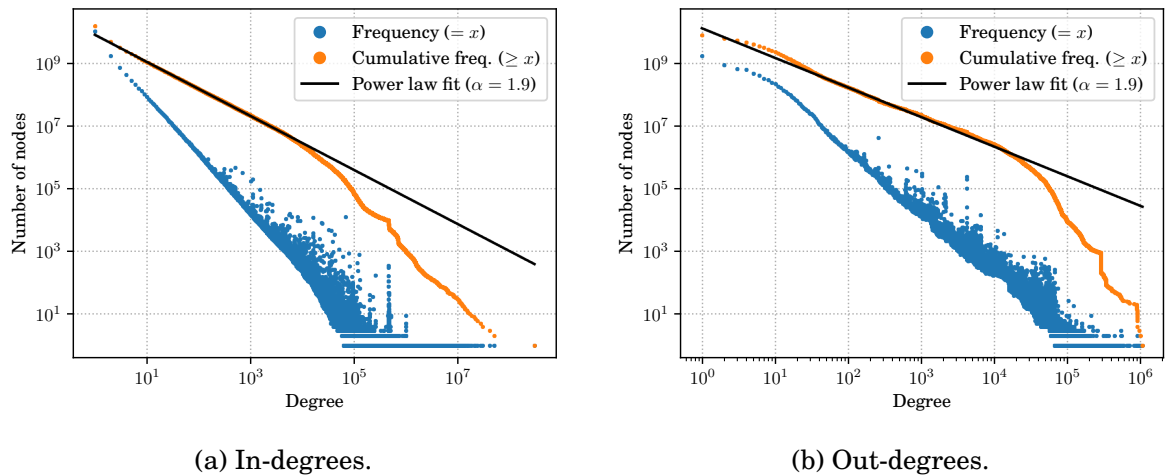


Figure 12.2: Degree distributions: Filesystem layer.

Figure 12.1 shows the frequency and cumulative frequency plots of in-degrees and out-degrees of the entire graph in log-log scale. These plots respectively show, for each degree d on the x-axis, the number of nodes that have a degree of exactly d , and a degree of d or more.

Figure 12.2 shows the frequency distributions for the filesystem layer. Its similarity with Figure 12.1 underscores that the filesystem layer largely dominates the frequency distribution of the entire graph. The out-degree distribution gives an idea of the number of objects present in the archived directories. Most directories seem to contain less than ten entries, with a threshold effect at around ten files after which the frequency drops at a faster pace. The points at the end of the tail are gigantic directories containing millions of entries. These are mostly binary files generated by scripts, or the result of user errors (e.g., the largest directory in the archive contains millions of generated shaders for an amateur video game, in a now deleted branch). The in-degree distribution represents the number of directories referencing some specific directory of content, and gives some insight on the deduplication process of directories and contents (as the in-degrees would always be one if no deduplication was happening). It appears to be very smooth, suggesting some strong underlying regularity to the deduplication process. The outliers at the end of the distribution are special objects that are omnipresent in software repositories: the empty file and the empty directory. Other remarkable outliers are directories containing an empty `.keep` or `.gitkeep` file, which is a common workaround to the fact that Git does not handle empty directories. It is also interesting to look at “bumps” in the distribution that break the regularity, as they are generally not isolated anomalies but consistent deviations centered around a range of values. In this in-degree distribution, there is a bump at around $d = 92,000$ which can be explained by the CocoaPods/Specs repository, a package manager that uses GitHub as a Content Delivery Network (CDN). This repository has more than

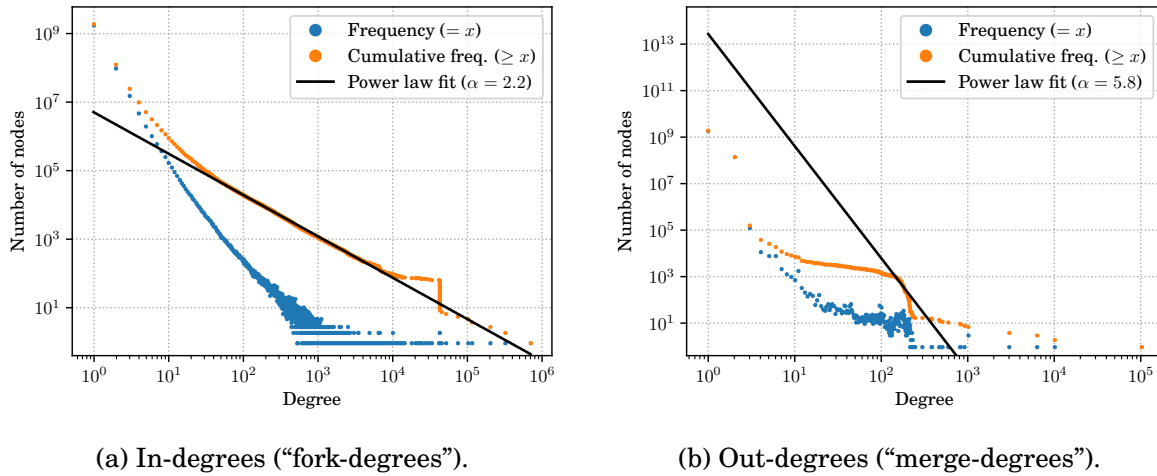


Figure 12.3: Degree distributions: Commit layer

368 k commits, each of them editing a single file in a giant directory containing all the packages of the distribution.

The degree distributions for the commit layer are shown in Figure 12.3. As the parent/children terminology can be confusing when dealing with commits (since *parent commits* are *children nodes* in the DAG), we generally refer to the in-degree of commits as the “fork-degree”, that is the number of commits that were based on a specific commit, and to the out-degree as the “merge-degree”, the number of commits that were merged in a specific commit. The fork-degree distribution is very smooth with no notable threshold effect. This can be explained by common development patterns: forks are generally feature branches that are based on the latest revision in the main development branch, which is generally random, so there is little reason to expect large deviations from the naturally resulting power law.

The merge-degree distribution has a large gap after $d = 2$. The vast majority of commits only have one parent, but occasionally two branches are merged back together, which creates a merge commit with two parents. These are the two most common cases, separated by one order of magnitude ($\approx 10^9$ simple commits, $\approx 10^8$ merge commits). Commits with more than one parent are called “octopus merges”, and are exceedingly rare occurrences, generally not a part of standard development workflows, which explains the gap of two orders of magnitude between $d = 2$ and $d = 3$. We expect most of these octopus merges with large degrees to be generated by scripts in very peculiar environments, so the irregularities observed in the tail of the distribution are not particularly surprising. As the shape of this distribution is dominated by outliers and irregularities, the power law fit likely cannot be interpreted.

The outliers in these distributions are usually experiments aiming to generate commits with inordinate amounts of parents. The largest degree in the out-degree distributions

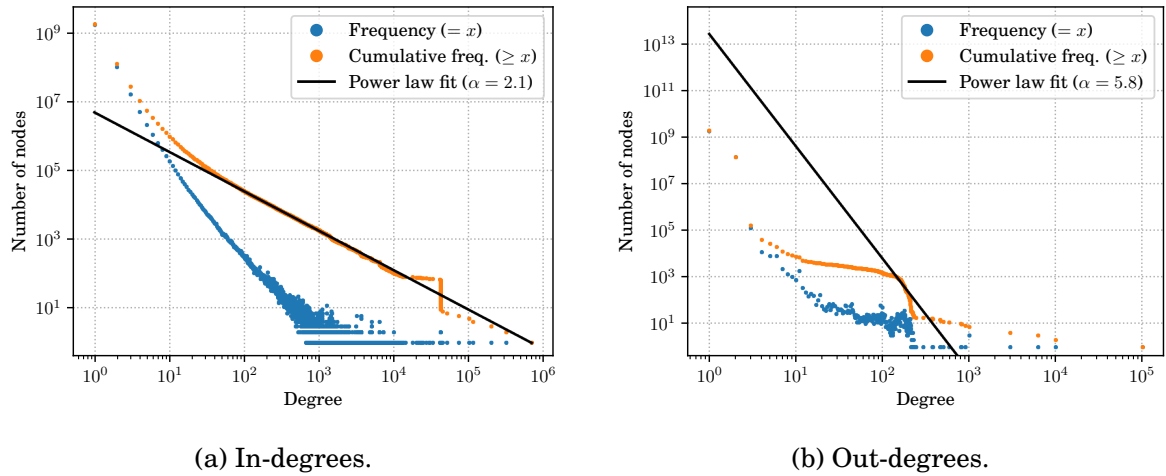


Figure 12.4: Degree distributions: History layer.

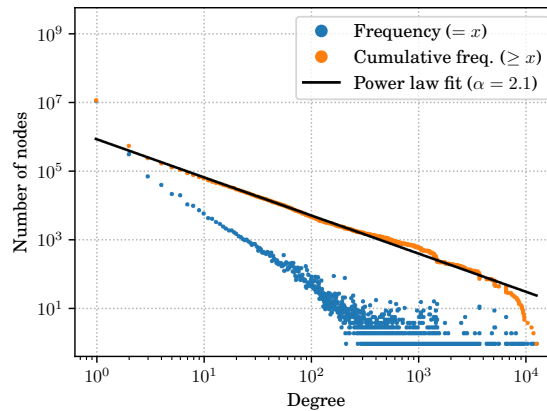


Figure 12.5: In-degrees of commits from releases.

is at $d = 10^6$, and comes from a GitHub repository called `test-commit-many-parents-1m`, which contains two commits linked together by one million of edges. It contains the most forked commit in the archive.

The history layer's degree distribution, shown in Figure 12.4, is extremely similar to the one of the commit layer as it is largely dominated by the commits it contains. However, it is still interesting to look at the in-degree distribution of the commit layer from the releases, i.e., the distribution of the number of releases that point to a given commit. It is shown in Figure 12.5. There is a noticeable threshold effect between $d = 1$ and the rest of the distribution, again attributable to development practices. Releases, or named tags, are generally used to denote specific versions of a software. It makes little sense to have two different versions pointing at a single commit, since there would be no code changes to justify the version increment. Occasionally releases can be used to annotate some specific milestones in a project in addition to its current version, so commits pointed by more than

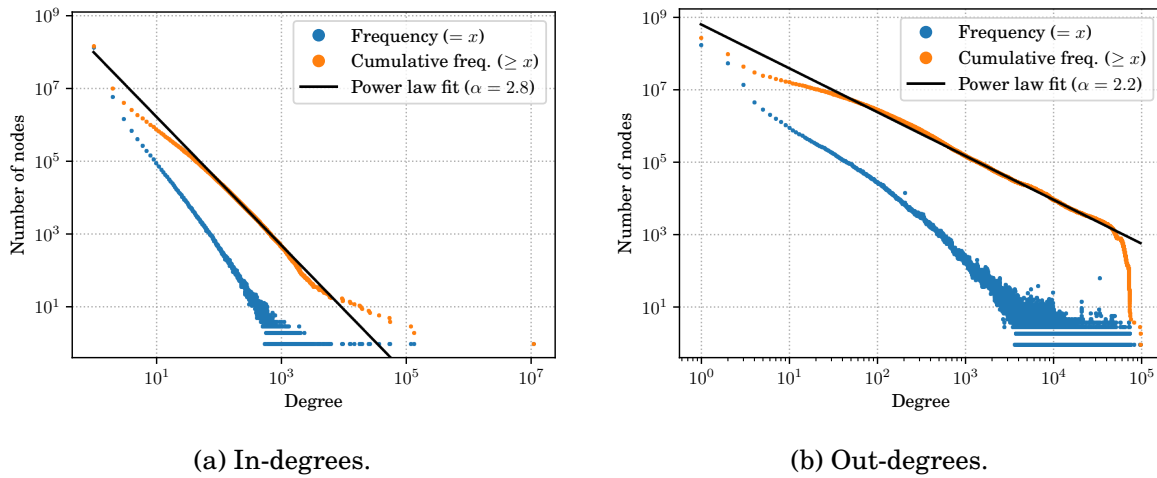


Figure 12.6: Degree distributions: Hosting layer.

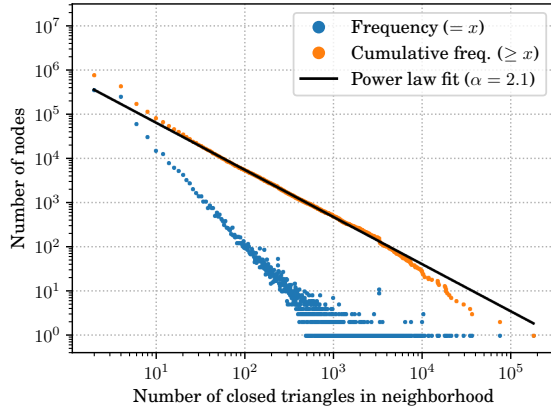
two releases do have some significance, although their importance diminishes rapidly in the distribution.

The distribution of the hosting layer is shown in Figure 12.6. Since within the history layer of the graph, an origin cannot have ancestors and a snapshot cannot have descendants, the two distributions show very different things. The out-degree distribution describes the number of snapshots associated to each origin. This is not an intrinsic property of development workflows because it is highly dependent on the crawling process of Software Heritage: if a repository changes constantly but is only visited once every month, the distribution will not capture how frequently the repository is updated, but rather how often the crawler visits it. On the other hand, the in-degree distribution describes the number of origins associated to each snapshot, that is, the number of “exact forks” of a given repository. This happens anytime someone makes an exact copy of a repository, for instance by clicking on the “fork” button in GitHub, without then updating it with new commits or branches: a new origin is created, but it points to the same repository state as the first origin.

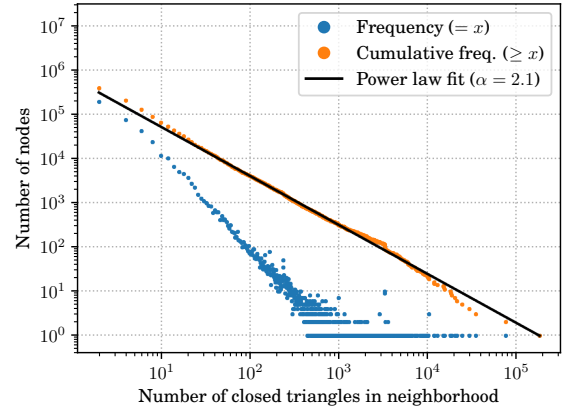
12.4.2 Local undirected clustering distribution

We compute the local clustering distribution on the different layers of the undirected graph, which describes the extent to which the nodes cluster together in tight cliques. In a DAG this coefficient is always 0 since a closed triangle corresponds to a cycle. However, the distribution of the number of closed triangles in the undirected version of the graph, shown in Figure 12.7(a) can be interpreted meaningfully.

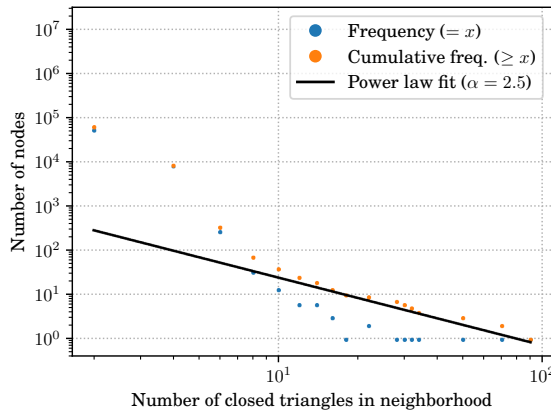
Again, the distribution for the full graph is completely dominated by the filesystem layer and cannot be interpreted on its own; each layer has to be looked at individually.



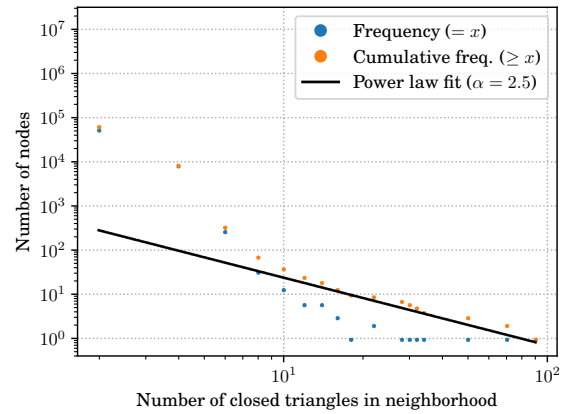
(a) Full graph.



(b) Filesystem layer.



(c) Commit layer.



(d) History layer.

Figure 12.7: Clustering distributions (0.1% uniform sample).

In the filesystem layer, a closed triangle corresponds to a file or a directory being both in a directory D and in a subdirectory D' of D . A common instance of this happening is when developers copy the contents of a directory in a “backup” or “old” directory to take a snapshot of a previous version of the directory. Figure 12.7(b) shows the frequency of these triangles forming in the filesystem layer, with an apparent scale-invariant regularity.

In the commit layer, the closed triangles correspond to a merge commit C with two parents A and B , with B being a parent of A . This often happens when merging multiple commits using the “no fast-forward” strategy (`--no-ff` in Git). Here, the distribution (Figure 12.7(c)) displays a similar pattern to what we observed in the out-degree distribution of the commit layer (Figure 12.3(b)): the two common cases are having one or two closed triangles, while having more triangles requires an octopus merge and is relatively rare in most development workflows, which explains the important threshold effect for $n > 2$.

Being a bipartite graph, the undirected hosting layer cannot contain closed triangles and its local clustering distribution is therefore not represented here.

Table 12.2: Connected components per layer.

Layer	# of WCC	Size of largest WCC	% of nodes in largest
Full graph	33,104,255	18,902,683,142	97.79%
Filesystem layer	46,286,502	16,565,521,611	97.16%
Commit layer	88,031,649	51,543,944	2.61%
History layer	88,040,059	52,176,239	2.62%
Hosting layer	108,342,722	13,841,855	4.82%

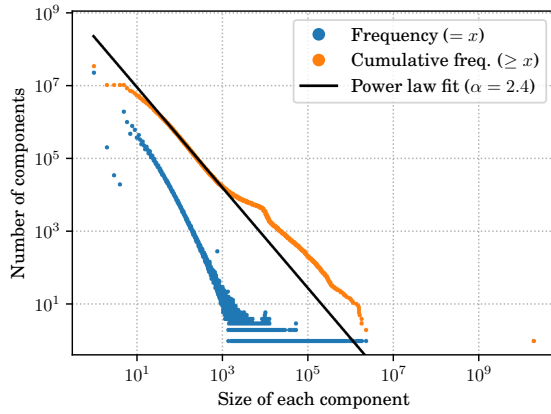
12.4.3 Connected components size distribution

In this experiment, we symmetrize the graph to treat it as an undirected graph and use a breadth-first traversal to compute the sizes of its weakly connected components (WCC). Table 12.2 shows a breakdown of the number of components and sizes of the largest components for each layer. In the entire graph, we find a giant component of 18.9 billion nodes, in which a whole 97.8% of the nodes in the graph are reachable from one another by following undirected edges. The size distribution for the full graph shown in Figure 12.8(a) clearly indicates the extent to which the largest connected component is an outlier that dominates the entire distribution, being 8345 times larger than the second largest.

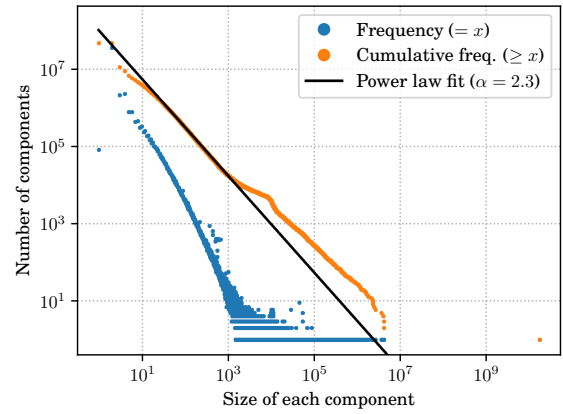
One could wonder whether this high connectivity results from a few highly-connected nodes, like the empty file which is present in millions of repositories. Surprisingly, this turns out not to be the case: repeating the same WCC experiment after removing the top 1 million of nodes with the largest in-degrees from the graph still yields a giant component of about the same order of magnitude (only about 5% smaller). This implies that the connectivity of the graph is resilient and does not depend on the existence of a few high-degree nodes, and that those highly reused software artifacts exist in a graph that is already well-connected without them. This finding is similar to what has been shown for the graph of the Web, where removing pages which are central hubs and have a high PageRank does not significantly reduce the connectivity of the graph [34].

These observations apply similarly to the filesystem layer which again dominates the distribution of the full graph. However, the distributions of the commit and history layers shown in Figure 12.8(c) and Figure 12.8(d) exhibit a very different graph connectivity. The largest component encompasses less than 3% of the graph, which indicates that the history layer can be separated in reasonably sized units. Furthermore, an in-depth investigation of this large component reveals that most of the commits it contains belong to various forks of the Linux kernel, which is suspected to be the largest open source software by number of commits across all its different forks. We can infer from this observation that the connected components of the history layer delineate structures of “fork networks” in the graph, by clustering together projects that have a shared development history.

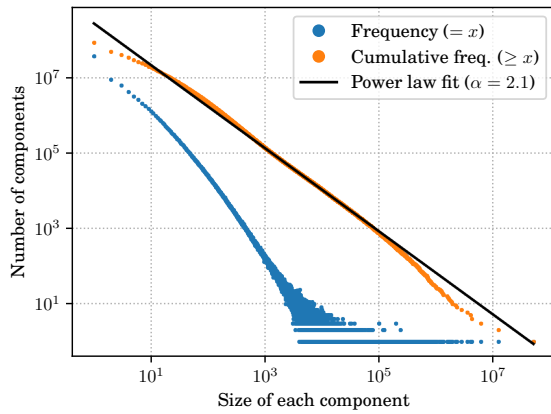
The largest component in the hosting layer contains a relatively small share of the layer’s nodes (around 3%), however it is still a relative outlier containing 2051 times



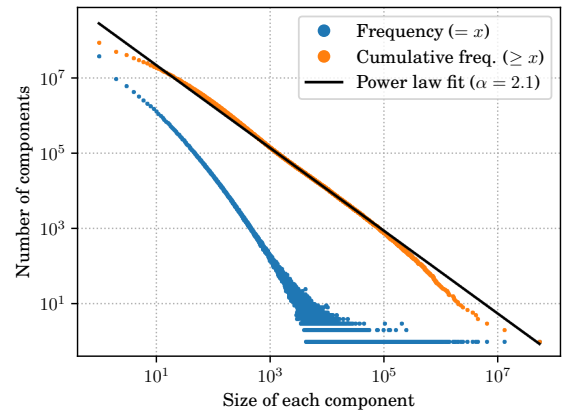
(a) Full graph.



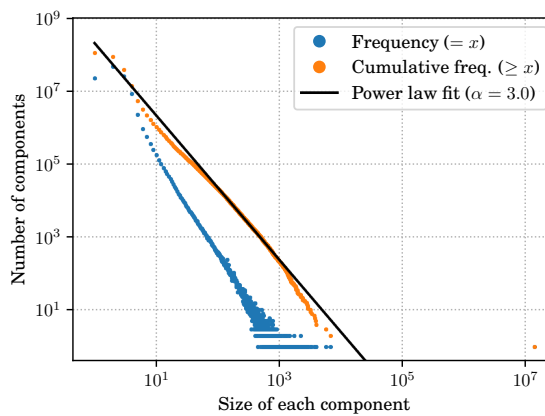
(b) Filesystem layer.



(c) Commit layer.



(d) History layer.



(e) Hosting layer.

Figure 12.8: Connected components distributions.

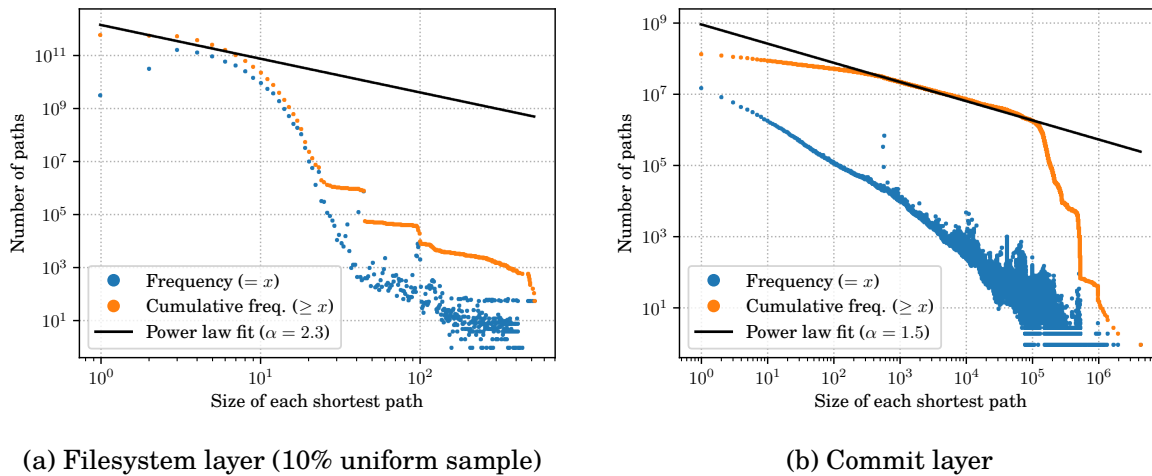


Figure 12.9: Shortest path length distributions

more nodes than the second-largest component. The larger components generally contain repositories that are forked a lot, for example the GitHub repository `jtleeek/datasharing` which has been forked more than 230 thousand times. Even so, these outlier repositories do not explain the existence of a component with more than seven million nodes, which is an order of magnitude higher than the most forked repository. By walking random paths in this component, it is possible to see some patterns that could explain the size of this component. One appears to be that beginners sometimes fork well-known repositories then rewrite their history to replace them with a completely different content. Every time this happens, it links together completely unrelated networks of forks, which all aggregate in this large component.

12.4.4 Shortest path lengths

The last topological property we look at is the average length of the shortest paths between root and leaf nodes in the filesystem (Figure 12.9(a)) and commit layer (Figure 12.9(b)), as defined in Section 12.3.2.5. These have directly transposable meanings in software development. In the filesystem layer, they correspond to the minimum directory depth at which a given blob can be found on average. In the commit layer, they are the lengths of the commit chains from the first commit of the project to the heads of the branches. These are particularly interesting alongside the degree distributions, as they help us understand the shape of the graph given its density.

We saw in Section 12.4.1 that the filesystem layer was dense, with nodes in close proximity with each other. The distribution obtained in Figure 12.9(a) gives an idea of the depth of the files in the directory trees, which interestingly appear in a very characteristic configuration. The distribution does not exhibit scale-invariant behavior, suggesting that

its mean does not diverge. Typically, most files seem to be at a depth of less than around 15, with the frequency of deeper files dropping sharply after that threshold. The most common depth is 3, and files less than 4 directory deep are less common than files at a depth between 4 and 8. This makes some intuitive sense, as the source files which are modified pretty often are generally organized inside directory hierarchies, and rarely at the top-level.

In contrast, Section 12.4.1 showed that the history layer was sparse with an average degree close to one, indicating that it was mainly consisting of degenerate strings of commits. Figure 12.9(b) shows us the distribution of the lengths of these strings, which appears to have some scale invariance. Because these lengths can be interpreted as the “age” of a given project measured in number of commits, it makes sense that they would follow a distribution with a high variance. A few outliers are also present in the tail of the distribution, mostly test projects like the GitHub repository `cirosantilli/test-many-commits-1m` which contains two million commits.

12.5 Discussion

Our results shed light on some properties of the software development graph which have important implications for empirical research and large-scale analysis.

The first salient characteristic is the large topological disparity between the different layers that constitute the graph, both at the local level and in their global structure. If we break down the graph in three layers, we see that they have dramatically different shapes, densities and connectivities.

The filesystem layer contains 90% of the nodes and 97% of the edges of the full graph, and thus largely dominates its high-level topological properties. This layer is dense and highly connected, due to the high amounts of deduplication of the software artifacts it contains. This high connectivity naturally leads to the existence of a giant connected component, containing more than 97% of all the files and directories in the graph that are all reachable from each other by simply following directory hierarchy vertices. The degree distributions in the layer have a heavy tail, with a very high frequency of nodes exceeding the average degree. The directory trees have a characteristic depth with a converging average, with only a few outliers with a hierarchy depth larger than around 20 nested directories.

In contrast, the history layer has virtually the exact opposite topological properties. It is sparse and mildly connected, mainly consisting of almost degenerate chains of commits, with relatively low deduplication compared to the filesystem layer. Its largest component is less than 3% the size of the entire graph, which implies that the nodes are well separated within the layer. Commits have a characteristic out-degree, with very few outliers that

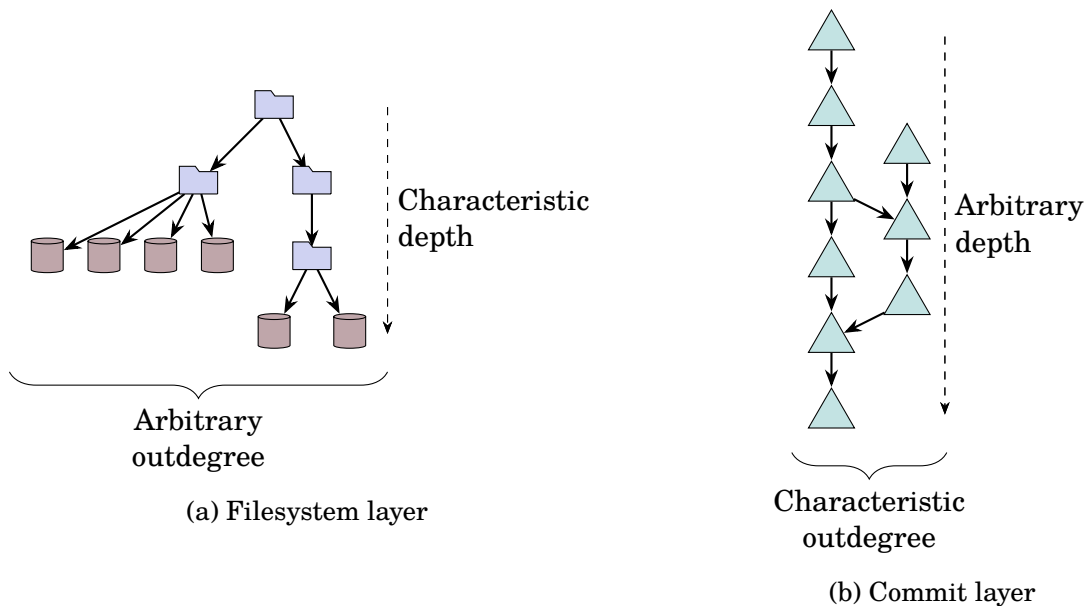


Figure 12.10: The filesystem and commit layers have drastically different topological properties: the filesystem layer has a characteristic depth (generally under 10 directories) but no characteristic out-degree, whereas the commit layer has a characteristic out-degree (generally two parent commits or less) but no characteristic depth.

merge more than 3 commits together. However, the commit chains do not have a bounded height, and the distribution of the shortest path lengths between the root commits and the branch heads has a heavy tail, with a high frequency of chains longer than the average height. This discrepancy between the topological properties of the two layers is illustrated in Figure 12.10.

Finally, the hosting layer is a bipartite graph containing a small fraction of the nodes in the graph. It is also sparse and minimally connected, with some deduplication for identical forks, which are a frequent occurrence in modern hosting platforms.

An important practical implication of these findings is that because the filesystem vertices largely aggregate in a giant component, it is not possible to apply strict component separation to partition the entire graph in smaller tightly connected clusters in order to perform scale-out computations. The filesystem layer should be seen as a dense network of highly connected software artifacts, and there is no obvious way to remove high-connectivity nodes to break it down into multiple disconnected subgroups. However, because the hosting and history layers are sparse, have low connectivity and smaller giant components, it is possible to easily separate them into multiple partitions that can be processed in parallel while retaining the performance advantages of exploiting node locality within them.

Another key point relevant for empirical software engineering studies is that the distributions studied in this article often have heavy tails and a generally high kurtosis (i.e., a high propensity to produce outliers). This implies that there is often no obvious rule to

filter out outliers after a given threshold, as they are an integral part of the distributions' nature. As such, empirical studies should be cautious to systematically justify how they filter outliers in their methodology, and consider sampling biases as potential threats to validity.

12.6 Threats to validity

12.6.1 Internal validity

This study closely follows the registered protocol described in Pietri et al. [123], and extracts the quantitative data required to answer its three research questions from the Software Heritage graph dataset. This data is extracted and analyzed using the algorithms and statistical tools described in the protocol, without any particular changes to the methodology. However, while writing the protocol, we underestimated the execution time of two algorithms, which we were not able to run on the entire graph.

According to our estimates, the path length distribution of the filesystem layer would have taken around two months to compute on an expensive machine, thus we restricted ourselves to a subsample of 10% of the nodes in the filesystem layer. However, we were able to run this algorithm on the entire commit layer in around 3 hours, without resorting to subsampling.

More importantly, we drastically underestimated the time required to compute the clustering coefficient of the entire undirected graph, which would have taken several years on our hardware. We had to analyze a subsample of 0.1% of the entire graph, which is in line with the sample sizes of clustering coefficient estimates in other analyzes of large networks.

As this study is exploratory in nature, we anticipated the possibility of having to resort to sampling in the protocol [123, Section 7].

One last potential internal threat is the validity of the dataset itself. Because this study is the first of its kind ever performed on the graph of software development, there is no existing dataset to which its properties can be cross-compared. We performed a series of manual robustness checks to ensure that the properties were consistent to our own expectations, which allowed us to iteratively find and correct errors in our dataset building pipeline. However, there is no way to guarantee that these checks were exhaustive.

12.6.2 External validity

While our data corpus is the largest dataset of software development history, and *aims* to be as exhaustive as materially possible, it remains a subsample of the entirety of public software commons, and as such the way it is constructed is a source of various potential bias.

Exhaustiveness of VCS and package managers. Software Heritage covers the most popular DVCS (Git, Mercurial, SVN...) as well as distribution and language-specific packages (dpkg, Nix, Python, NodeJS...), and regularly adds support for new systems. The dataset does not cover all the less commonly used systems (Bazaar, Darcs, CVS, ...). If software development patterns on these platforms are significantly different, this study cannot properly capture them in a representative manner.

Exhaustiveness of data sources. Likewise, the representativeness of the study is limited by the extent of the data source coverage of Software Heritage. The archive contains the main centralized software forges and package repositories (GitHub, GitLab, Bitbucket, PyPI, Debian, NixOS, ...), as well as instances of decentralized forges (e.g., various self-hosted GitLab or Phabricator instances). As the archive cannot realistically cover the long tail of smaller self-hosted forges, this is another source of popularity bias in the input dataset.

Archival process. The process of listing data sources and loading repositories and software packages in the Software Heritage archive is heterogeneous across data sources, which can skew the representativeness of the data. Data sources are crawled at varying frequencies depending on multiple factors: for instance, some forges support subscription-based APIs that allow the listers to crawl repositories as soon as a change is pushed to them. Some repositories are considered more critical to software infrastructures and are crawled daily. Other scheduling heuristics are in place to maximize resource usage efficiency of data crawlers. Overall, this means that the topology of the “hosting” layer is endogenous to the archival process, rather than being an intrinsic property of software development. This is mainly reflected in the number of snapshots that neighbor a given origin, since more frequent crawling generally produces more snapshots.

Non-software data. We acknowledge the habit of developers to use software development platforms and hosts for non-software projects (e.g., collaborative writing, websites, open datasets, art assets, etc.). However, we expect software development to be the dominant content hosted in these platforms. We also assume that the results of this work would be most useful for researchers when applied to similar corpuses, which would contain the same kind of non-software data, as opposed to carefully curated ones.

Identifying Software Forks

In the previous chapter, we have described the structure of the graph of software development by analyzing robust measures of network topology: degrees, connected components, shortest path lengths and clustering coefficients. These properties qualify the graph at a low-level in a generic way, enabling direct comparison with various other real-world networks. At a higher level of abstraction, we can also analyze how software development is semantically organized in *projects* which, due to collaborative development and code reuse patterns, tends to be arranged in groups of “forks” of varying sizes. Describing this high-level structure is a key part of the overarching quest for a deeper understanding of the graph of software development, and has direct methodological applications for empirical software engineering.

This chapter is based on an article [124] accepted at the 17th International Conference on Mining Software Repositories (MSR 2020).

13.1 Introduction

How developers and software communities work on their projects, and how this relationship evolves over time, have been topics of interest in software engineering research for many decades.

Historically, *software “forking”* [114] has been intended as the practice of taking the source code and development history of an existing software product to create a new, competing product, whose development will happen elsewhere and taken to different directions. This kind of “hard fork” is enabled by free/open source software (FOSS) licensing [55] and its possibility is an asset that guarantees freedom of development, while the actual occurrence

of a hard fork has generally been considered a liability [135] for project sustainability [115, 113, 130].

In the past decade, the rise in popularity of DVCS [148] introduced a significant shift of paradigm and terminology. The expression “fork” is now generally intended [174] to refer to the mere technical act of creating a new VCS repository that contains the full history (at the time of fork) of a pre-existing repository, without an implicit negative connotation (also called “development forks” [55]). Repository forks can be created on social coding platforms [42, 158] with as little as a click of a button. Then, while a forked repository *can* be used to hard fork a project, often it is just a way to work on software improvements that will be eventually sent back to the originating project as pull requests [67] for integration.

Consequently, the literature on software health and evolution has been recently focusing on studying this plurality of new forks to better understand what they can signal on the state of a software project. The forking process often exhibits some patterns that can be used to determine common criteria of healthy software projects, e.g., by measuring contributing activity using forks as a proxy, or by comparing successful hard forks to their inactive counterparts. Those metrics are particularly useful in that they are conceptually independent of the VCS used by the development team, which makes them robust across different workflows.

Likely as a consequence of the prevalence of social coding platforms, recent literature on forks has focused on a single source of truth to determine what constitutes a fork: metadata provided by code hosting providers, and most notably GitHub. In addition to cloning development history into a new repository, clicking the fork button on GitHub also registers an “is forked from” relationship between the new repository and its parent. This relationship forms an ancestry graph that GitHub makes available through its API and that is what has traditionally been studied as a large, easily exploitable fork network. However, using this forge-level information can pose some important methodological challenges for empirical studies.

The first drawback of trusting platform metadata as source of truth for what repository is a fork is that it is platform-specific. Repositories hosted on GitHub that have been forked from, say, GitLab, or more generally non-GitHub hosted repositories, cannot be identified as forks and vice-versa. Similarly, although arguably less relevant from a quantitative point of view, one cannot recognize as forks, say, Git repositories used to collaborate with Subversion repositories via the `git-svn` bridge.¹ For a fork ecosystem to be properly studied via the current approach, all the parallel development must happen using the same VCS and on the same platform. While the prevalence of Git does not seem to be waning, Git code hosting diversity is increasing, making the platform-specific part of this problem potentially severe.

¹<https://git-scm.com/docs/git-svn>

A second, more subtle methodological drawback is that trusting platform metadata introduces a selection bias on both the amount and type of forks that are considered. The number of forks is inflated by the fact that social coding platforms strongly encourage, and sometimes even automate, the creation of forked repositories as the main way to contribute even the smallest one-liner change. Many of these (soft) forks will be short-lived in terms of development activity. Hard forks will comparatively be more long-lived and will not necessarily reside on the same code hosting platform. The example of the Linux kernel community is revealing in this respect: several copies of the full development history of Linux exist on GitHub, but are not recognizable as forks of `torvalds/linux` according to platform metadata, because kernel development does not primarily happen on GitHub and kernel developers create their repositories using `git clone`.

Fork inflation also results in increased duplication of software artifacts (source code files or directories, commits, ...) across repositories [137], which has a significant impact on fork studies that rely on metrics as simple as repository size (measured as the number of hosted commits). Filtering out forked repositories is a common solution to this problem, which calls into question *how* to properly identify forks. It has been shown [137] that a partition of commits within origins sharing at least 1 commit (“most fit fork” partition), could decrease by several orders of magnitude the tail of the distribution of origin size, and impact origins of all sizes greater than 100 commits.

The absence of extensive, homogeneous fork research has been pointed out in the past as a missing piece [135] in the literature. In this chapter we try to provide methodological tools to enable fork studies that do not restrict themselves to platform metadata to recognize forks, thereby removing the constraint of analyzing a single platform and mitigating the risk of selection biases.

As an alternative to relying on platform metadata to recognize forks we propose to compare the content of VCSs and consider as forks repositories that share artifacts such as commits or entire source trees. We will explore different definitions of forking and compare their impact in terms of the amount and structure of forks identified using platform metadata. Specifically, we will answer the following research questions:

Research question 13.1. How do code hosting platform information about which VCS repositories are forks compare to the presence of shared source code artifacts in repositories?

Research question 13.2. How are (a) the amount of forks and (b) the structure of fork networks affected by fork definitions based on VCS artifact sharing?

RQ 13.1 will intuitively assess the level of trustworthiness of platform fork metadata: for instance, if many repositories share commits but are not identified as forks by platform metadata, then relying on this metadata alone would appear to be methodologically dangerous. As one might consider different types of shared VCS artifacts (commits, source

tree directories, individual files, ...) as fork evidence, RQ 13.2 will provide an empirical evaluation of the effects of basing fork definitions on one or the other.

Replication package A replication package for this chapter is available from Zenodo at <https://zenodo.org/record/3610708>.

13.2 What is a fork?

In this section we explore the spectrum of possible definitions of what constitutes a *fork*. In the following we will use the term “fork” to mean a forked software *repository*, without discriminating between “hostile” (or hard forks, according to the terminology of Zhou et al. [174]) and development forks. We propose three definitions, corresponding to three types of forks—type 1 to 3, reminiscent of code clone classification [138, 132]—along a spectrum of increased sharing of artifacts commonly found in VCSs, such as commits and source code directories.

The first definition, of type 1 forks, relies solely on code hosting platform information and requires no explicit VCS artifact sharing between repositories to be considered forks (although it allows it):

Definition 13.1 (Type 1 fork, or forge fork). A repository B hosted on code hosting platform P is a *type 1 fork* (or *forge fork*) of repository A hosted on the same platform, written $A \rightsquigarrow_1 B$, if B has been created with an explicit “fork repository A ” action on platform P .

Although informal and seemingly trivial, this definition is both meaningful and actionable on current major code hosting platforms. For example, GitHub stores an explicit “forked from” relationship and makes it available via its repositories API:²

The parent and source objects are present when the repository is a fork. `parent` is the repository this repository was forked from, `source` is the ultimate source for the network.

GitLab does the same and exposes type 1 fork information via its projects API:³

If the project is a fork, and you provide a valid token to authenticate, the `forked_from_project` field will appear in the response.

which corresponds to exploitable JSON metadata such as:

²<https://developer.github.com/v3/repos/>, retrieved 2020-01-13.

³<https://docs.gitlab.com/ee/api/projects.html>, retrieved 2020-01-13

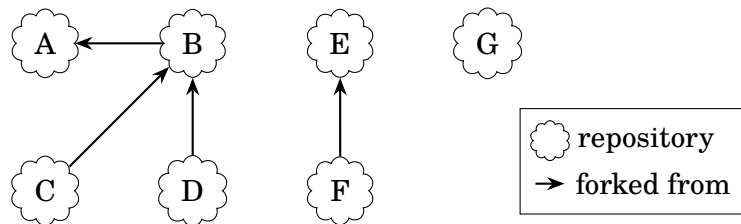


Figure 13.1: Type 1 forks, or forge forks, as declared on code hosting platforms. Repository B is a forge fork of A , C and D are forge fork of B , F of E , while no repository is a forge fork of G . Note how this definition induces a global, directed, forge fork graph (specifically: a forest of disjoint trees).

```

{
  "id":3,
  ...
  "forked_from_project":
  {
    "id":13083,
    "description":"GitLab Community Edition",
    "name":"GitLab Community Edition",
    ...
    "path":"gitlab-foss",
    "path_with_namespace":"gitlab-org/gitlab-foss",
    "created_at":"2013-09-26T06:02:36.000Z",
    ...
  }
}

```

Without getting too formal we observe that each repository is the forge fork of at most one repository (its parent) and that the relation of being a forge fork is: not reflexive ($A \not\rightsquigarrow_1 A$), not symmetric ($A \rightsquigarrow_1 B$ does not imply—and, in fact, excludes—that $B \rightsquigarrow_1 A$), not transitive ($A \rightsquigarrow_1 B$ and $B \rightsquigarrow_1 C$ does not imply—and in fact, due to parent uniqueness, excludes—that $A \rightsquigarrow_1 C$). The latter might seem surprising at first but is consistent with the definition, because the action resulting on the creation of C happened on B , not A . (We will introduce later a related notion of repository relationship that captures transitivity.)

Forge forks induce a global directed graph on repositories, specifically a forest of disjoint fork-labeled trees, as depicted in Figure 13.1.

Type 2 forks, or *shared commit forks*, are based on the ability offered by most VCSs (and all DVCSs) of globally identifying commits across any number of repositories, usually by the means of intrinsic commit identifiers based on cryptographic hashes [148, 46]. Given the ability to identify commits across different repositories we can define type 2 forks as follows:

Definition 13.2 (Type 2 fork, or shared commit fork). A repository B is a *type 2 fork* (or

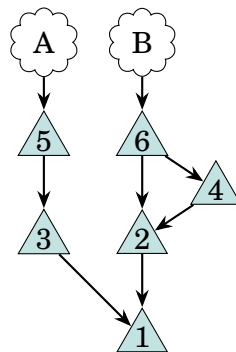


Figure 13.2: Type 2 forks, or shared commit forks. Repository A is a fork of B and vice versa, since they share commit 1.

shared commit fork) of repository A , written $A \rightsquigarrow_2 B$ if it exists a commit c contained in the development histories of both A and B .

Figure 13.2 shows an example of 2 repositories, A and B that are type 2 forks of each other, due to the fact they have in common commit 1, the initial commit; their respective development histories diverged immediately after that commit and never shared any other commits. In the general case shared commit forks will share many more commits: all the commits that were available at the time of the most recent development history divergence.

Differently from type 1 forks, the relation of being a type 2 fork is symmetric ($A \rightsquigarrow_2 B$ implies $B \rightsquigarrow_2 A$), but still not transitive (as three repositories A , B and C can have shared artifacts between A and B and between B and C without there necessarily being a shared artifact between A and C).

Intuitively, the notion of shared commit forks is more robust than that of forge forks because it allows recognizing as forks—in the broad sense of “repositories that collaborate with one another”—repositories that are hosted on different platforms. A repository hosted on GitLab.com, or your personal Git repository on your homepage, can be recognized as a fork of another hosted on GitHub. The price to pay is that, due to symmetry, the definition *alone* is not enough to orient the relationship; it does not capture which repository “came first”.

We can push this idea further, trying to make it even more robust, and capable of recognizing as forks repositories that have no recognizable shared commits, but do share entire source trees. That is of interest when, for example, collaboration happens using different version control systems (e.g., a developer using `git-svn` to participate in the development of a Subversion based project). Type 3 forks, or *shared root (directory) forks*, allow to capture those situations:

Definition 13.3 (Type 3 fork, or shared root fork). A repository B is a *type 3 fork* (or *shared root fork*) of a repository A , written $A \rightsquigarrow_3 B$, if there exist a commit c_A in the development

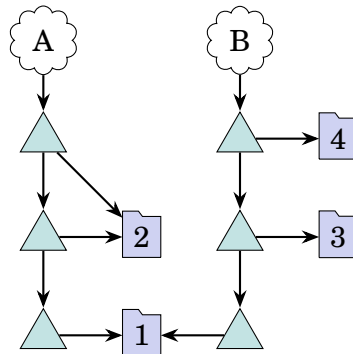


Figure 13.3: Type 3 fork, or shared root fork. Repository A is a fork of B and vice versa, since they share root directory 1. As per shared commit forks, shared root forks are symmetric.

history of A and a commit c_B in that of B such that the full source code trees of the two commits are identical.

The intuition behind type 3 forks is depicted in Figure 13.3. Note that it is not enough for the two repositories to share any arbitrary *sub*-directory to be considered forks, as that would consider as forks repositories that embed third-party libraries, an arguably undesired consequence; we need the *root* directories of two commits to be (recursively) equal for establishing a shared root fork relationship.

The same properties of type 2 forks apply to type 3 forks: the shared root fork relation is also symmetric. In most VCSs, and in all modern DVCSs, type 2 forks is also a strictly larger relation than type 3 forks: $A \rightsquigarrow_2 B$ implies $A \rightsquigarrow_3 B$, because if there exists a shared commit c that makes A and B shared commit forks, then the root directory pointed by c also makes A and B shared root forks (due to the cryptographic properties of intrinsic commit identifiers in DVCSs). This property of inclusion, in the sense of one definition implying the other, is at the heart of the analysis made in Section 13.4.3, studying the aggregation processes of networks and cliques.

In theory, we could go further, and introduce an even more lax notion of fork, that equates repositories sharing as little as a single file, but that would exacerbate the problematic behavior we discussed for sharing sub-directories.

Armed with these definitions we will be able to answer RQ 13.1, by comparing the number of forks identified by Definition 13.1 with those identified by Definition 13.2 and/or Definition 13.3 (that we refer to as *intrinsic* forks). To fully address RQ 13.2 on the other hand we need to capture the notion of “community” of repositories used for collaboration, as follows:

Definition 13.4 (Type T fork network). The *type T fork network* of a repository A is the smallest set \mathcal{N}_A^T such that:

- $A \in \mathcal{N}_A^T$

- $\forall B \in \mathcal{N}_A^T, B \rightsquigarrow_T C \implies C \in \mathcal{N}_A^T$
- $\forall B \in \mathcal{N}_A^T, C \rightsquigarrow_T B \implies C \in \mathcal{N}_A^T$

That is, a fork network is the set of all repositories reachable from a given one, following both forked from (parents) and forked to repositories (children). The definition is parametric in the type of forks, so we have type 1 fork networks (\mathcal{N}^1), type 2 fork networks (\mathcal{N}^2), and type 3 fork networks (\mathcal{N}^3).

A stricter notion that will come in handy is that of repository cliques, sets of repositories that are all direct forks (i.e., neither transitive nor reverse transitive) of each other:

Definition 13.5 (Type T fork clique). The *type T fork clique* of a repository A is the largest set \mathcal{C}_A^T such that:

- $A \in \mathcal{C}_A^T$
- $\forall C, (\forall B \in \mathcal{C}_A^T, B \rightsquigarrow_T C \wedge C \rightsquigarrow_T B) \implies C \in \mathcal{C}_A^T$

Note that, while this definition is parametric in the type of forks too, fork cliques make intuitive sense only for type 2 and type 3 forks; type 1 forks (forge forks) only have singleton cliques as the relation is not symmetric.

13.3 Methodology

13.3.1 Dataset

In this study, our goal is to experimentally determine the amount and structure of forks for the various definitions we have introduced. To do, so we will use two datasets: the Software Heritage Property Graph Dataset described in Chapter 8, which contains the development history needed to find intrinsic fork relationships, and a reference forge-specific dataset, GHTorrent [68], which contains the fork ancestry relationships as captured by GitHub.

GHTorrent GitHub is the largest public software forge, and is therefore the candidate of choice to study forge forks (type 1). GHTorrent [68] crawls and archives GitHub via its REST API and makes periodical data dumps available in a relational table format. In its database schema, the project table contains a unique identifier for each repository, and a `forked_from` column contains the ID of the repository it has been forked from if the repository is considered to be a forge forks. A single SQL query on this table allows to extract the full graph of GitHub-declared forks, e.g.:⁴

⁴Additional URL gymnastics are needed in the query to cross-reference GHTorrent project URLs with Software Heritage ones; we refer to the replication package for this kind of details.

```
select parents.url as parent,
       projects.url as child
from projects
inner join projects as parents
       on projects.forked_from = parents.id
```

Software Heritage Graph Dataset The Software Heritage Property Graph Dataset data model maps the traditional concepts of VCSs as nodes in a Merkle DAG [100], as described in Chapter 4 and Chapter 8. As a consequence, all the development artifacts, including commits and source trees, are natively deduplicated within and across projects. This property is particularly useful to find intrinsic forks, as it enables tracking the relevant artifacts (revisions and directories) across the entire dataset and link them back to their source origins.

The dataset contains two intermediate layers between origins and the commit graph they point to, snapshots and tags. As none of our fork definitions depend on these artifacts, the two layers can be flattened out so that the origins point directly to the revision graph. Likewise, the blob layer and the directory layer are not needed to find shared commit forks (Definition 13.2), while shared root forks (Definition 13.3) only require the root directory of each revision. Filtering out the unnecessary nodes reduces the graph to a more reasonable size of 2 billion nodes (down from 10 billion), which makes it easier to process on a single machine. The structure of the resulting subgraph closely matches the examples in Figure 13.2 and Figure 13.3, making it easy to verify the intrinsic definitions.

We run the experiments on the compressed version of the two graph datasets, using the WebGraph framework, as described in Chapter 9. The Software Heritage Property Graph Dataset is already distributed as a compressed BVGraph and does not need any additional processing to be exploited. The GHTorrent can be compressed from its relational database format using the same graph compression framework introduced in Chapter 9.

13.3.2 Fork networks

The easiest way to get a first sense of the amount and structure of forks according to the various definitions is to find all fork networks, as per Definition 13.4. This can be done in linear time with a simple graph traversal with linear complexity: two repositories are in the same network if and only if there exists a path between them in the undirected subgraph of origins and revisions. (We recall from the dataset section that we have removed the snapshot and revision layers, so that root commits are directly pointed by repository nodes.) Finding all the fork networks is therefore equivalent to computing the connected components on this subgraph, as exemplified in Figure 13.4.

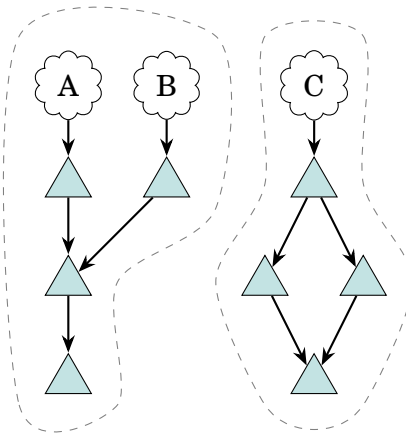


Figure 13.4: Fork networks identified as connected components, for the case of shared commit (type 2) forks. Connected components are computed on the undirected version of the shown Merkle DAG. Measuring network sizes as the number of contained origin nodes we obtain that: repositories A and B are forks of each other and members of a network of size 2, while repository C is in its own singleton network.

Using fork networks has the advantage of allowing easy interpretations of the results. First, it is trivial to quantify how many repositories are forks by counting the number of repositories that belong to non-singleton networks. Besides, a direct comparison can be made between the distribution of forge forks and shared commit or root forks, as networks provide a partition method for both graphs. The network sizes can be directly compared between the three definitions while keeping the invariant of the number of total repositories. This is not the case when looking at fork cliques, since the same repository can be found in multiple cliques, which makes comparison harder.

In GHTorrent origins are already linked together in a global graph where the edges represent the forge-level forking relationships. We can partition this forge fork graph in fork networks similarly by computing all its connected components.

Our experimental design is therefore as follows: first, we list the common non-empty repositories between the Software Heritage Property Graph Dataset and GHTorrent. We then extract the aforementioned subgraphs: the development history graph for Software Heritage (origins \rightarrow {revisions, releases} \rightarrow commits) and the fork graph for GHTorrent (origins \rightarrow origins). We then compute the connected components of each graph using a simple depth-first traversal algorithm, then output the origins contained in each component.

13.3.3 Fork cliques

While partitioning the corpus in fork networks gives a good idea of how intrinsic forks are linked together, it can group together repositories that are not forks of each other, as the intrinsic fork relationship is not transitive. Figure 13.5 shows a pattern that we

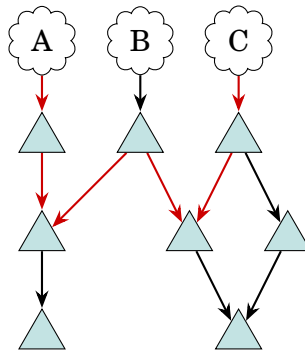


Figure 13.5: Example of misleading clustering of fork networks. Here, repositories A and C are in the same network because there is a path between them, even though they do not share common development history.

have verified as commonly found in the wild, where two different cliques will be merged in the same fork network—A and B are part of the same clique as they share development history; the same applies to B and C; whereas A and C do not share any part of their respective development histories but will end up in the same network. In simpler terms, this means that a repository containing a merge commit with two parents from otherwise unrelated repositories will make those repositories belong to the same fork network. We expect this effect to merge cliques into giant components, which will make the size of the largest networks hard to interpret.

The other interesting metric that can be looked at is the distribution of fork cliques, as defined in Definition 13.5. While cliques do not provide a partition function for the graph, they allow narrowing down the actual extent of forking relationships within large fork networks.

Due to the fact that shared commit fork cliques are defined pairwise, the naive algorithm to find all the inclusion-maximal cliques is superlinear: for each repository, walk through its commit history and add all the commits to a queue, then take the transposed graph to walk through the commit history backwards and list all repository leaves. The time complexity of this algorithm is highly impractical: in the worst case, if all the repositories are forks of each other, it has time complexity of $O(R \times C)$ where C is the number of commits and R the number of repositories in the graph.

However, clever use of some properties on the DAG structure of the commit graph can substantially speed up the algorithm. First, fork cliques can be generated by iterating on the common ancestors instead of the repositories: for each commit c , if it has more than one repository leaf when doing a traversal on the transposed graph, then c was a common commit ancestor, and the generated set of repositories is a fork clique. Besides, since the ancestry relationship is transitive, the clique with commit c as a common ancestor is the same as the clique generated by running the traversal on its parents. By induction, it is

Algorithm 3 Find all the fork cliques.

```
function FINDORIGINLEAVES( $r$ )
   $S_O \leftarrow$  empty set
  for all  $n \in$  ANCESTORSDFS( $r$ ) do
    if TYPE( $n$ ) = ORIGIN then
      add  $n$  to  $S_O$ 
    end if
  end for
  return  $S_O$ 
end function

function FINDCLIQUES( $G$ )
   $S_F \leftarrow$  empty set
   $S_C \leftarrow$  empty set
  for all  $n \in G$  do
    if TYPE( $n$ ) = REVISION and  $n$  has no parents then
       $c \leftarrow$  FINDORIGINLEAVES( $n$ )
       $f_c \leftarrow$  FINGERPRINT( $c$ )
      if  $f_c \notin S_F$  then
        add  $f_c$  to  $S_F$ 
        add  $c$  to  $S_C$ 
      end if
    end if
  end for
  return  $S_C$ 
end function
```

possible to compute all the cliques simply by doing one traversal per “root” commit.

The resulting algorithm is Algorithm 3: for each root commit with no parents, we generate the clique of all repositories that contain it. We use a cryptographic hash fingerprint to avoid adding multiple times the same clique if it has multiple root commits. While this algorithm technically does not change the worst case complexity on arbitrary graphs, it is still a huge speed improvement in our case, as commit chains tend to be degenerate (i.e., very long chains with in-degrees and out-degrees close to 1 on average). Algorithm 3 has a best-case complexity of $\Theta(C)$, equivalent to a single DFS traversal. The commit graph is largely close to this best-case scenario, making the algorithm run in just a few hours on the entire corpus.

While Algorithm 3 works well for shared commit forks, the speedup does not apply to shared root forks: the induction property no longer works for root directories, as they are not organized in nearly-degenerate chains. The time complexity for type 3 forks is closer to the worst case of $O(C \times R)$, which makes the clique analysis impractical for this kind of forks.

Algorithm 4 Compute the p-cliques partition function.

```

function CLIQUESTOPARTITION( $L_C$ )
  REVERSEIZESORT( $L_C$ )                                ▷ Process larger cliques first
   $I \leftarrow$  empty map                                ▷ Build reverse index
  for all  $c_i \in L_C$  do
    add  $\{i \rightarrow c_i\}$  to  $I$ 
  end for
  for all  $c_i \in L_C$  do
    for all  $r_j \in c_i$  do
      for all  $s \in I[r_j]$  do                          ▷ Remove subsequent occurrences of  $r_j$ 
        if  $k > i$  then
          remove  $r_j$  from  $s$ 
        end if
      end for
    end for
  end for
   $L_C \leftarrow$  REMOVEEMPTYSETS( $L_C$ )                ▷ Remove cliques left empty
  return  $L_C$ 
end function

```

P-clique partition function While cliques do not directly provide a way to partition the corpus in several fork clusters (because a single origin can be contained in multiple cliques), it is possible to define a partition function based on them, e.g., by always assigning repositories to the largest clique they belong to. As repositories belonging to multiple cliques appear to be a quite rare occurrence (as they require the equivalent of a `git merge --allow-unrelated-histories` on two completely different repositories), the arbitrary criterion choice is not expected to be a significant caveat to interpret the results.

We use Algorithm 4 to generate the partition function of the graph based on cliques. To implement the criterion of attributing repositories to their largest cliques, cliques are processed in decreasing order of size. Building a reverse index of “repository \rightarrow clique it belongs to” allows direct access to the subsequent occurrences of repositories in smaller cliques to remove them. After doing so, the cliques left empty are removed and the newly generated graph partition can be returned.

This algorithm generates as its output a set of sets of origins that are subsets of the input fork cliques. We call this set “*fork p-cliques*” to emphasize the fact that they form a partition of the repository set in which all the groups are fork cliques.

Once this p-clique graph partition is established, the fork definition can once again be compared with the forge definition, by looking at the difference between the size distribution of the partitioned cliques of type 2 forks and the size distribution of networks for type 1 forks.

Table 13.1: Number of forks and networks by fork type.

Fork type	# forks	# networks
Forge forks (type 1)	18.5 M (44.7%)	25.3 M
Shared commit forks (type 2)	20.1 M (48.4%)	24.0 M
Shared root forks (type 3)	25.3 M (61.1%)	18.5 M

13.4 Results

We identified 71.9 M repositories in common between the Software Heritage Property Graph Dataset and GHTorrent, 41.4 M of which are non-empty. We focused our experiments on these repositories.

13.4.1 Fork networks

In the GHTorrent graph, we found 25.3 M different connected components, among which 22.9 M repositories isolated in their own component, which means they are not forge forks (type 1) of other repositories. The other 2.4 M connected components contain the remaining 18.5 M repositories, which are all in fork networks. These forge forks represent 44.74% of all repositories.

In the Software Heritage Property Graph Dataset, we found 24.0 M connected components, among which 21.3 M isolated repositories. The remaining 2.6 M components contain 20.1 M shared commit forks (type 2), i.e., 48.44% of all repositories. We have hence almost 9% *more* shared commit forks than forge forks, which is a significant divergence for the strictest definition of forks based on shared VCS artifacts.

For shared root forks (type 3), we found 18.5 M connected components, among which 16.1 M isolated repositories and 2.4 M intrinsic forks (61.08% of all repositories), which is almost 37% more than the forge forks. These results are summarized in Table 13.1. They suggest that **between 1.6 M (3.8% of total) and 6.8 M (16%) repositories might be overlooked when studying forks using only GitHub metadata** as a source of truth for what is a fork.

Figure 13.6 shows the cumulative frequency distribution of fork networks for intrinsic forks and forge forks. That is, for each fork network size x , the number of repositories in networks of size $\geq x$ is shown. At first glance, the distribution of forge forks and shared commit forks appear to be pretty similar (although the log scale minimizes the differences between the two), which is a good sign that the two definitions are not returning vastly different results. The average size of fork networks is also about the same (≈ 7.6 for type 2 forks, ≈ 7.7 for type 1). The situation appears to be quite different for shared root forks, where the average size is ≈ 10.5 and the frequency distribution is significantly farther from the reference distribution of forge forks.

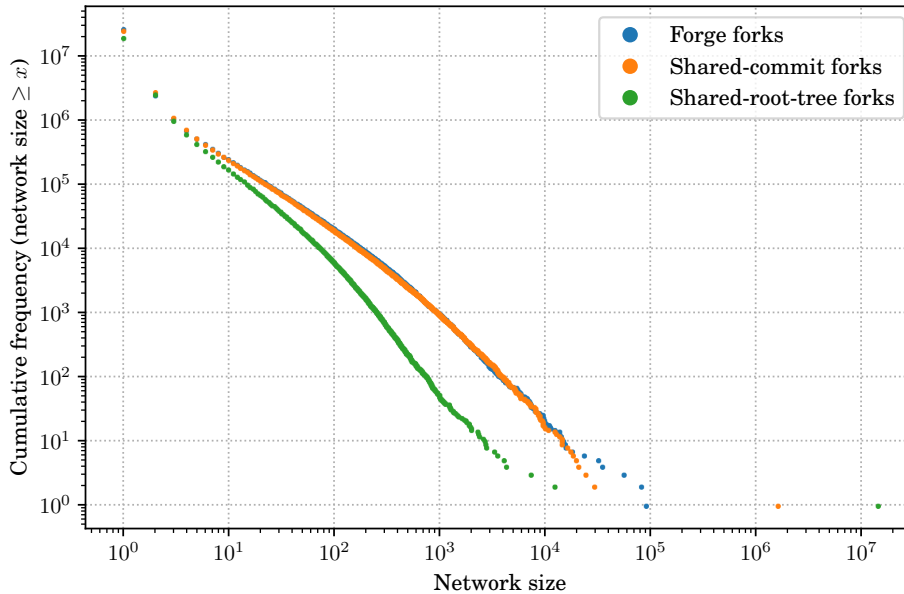


Figure 13.6: Cumulative frequency distribution of fork network sizes.

One distinguishing feature of each distribution of type 2 and type 3 forks is the size of the largest connected component, which is significantly larger than the largest networks of forge forks (by a factor of 17 for shared revision forks, and 157 for shared root forks). As discussed in Section 13.3.3, this is an expected outcome of our use of networks as a quantification metric and confirms the need for further analysis through fork cliques. This does not however have any implications on the quantification aspect of the experiment, as partitioning this network further using fork cliques would still yield the same number of non-isolated repositories.

13.4.2 Fork cliques

As expected, running Algorithm 3 to generate the shared-revision cliques on the compressed graph does not take more than an hour, which is the same order of magnitude as the time needed for a simple full traversal of the revision graph [27]. This confirms our prediction that in the shared-revision case, the average-case runtime of the algorithm is close to $\Theta(R)$.

The algorithm finds 24.5 M cliques, although the results are difficult to interpret in this current state as the cliques overlap together. A few key observations can nevertheless already be made, notably the absence of very large cliques: the largest clique contains 92.4 M repositories, which is very similar to the largest forge fork network (which contains 90.2 M repositories). This is consistent with our intuition expressed in Section 13.3.3 that the largest intrinsic fork networks are a specific feature of networks (as seen in Figure 13.5), and that these artifacts disappear when looking at the cliques. It is also possible to measure how

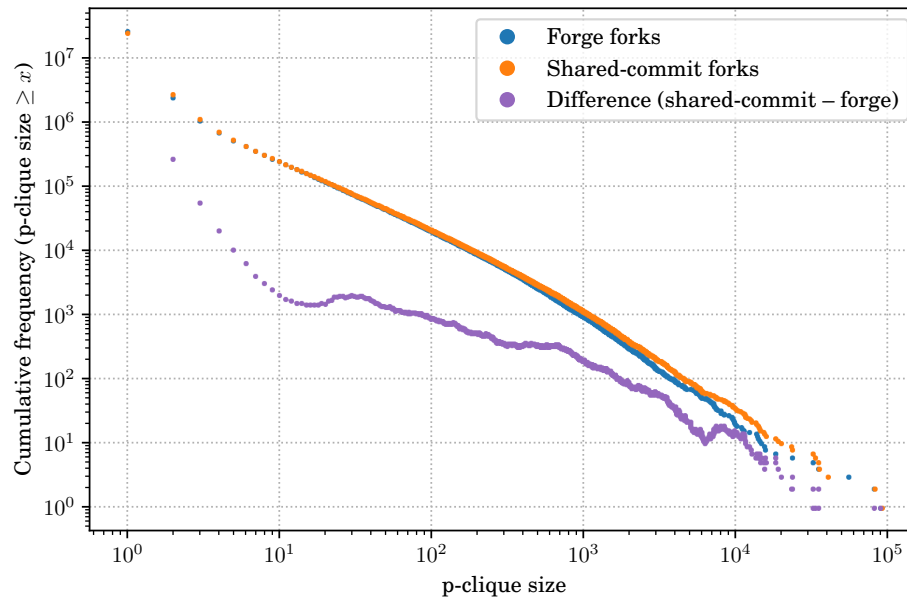


Figure 13.7: Cumulative frequency distribution of intrinsic fork p-cliques compared to forge fork networks

the cliques overlap: 28 M repositories are present in a single clique, while the remaining 13.3 M appear two times or more. On average, each repository appears in ≈ 1.47 cliques.

Computing the p-clique partition function using Algorithm 4 removes this overlap to allow a direct comparison with the forge fork networks. This algorithm takes a few minutes to process the 24 million cliques and returns the p-clique partition directly, restoring the invariant of the total number of repositories (41.5 M).

There are 24.0 M of p-cliques partitioning the graph, which is pretty close to the number of forge fork networks in GitHub (25.3 M). 21.3 M repositories are isolated in their own p-clique (51.6%), and the remaining 48.4% are in cliques of size larger than one, which is consistent with the findings of Section 13.4.1 which uses fork networks as a quantification mechanism.

Figure 13.7 shows the cumulative frequency distribution of the sizes of the shared-commit fork cliques, compared to the baseline of forge fork networks. As before, the graph can be read as: “for each clique (resp. forge fork network) of size x , the number of repositories found in cliques (resp. networks) of size $\geq x$ ”.

The visual similarity between the two distributions is striking: while the p-clique distribution of shared-commit forks seems to be consistently above the forge fork network baseline for groups of size ≥ 2 , they always appear to be very close to each other, even farther in the tail. This **suggests that type 2 forks capture well what developers typically recognize as forks**.

To formally assess this similarity, the graph also exhibits the cumulative difference

between the clique distribution and the baseline. This is, in essence, the cumulative size distribution of the cliques of forks *overlooked* when using only the GitHub metadata. This cumulative distribution **mostly stays positive, suggesting that using DVCS data to identify forks is overall a net gain in coverage**. It also appears that the difference is typically at least one order of magnitude less than the size of the clusters, emphasizing the proximity between the two definitions.

13.4.3 Aggregation process

Two repositories having a common commit ancestor necessarily have a common root source tree (the root source tree of that common commit ancestor), so all the repositories that belong to the same shared-commit network also belong to the same shared-directory network. Similarly, we expect that most origins declared as forge forks will be in the same shared commit and shared root source tree fork networks. By switching from one definition to another, we expect the clusters to aggregate together smaller clusters from the previous definitions.

To characterize this aggregation process into fork clusters at different granularities, we compute the Kolmogorov-Smirnov (KS) distance between the weighted cumulative distributions function of the clique or network size.

We note δO the KS difference between a fork definition A and a fork definition B, and represent it as a function of the size of the network (or partitioned clique). By definition δO is always equal to zero for sizes $s = 1$, since all the forks are in clusters of size $s \geq 1$, and $s = \max(\text{cluster sizes})$, since there are no clusters larger than this size.

Because the total number of repositories is invariant, we can plot the KS distance weighted by repositories to see how the repositories found in fork networks (or cliques) of a given size will progressively aggregate into fork networks (or cliques) of different sizes. Figure 13.8 represents δO between the forge fork definition baseline and: shared commit fork networks (top), shared commit p-cliques (bottom), and shared root tree fork networks (middle).

While this analysis shows the flux of repositories between clusters identified by the different definitions, it can mask some compensating phenomena by merging independent processes, as some repositories can migrate from larger to smaller clusters, sometimes leading to $\delta O < 0$ (Figure 13.8, bottom, size $\approx 10^5$).

To narrow down this phenomenon, we specifically focus on the largest shared-commit fork network to see how it contributes to the global flux. By taking the repositories in this network and the size distribution of the forge fork networks, we show in Figure 13.9 the repository flux, as defined above, and compare it to the corresponding global flux.

Several points are noteworthy. First, only 15% (200 k origins over 1.53 M, the red dotted line) of the origins that were isolated (blue dot for $size = 2$) in forge fork networks are

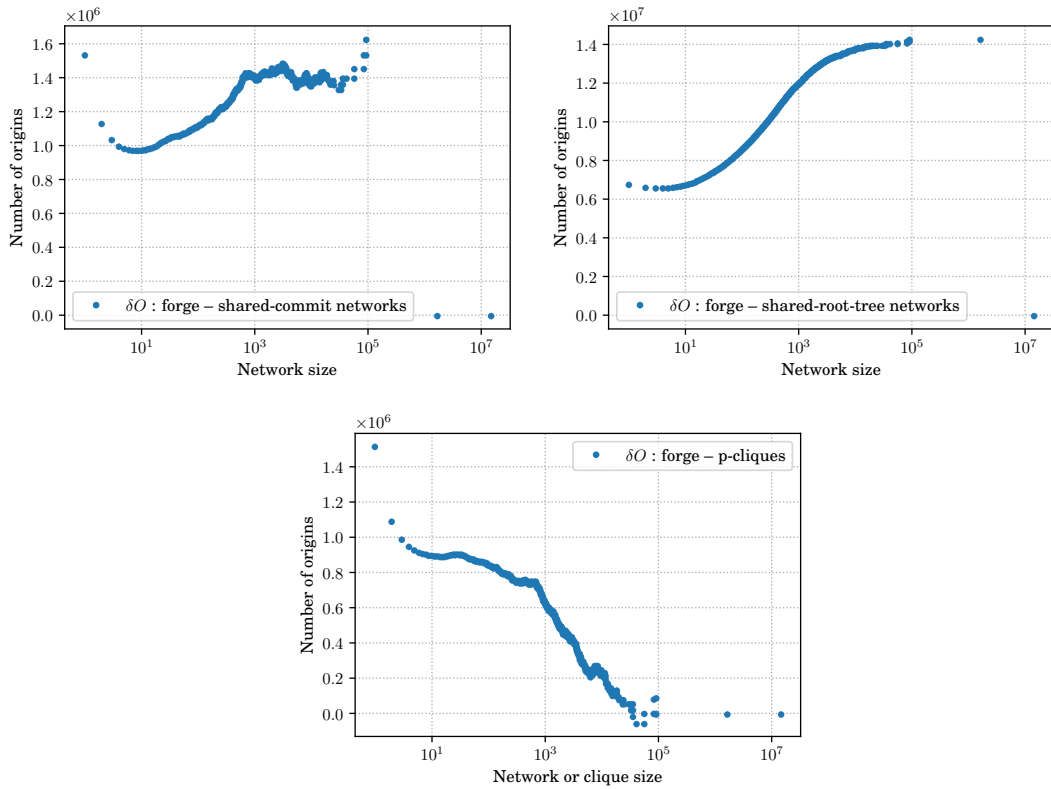


Figure 13.8: Complementary Cumulative Weighted Distribution Functions Differences between forge fork network and shared-commit fork network (top left), shared root source tree fork network (top right), and p-cliques based fork network (bottom).

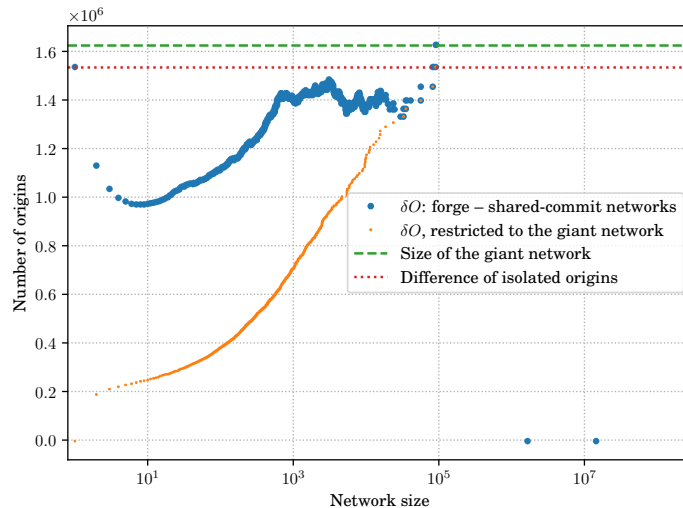


Figure 13.9: Contribution (orange dots) of the giant (largest) network that appears using the shared commit fork definition w.r.t. δO : forge fork – shared commit fork (same as Figure 13.8, top).

aggregated in the giant network of shared commit fork. This shows that the aggregation mechanism is not only to this “giant” network, since $\approx 85\%$ of the origins are aggregated within smaller networks.

Then, the flux for the 6 largest networks (isolated blue and orange dots around 10^5 that overlap) is almost the same whether we restrict ourselves to the origins of the giant network (orange line) or to all the networks (blue line). We conclude that aggregation for the large network sizes is dominated by absorption into the giant cluster, without any redistribution to smaller networks.

This confirms that the “aggregation/merge” mechanism which happens when changing the definition is not just an absorption phenomenon into a “super attractor”, but concerns all network sizes, with larger networks absorbing networks of any size.

13.5 Threats to validity

Internal validity Aside from forge forks (type 1) we have no certainty on how well the proposed fork definitions capture what developers would recognize as forks. While shared commit (type 2) and shared root (type 3) forks make intuitive sense, the sheer volume of data to be analyzed makes it very hard to rule out the existence of pathological cases. Certainly type 2 and type 3 fork definitions can be “gamed”, making unrelated repositories appear as forks when they intuitively are not. Unusual development workflows might also induce topology in the global development graph that merge together repositories that would not be considered forks by developers. There exists an apparent trade-off here between fully automatable definitions based on VCS artifact sharing, and qualitative assessment by developers that does not scale to datasets like the one studied here.

As a consequence of the above we do not feel confident at this stage in making a methodological judgment call on whether type 2 or type 3 fork definitions “better” capture the essence of a fork. We simply warn scholars about the extent of the discrepancies between the number of forks detectable via shared VCS artifacts and forge-level metadata. Further work, of both statistical nature (looking for outliers) and based on structured interviews with developers (to review uncommon cases), is needed to improve over this point.

External validity The datasets used in this study do not capture the full extent of publicly available development history, notably due to their snapshot nature and their assembly through periodic crawling processes. The Software Heritage Property Graph Dataset only contains data from various forges to the extent of what is covered by the Software Heritage archive, which might lag behind the tracked forges, and GHTorrent is GitHub-specific. As we need comparable samples we were limited by the intersection of the two datasets in this study, which composes these limitations. Still, to the best of our

knowledge this is one of the largest quantitative fork studies to date, having considered more than 40 M public version control system repositories.

In the future it would be interesting to extend this approach to forges that are rising in popularity, and most notably GitLab.com. For that we would need a GHTorrent equivalent (or corresponding ad hoc crawling of that forge for the purposes of the study only).

13.6 Related work

Accounts of the history of forking have been given by Nyman [114] and Zhou et al. [174, Section 2]. The latter also covers the terminological and cultural shift from hard forks (to be avoided) to forking as the mere technical act of duplicating VCS history, possibly as a basis for future collaboration. The present work is agnostic to which interpretation prevails, as in both cases the main observable effect of forking are VCS repositories that share parts of an initially common development history.

Hard forks Hard forks have been studied extensively in seminal works by Nyman [115, 116, 113, 114], covering historical origins, motivations for forking (or not), and sustainability considerations in the socioeconomic context of free/open source software (FOSS) development. Robles and González-Barahona [135] give a detailed account of famous hard forks, covering history, reasons, and outcomes.

These and other studies of hard forks are qualitative and focused in nature. This chapter is complementary to them as it proposes tools to identify and quantitatively measure and observe forks, addressing the need of more extensive and homogeneous fork research already observed in [135]. As far as we could determine without fully replicating the corresponding studies, the VCS repositories involved in the hard fork cases cited thus far would be correctly identified as either type 2 or type 3 forks.

Development forks With the advent of DVCSs and social coding [91], there has been a significant amount of empirical research devoted to development forks. Motivations for forking on GitHub have been studied by Jiang et al. [81].

The structure of forks on GitHub has been analyzed in several studies. Thung et al. [158] have characterized the network structure of social coding of GitHub, including forks. Padhye et al. [120] have measured external contributions from non-core developers. Biazzini and Baudry [22] have proposed metrics to quantify and classify collaboration in GitHub repositories pertaining to the same fork tree. Rastogi and Nagappan [130]—as well as Stanciulescu et al. [151] for firmware projects—have characterized forks on GitHub based on the flow of commits between them and the originating repository.

Various performance aspects of the pull request development model [67, 69] have been also studied. Latency in acceptance has been a popular one [171, 160]; the amount of

generated community engagement [42, 41] another. A more general accounting of efficient forking patterns has recently been given by Zhou et al. [174].

To the extent we could determine it without full replication, all aforementioned studies on forking for social coding purposes rely on platform (and more specifically GitHub’s) metadata to determine which repository is a fork (of which other). As such, involved repositories would be recognized as type 1 forks, and non type 1 forks (but nonetheless type 2 or 3 forks) might have been overlooked in the studies. To be clear: we have no reason to believe that the findings in those studies would turn out to be different by enlarging the set of considered forks using the alternative definitions. We simply propose to acknowledge fork type discrepancy as an internal validity threat in future studies.

Fork definitions Aside from the already discussed hard forks vs. development forks distinction, the only other work we are aware of on formal or semi-formal fork characterization is [137], which introduces the notion of *most fit fork*: a repository that, within a group of VCS repositories that share commits, contain the largest number of commits. The notion is proposed as a long-term approximation of the main development line of a forked (hardly or otherwise) project. Our notion of type 2 fork cliques captures the same idea; additionally we show how to use it to partition the global set of VCS repositories into independent clusters instead of partitioning the global set of commits.

Methodology Methodological issues and risks in analyzing GitHub were pointed out by Kalliamvakou et al. [83]. While not directly addressed as an explicit risk, forks not recognized as such are echoed by perils “*I: A repository is not necessarily a project*” and “*IX: Many active projects do not conduct all their software development in GitHub*” in that article. Proposed mitigations were, respectively, “*consider the activity in both the base repository and all associated forked repositories*” and “*Avoid projects that have a high number of committers who are not registered GitHub users and projects which explicitly state that they are mirrors in their description*”.

Half a decade later, it is arguably *less* of a risk that development happens elsewhere and that a high number of committers are not registered GitHub users (due to the current market dominance of GitHub). However, it is still a risk and might be about to increase again due to push back against centralized services among FOSS developers. In this chapter we provide methodological tools and improve upon the mitigation techniques proposed back then. Instead of avoiding projects, one can start from cross-platform datasets [3, 137, 126, 92] and measure the amount of shared VCS artifacts in the available repositories.

13.7 Discussion

When relying only on forge-specific features and metadata to identify forked repositories, empirical studies on software forks might incur into selection and methodological biases. This is because repository forking can happen exogenously to any specific code hosting platform and out of band, especially when using DVCSs, which are currently very popular among developers.

To mitigate these risks we proposed two different ways to identify software forks solely based on intrinsic VCS data and development history: a *shared commit forks* (type 2) and a *shared root directory forks* (type 3) definition of software forks, as opposed to *forge forks* (type 1) which are identifiable only when created on specific code hosting platforms by, e.g., clicking on a “fork” UI element. We also introduced the notions of *fork cliques* (set of repositories that share parts of a common development history) and *fork networks* (repositories linked together by pairwise fork relationships) as ways to understand and quantify larger sets of forks when using non-transitive definitions of forking.

Via empirical analysis of 40+ M repositories using the GHTorrent and Software Heritage datasets we quantified the amount of type 2 and type 3 forks that are not recognizable as type 1 forks on GitHub, which appears to be substantial: +9% forks for type 2 forks, +37% more for type 3.

We also showed that the aggregation/merge dynamics into larger clusters of related repositories upon changing fork definitions is not just an absorption phenomenon into a “super attractor” cluster, but that it concerns all clusters: smaller ones are absorbed into larger ones of any size.

The methodological implications of our findings are that:

- Empirical software engineering studies on software forks aiming to be exhaustive in their coverage of forked repositories should consider using fork definitions based on *shared VCS history* rather than trusting forge-specific metadata.
- Depending on the research question at hand, the objects of studies to consider when looking at repositories involved in forks are either *fork networks* or *fork cliques*. The latter have the advantage of excluding cases that exist in the wild (e.g., on GitHub) in which repositories that do not share VCS artifacts might end up in the same fork network due to transitivity.
- Any set of repositories can be partitioned in accordance with its relevant shared commit fork cliques by computing its *fork p-clique* partition function. This way of grouping together repositories that are all type 2 forks of each other is easily substitutable to partition approaches based on forge fork metadata.

CONCLUSION

Conclusion

This thesis retraces my journey of trying to make an immense corpus, the entire *graph of public software development*, accessible for software mining research. This endeavor proved to be challenging. The sheer scale of the corpus, a graph of more than 20 billion nodes and 200 billion edges, precludes the use of most conventional approaches for data processing, and necessitates state-of-the-art techniques in big data and graph analysis. While venturing to organize this development graph in a way that could be empirically studied, I encountered and tackled challenging issues in both the technical and research aspects of my work, which I have described extensively in this thesis. In this final chapter, I summarize my main academic contributions to the fields of empirical software engineering and software mining, and discuss open questions and future research directions.

14.1 Summary of academic contributions

I have made several contributions to the scientific literature that I described in this thesis, taking the form of novel research questions, techniques and empirical studies. These contributions were, to the best of my knowledge, never studied to this extent in the extant literature.

Contextualization The first original contribution in my work stems from an inquiry into how making a large corpus of software development data accessible to researchers can be the most useful to software mining research. For this purpose, I realized a literature review (Chapter 1 and Section 5.1.1) which helped determine the most effective ways in which making the Software Heritage data accessible to researchers can be of use to future

software mining studies (Section 5.2). From this, I derived a concrete roadmap [127] to create a software mining platform for the Software Heritage archive (Section 5.4).

Small-scale artifact processing I presented the designs and implementations for two techniques to access and process small to medium sets of software artifacts (up to tens of thousands) from a large remote repository of artifacts, one based on preparing bundles of artifacts on the server-side (Chapter 6) and another using a virtual filesystem [7] to provide lazy access to objects (Chapter 7). The latter is, as far as I know, the first attempt to create a virtual filesystem for a universal corpus of source code and software development.

Scale-out processing of the relational graph As described in Chapter 8, I have made available several exports of the graph of development history as an exploitable dataset [126] in a relational format, along with a reusable pipeline to produce future versions of this dataset. I have shown how this dataset can be leveraged with OLAP platforms by answering concrete examples of research questions on the entire corpus. I have also made the dataset available on public clouds in a way that can be queried by researchers with minimal setup, further reducing the friction of analyzing the full graph of development history. The usability of these graph exports for software mining studies were further demonstrated by making it the object of study in the MSR 2020 mining challenge [125]. While other databases of software development history were already accessible in a similar manner ([68, 78, 92]), this is the first time such a vast corpus of deduplicated software artifacts archived from such a variety of different sources was made available for study, in a way that is demonstrably exploitable for practical uses.

Graph compression and exploitation In Chapter 9 I introduce graph compression as a way to fit the entire graph of software development in the RAM of a single machine [27], which makes it possible to run complex graph algorithms that are not necessarily easy to parallelize on large clusters. While the techniques of graph compression are not new, applying them to the deduplicated graph of software development is a novel approach in the field of software mining: all previous works in the literature relied on scale-out approaches for large datasets. Graph compression opens new possibilities for systematic empirical analyses of large graphs of software development artifacts. These research opportunities are further broadened by the addition of node and edge properties to the compressed graph; in Chapter 10 I describe methods to store these graph properties compactly and so that they can be queried efficiently from the compressed graph. In Chapter 11, I present two ways of querying the compressed graph and its properties at a high level; one involves a simple traversal API, while the other relies on graph query languages. I also describe different techniques to subsample the graph into smaller coherent corpuses, as well as a way to estimate the size of the obtained subgraphs using empirical measurements.

Topological properties Chapter 12 describes how I conducted the first empirical exploratory study [123] on the intrinsic structure of the graph of software development and its most salient topological properties as a complex network. I analyzed the following robust measures of network topology: degree distributions, connected components, shortest path lengths and clustering coefficients. Computing these metrics efficiently was made possible by the graph compression techniques presented in Chapter 9, and thus I do not expect that this study could have been realized in the absence of these contributions. From these metrics, I derived concrete implications for software mining research by accurately describing the general form of the different layers in the graph (filesystem, development history and hosting layers), notably by looking at which distributions could be considered scale-free.

Identification of software forks Finally, I conducted a second extensive empirical study [124] on the semantic organization of software projects in groups of forks of varying sizes, described in Chapter 13. I introduce two different notions of “forks” that can be robustly identified without relying on external forge-level metadata, instead using shared software artifacts: “shared commit” forks and “shared root” forks. I quantitatively compare how these definitions of forking match with the definition based on external forking metadata. In addition, I empirically analyze how these forks organize in fork networks and fork cliques of different sizes. I then propose a computationally feasible way to partition the set of origins in the graph by grouping together repositories that are all shared-commit forks of each other.

14.2 Empirical findings and impact on software mining research

While many of the contributions presented in this thesis are new techniques for large-scale software mining, I also conducted empirical studies on the graph of software development, which yielded potentially impactful insights for this research field. These findings are summarized below as stylized facts.

Per-layer topology While assessing the topological properties of the graph by decomposing it into different layers, one important finding is that there is a large disparity in the low-level structure of these layers. As detailed in Section 12.5, the three main layers have dramatically different shapes, densities and connectivities, and each take up very uneven shares of the total size of the graph. The properties of the full graph are largely dominated by those of the filesystem layer, which represents the vast majority of nodes (90%) and edges (97%) in the graph. This highlights the importance of studying graph

layers separately when assessing the graph properties, as it can have a considerable impact on the findings.

Graph connectivity and sharding Another important empirical finding from the low-level topology analysis is that the graph is generally well-connected due to the deduplication of the software artifacts it contains. This is particularly true at the level of the filesystem, where 97% of all the files and directories in the graph are all reachable from each other and form a giant connected component. This giant component precludes the use of naive partition techniques, such as regrouping nodes in small tightly connected clusters, as a way to “shard” the graph on multiple machines in order to perform scale-out computations. We observe that this high connectivity is resilient, and does not depend on the existence of a few nodes with large in-degrees; removing these nodes is of little help to partition the giant component.

However, restricting our view to the history layer yields a very different picture: revision chains are less connected, with the giant component only encompassing 3% of the total number of revisions. This indicates that the history layer can easily be partitioned in reasonably sized groups of connected clusters, which is impactful for empirical studies looking at distributing the nodes in the history layer on multiple machines.

Kurtosis, scale invariance and heavy tails Some of the frequency distributions studied in this thesis have heavy tails, a relatively high kurtosis and exhibit scale-invariant behavior. This has important implications for empirical software engineering studies using data filtering and sampling as a way to get a representative set of software artifacts to analyze. Notably, it means that there is often no obvious rule to filter out outliers that cross a certain threshold, as they are an integral part of the distributions’ nature. This emphasizes that empirical software mining studies should be cautious to systematically justify their data selection methodology, and consider sampling biases as potential threats to validity.

Fork identification A key empirical result of the fork identification study is that relying on external forge-level metadata as the sole source of information to determine whether two repositories are forks of one another tends to miss a significant share of what people intuitively think of as software forks. By instead using the “shared commits” definition of a fork, we identified around 9% more forks than when using the GitHub forking information, suggesting that a non-negligible number of repositories are forked without using forge forking tools (e.g., by manually cloning a repository and pushing it to a different one). We expect that this number would be even higher when including data from other hosting places, as there is no external source of information to identify forks across different forges. Empirical software mining studies on software forks aiming to be exhaustive

in their coverage and reduce potential sources of bias should thus consider using fork definitions based on *shared VCS history* rather than trusting forge-level metadata, using the identification technique based on the compressed graph described in Chapter 13.

14.3 Future work

The work presented in this thesis improves the state-of-the-art of large-scale software mining platforms, and by doing so raises a significant number of open questions and enables new avenues for future research. Below is a summary of what we consider to be the most promising next steps in pursuing this research.

Incremental graph compression An impactful research problem that could be tackled is to produce a functional design for *incremental graph compression*, i.e., having an updating pipeline constantly maintaining the compressed representation of the graph up-to-date with the latest software artifact data. Doing so would considerably reduce the delay between the current state of the archive and the data stored in the compressed graph, from entire months down to a few hours or less. This would allow the research platform to answer complex queries on the archive by using compressed graph algorithms only, and without having to fallback on slower algorithms for artifacts that were not present at the time of compression.

Incremental graph compression would be a very challenging endeavor, as most of the components in the compression pipeline relies on a fixed number of nodes. For instance, one of the core steps of the process is the use of MPH functions, which rely on a static number of elements and cannot be dynamically updated. One way to address this problem could be to use *dynamic perfect hashing* algorithms [57, 49], although these have fairly different properties, including the non-contiguity of their output, which pose their own set of challenges.

Typed graph compression While the LLP algorithm (Section 9.3.4) for graph reordering achieves impressive compression ratios, we strongly suspect that even better results could be achieved by leveraging *typed* graph compression, i.e., compressing all the nodes and edges of each of the six node types in their own separate graph. This is because there is a higher chance that copy-lists used in the compressed BVGraph would be more similar between nodes of the same type, but this locality cannot be exploited with algorithms based solely on node proximity and clustering.

Graph querying Section 11.1 describes a simple traversal API for remote graph querying, and outlines its limitations by showing examples of queries that it cannot easily answer. It also presents graph query languages as a possible solution to limited expressiveness,

but these were not actually implemented as an interface to the compressed graph. Doing so could significantly improve the accessibility of the research platform for more complex experiments. In a comprehensive platform, this could then further be extended to include computation primitives on the objects themselves, down to the file contents. One could imagine writing a single query to get the average number of lines of codes of all the files in the archive in a path matching `fr_FR/*.po`. This would require a comprehensive integration of all the components of the research platform: graph structure querying, content fetching and processing, online distributed computing, etc. While this is a long term goal, this thesis provides the basic building blocks to achieve such a feat.

Scheduling predictions This thesis presents a work which leverages the Software Heritage archive to build an analysis platform open to all software mining researchers. Among them, one of the primary beneficiaries of this platform could be the Software Heritage initiative itself, as analyzing the data in the archive could yield valuable insights that could be used to fine-tune archiving behavior. One example of this is for scheduling predictions: the compressed graph augmented with commit timestamps can efficiently compute the frequency at which a given repository is updated; feeding that information back to the scheduler could help prioritizing the crawling of repositories that get updated more often, and reduce the crawling frequency of repositories that do not. This could alleviate the total load on the archive workers and increase its throughput.

Graph partitioning techniques In Section 12.5, we discuss how the high connectivity of the graph precludes the use of naive graph partitioning techniques to shard the graph into smaller isolated units, as most of the nodes in the graph are aggregated in a giant connected component that cannot easily be broken down. While this remains generally true, if sharding is needed it is still possible to try to regroup nodes using their locality characteristics to decide how to partition the graph. In Chapter 9 we use two approaches, BFS and LLP, to generate a node ordering which preserves locality. It would be interesting to partition the graph nodes by splitting these locality-preserving orders in chunks of equal size. These partitions could then be benchmarked for sharding efficiency to find out whether parallel graph querying workloads would require many synchronization points between the different shards, or just a few. This could be extended to other clustering techniques that were not investigated in this thesis, such as using modular decomposition [59, 98, 110] or Louvain community detection [24] to delimitate groups of nodes.

Derived graphs Another interesting research opportunity would be to make accessible data that can be derived from the graph for software mining research. One such example is the graph of collaboration in software development, where the nodes are unique persons and the edges represent whether two authors collaborated on a common project; this could

be generated using a linear traversal on the compressed graph, and would represent a real-world social network whose structure has, to our knowledge, not yet been extensively studied in the social network literature. Another example would be to compute the “diff graph” of development history, where instead of each revision pointing to the root directory of its source tree, it would instead point to the changes that happened since the last revision (i.e., new, deleted, modified and renamed files). This could help answer more research questions efficiently, such as finding the list of revisions in which a given file was added for the first time. Constructing this derived graph has been attempted at a reasonably large scale (10% of the edges in the graph) as part of an unpublished study with Wellenzohn et al., delivering promising early results.

14.4 Publications

This section lists the academic articles that I co-wrote in the context of this thesis, in chronological order of publication.

1. Antoine Pietri and Stefano Zacchiroli. “Towards Universal Software Evolution Analysis”. In: *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop, Delft, the Netherlands, December 10th - to - 11th, 2018*. Ed. by Georgios Gousios and Joseph Hejderup. Vol. 2361. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 6–10. URL: <http://ceur-ws.org/Vol-2361/short2.pdf>
2. Antoine Pietri et al. “The Software Heritage Graph Dataset: public software development under one roof”. In: *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. Ed. by Margaret-Anne D. Storey et al. IEEE, 2019, pp. 138–142. DOI: [10.1109/MSR.2019.00030](https://doi.org/10.1109/MSR.2019.00030)
3. Antoine Pietri et al. “The Software Heritage Graph Dataset: Large-scale Analysis of Public Software Development History”. In: *MSR ’20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim et al. ACM, 2020, pp. 1–5. DOI: [10.1145/3379597.3387510](https://doi.org/10.1145/3379597.3387510)
4. Paolo Boldi et al. “Ultra-Large-Scale Repository Analysis via Graph Compression”. In: *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. Ed. by Kostas Kontogiannis et al. IEEE, 2020, pp. 184–194. DOI: [10.1109/SANER48275.2020.9054827](https://doi.org/10.1109/SANER48275.2020.9054827)
5. Antoine Pietri et al. “Forking Without Clicking: on How to Identify Software Repository Forks”. In: *MSR ’20: 17th International Conference on Mining Software Reposito-*

- ries, Seoul, Republic of Korea, 29-30 June, 2020. Ed. by Sunghun Kim et al. ACM, 2020, pp. 277–287. DOI: [10.1145/3379597.3387450](https://doi.org/10.1145/3379597.3387450)
6. Antoine Pietri et al. “Determining the Intrinsic Structure of Public Software Development History”. In: *MSR ’20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim et al. OSF registration available online at: <https://osf.io/7r2w4>. ACM, 2020, pp. 602–605. DOI: [10.1145/3379597.3387506](https://doi.org/10.1145/3379597.3387506)
7. Thibault Allançon et al. “The Software Heritage Filesystem (SwhFS): Integrating Source Code Archival with Development”. In: *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 45–48. DOI: [10.1109/ICSE-Companion52605.2021.00032](https://doi.org/10.1109/ICSE-Companion52605.2021.00032)

14.5 Software

All the software I developed for this thesis is available in the Software Heritage forge,¹ and released under a free software license (GPLv3). In particular, the following software packages implement the various systems described in this thesis:

- `swh-vault` implements the Vault (see Chapter 6).
- `swh-fuse` implements SwhFS (see Chapter 7).
- `swh-dataset` implements the export pipeline for the graph dataset (see Chapter 8).
- `swh-graph` implements graph compression, manipulation of graph property data and an RPC API to query the compressed graph (see Chapters 9 to 11).

Additionally, the empirical studies described in Chapters 12 and 13 can be repeated using their respective replication packages.^{2,3} All plots shown in this thesis can be reproduced by running the Jupyter notebooks provided in its public repository,⁴ which also includes the raw experimental data in textual format.

¹<https://forge.softwareheritage.org/>

²<https://zenodo.org/record/3610708>

³<https://github.com/seirl/swh-graph-structure>

⁴<https://github.com/seirl/thesis>

Acronyms

- BFS** Breadth-First Search. 124, 128, 129, 161, 164, 228
- CDN** Content Delivery Network. 186
- CI** Continuous Integration. 61
- DAG** Directed Acyclic Graph. 43–45, 52–54, 56, 119, 183
- DFS** Depth-First Search. 161
- DVCS** Distributed Version Control System. 24, 28, 30, 44, 200, 203, 205, 215, 218, 220
- ESE** Empirical Software Engineering. 61
- HDD** Hard Disk Drive. 101
- IPC** Inter-process communication. 164
- LLP** Layered Label Propagation. 124, 128–130, 143, 227, 228
- MPH** Minimal Perfect Hash. 125, 143–145, 148, 151–153, 158, 227
- OLAP** Online Analytical Processing. 103, 106, 224
- OLTP** Online Transactional Processing. 103
- PII** Personally Identifiable Information. 96
- RDBMS** Relational Database Management System. 59, 102
- RDF** Resource Description Framework. 167
- SSD** Solid State Drive. 101
- SWHID** Software Heritage Persistent Identifier. 57, 58, 75–77, 103, 125, 142–146, 154, 158, 163, 165, 171
- UDF** User-Defined Function. 114
- VCS** Version Control System. 19, 24, 25, 31–33, 35–41, 43–45, 47, 50–52, 75, 76, 81, 94, 98, 124, 151, 177–179, 200–203, 205, 207, 212, 217–220, 227

Bibliography

- [1] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. “Mining Component Repositories for Installability Issues”. In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. Ed. by Massimiliano Di Penta, Martin Pinzger, and Romain Robbes. IEEE Computer Society, 2015, pp. 24–33. ISBN: 978-0-7695-5594-2. DOI: [10.1109/MSR.2015.10](https://doi.org/10.1109/MSR.2015.10). URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7180033>.
- [2] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. “Strong dependencies between software components”. In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 89–99.
- [3] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. “Building the Universal Archive of Source Code”. In: *Communications of the ACM* 61.10 (Sept. 2018), pp. 29–31. ISSN: 0001-0782. DOI: [10.1145/3183558](https://doi.org/10.1145/3183558).
- [4] Bram Adams, Eleni Constantinou, Tom Mens, and Gregorio Robles, eds. *Proceedings of the 1st International Workshop on Software Health, SoHeal@ICSE 2018, Gothenburg, Sweden, May 27, 2018*. ACM, 2018. ISBN: 978-1-4503-5730-2. DOI: [10.1145/3194124](https://doi.org/10.1145/3194124).
- [5] Réka Albert and Albert-László Barabási. “Statistical mechanics of complex networks”. In: *Reviews of modern physics* 74.1 (2002), p. 47.
- [6] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. “Redundancy-free analysis of multi-revision software artifacts”. In: *Empirical Software Engineering* 24.1 (2019), pp. 332–380. DOI: [10.1007/s10664-018-9630-9](https://doi.org/10.1007/s10664-018-9630-9).
- [7] Thibault Allançon, Antoine Pietri, and Stefano Zacchiroli. “The Software Heritage Filesystem (SwhFS): Integrating Source Code Archival with Development”. In: *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 45–48. DOI: [10.1109/ICSE-Companion52605.2021.00032](https://doi.org/10.1109/ICSE-Companion52605.2021.00032).
- [8] Amazon. *Amazon Athena*. 2019. URL: <https://aws.amazon.com/athena/>.

- [9] Renzo Angles. “The Property Graph Database Model”. In: *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*. Ed. by Dan Olteanu and Barbara Poblete. Vol. 2100. CEUR Workshop Proceedings. CEUR-WS.org, 2018. URL: <http://ceur-ws.org/Vol-2100/paper26.pdf>.
- [10] Renzo Angles et al. “G-CORE: A Core for Future Graph Query Languages”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1421–1432. DOI: [10.1145/3183713.3190654](https://doi.org/10.1145/3183713.3190654).
- [11] Apache. *Apache Parquet*. 2019. URL: <https://parquet.apache.org>.
- [12] Alberto Apostolico and Guido Drovandi. “Graph Compression by BFS”. In: *Algorithms* 2.3 (2009), pp. 1031–1044. ISSN: 1999-4893. DOI: [10.3390/a2031031](https://doi.org/10.3390/a2031031).
- [13] Michael Armbrust et al. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1383–1394. DOI: [10.1145/2723372.2742797](https://doi.org/10.1145/2723372.2742797).
- [14] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein. “BLAKE2: simpler, smaller, fast as MD5”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.
- [15] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. “Four degrees of separation”. In: *Proceedings of the 4th Annual ACM Web Science Conference*. 2012, pp. 33–42.
- [16] Albert-László Barabási and Eric Bonabeau. “Scale-free networks”. In: *Scientific american* 288.5 (2003), pp. 60–69.
- [17] Victor Basili, Lionel Briand, Steven Condon, Yong-Mi Kim, Walcélío L Melo, and Jon D Valen. “Understanding and predicting the process of software maintenance releases”. In: *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE. 1996, pp. 464–474.
- [18] Donald Beagle. “Conceptualizing an information commons”. In: *The Journal of Academic Librarianship* 25.2 (1999), pp. 82–89.
- [19] Laszlo A. Belady and Meir M Lehman. “A model of large program development”. In: *IBM Systems journal* 15.3 (1976), pp. 225–252.

-
- [20] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. “Theory and practice of monotone minimal perfect hashing”. In: *Journal of Experimental Algorithmics (JEA)* 16 (2008), pp. 3–1.
- [21] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. “Comparison and evaluation of clone detection tools”. In: *IEEE Transactions on software engineering* 33.9 (2007), pp. 577–591.
- [22] Marco Biazzini and Benoit Baudry. “May the fork be with you: novel metrics to analyze collaboration on github”. In: *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. ACM. 2014, pp. 37–43.
- [23] Twitter Blog. “Dremel made simple with parquet”. In: *dated Sep 11 (2013)*, p. 12. URL: https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.
- [24] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. “Fast unfolding of communities in large networks”. In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.
- [25] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. “BUbiNG: Massive Crawling for the Masses”. In: *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.
- [26] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. *Replication package: Ultra-large-scale Repository Analysis via Graph Compression*. Zenodo. 2019. doi: [10.5281/zenodo.3574459](https://doi.org/10.5281/zenodo.3574459).
- [27] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. “Ultra-Large-Scale Repository Analysis via Graph Compression”. In: *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. Ed. by Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou. IEEE, 2020, pp. 184–194. doi: [10.1109/SANER48275.2020.9054827](https://doi.org/10.1109/SANER48275.2020.9054827).
- [28] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. “Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks”. In: *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. Ed. by Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar. ACM, 2011, pp. 587–596. doi: [10.1145/1963405.1963488](https://doi.org/10.1145/1963405.1963488).

- [29] Paolo Boldi and Sebastiano Vigna. “The Webgraph Framework I: compression techniques”. In: *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. Ed. by Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills. Manhattan, USA: ACM, 2004, pp. 595–602. ISBN: 1-58113-844-X. DOI: [10.1145/988672.988752](https://doi.org/10.1145/988672.988752).
- [30] Paolo Boldi and Sebastiano Vigna. “The WebGraph Framework II: Codes For The World-Wide Web”. In: *2004 Data Compression Conference (DCC 2004), 23-25 March 2004, Snowbird, UT, USA*. IEEE Computer Society, 2004, p. 528. ISBN: 0-7695-2082-0. DOI: [10.1109/DCC.2004.1281504](https://doi.org/10.1109/DCC.2004.1281504). URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9014>.
- [31] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018. DOI: [10.2200/S00873ED1V01Y201808DTM051](https://doi.org/10.2200/S00873ED1V01Y201808DTM051).
- [32] Nieves R. Brisaboa, Ana Cerdeira-Pena, Guillermo de Bernardo, and Gonzalo Navarro. “Compressed representation of dynamic binary relations with applications”. In: *Information Systems* 69 (2017), pp. 106–123.
- [33] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. “Compact representation of Web graphs with extended functionality”. In: *Information Systems* 39.1 (2014), pp. 152–174. DOI: [10.1016/j.is.2013.08.003](https://doi.org/10.1016/j.is.2013.08.003).
- [34] Andrei Broder et al. “Graph structure in the web”. In: *Computer networks* 33.1-6 (2000), pp. 309–320.
- [35] Simon Brown. *The C4 model for Software Architecture*. Accessed 2020-11-16. 2018. URL: <https://c4model.com/>.
- [36] Matthieu Caneill, Daniel M. Germáin, and Stefano Zacchiroli. “The Debsources Dataset: Two Decades of Free and Open Source Software”. In: *Empirical Software Engineering* 22 (June 2017), pp. 1405–1437. ISSN: 1382-3256. DOI: [10.1007/s10664-016-9461-5](https://doi.org/10.1007/s10664-016-9461-5).
- [37] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. “On compressing social networks”. In: *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. Paris, France: ACM, 2009, pp. 219–228. ISBN: 978-1-60558-495-9. DOI: <http://doi.acm.org/10.1145/1557019.1557049>.
- [38] Christian Collberg and Todd A. Proebsting. “Repeatability in computer systems research”. In: *Communications of the ACM* 59.3 (Feb. 2016), pp. 62–69. DOI: [10.1145/2812803](https://doi.org/10.1145/2812803).
- [39] Yann Collet. *Zstandard*. 2015. URL: <https://facebook.github.io/zstd/>.

- [40] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. “Gitana: A software project inspector”. In: *Sci. Comput. Program.* 153 (2018), pp. 30–33. doi: [10.1016/j.scico.2017.12.002](https://doi.org/10.1016/j.scico.2017.12.002).
- [41] Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb. “Leveraging transparency”. In: *IEEE software* 30.1 (2012), pp. 37–43.
- [42] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. “Social coding in GitHub: transparency and collaboration in an open software repository”. In: *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM. 2012, pp. 1277–1286.
- [43] Nicolas Dandrimont. *[ceph-users] Ceph behavior on (lots of) small objects (RGW, RADOS + erasure coding)?* Ceph Users Mailing List. 2018. URL: <https://marc.info/?l=ceph-users%5C&m=153013955112932>.
- [44] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. “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. 2016, p. 2.
- [45] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. “Compressing Graphs and Indexes with Recursive Graph Bisection”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. New York, NY, USA: ACM, 2016, pp. 1535–1544. doi: [10.1145/2939672.2939862](https://doi.org/10.1145/2939672.2939862).
- [46] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. “Identifiers for Digital Objects: the Case of Software Source Code Preservation”. In: *Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, USA*. Sept. 2018. doi: [10.17605/OSF.IO/KDE56](https://doi.org/10.17605/OSF.IO/KDE56).
- [47] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. “Referencing Source Code Artifacts: a Separate Concern in Software Citation”. In: *Computing in Science and Engineering* 22.2 (Mar. 2020), pp. 33–43. issn: 1521-9615. doi: [10.1109/MCSE.2019.2963148](https://doi.org/10.1109/MCSE.2019.2963148).
- [48] Roberto Di Cosmo and Stefano Zacchiroli. “Software Heritage: Why and How to Preserve Software Source Code”. In: *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*. Sept. 2017. URL: <https://hal.archives-ouvertes.fr/hal-01590958/>.

- [49] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. “Dynamic perfect hashing: Upper and lower bounds”. In: *SIAM Journal on Computing* 23.4 (1994), pp. 738–761.
- [50] Robert Dyer, Hoan Anh Nguyen, Hridayesh Rajan, and Tien N Nguyen. “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 422–431.
- [51] Peter Elias. “Efficient Storage and Retrieval by Content and Address of Static Files”. In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260.
- [52] Arash Farzan and J. Ian Munro. “Succinct encoding of arbitrary graphs”. In: *Theoretical Computer Science* 513.Supplement C (2013), pp. 38–52. ISSN: 0304-3975. DOI: <https://doi.org/10.1016/j.tcs.2013.09.031>.
- [53] Nicola Ferro and Diane Kelly. “SIGIR initiative to implement ACM artifact review and badging”. In: *ACM SIGIR Forum*. Vol. 52. ACM New York, NY, USA. 2018, pp. 4–10.
- [54] PUB FIPS. “180-2: Secure hash standard (SHS)”. In: *US Department of Commerce, National Institute of Standards and Technology (NIST)* (2012).
- [55] Karl Fogel. *Producing open source software: How to run a successful free software project*. O’Reilly Media, Inc., 2005.
- [56] Nadime Francis et al. “Cypher: An Evolving Query Language for Property Graphs”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1433–1445. DOI: [10.1145/3183713.3190657](https://doi.org/10.1145/3183713.3190657).
- [57] Michael L Fredman, János Komlós, and Endre Szemerédi. “Storing a sparse table with $O(1)$ worst case access time”. In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 538–544.
- [58] Michael Freeman. *Programming Skills for Data Science: Start Writing Code to Wrangle, Analyze, and Visualize Data with R*. Boston: Addison-Wesley, 2019. ISBN: 0135133106.
- [59] Tibor Gallai. “Transitiv orientierbare graphen”. In: *Acta Mathematica Hungarica* 18.1-2 (1967), pp. 25–66.
- [60] Yongqin Gao, Matthew VanAntwerp, Scott Christley, and Greg Madey. “A research collaboratory for open source software research.” In: *2007 Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS’07*. IEEE, 2007.

- [61] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. “Fast Scalable Construction of (Minimal Perfect Hash) Functions”. In: *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*. Ed. by Andrew V. Goldberg and Alexander S. Kulikov. Springer, 2016, pp. 339–352.
- [62] Daniel M German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. “Code siblings: Technical and legal implications of copying code between applications”. In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE. 2009, pp. 81–90.
- [63] Sanjay Ghemawat and Jeff Dean. *LevelDB*. 2011.
- [64] Michael W Godfrey, Daniel M German, Julius Davies, and Abram Hindle. “Determining the provenance of software artifacts”. In: *Proceedings of the 5th International Workshop on Software Clones*. 2011, pp. 65–66.
- [65] Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M German. “Macro-level software evolution: a case study of a large software compilation”. In: *Empirical Software Engineering* 14.3 (2009), pp. 262–285.
- [66] Georgios Gousios. *The Issue 32 incident – An update*. 2016. URL: <https://gousios.gr/blog/Issue-thirty-two.html>.
- [67] Georgios Gousios, Martin Pinzger, and Arie van Deursen. “An exploratory study of the pull-based software development model”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 345–355.
- [68] Georgios Gousios and Diomidis Spinellis. “GHTorrent: Github’s data from a firehose”. In: *9th IEEE Working Conference of Mining Software Repositories, MSR*. Ed. by Michele Lanza, Massimiliano Di Penta, and Tao Xie. IEEE Computer Society, 2012, pp. 12–21. ISBN: 978-1-4673-1761-0. DOI: 10.1109/MSR.2012.6224294. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6220358>.
- [69] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. “Work practices and challenges in pull-based development: The integrator’s perspective”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 358–368.
- [70] Reilly Grant. *Filesystem interface for the git version control system-final report*. Tech. rep. <https://www.stwing.upenn.edu/~rm/figfs/final.pdf>, accessed 2020-11-20. University of Pennsylvania, 2009.

- [71] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. “Toward Large-Scale Vulnerability Discovery Using Machine Learning”. In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY '16. New Orleans, Louisiana, USA: ACM, 2016, pp. 85–96. ISBN: 978-1-4503-3935-3. DOI: [10.1145/2857705.2857720](https://doi.org/10.1145/2857705.2857720).
- [72] Torben Hagerup. “Fast breadth-first search in still less space”. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 2019, pp. 93–105.
- [73] Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi, eds. *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*. Vol. 378. IFIP Advances in Information and Communication Technology. Springer, 2012. ISBN: 978-3-642-33441-2. DOI: [10.1007/978-3-642-33442-9](https://doi.org/10.1007/978-3-642-33442-9).
- [74] Ahmed E Hassan. “Mining software repositories to assist developers and support managers”. In: *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 339–342.
- [75] Ahmed E Hassan. “The road ahead for mining software repositories”. In: *Frontiers of Software Maintenance, 2008. FoSM 2008*. IEEE, 2008, pp. 48–57.
- [76] Ahmed E. Hassan and Richard C. Holt. “The Small World of Software Reverse Engineering”. In: *11th Working Conference on Reverse Engineering, WCRE 2004*. IEEE Computer Society, 2004, pp. 278–283. DOI: [10.1109/WCRE.2004.37](https://doi.org/10.1109/WCRE.2004.37).
- [77] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesús M. González-Barahona. “The evolution of the laws of software evolution: A discussion based on a systematic literature review”. In: *ACM Computing Surveys (CSUR)* 46.2 (2013), p. 28.
- [78] Felipe Hoffa. *GitHub on BigQuery: Analyze all the open source code*. 2016. URL: <https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code>.
- [79] James Howison, Megan Conklin, and Kevin Crowston. “FLOSSmole: A collaborative repository for FLOSS research data and analyses”. In: *International Journal of Information Technology and Web Engineering (IJITWE)* 1.3 (2006), pp. 17–26.
- [80] Yin Huai et al. “Major technical advancements in apache hive”. In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu. ACM, 2014, pp. 1235–1246. DOI: [10.1145/2588555.2595630](https://doi.org/10.1145/2588555.2595630).

-
- [81] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet Singh Kochhar, and Li Zhang. “Why and how developers fork what from whom in GitHub”. In: *Empirical Software Engineering* 22.1 (2017), pp. 547–578.
- [82] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. “A survey and taxonomy of approaches for mining software repositories in the context of software evolution”. In: *Journal of software maintenance and evolution: Research and practice* 19.2 (2007), pp. 77–131.
- [83] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. “The promises and perils of mining GitHub”. In: *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.
- [84] Chris F. Kemerer and Sandra Slaughter. “An empirical approach to studying software evolution”. In: *IEEE transactions on software engineering* 25.4 (1999), pp. 493–509.
- [85] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. “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*. Ed. by Jesús M. González-Barahona, Abram Hindle, and Lin Tan. IEEE Computer Society, 2017, pp. 102–112. ISBN: 978-1-5386-1544-7. DOI: [10.1109/MSR.2017.55](https://doi.org/10.1109/MSR.2017.55). URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7959735>.
- [86] Nancy Kranich and Jorge Reina Schement. “Information Commons”. In: *Annual Review of Information Science and Technology* 42.1 (2008), pp. 546–591.
- [87] Shriram Krishnamurthi and Jan Vitek. “The Real Software Crisis: Repeatability As a Core Value”. In: *Communications of the ACM* 58.3 (Feb. 2015), pp. 34–36. DOI: [10.1145/2658987](https://doi.org/10.1145/2658987).
- [88] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. “What is Twitter, a social network or a news media?” In: *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*. Ed. by Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti. ACM, 2010, pp. 591–600. DOI: [10.1145/1772690.1772751](https://doi.org/10.1145/1772690.1772751).
- [89] Nathan LaBelle and Eugene Wallingford. “Inter-package dependency networks in open-source software”. In: *arXiv preprint cs/0411096* (2004). URL: <https://arxiv.org/abs/cs/0411096>.

- [90] Frank Li and Vern Paxson. “A Large-Scale Empirical Study of Security Patches”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, 2017, pp. 2201–2215. ISBN: 978-1-4503-4946-8. doi: [10.1145/3133956.3134072](https://doi.org/10.1145/3133956.3134072).
- [91] Antonio Lima, Luca Rossi, and Mirco Musolesi. “Coding together at scale: GitHub as a collaborative social network”. In: *Eighth International AAAI Conference on Weblogs and Social Media*. 2014.
- [92] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. “World of code: an infrastructure for mining the universe of open source VCS data”. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press. 2019, pp. 143–154.
- [93] Yuxing Ma et al. “World of code: Enabling a research workflow for mining and analyzing the universe of open source vcs data”. In: *Empirical Software Engineering* 26.2 (2021), pp. 1–42.
- [94] Damiano Di Francesco Maesa, Andrea Marino, and Laura Ricci. “Data-driven analysis of Bitcoin properties: exploiting the users graph”. In: *International Journal of Data Science and Analytics* 6.1 (2018), pp. 63–80.
- [95] Thomas Maillart, Didier Sornette, Sebastian Spaeth, and Georg von Krogh. “Empirical tests of Zipf’s law mechanism in open source Linux distribution”. In: *Physical Review Letters* 101.21 (2008), p. 218701.
- [96] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. “Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset”. In: *Empirical Software Engineering* 22.4 (2017), pp. 1936–1964. doi: [10.1007/s10664-016-9470-4](https://doi.org/10.1007/s10664-016-9470-4).
- [97] Matias Martinez and Martin Monperrus. “Mining software repair models for reasoning on the search space of automated program fixing”. In: *Empirical Software Engineering* 20.1 (2015), pp. 176–205. doi: [10.1007/s10664-013-9282-8](https://doi.org/10.1007/s10664-013-9282-8).
- [98] Ross M McConnell and Fabien De Montgolfier. “Linear-time modular decomposition of directed graphs”. In: *Discrete Applied Mathematics* 145.2 (2005), pp. 198–209.
- [99] Tom Mens. “Introduction and roadmap: History and challenges of software evolution”. In: *Software evolution*. Springer, 2008, pp. 1–11.
- [100] Ralph C. Merkle. “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*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Com-

- puter Science. Springer, 1987, pp. 369–378. ISBN: 3-540-18796-0. DOI: [10.1007/3-540-48184-2_32](https://doi.org/10.1007/3-540-48184-2_32).
- [101] Robert Meusel, Sebastiano Vigna, Oliver Lehmborg, and Christian Bizer. “The graph structure in the web—analyzed on different aggregation levels”. In: *The Journal of Web Science* 1 (2015).
- [102] Jan Michels, Keith Hare, and Jim Melton. *Standardizing Graph Database Functionality*. 2017. URL: <https://www.gqlstandards.org/>.
- [103] Microsoft. *Azure Databricks*. 2019. URL: <https://azure.microsoft.com/en-in/services/databricks/>.
- [104] Microsoft. *Scalar*. Accessed 2020-09-29. URL: <https://github.com/microsoft/scalar>.
- [105] Microsoft. *VFS for Git*. Accessed 2020-09-29. URL: <https://vfsforgit.org/>.
- [106] Audris Mockus. “Amassing and indexing a large sample of version control systems: Towards the census of public source code history”. In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE. 2009, pp. 11–20.
- [107] Audris Mockus, Diomidis Spinellis, Zoe Kotti, and Gabriel John Dusing. “A Complete Set of Related Git Repositories Identified via Community Detection Approaches Based on Shared Commits”. In: *ArXiv preprint abs/2002.02707* (2020). URL: <https://arxiv.org/abs/2002.02707>.
- [108] Christopher R Myers. “Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs”. In: *Physical Review E* 68.4 (2003), p. 046116.
- [109] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. “Information network or social network? The structure of the Twitter follow graph”. In: *Proceedings of the 23rd International Conference on World Wide Web*. 2014, pp. 493–498.
- [110] Chemseddine Nabti and Hamida Seba. “Querying massive graph data: A compress and search approach”. In: *Future Generation Computer Systems* 74 (2017), pp. 63–75. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2017.04.005>.
- [111] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. “Mining metrics to predict component failures”. In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 452–461.
- [112] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

- [113] Linus Nyman. “Hackers on Forking”. In: *Proceedings of The International Symposium on Open Collaboration, OpenSym 2014, Berlin, Germany, August 27 - 29, 2014*. Ed. by Dirk Riehle et al. ACM, 2014, 6:1–6:10. ISBN: 978-1-4503-3016-9. DOI: [10.1145/2641580.2641590](https://doi.org/10.1145/2641580.2641590). URL: <http://dl.acm.org/citation.cfm?id=2641580>.
- [114] Linus Nyman and Mikael Laakso. “Notes on the History of Fork and Join”. In: *IEEE Annals of the History of Computing* 38.3 (2016), pp. 84–87. DOI: [10.1109/MAHC.2016.34](https://doi.org/10.1109/MAHC.2016.34).
- [115] Linus Nyman and Tommi Mikkonen. “To Fork or Not to Fork: Fork Motivations in SourceForge Projects”. In: *IJOSSP* 3.3 (2011), pp. 1–9. DOI: [10.4018/jossp.2011070101](https://doi.org/10.4018/jossp.2011070101).
- [116] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. “Perspectives on Code Forking and Sustainability in Open Source Software”. In: *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*. Ed. by Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi. Vol. 378. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 274–279. ISBN: 978-3-642-33441-2. DOI: [10.1007/978-3-642-33442-9_21](https://doi.org/10.1007/978-3-642-33442-9_21).
- [117] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
- [118] Giuseppe Ottaviano and Rossano Venturini. “Partitioned Elias–Fano Indexes”. In: *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’14, Gold Coast , QLD, Australia - July 06 - 11, 2014*. Ed. by Shlomo Geva, Andrew Trotman, Peter Bruza, Charles L. A. Clarke, and Kalervo Järvelin. SIGIR ’14. New York, NY, USA: ACM, 2014, pp. 273–282. ISBN: 978-1-4503-2257-7. DOI: [10.1145/2600428.2609615](https://doi.org/10.1145/2600428.2609615).
- [119] Mike Owens. *The definitive guide to SQLite*. Apress, 2006.
- [120] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. “A study of external community contribution to open-source projects on GitHub”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 332–335.
- [121] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. “Semantics and complexity of SPARQL”. In: *ACM Transactions on Database Systems (TODS)* 34.3 (2009), pp. 1–45.
- [122] Giulio Ermanno Pibiri and Rossano Venturini. “Dynamic Elias-Fano Representation”. In: *28th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 2017, 30:1–30:14. DOI: [10.4230/LIPIcs.CPM.2017.30](https://doi.org/10.4230/LIPIcs.CPM.2017.30).

- [123] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. “Determining the Intrinsic Structure of Public Software Development History”. In: *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup. OSF registration available online at: <https://osf.io/7r2w4>. ACM, 2020, pp. 602–605. DOI: [10.1145/3379597.3387506](https://doi.org/10.1145/3379597.3387506).
- [124] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. “Forking Without Clicking: on How to Identify Software Repository Forks”. In: *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup. ACM, 2020, pp. 277–287. DOI: [10.1145/3379597.3387450](https://doi.org/10.1145/3379597.3387450).
- [125] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. “The Software Heritage Graph Dataset: Large-scale Analysis of Public Software Development History”. In: *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup. ACM, 2020, pp. 1–5. DOI: [10.1145/3379597.3387510](https://doi.org/10.1145/3379597.3387510).
- [126] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. “The Software Heritage Graph Dataset: public software development under one roof”. In: *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. Ed. by Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc. IEEE, 2019, pp. 138–142. DOI: [10.1109/MSR.2019.00030](https://doi.org/10.1109/MSR.2019.00030).
- [127] Antoine Pietri and Stefano Zacchiroli. “Towards Universal Software Evolution Analysis”. In: *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop, Delft, the Netherlands, December 10th - to - 11th, 2018*. Ed. by Georgios Gousios and Joseph Hejderup. Vol. 2361. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 6–10. URL: <http://ceur-ws.org/Vol-2361/short2.pdf>.
- [128] Presslabs. *gitfs: Version controlled file system*. Accessed 2020-11-17. 2014. URL: <https://github.com/presslabs/gitfs>.
- [129] Keith Randall, Raymie Stata, Rajiv Wickremesinghe, and Janet L. Wiener. *The LINK database: Fast access to graphs of the Web*. Research Report 175. Compaq Systems Research Center, Palo Alto, CA, 2001.
- [130] Ayushi Rastogi and Nachiappan Nagappan. “Forking and the Sustainability of the Developer Community Participation—An Empirical Investigation on Outcomes and Reasons”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 102–111.
- [131] Nikolaus Rath. *Filesystem in Userspace (FUSE)*. Accessed 2020-09-29. URL: <https://github.com/libfuse/libfuse>.

- [132] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. “Software clone detection: A systematic review”. In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199.
- [133] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. “A large scale study of programming languages and code quality in github”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 155–165.
- [134] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. “PGQL: a property graph query language”. In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 2016, pp. 1–6.
- [135] Gregorio Robles and Jesús M. González-Barahona. “A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes”. In: *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*. Ed. by Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi. Vol. 378. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 1–14. ISBN: 978-3-642-33441-2. DOI: [10.1007/978-3-642-33442-9_1](https://doi.org/10.1007/978-3-642-33442-9_1).
- [136] Marko A Rodriguez. “The gremlin graph traversal machine and language (invited talk)”. In: *Proceedings of the 15th Symposium on Database Programming Languages*. 2015, pp. 1–10.
- [137] Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. “Software Provenance Tracking at the Scale of Public Source Code”. In: *Empirical Software Engineering* (2020). to appear. ISSN: 1382-3256.
- [138] Chanchal Kumar Roy and James R Cordy. *A survey on software clone detection research*. Tech. rep. 115. Queen’s School of Computing, 2007, pp. 64–68.
- [139] Vitalis Salis and Diomidis Spinellis. “RepoFS: File system view of Git repositories”. In: *SoftwareX* 9 (2019), pp. 288–292.
- [140] Thomas Schank and Dorothea Wagner. “Approximating clustering coefficient and transitivity.” In: *Journal of Graph Algorithms and Applications* 9.2 (2005), pp. 265–275.
- [141] Jonatan Schroeder. “GitOD: An on demand distributed file system approach to Version Control”. In: *CTS 2012: International Conference on Collaboration Technologies and Systems*. IEEE, 2012, pp. 613–615. DOI: [10.1109/CTS.2012.6261115](https://doi.org/10.1109/CTS.2012.6261115).

- [142] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. “CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization”. In: *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*. 2017, pp. 654–659. doi: [10.1109/APSEC.2017.80](https://doi.org/10.1109/APSEC.2017.80).
- [143] Raghav Sethi et al. “Presto: SQL on everything”. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 1802–1813.
- [144] Param Vir Singh. “The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success”. In: *ACM Trans. Softw. Eng. Methodol.* 20.2 (2010), 6:1–6:27. doi: [10.1145/1824760.1824763](https://doi.org/10.1145/1824760.1824763).
- [145] Peter Snyder. “tmpfs: A virtual memory file system”. In: *Proceedings of the autumn 1990 EUUG Conference*. 1990, pp. 241–248.
- [146] source{d}. *gitbase: a SQL database interface to Git repositories*. Accessed 2020-11-17. 2016. URL: <https://github.com/src-d/gitbase>.
- [147] Diomidis Spinellis. “A repository of Unix history and evolution”. In: *Empirical Software Engineering* 22.3 (2017), pp. 1372–1404. doi: [10.1007/s10664-016-9445-5](https://doi.org/10.1007/s10664-016-9445-5).
- [148] Diomidis Spinellis. “Version control systems”. In: *IEEE Software* 22.5 (2005), pp. 108–109.
- [149] Megan Squire. “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*. Ed. by Lorraine Morgan. ACM, 2017, 15:1–15:8. ISBN: 978-1-4503-5187-4. doi: [10.1145/3125433.3125468](https://doi.org/10.1145/3125433.3125468). URL: <http://dl.acm.org/citation.cfm?id=3125433>.
- [150] Megan Squire and David Williams. “Describing the software forge ecosystem”. In: *System Science (HICSS), 2012 45th Hawaii International Conference on*. IEEE. 2012, pp. 3416–3425.
- [151] Stefan Stanciulescu, Sandro Schulze, and Andrzej Wasowski. “Forked and integrated variants in an open-source firmware project”. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 151–160.
- [152] Secure Hash Standard. “FIPS Pub 180-1”. In: *National Institute of Standards and Technology* 17 (1995), p. 15.
- [153] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. “The first collision for full SHA-1”. In: *Annual International Cryptology Conference*. Springer. 2017, pp. 570–596.

- [154] Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, eds. *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. IEEE, 2019. ISBN: 978-1-7281-3412-3. URL: <https://dl.acm.org/citation.cfm?id=3341883>.
- [155] Jeffrey Svajlenko and Chanchal Kumar Roy. “Fast and flexible large-scale clone detection with CloneWorks”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 2017, pp. 27–30. DOI: [10.1109/ICSE-C.2017.3](https://doi.org/10.1109/ICSE-C.2017.3).
- [156] M. M. Mahbubul Syeed, Imed Hammouda, and Tarja Systä. “Evolution of Open Source Software Projects: A Systematic Literature Review”. In: *JSW* 8.11 (2013), pp. 2815–2829.
- [157] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. “An empirical study on the maintenance of source code clones”. In: *Empirical Software Engineering* 15.1 (2010), pp. 1–34. DOI: [10.1007/s10664-009-9108-x](https://doi.org/10.1007/s10664-009-9108-x).
- [158] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. “Network structure of social coding in github”. In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 323–326.
- [159] Milo Z Trujillo, Laurent Hébert-Dufresne, and James P Bagrow. “The penumbra of open source: projects outside of centralized platforms are longer maintained, more academic and more collaborative”. In: *ArXiv preprint abs/2106.15611* (2021). URL: <https://arxiv.org/abs/2106.15611>.
- [160] Jason Tsay, Laura Dabbish, and James Herbsleb. “Influence of social and technical factors for evaluating contribution in GitHub”. In: *Proceedings of the 36th international conference on Software engineering*. ACM. 2014, pp. 356–366.
- [161] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. “The anatomy of the facebook social graph”. In: *ArXiv preprint abs/1111.4503* (2011). URL: <https://arxiv.org/abs/1111.4503>.
- [162] Sergi Valverde and Ricard V Solé. “Hierarchical small worlds in software architecture”. In: *arXiv preprint cond-mat/0307278* (2003).
- [163] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. “To FUSE or Not to FUSE: Performance of User-Space File Systems”. In: *15th USENIX Conference on File and Storage Technologies, FAST 2017*. USENIX Association, 2017, pp. 59–72. URL: <https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor>.

- [164] Enrique Larios Vargas, Joseph Hejderup, Maria Kechagia, Magiel Bruntink, and Georgios Gousios. “Enabling real-time feedback in software engineering”. In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Andrea Zisman and Sven Apel. ACM, 2018, pp. 21–24. ISBN: 978-1-4503-5662-6. DOI: [10.1145/3183399.3183416](https://doi.org/10.1145/3183399.3183416).
- [165] C. Vendome. “A Large Scale Study of License Usage on GitHub”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. May 2015, pp. 772–774. DOI: [10.1109/ICSE.2015.245](https://doi.org/10.1109/ICSE.2015.245).
- [166] Duncan J Watts and Steven H Strogatz. “Collective dynamics of ‘small-world’ networks”. In: *Nature* 393.6684 (1998), pp. 440–442.
- [167] Lian Wen, Diana Kirk, and R. Geoff Dromey. “Software Systems as Complex Networks”. In: *Proceedings of the Six IEEE International Conference on Cognitive Informatics, ICCI 2007, August 6-8, Lake Tahoe, CA, USA*. Ed. by Du Zhang, Yingxu Wang, and Witold Kinsner. IEEE Computer Society, 2007, pp. 106–115. DOI: [10.1109/COGINF.2007.4341879](https://doi.org/10.1109/COGINF.2007.4341879).
- [168] Igor Scaliante Wiese, José Teodoro Da Silva, Igor Steinmacher, Christoph Treude, and Marco Aurélio Gerosa. “Who is who in the mailing list? Comparing six disambiguation heuristics to identify multiple addresses of a participant”. In: *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2016, pp. 345–355.
- [169] Chadd C Williams and Jeffrey K Hollingsworth. “Automatic mining of source code repositories to improve bug finding techniques”. In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 466–480.
- [170] Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. Germán, and Katsuro Inoue. “Analysis of license inconsistency in large collections of open source projects”. In: *Empirical Software Engineering* 22.3 (2017), pp. 1194–1222. DOI: [10.1007/s10664-016-9487-8](https://doi.org/10.1007/s10664-016-9487-8).
- [171] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. “Wait for it: determinants of pull request evaluation latency on GitHub”. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 367–371.
- [172] Stefano Zacchiroli. “Gender Differences in Public Code Contributions: a 50-year Perspective”. In: *IEEE Software* 38.2 (2021), pp. 45–50. ISSN: 0740-7459. DOI: [10.1109/MS.2020.3038765](https://doi.org/10.1109/MS.2020.3038765).

BIBLIOGRAPHY

- [173] Matei Zaharia et al. “Apache Spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.
- [174] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. “What the fork: a study of inefficient and efficient forking practices in social coding”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2019, pp. 350–361.
- [175] Jiaxin Zhu and Jun Wei. “An empirical study of multiple names and email addresses in oss version control repositories”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 409–420.
- [176] T. Zimmermann, R. Premraj, and A. Zeller. “Predicting Defects for Eclipse”. In: *Predictor Models in Software Engineering, 2007. PROMISE’07: ICSE Workshops 2007. International Workshop on*. May 2007, pp. 9–9. DOI: [10.1109/PROMISE.2007.10](https://doi.org/10.1109/PROMISE.2007.10).
- [177] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. “Mining Version Histories to Guide Software Changes”. In: *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*. Ed. by Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum. IEEE Computer Society, 2004, pp. 563–572. ISBN: 0-7695-2163-0. DOI: [10.1109/ICSE.2004.1317478](https://doi.org/10.1109/ICSE.2004.1317478). URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9201>.

Copyright © 2021 Antoine Pietri

This work is licensed under the Creative Commons Attribution 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.