Querying Public Software Development at Scale

Thibault Allançon, Antoine Pietri, Stefano Zacchiroli

University Paris Diderot and Inria, Paris

9 September 2019
IRIF, University Paris Diderot
Paris, France

Context — Software Heritage

Software Heritage

THE GREAT LIBRARY OF SOURCE CODE

Our mission

Collect, preserve, and share the source code of all (publicly available) software, together with its full development history.

Archiving goals

Targets: Version Control System (VCS) repositories & source code distribution platforms (e.g., package repositories).

We DO archive

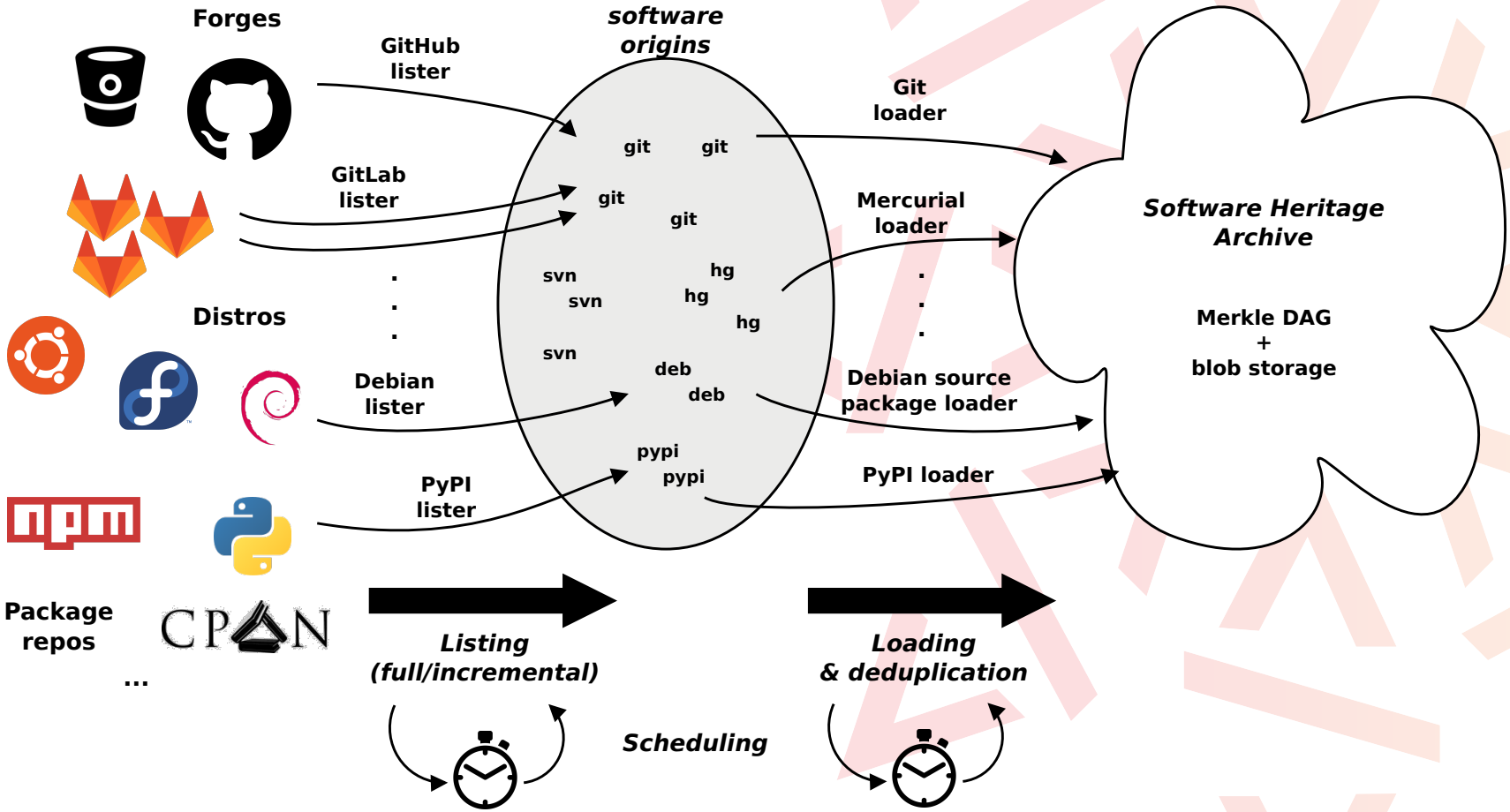
- file content (= blobs)
- revisions (= commits), with full metadata
- releases (= tags), ditto
- where (origin) & when (visit) we found any of the above

. . . in a VCS-/archive-agnostic canonical data model

We DON'T archive

- homepages, wikis
- BTS/issues/code reviews/etc.
- mailing lists

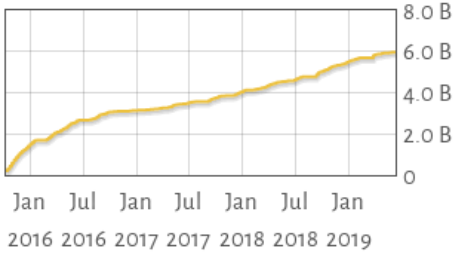
Data flow


Forges
GitHub lister
GitLab lister
Distros
Debian lister
PyPI lister
Package repos
CPAN
...
software origins
git
git
git
git
svn
svn
svn
hg
hg
hg
deb
deb
pypi
pypi
Git loader
Mercurial loader
Debian source package loader
PyPI loader
Software Heritage Archive
Merkle DAG
+
blob storage
Listing (full/incremental)
Scheduling
Loading & deduplication

Archive coverage — archive.softwareheritage.org

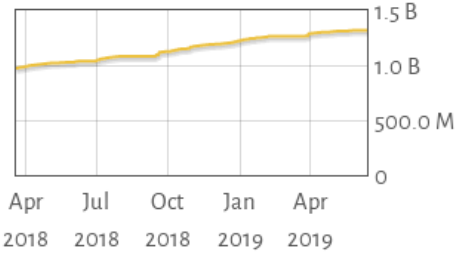
Source files

6,006,503,960



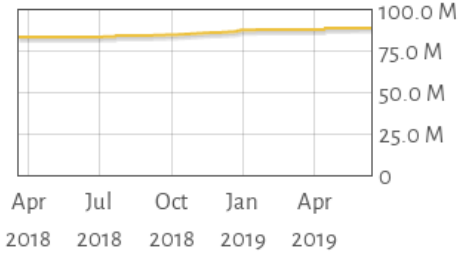
Commits

1,326,776,432



Projects

89,301,694

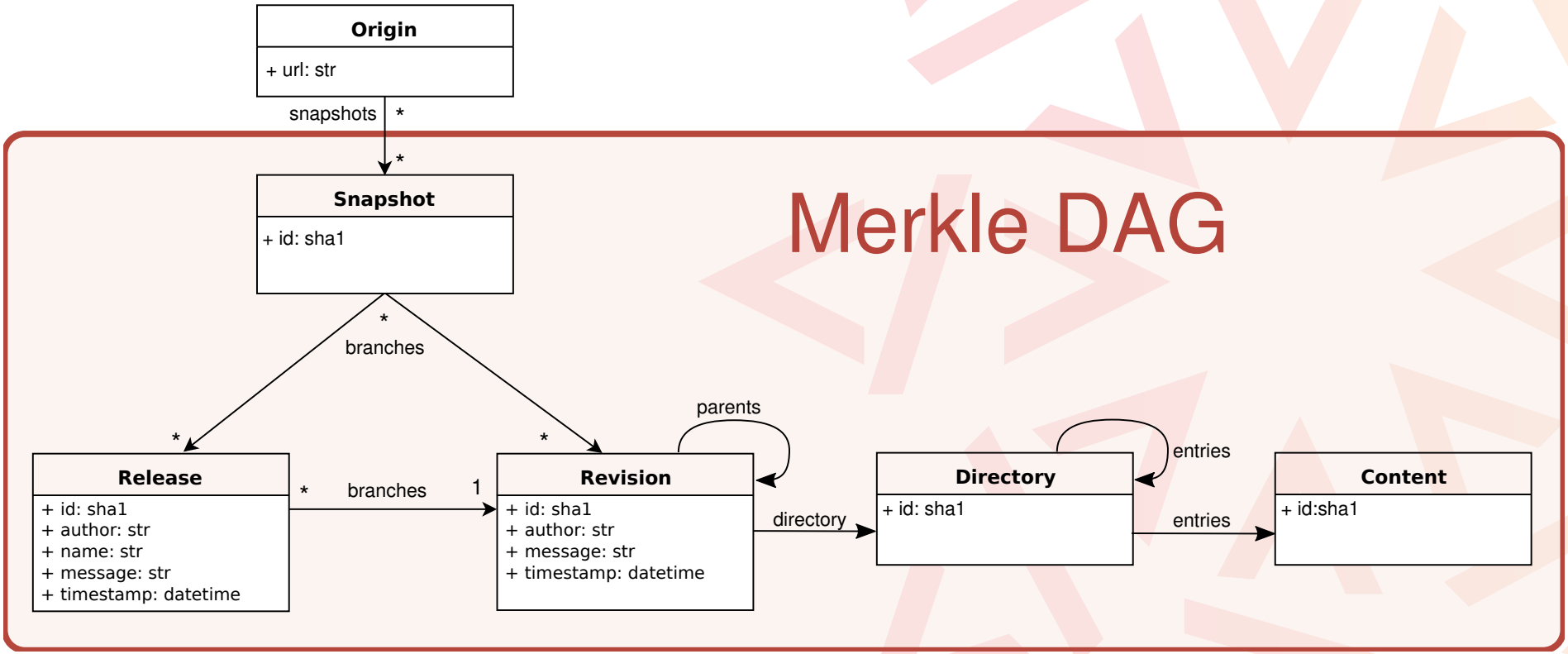


GitHub debian GitLab npm Google code {P}

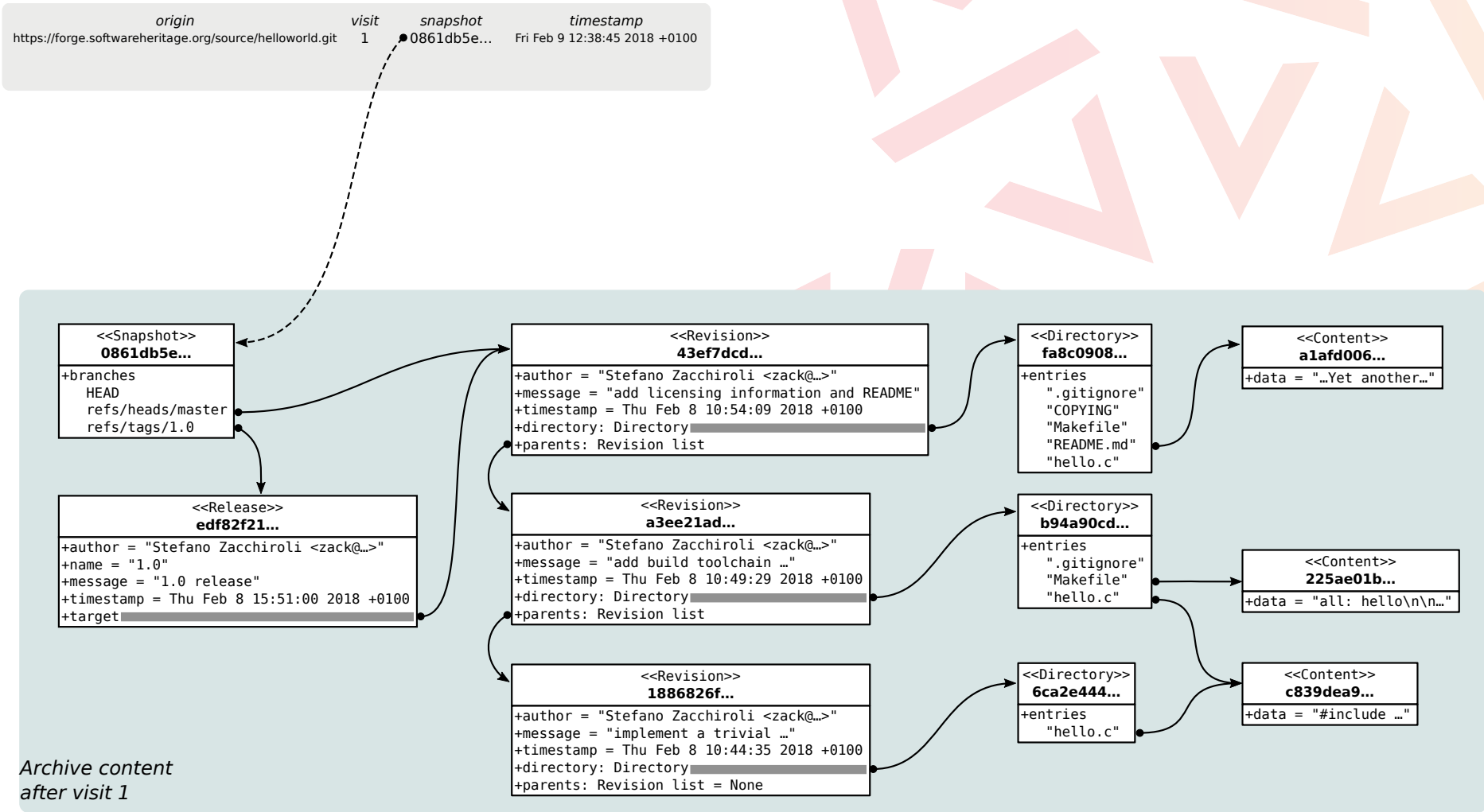
GITORIOUS GNU HAL archives-ouvertes.fr Inria inventeurs du monde numérique python Package Index

- ~600 TB of (uncompressed) source code
- The *richest* public source code archive, . . . and growing daily!

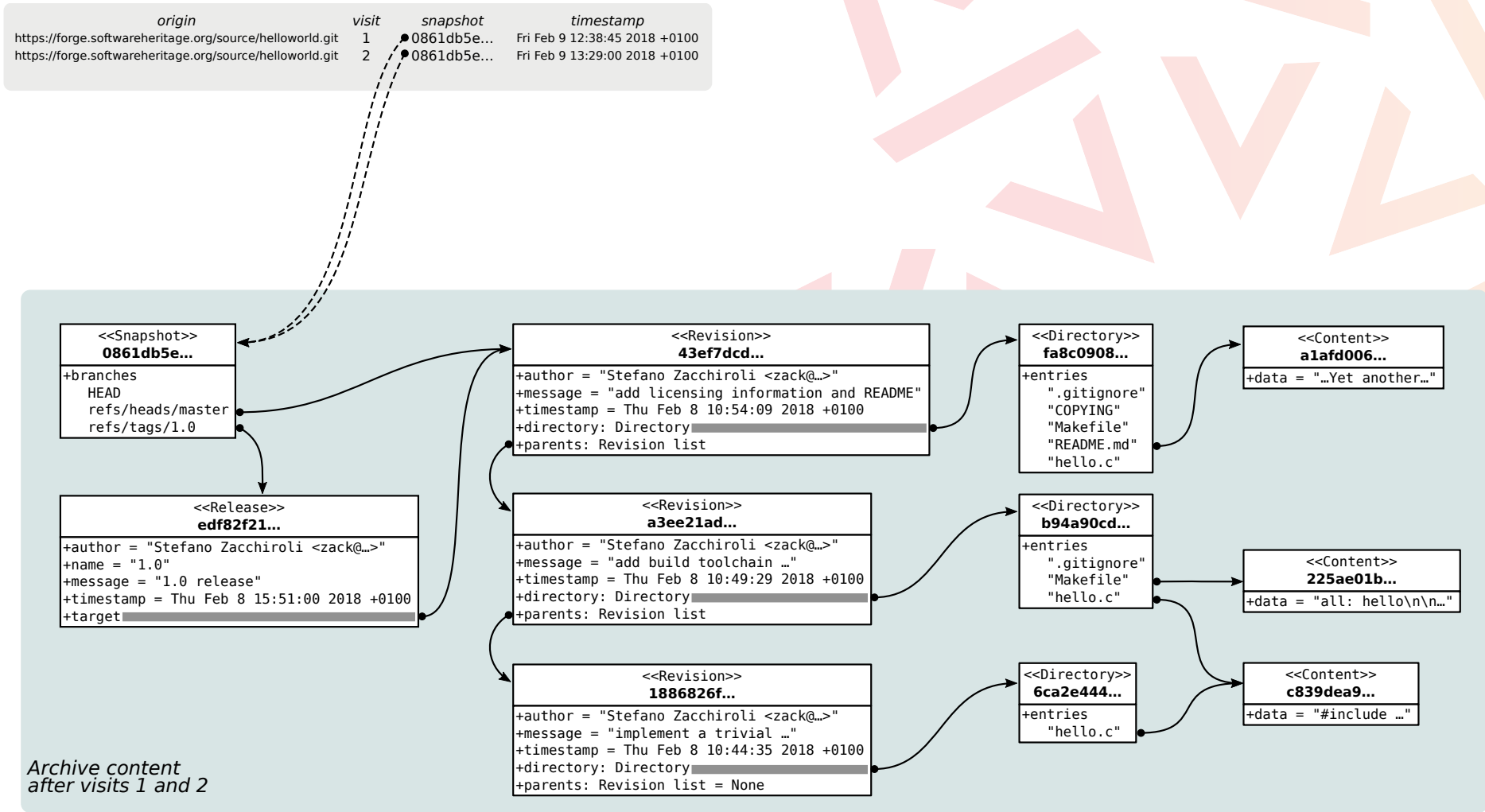
The archive: a (giant) Merkle DAG



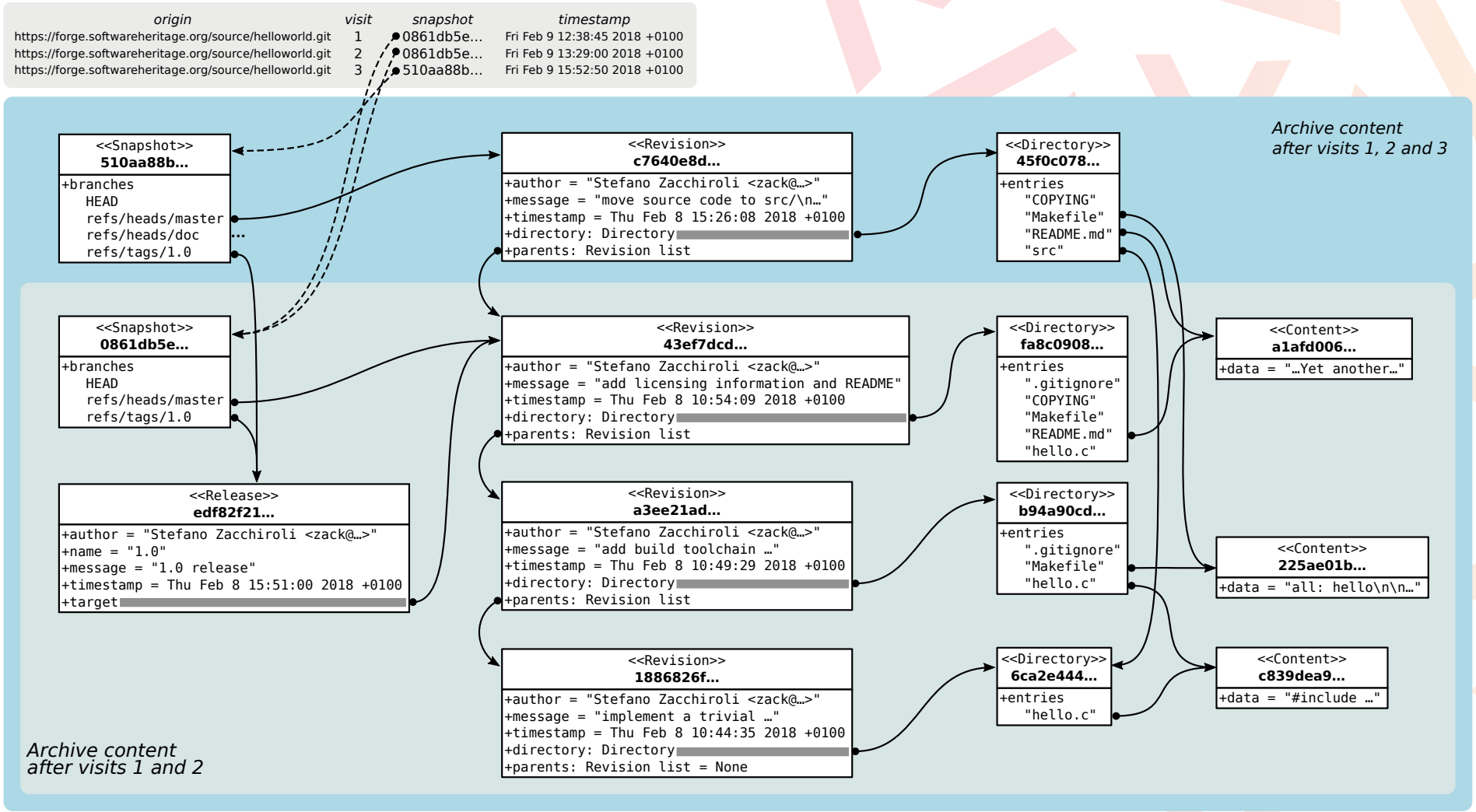
The archive: a (giant) Merkle DAG



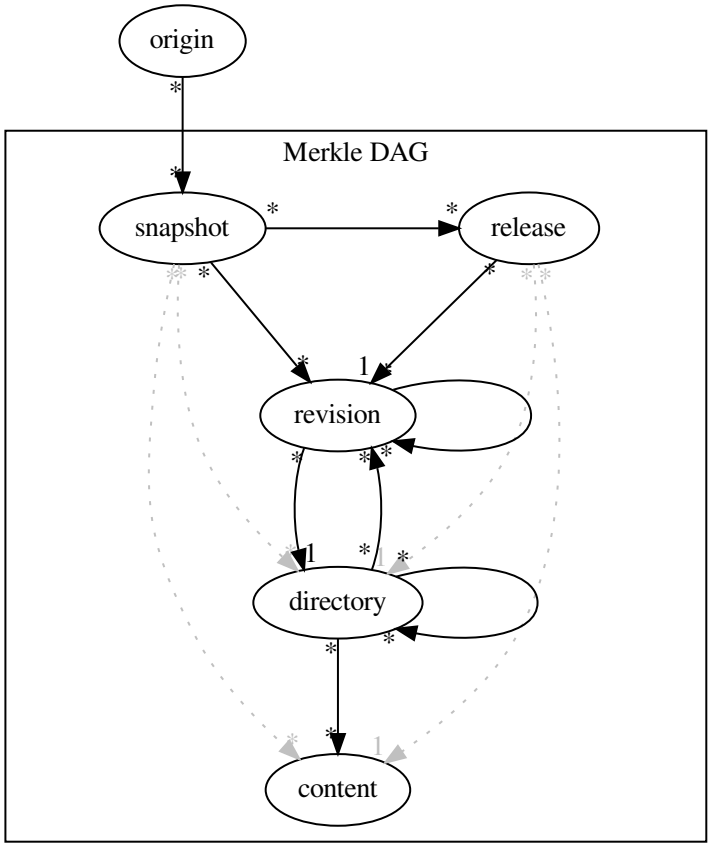
The archive: a (giant) Merkle DAG



The archive: a (giant) Merkle DAG



Software Heritage graph — topology & size

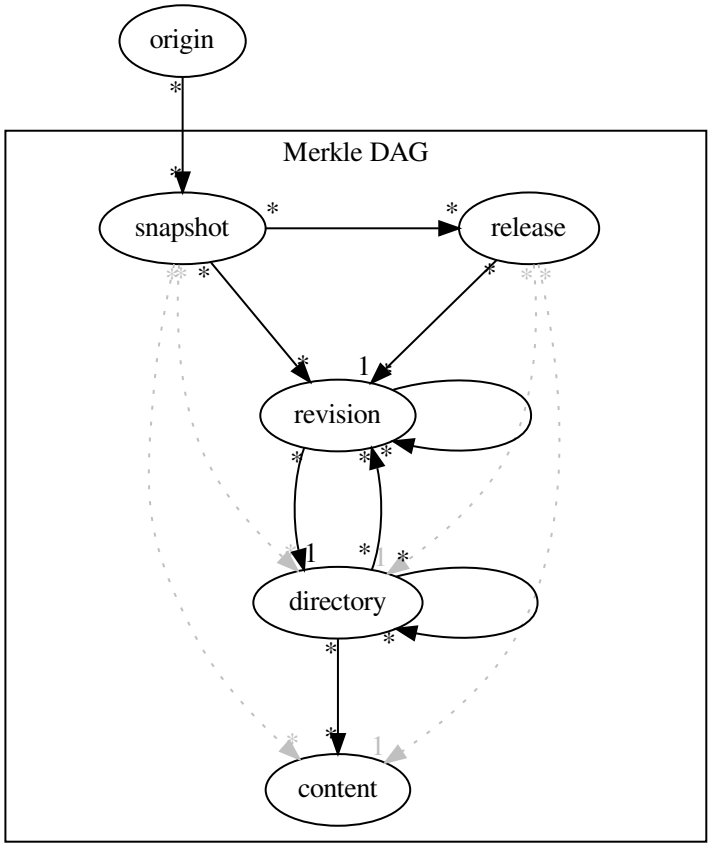


Nodes	
origins (ori)	85 M
snapshots (snp)	57 M
releases (rel)	9.9 M
revisions (rev)	1.1 B
directories (dir)	4.4 B
contents (cnt)	5.0 B
	≈11 B nodes

Edges	
ori→snp	74 M
snp→rev	616 M
rev→rev	1.2 B
rev→dir	1.2 B
dir→dir	49 B
dir→cnt	113 B
	≈165 B edges

- stats for archive snapshot dated 25 Sep 2018
- current size: +22–27% (cnt–rev) in 11 months (≈10M new nodes/day)
- in https://arxiv.org/abs/1906.08076 we found similar evidence of exponential growth, doubling every 22–30 months

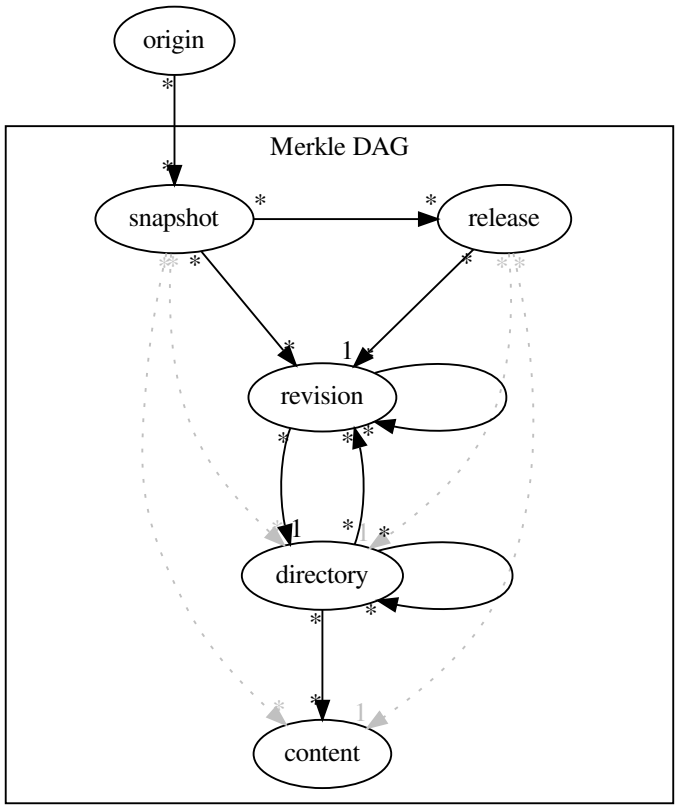
Software Heritage graph — topology & size



Nodes		Edges	
origins (ori)	85 M	ori→snp	74 M
snapshots (snp)	57 M	snp→rev	616 M
releases (rel)	9.9 M	rev→rev	1.2 B
revisions (rev)	1.1 B	rev→dir	1.2 B
directories (dir)	4.4 B	dir→dir	49 B
contents (cnt)	5.0 B	dir→cnt	113 B
	≈11 B nodes		**≈165 B edges**

- stats for archive snapshot dated 25 Sep 2018
- current size: +22–27% (cnt–rev) in 11 months (≈10M new nodes/day)
- in `https://arxiv.org/abs/1906.08076` we found similar evidence of exponential growth, doubling every 22–30 months

Use cases

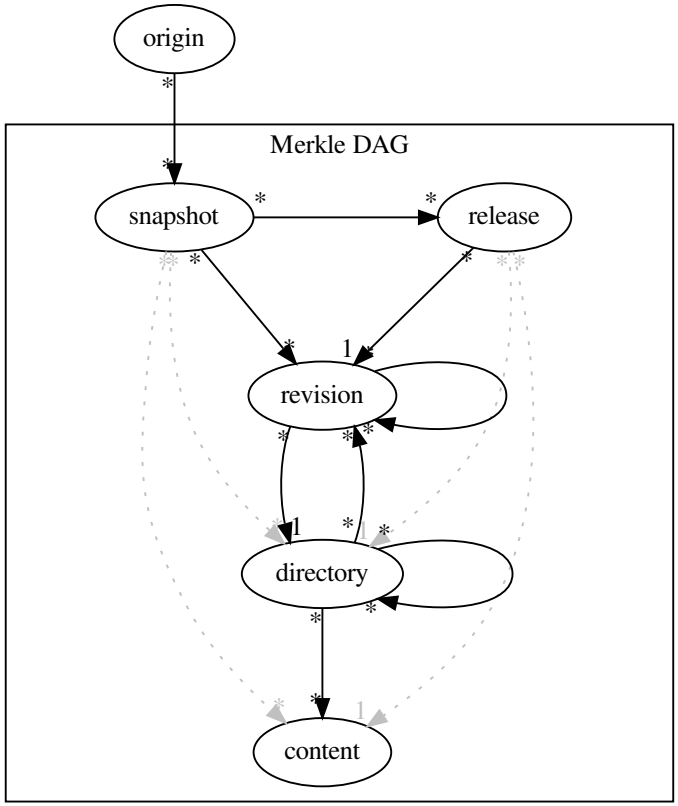


Product needs, e.g., for `https://archive.softwareheritage.org`

- Browsing
  - `ls`
  - `git log` (Linux: 800K+ commits)
- Wayback machine
  - tarball
  - `git bundle` (Linux: 7M+ nodes)
- Provenance tracking
  - commit provenance (one/all)
    - requires backtracking
  - origin provenance (one/all)

Note: we need both the graph and its transposed.

Use cases (cont.)



Research questions

- for the sake of it
  - local graph topology
  - connect component size
    - ⋆ enabling question to find the best approach to conduct large-scale analyses
  - any other emerging property
- software engineering needs
  - software provenance analysis at this scale is pretty much unexplored yet
  - industry frontier: increase granularity down to the individual line of code
  - ...

Current graph storage

- relational DB (Postgres)
- (roughly) 1 table per node type
- intrinsic Merkle DAG identifiers as primary keys
- ≈ 6 TB on SSDs
  - indexes don't fit in memory
  - random access pattern (due to crypto checksum properties)
  - ⇒ bad performance

What if we can fit the entire graph (structure) in memory?

⇒ graph compression

Current graph storage

- relational DB (Postgres)
- (roughly) 1 table per node type
- intrinsic Merkle DAG identifiers as primary keys
- ≈ 6 TB on SSDs
  - indexes don't fit in memory
  - random access pattern (due to crypto checksum properties)
  - ⇒ bad performance

What if we can fit the entire graph (structure) in memory?

⇒ graph compression

Approach

Boldi, P. and and Vigna, S., 2004.
The WebGraph framework I: compression techniques.
In Proceedings of the 13th international conf. on World Wide Web (pp. 595-602). ACM.

Apostolico, A. and Drovandi, G., 2009.
Graph compression by BFS.
Algorithms, 2(3), pp.1031-1044.

In practice: `http://webgraph.di.unimi.it/`

- Note: while the techniques are known to be used at scale (Google, Facebook, etc.), *this* specific implementation had never been used before at Software Heritage scale.

Approach

Boldi, P. and and Vigna, S., 2004.
The WebGraph framework I: compression techniques.
In Proceedings of the 13th international conf. on World Wide Web (pp. 595-602). ACM.

Apostolico, A. and Drovandi, G., 2009.
Graph compression by BFS.
Algorithms, 2(3), pp.1031-1044.

In practice: `http://webgraph.di.unimi.it/`

- Note: while the techniques are known to be used at scale (Google, Facebook, etc.), *this* specific implementation had never been used before at Software Heritage scale.

Approach

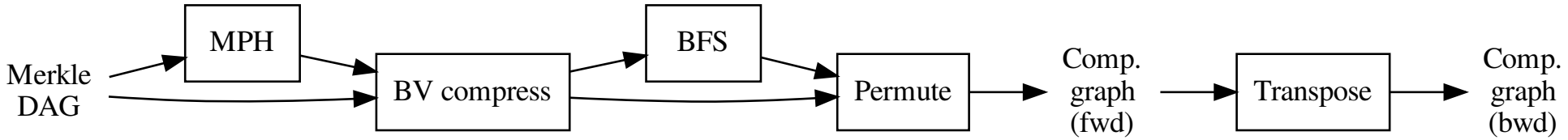
Boldi, P. and and Vigna, S., 2004.
The WebGraph framework I: compression techniques.
In Proceedings of the 13th international conf. on World Wide Web (pp. 595-602). ACM.

Apostolico, A. and Drovandi, G., 2009.
Graph compression by BFS.
Algorithms, 2(3), pp.1031-1044.

In practice: `http://webgraph.di.unimi.it/`

- Note: while the techniques are known to be used at scale (Google, Facebook, etc.), *this* specific implementation had never been used before at Software Heritage scale.

Compression: pipeline and benchmark



	Timings	Max mem usage
`MPH`	3h30	10GB
`BV Compress`	103h	15GB
`BFS`	10h	1057GB
`Permute`	24h40	115GB
`Stats`	4h	102GB
`Transpose`	21h30	19GB
Total	**167h** (7 days)	**1TB**

Compression results

And the winner is. . .

- 4.913 bit/edge
  - compare with 3.08–5.40 bit/edge for the largest dataset in [1] (100x smaller)
- compressed size
  - direct: 91 GiB
  - transposed: 83 GiB
  - total: 174 GiB

(reminder: that's for 11 B nodes / 165 B edges)

Practical takeaways

- on a powerful workstation we can load in main memory one direction of the graph
- on a relatively cheap server we can load both

Compression results

And the winner is. . .

- 4.913 bit/edge
  - compare with 3.08–5.40 bit/edge for the largest dataset in [1] (100x smaller)
- compressed size
  - direct: 91 GiB
  - transposed: 83 GiB
  - total: 174 GiB

(reminder: that's for 11 B nodes / 165 B edges)

Practical takeaways

- on a powerful workstation we can load in main memory one direction of the graph
- on a relatively cheap server we can load both

Compression results

And the winner is. . .

- 4.913 bit/edge
  - compare with 3.08–5.40 bit/edge for the largest dataset in [1] (100x smaller)
- compressed size
  - direct: 91 GiB
  - transposed: 83 GiB
  - total: 174 GiB

(reminder: that's for 11 B nodes / 165 B edges)

Practical takeaways

- on a powerful workstation we can load in main memory one direction of the graph
- on a relatively cheap server we can load both

Deployment: swh-graph

Software Heritage graph service

- https://forge.softwareheritage.org/source/swh-graph/

- Docker-ized compression pipeline
  - input: .nodes/.edges text files (SWH PIDs[1])
  - output:
    - WebGraph compression output
    - mappings SWH PID ↔ long
    - mapping long → node type (for selective edge traversal)
- Java server: bridge Webgraph ↔ REST API
- Python client for the REST API

[1] https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html

swh-graph — use cases and API

Browsing

- **`ls`**

  `/graph/neighbors/:DIR_ID?edges=dir:cnt,dir:dir`

- **`git log`**

  `/graph/visit/nodes/:REV_ID?edges=rev:rev`

Wayback machine

- **tarball** (i.e., `ls -R`)

  `/graph/visit/paths/:DIR_ID?edges=dir:cnt,dir:dir`

- **`git bundle`**

  `/graph/visit/nodes/:NODE_ID?edges=*`

swh-graph — use cases and API

Browsing

- **`ls`**

  `/graph/neighbors/:DIR_ID?edges=dir:cnt,dir:dir`

- **`git log`**

  `/graph/visit/nodes/:REV_ID?edges=rev:rev`

Wayback machine

- **tarball** (i.e., `ls -R`)

  `/graph/visit/paths/:DIR_ID?edges=dir:cnt,dir:dir`

- **`git bundle`**

  `/graph/visit/nodes/:NODE_ID?edges=*`

swh-graph — use cases and API (cont.)

Provenance

- **commit provenance** (one commit)

  ```
  /graph/walk/:NODE_ID/rev?direction=backward\
      &edges=dir:dir,cnt:dir,dir:rev
  ```

- **commit provenance** (all commits)

  ```
  /graph/leaves/:NODE_ID?direction=backward\
      &edges=dir:dir,cnt:dir,dir:rev
  ```

- **origin provenance** (one origin)

  ```
  /graph/walk/:NODE_ID/ori?direction=backward&edges=*
  ```

- **origin provenance** (all origins)

  ```
  /graph/leaves/:NODE_ID?direction=backward&edges=*
  ```

swh-graph — use cases and API (cont.)

Provenance

- **commit provenance** (one commit)

```
/graph/walk/:NODE_ID/rev?direction=backward\
    &edges=dir:dir,cnt:dir,dir:rev
```

- **commit provenance** (all commits)

```
/graph/leaves/:NODE_ID?direction=backward\
    &edges=dir:dir,cnt:dir,dir:rev
```

- **origin provenance** (one origin)

```
/graph/walk/:NODE_ID/ori?direction=backward&edges=*
```

- **origin provenance** (all origins)

```
/graph/leaves/:NODE_ID?direction=backward&edges=*
```

Preliminary benchmarks — edge traversal

TL;DR

Experimentally verified per-edge traversal time (1.4–3.8μs) is consistent with what is reported in the literature on much smaller graphs (e.g., 2.1–3.1μs in [2] for a graph 500x smaller).

Preliminary benchmarks — edge traversal (cont.)

Browsing

	ls μs/edge	**ls -R** μs/edge	**git log** μs/edge
average	3.0	3.8	1.5
median	1.6	1.7	3.2
standard deviation	20	5.0	3.4

	ls	**ls -R**	**git log**
average edge accessed	30	2566	21081

Preliminary benchmarks — edge traversal (cont.)

Wayback machine

	git bundle μs/edge
average	2.9
median	0.9
standard deviation	0.2

	git bundle
average edge accessed	456418

Preliminary benchmarks — edge traversal (cont.)

Provenance

	commit pr. μs/edge	**complete commit pr.** μs/edge	**origin pr.** μs/edge	**complete origin pr.** μs/edge
average	3.5	1.7	1.4	1.4
median	2.4	1.7	1.1	1.4
standard deviation	3.0	1.7	6.2	1.2

	commit pr.	**complete commit pr.**	**origin pr.**	**complete origin pr.**
average edge accessed	38	12	2799	12

Directions

- incrementality: cope with frequent (10 M nodes/day additions) graph updates without having to wait for full re-compression
- attributes: we currently only have node→type information in memory; we want to add attribute *on edges*, e.g., timestamps that could be used for...
- optimization during visits: e.g., to track the *earliest occurrence* of any given node without ad hoc out-of-graph tracking
- granularity: add the line of code layer, assuming 10 SLOC/content ratio, that would add 50 B new nodes and hundred B new edges; is it still workable?

- Software Heritage is a huge graph emergent from public software development
- It's a novel object of study that compresses well with Web graph techniques
- It poses challenges of incrementality and visit optimization at scale

Advertisement

We have an open post-doc position on this topic:
`https://www.softwareheritage.org/jobs/postdoc-provenance/`

Thanks!

Questions?

Stefano Zacchiroli
zack@upsilon.cc

about the slides:
available at `https://upsilon.cc/~zack/talks/2019/2019-09-09-gccr2019.pdf`