Git at GitHub Scale

Taylor Blau (@ttaylorr), GitHub
Git Merge 2022

Taylor Blau

@ttaylorr

Staff Software Engineer, GitHub

My work at GitHub

50% of time on open-source

Triaging mailing list, responding to bugs, submitting patches, PLC work.

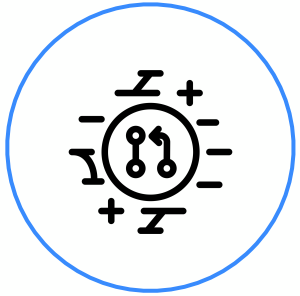
50% of time on “Git at GitHub”

Responding to escalations, identifying pain points, writing code, working with internal teams.

Today's agenda

"Git at GitHub"

Our fork model, where and why we use Git.



Git ➡️ GitHub

Features from the open-source project we use at GitHub.

GitHub ➡️ Git

Features developed at GitHub that we contributed to Git.

Git at GitHub

Some numbers...

200M+

public+private repositories

2.6B+

annual contributions

Git at GitHub

- (Lightly) modified fork of git/git, called “github/git”
- Powers many internal APIs and processes:
  - pushes, fetches, clones
  - periodic repacking
  - many internal RPCs (e.g., get the contents of this README, count of branches, merges, etc.)
- libgit2
  - remaining internal RPCs (e.g., does this branch exist?, create an object, etc.)

github/git

- Lightly modified fork based on the open-source Git project.
- Handful of “uninteresting” permanent patches (logging, metrics, internal services)
- Home of new feature development at GitHub
  - multi-pack bitmaps
  - staging ground for commit-graph changes
  - tree-level `git blame` implementation
- Continuous deployment to GitHub.com
- Back-merges with upstream Git, usually 1-2 major versions behind

github/git

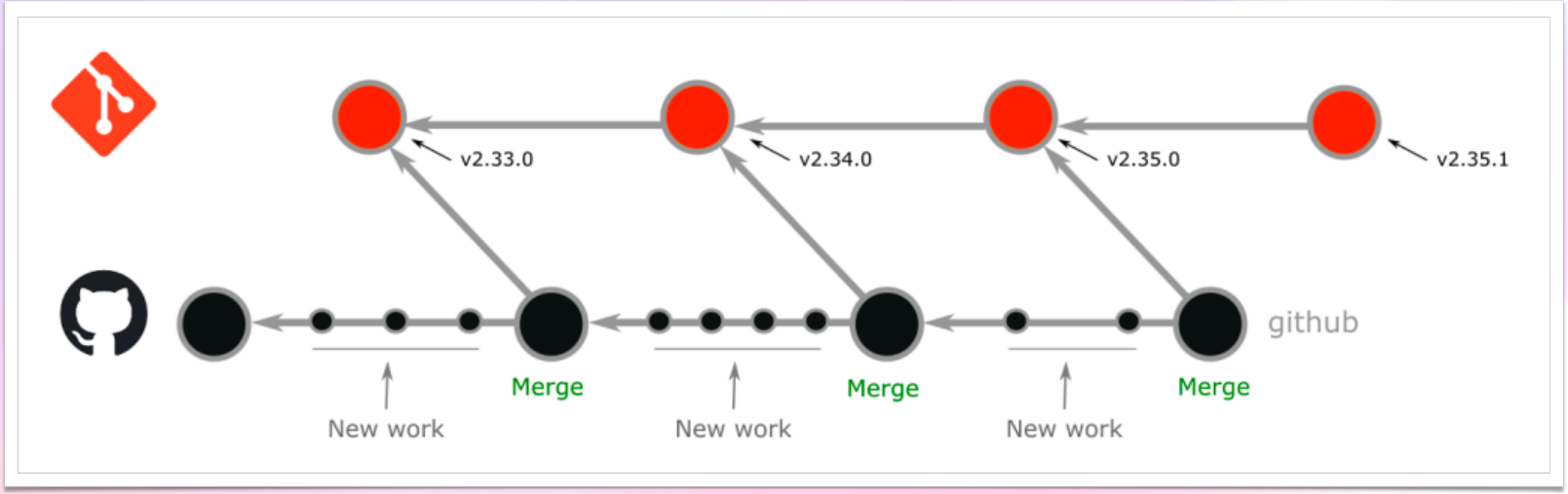


Image credit: Lessley Dennington, GitHub

Why Git?

- Could have built “Git” operations on any technology.
- Git is:
  - fast, and getting faster
  - battle-tested, and reliable
  - secure
  - mutually-beneficial

Git ➡️ GitHub

Upstream Git features at GitHub

commit-graph and
changed-path Bloom filters

partial clones

merge-ort

commit-graphs & changed-path Bloom filters

commit-graphs & changed-path Bloom filters

- On-disk serialization of commit data:
  - Root tree ID
  - Date
  - Parent(s), and octopus edges
- Upstream feature developed at Microsoft by Derrick Stolee

commit-graphs & changed-path Bloom filters

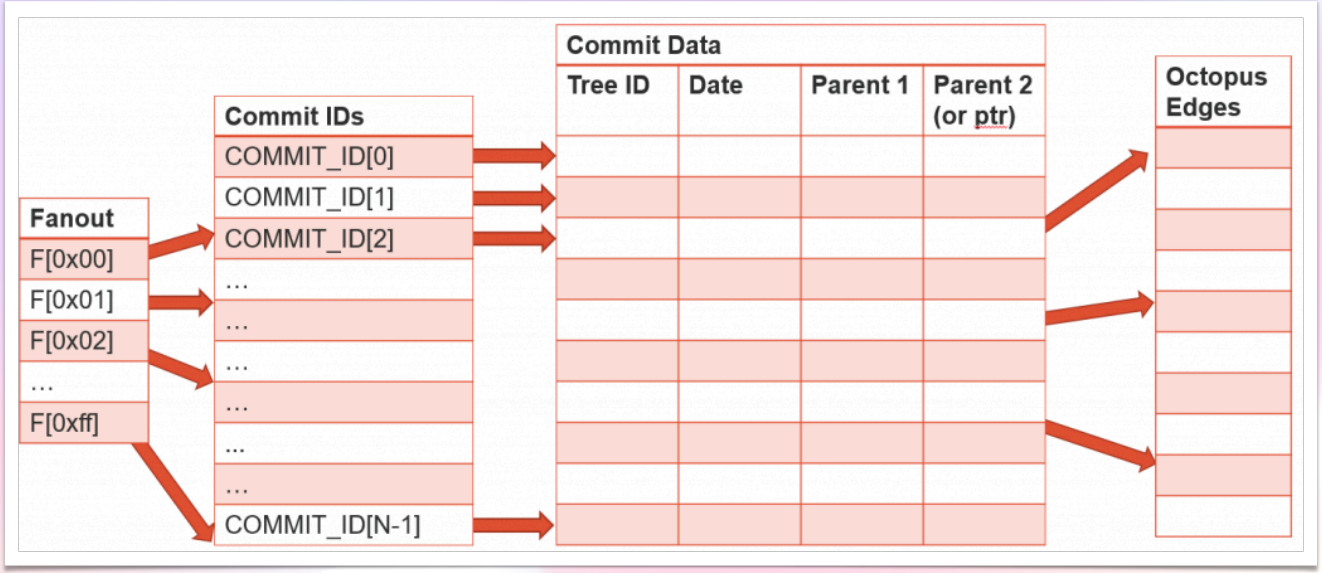


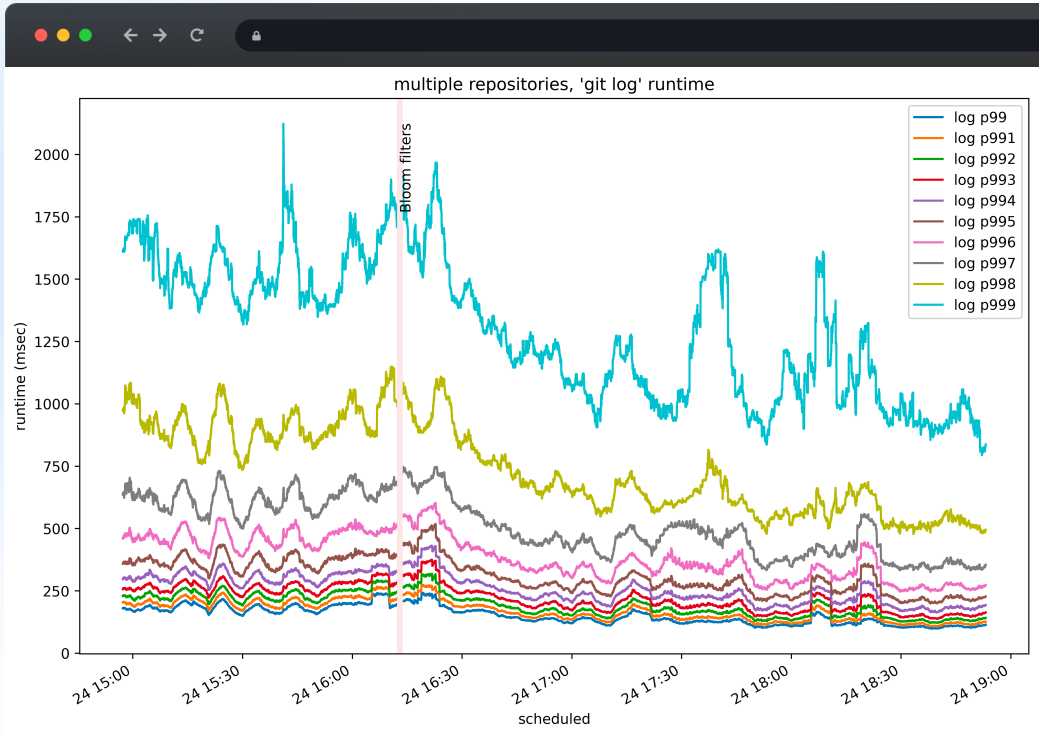
Image credit: Derrick Stolee, GitHub

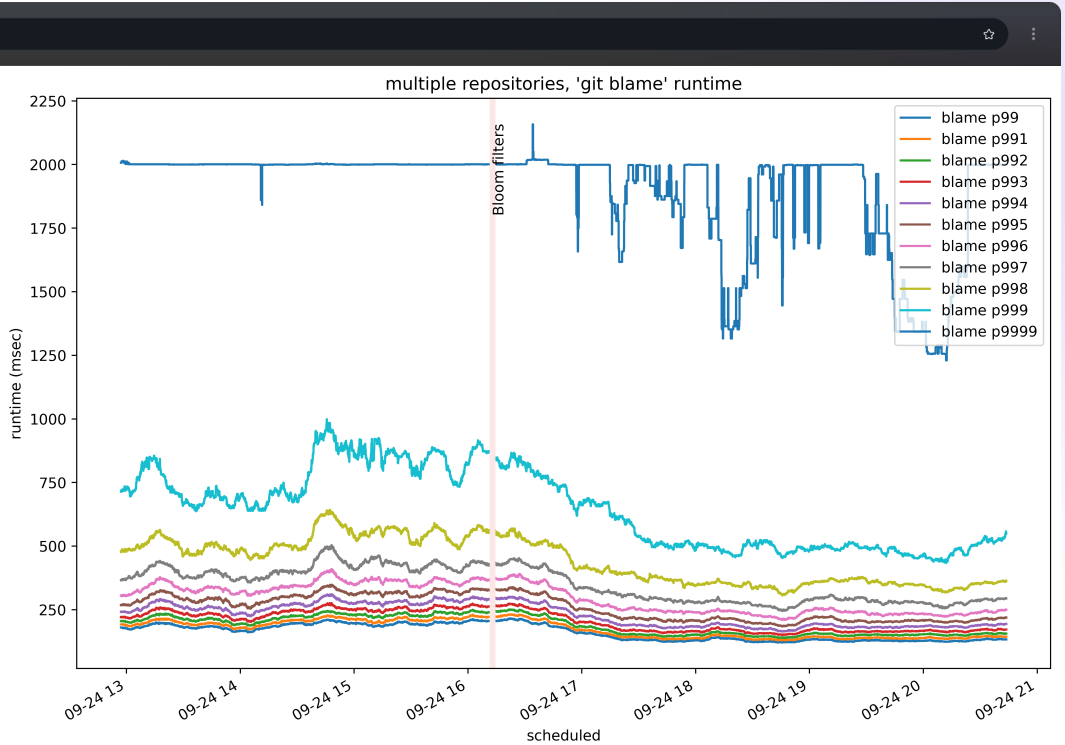
commit-graphs & changed-path Bloom filters

- GitHub updates the commit-graph on each new push
  - Each update adds one new “layer” to the commit-graph chain
  - Occasionally updates cause us to “merge” previous layers
  - Changed-path Bloom filters are computed for incoming commits up to a threshold

commit-graphs & changed-path Bloom filters

- `git log` runtimes, p99 through p999
- 1.75s p999 -> 1s p999



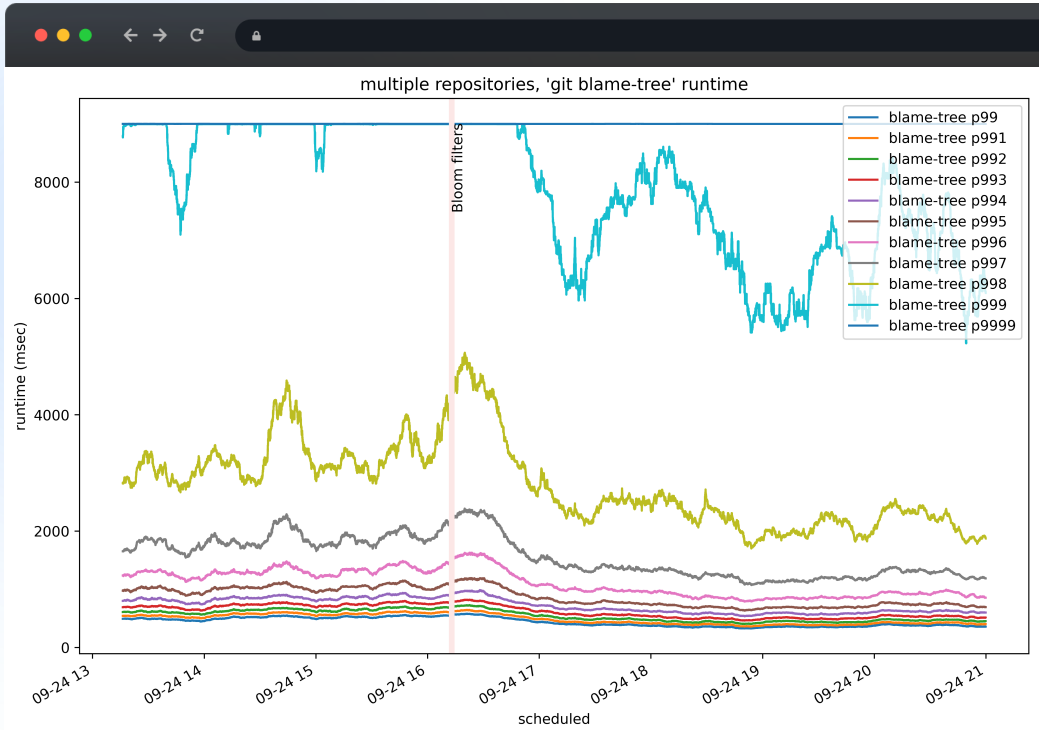


commit-graphs & changed-path Bloom filters

- `git blame` runtimes, p99 through p9999
- ~40% reduction p999
- fewer timeouts in p9999

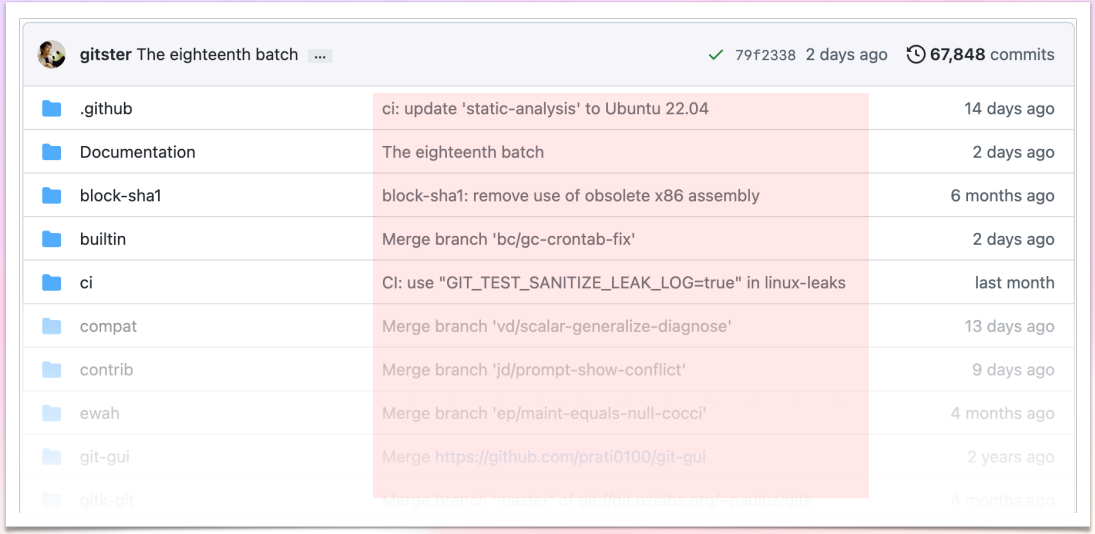
commit-graphs & changed-path Bloom filters

- `git blame-tree` runtimes, p99 through p9999
- 3.5s -> 2s p998
- fewer timeouts p999, p9999



commit-graphs & `git blame-tree`

- git blame-tree is a custom command that provides a tree-level blame

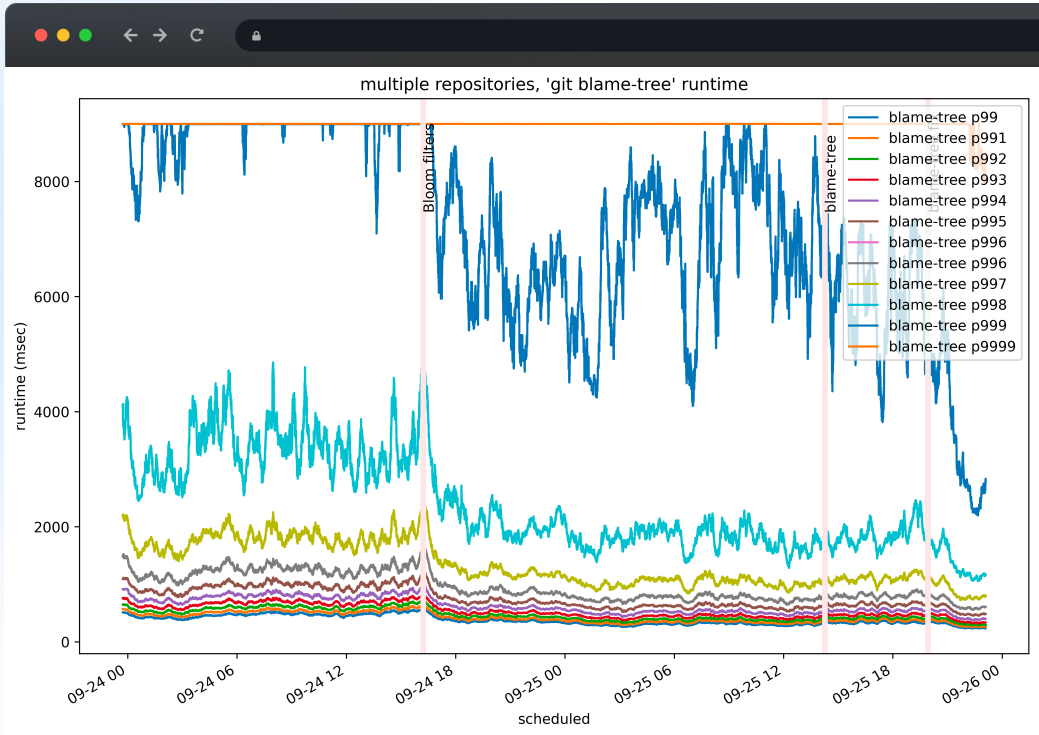


commit-graphs & `git blame-tree`

- Existing algorithm:
  - Until all paths are blamed, walk along history and compute a tree-level diff at each level
- New algorithm:
  - Only compute a tree-level diff for unblamed paths
  - Skip over parts of history where possible with Bloom filters
  - Pass unblamed paths to parent(s)
- Collaboration between Derrick Stolee and myself

commit-graphs & `git blame-tree`

- `git blame-tree` runtimes, p99 through p9999
- further reduction p998, 4s -> 2s
- p999, p9999 timeout reduction



Partial clones

Partial clones

- Ability to clone specific part(s) of your repository
- Dictated by different `--filter` options when cloning
- Developed upstream by Jeff Hostetler and Jonathan Tan
- Integrated with bitmaps by GitHub

Partial clones

- full clone runtime 4m43s

```
$ best-of-five -p 'rm -rf linux.git' \
  sh -c 'git clone --bare \
    git@github.com:torvalds/linux.git'

Attempt 1: 283.75
Attempt 2: 283.97
Attempt 3: 297.601
Attempt 4: 299.141
Attempt 5: 323.365

real 4m43.750s
user 5m23.133s
sys  1m5.691s
```


Partial clones

- full clone runtime 4m43s
- partial clone runtime 1m57s

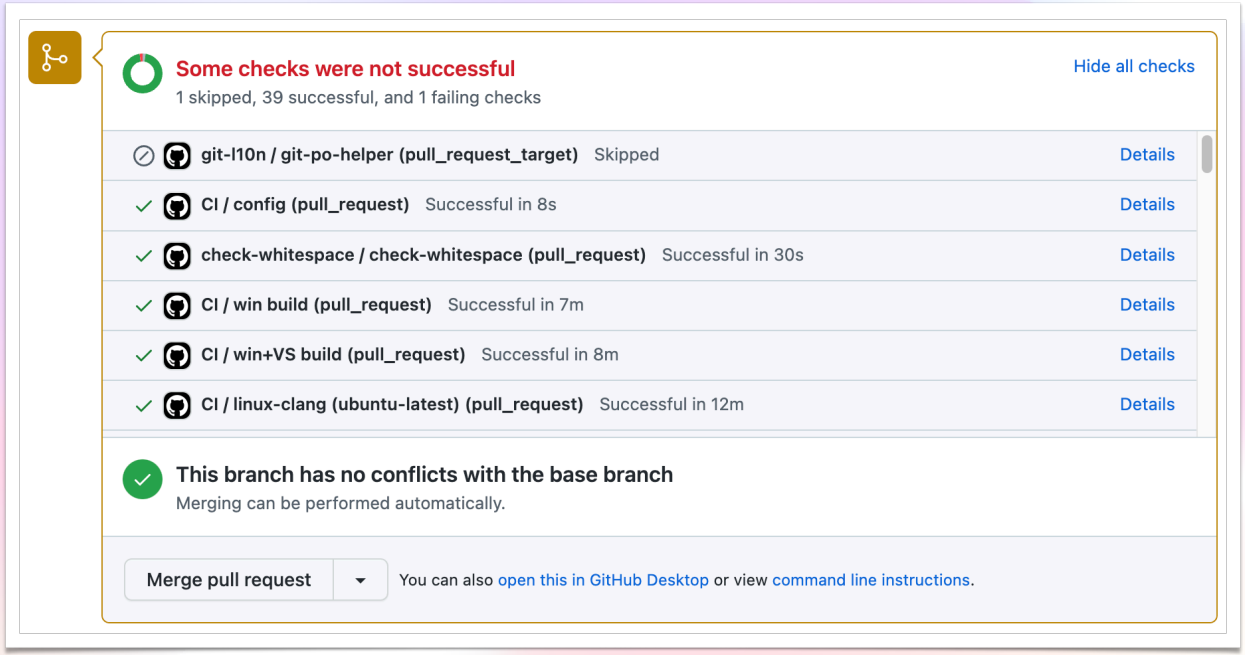
```
$ best-of-five -p 'rm -rf linux.git' \
  sh -c 'git clone --bare --filter=blob:none \
    git@github.com:torvalds/linux.git'

Attempt 1: 124.282
Attempt 2: 127.547
Attempt 3: 134.818
Attempt 4: 125.464
Attempt 5: 117.205

real 1m57.205s
user 1m16.124s
sys  0m25.912s
```


merge-ort

merge-ort

- Elijah's talk explained many/all of the details here
- Merges are computed proactively/manually in the web UI

merge-ort


Some checks were not successful
1 skipped, 39 successful, and 1 failing checks
Hide all checks
git-l10n / git-po-helper (pull_request_target) Skipped Details
CI / config (pull_request) Successful in 8s Details
check-whitespace / check-whitespace (pull_request) Successful in 30s Details
CI / win build (pull_request) Successful in 7m Details
CI / win+VS build (pull_request) Successful in 8m Details
CI / linux-clang (ubuntu-latest) (pull_request) Successful in 12m Details
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

merge-ort

- merge-recursive requires working copy to represent conflicts
  - originally created a temporary working copy to perform a merge
  - then implemented merges in libgit2 to eliminate the need for a working copy
- `git merge-tree` gained the ability to do “server-side” merges
  - Collaboration between Johannes Schindelin and Elijah Newren
- now merge-ort powers merges on GitHub.com
  - Work here done by Johannes Schindelin and Greg Hurrell


rpc.git.dist.time p99
merge-ort deploy
merge-ort 100%
Seconds
10
9
8
7
6
5
4
3
2
1
0
Tue 9
Thu 11
Sat 13
Mon 15
Wed 17
Fri 19
Aug 21
Tue 23
Thu 25
Sat 27
Mon 29
Wed 31
September
Sat 3
Mon 5
Wed 7
July
August
September

GitHub ➡️ Git

GitHub features in upstream Git

- multi-pack reachability bitmaps
- On-disk reverse indexes
- Geometric repacking
- Cruft packs

multi-pack reachability bitmaps

Packs & maintenance

- Each time a repository is pushed to, a new pack is added to the repository
- As more packs are added, performance degrades over time
- To keep repositories running smoothly, schedule a periodic “maintenance” routine on active and/or under-maintained repositories
- Maintenance compacts all objects into a single pack

Repository maintenance

- Running something similar to `git repack -adk --write-bitmap-index`.
- Why a single pack?
- Any operation which performs object lookups needs only to consult a single pack (+ any loose object, of which there are generally few)
- Key point: reachability bitmaps.

Reachability bitmaps

- Reachability bitmaps allow us to quickly answer: “what object(s) are reachable from this commit?”
- Eliminates the need for object traversal, which is unbounded
- Can be combined in intuitive ways:
  - The union of reachable objects among multiple bitmaps is a bitwise-OR
  - The set difference (e.g., for haves and wants) is a bitwise-AND/NOT

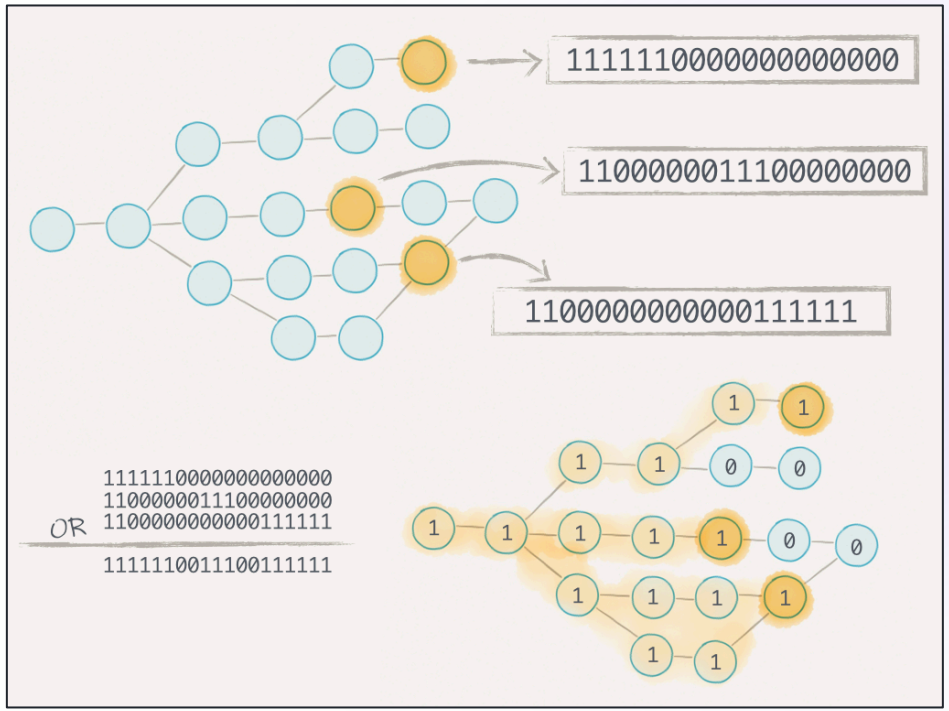


Image credit: Vicent Martí, GitHub

Reachability bitmaps

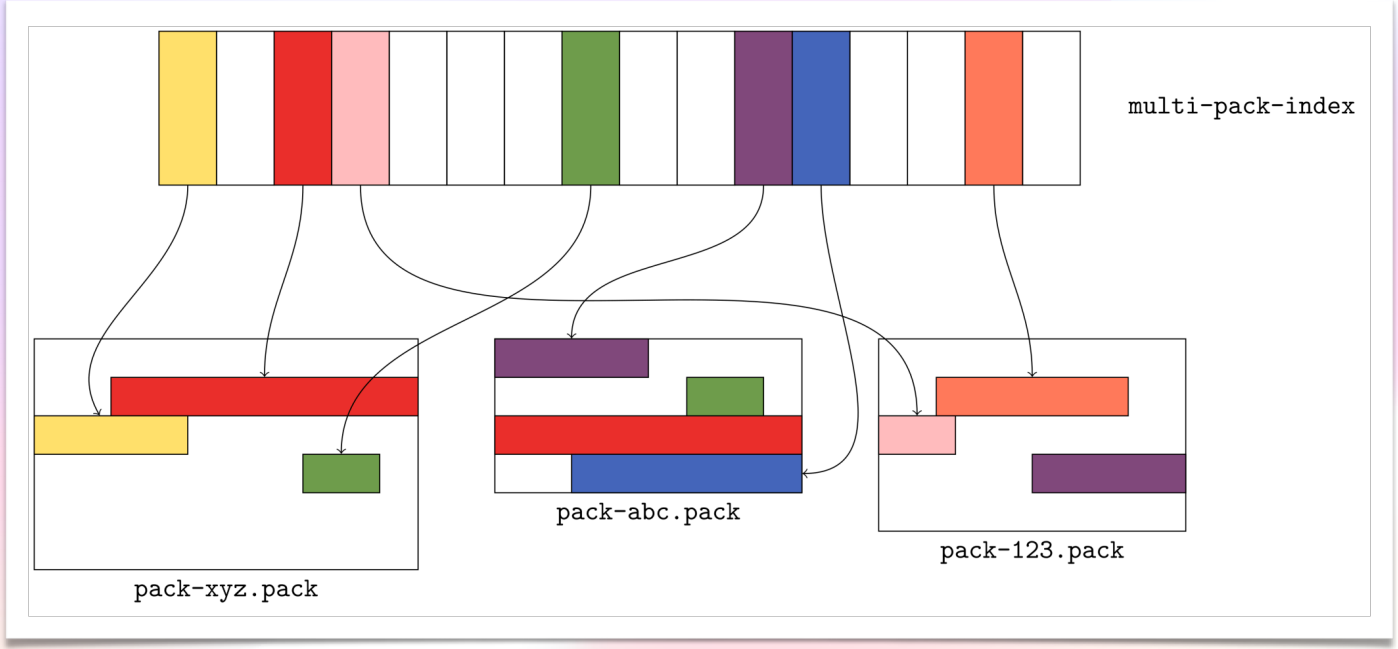
Reachability bitmap limitations

- Problem: can only encode information about objects in a single pack
- Requires us to repack all objects in a repository into a single pack
- Prohibitively expensive as repositories accumulate more and more objects

multi-pack index (MIDX)

- (Partial-)solution: multi-index index (MIDX)
  - Upstream feature contributed by Derrick Stolee
- Acts like a single index over multiple packfiles

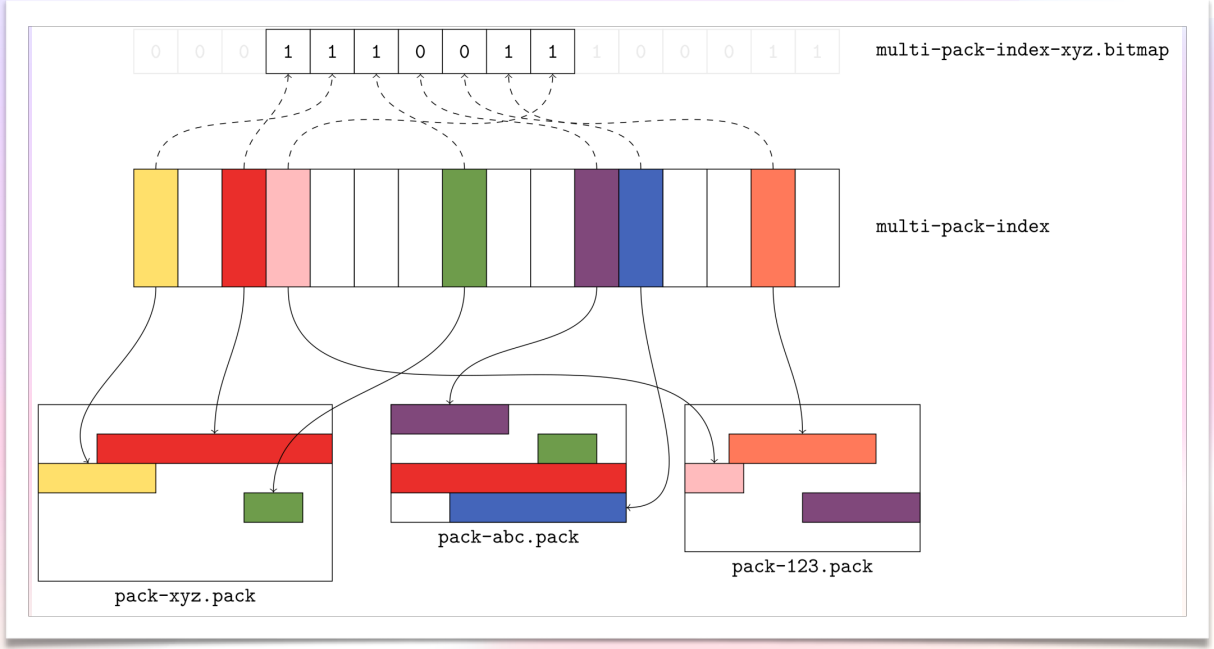
multi-pack index (MIDX)



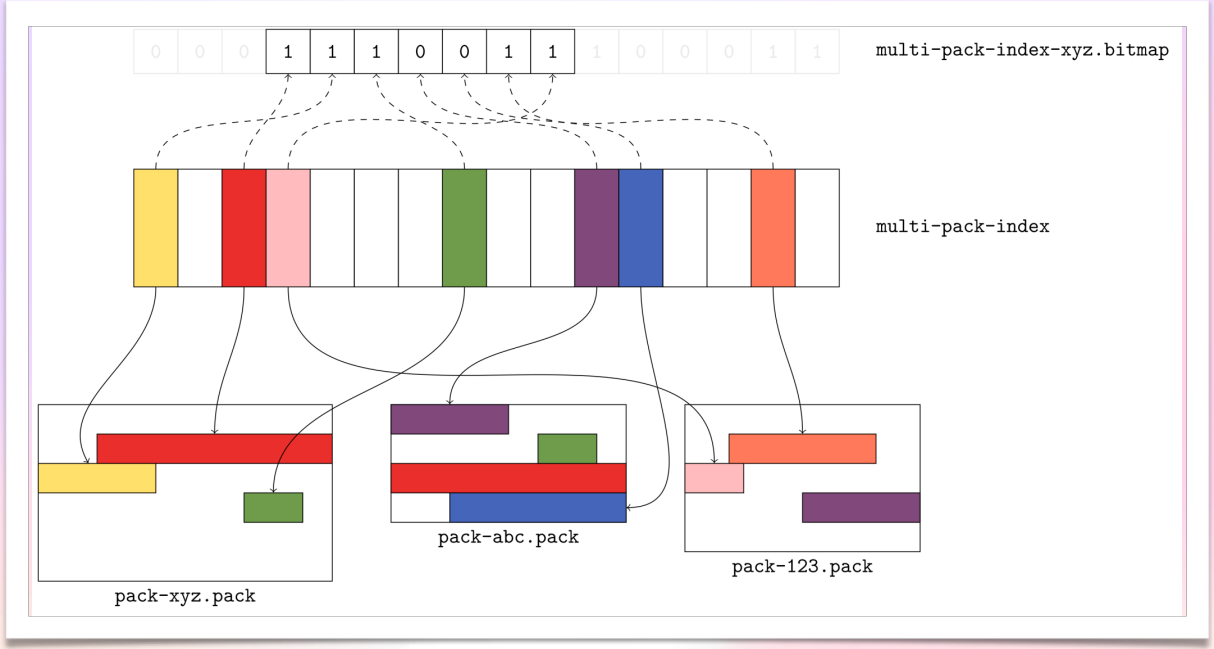
MIDX pseudo-pack order

- Could we define an object order over the objects in a MIDX that we then use to generate a bitmap?
- If so:
  - Could repack a repository into arbitrary pack structure
  - Write a MIDX containing just the packs we want to keep
  - Write a bitmap covering the objects over those packs

MIDX pseudo-pack order



MIDX pseudo-pack order



- How to map bit #5 to blue object?
- Could use number of objects in each pack to find the pack-relative position in `pack-abc`.
- Problem: need to know unique object count in each pack.
  - Red object is only stored once.
- Need something better.

On-disk reverse indexes

Packs, indexes, reverse indexes

- Packs (`*.pack` files) contain a continuous sequence of objects in a (semi-)arbitrary order
- Indexes (`*.idx` files) map objects in lexicographic order to their offset in the corresponding `.pack` file
- Conceptually, reverse indexes map objects in their pack order to lexicographic order

Reverse indexes

- Reverse indexes already exist in Git
  - E.g., `git cat-file --batch-check='%(objectsize:disk)'`
- Computed on-the-fly by (radix) sorting an array of `(object_id, offset)` pairs
  - Works, and uses an efficient sort
  - But still takes time proportional to the number of packed objects
  - Memory intensive
- GitHub had an on-disk version of this as an optional extension in the `.bitmap` file

Reverse indexes

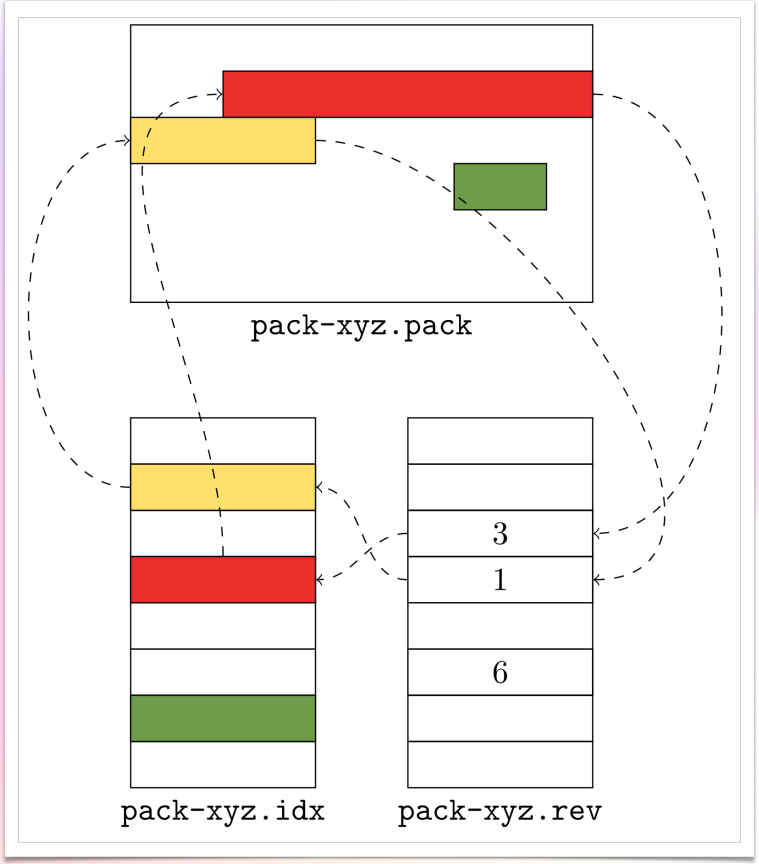
- Bitcache extension
  - Appears at the end of a .bitmap file as an optional extension
  - Table of 4-byte (unsigned) integers, each corresponding to a packed object
  - Stored in pack order (corresponding to objects by ascending pack offsets)
  - Value is the lexicographic index for each object
- Could upstream this, and use it for multi-pack reachability bitmaps
- But requires using a bitmap to be useful

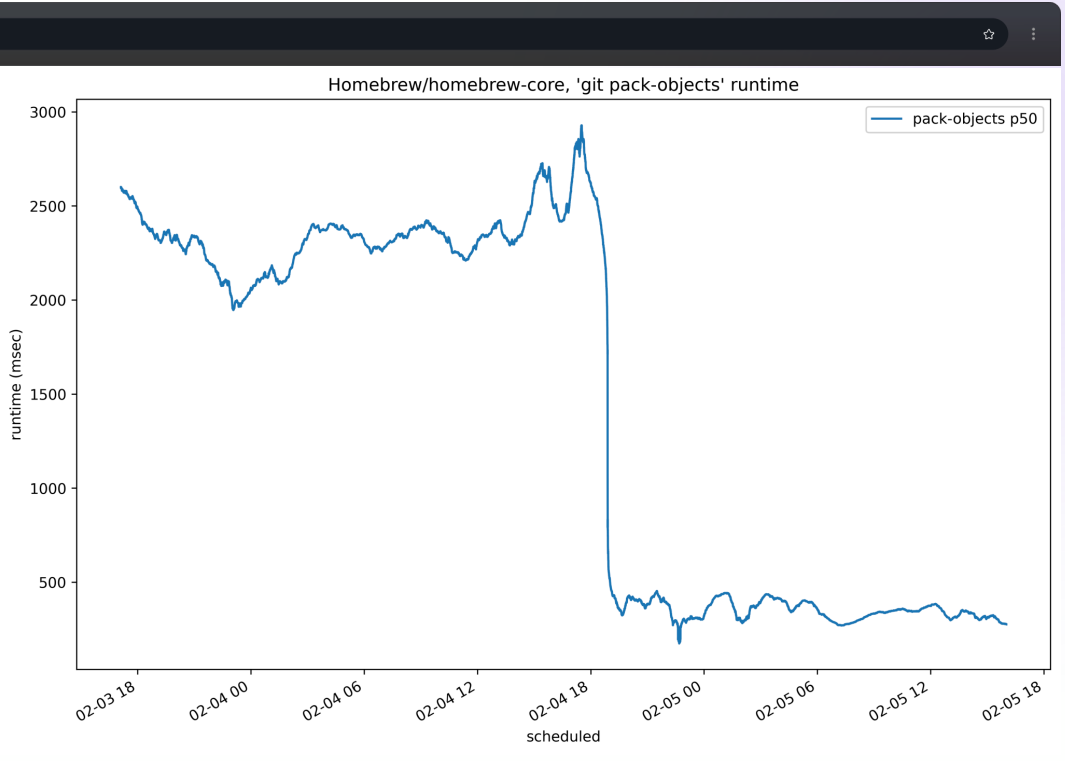
.rev files

- Instead, implement a new on-disk format used outside of the `.bitmap` file
- Called `.rev`, corresponds to both packs and MIDXs
  - For packs, can be used with or without bitmaps
  - For MIDXs, not useful without a bitmap

.rev files

- Instead, implement a new on-disk format used outside of the `.bitmap` file
- Called `.rev`, corresponds to both packs and MIDXs
  - For packs, can be used with or without bitmaps
  - For MIDXs, requires a bitmap



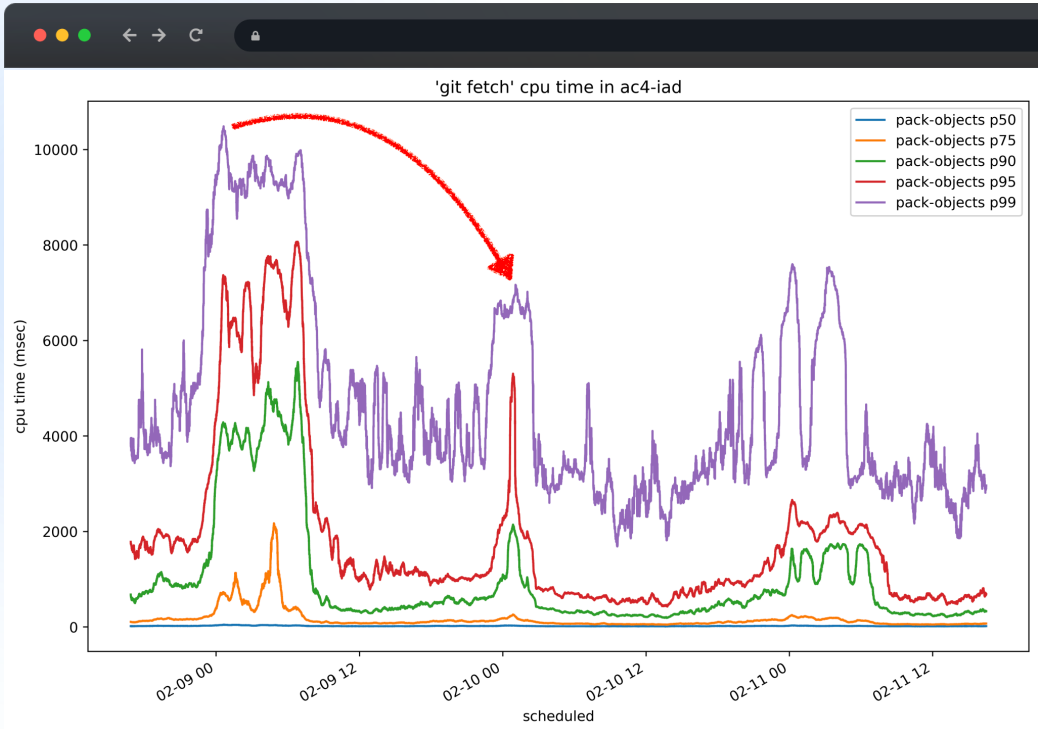


.rev files at GitHub

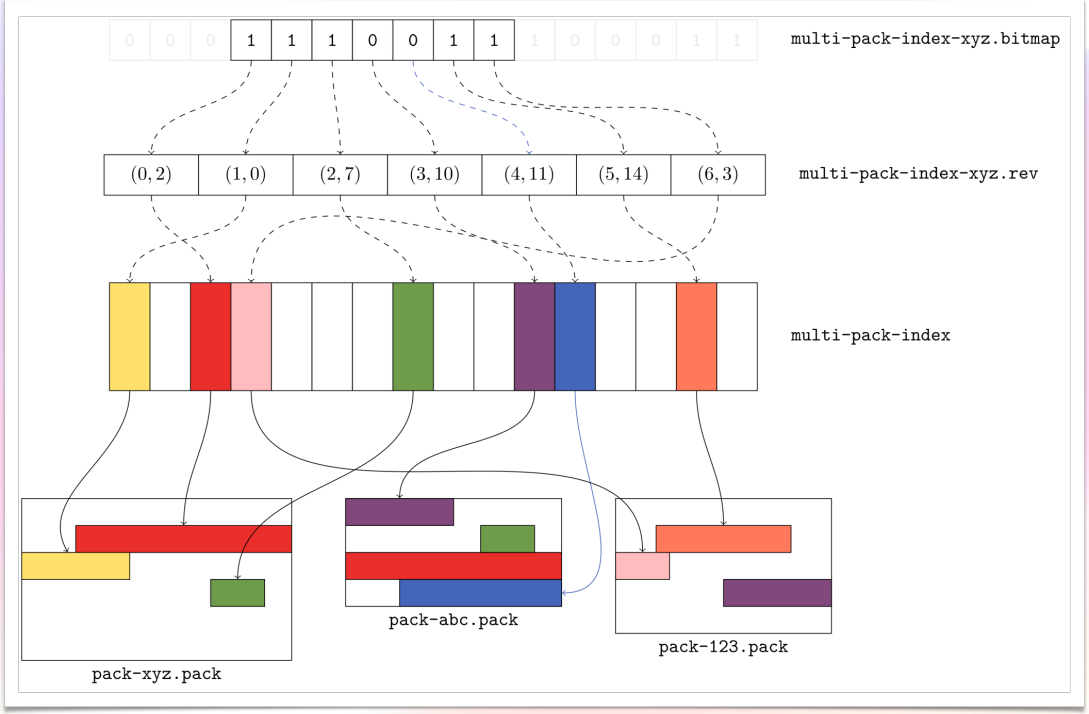
`git pack-objects` p50 CPU time,
(`Homebrew/homebrew-core`)

.rev files at GitHub

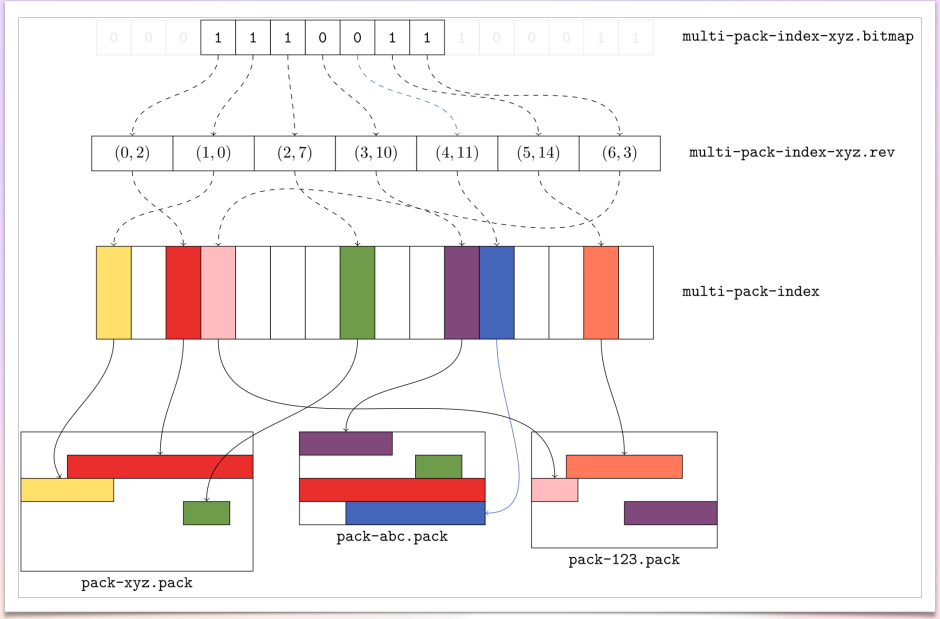
`git fetch` p50-p99 CPU time,
(all repos in one site)



MIDX .rev files



MIDX .rev files



- Define a pseudo-pack order:
  - Objects in pack-order
  - Arrange packs according to how new they are
  - Eliminate duplicate objects, resolve in favor of a “preferred” pack
- Store this order in a `.rev` file for the MIDX
- Can translate from bit positions by reading the corresponding entry

Geometric repacking

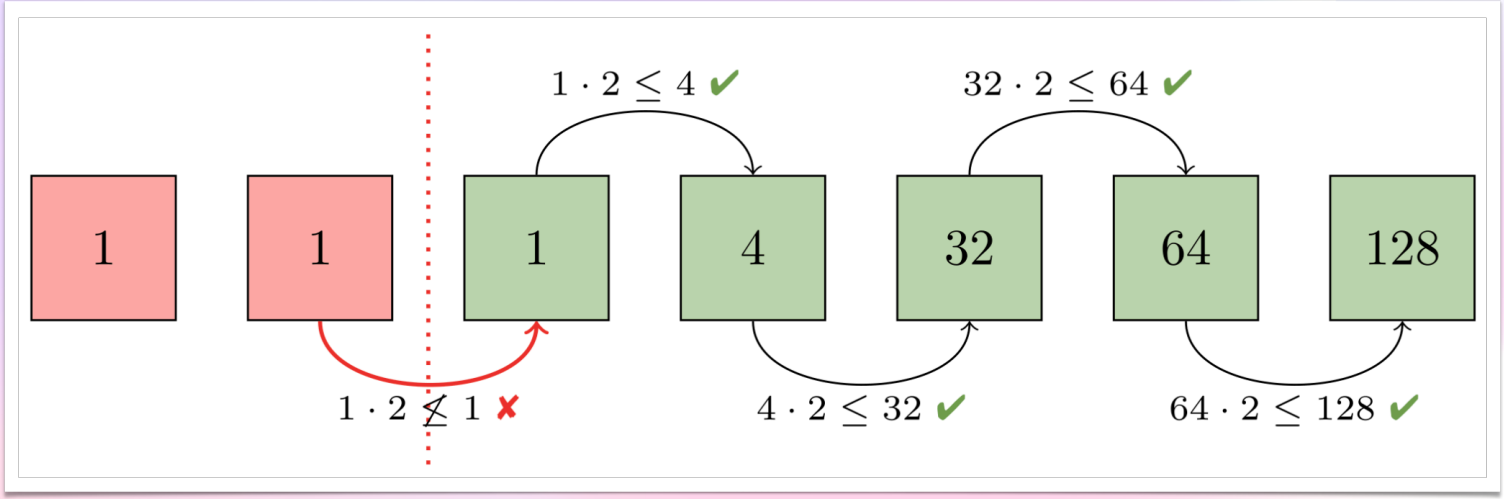
Geometric repacking

- Now that we can repack a repository into arbitrary structure(s)… what strategy should we use?
- Want two properties:
  - On average, usually few packs remain after repacking
  - On average, work is proportional to number of new objects since previous maintenance
- Simplest approach that captures the above two: make each remaining pack have twice the number of objects as the next-largest pack

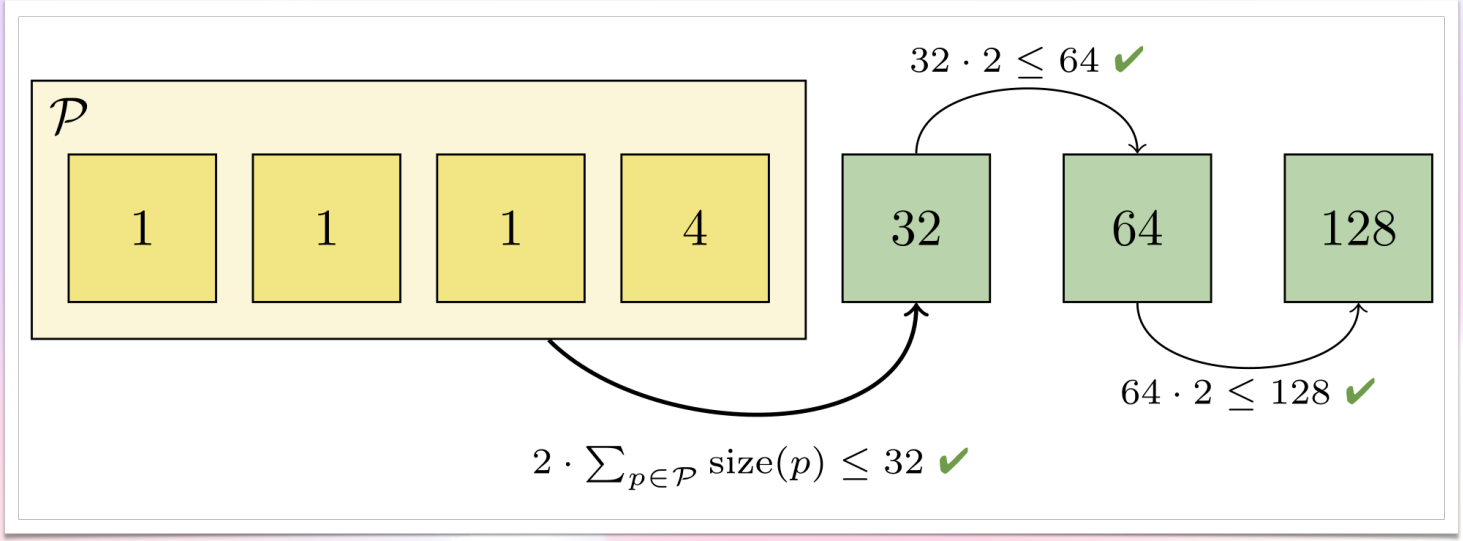
Geometric repacking

- First, organize packs by their object count in ascending order
- Then, determine how many large packs are already in progression
- Adjust based on rolling up remaining packs

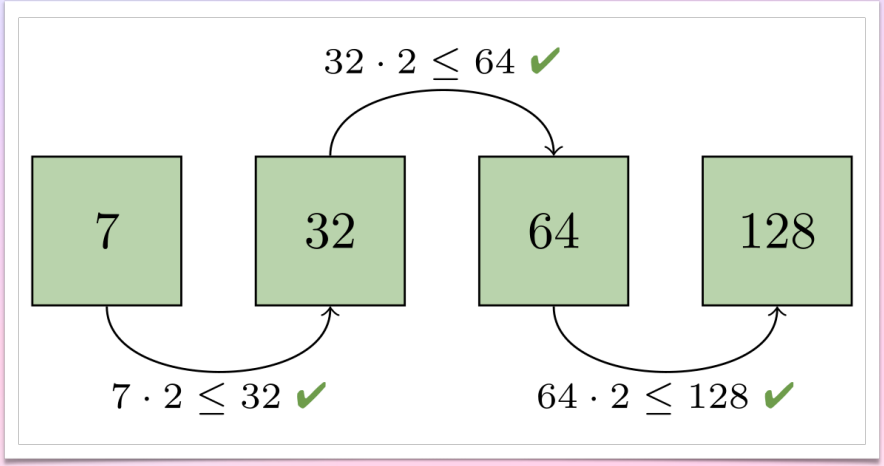
Geometric repacking



Geometric repacking



Geometric repacking



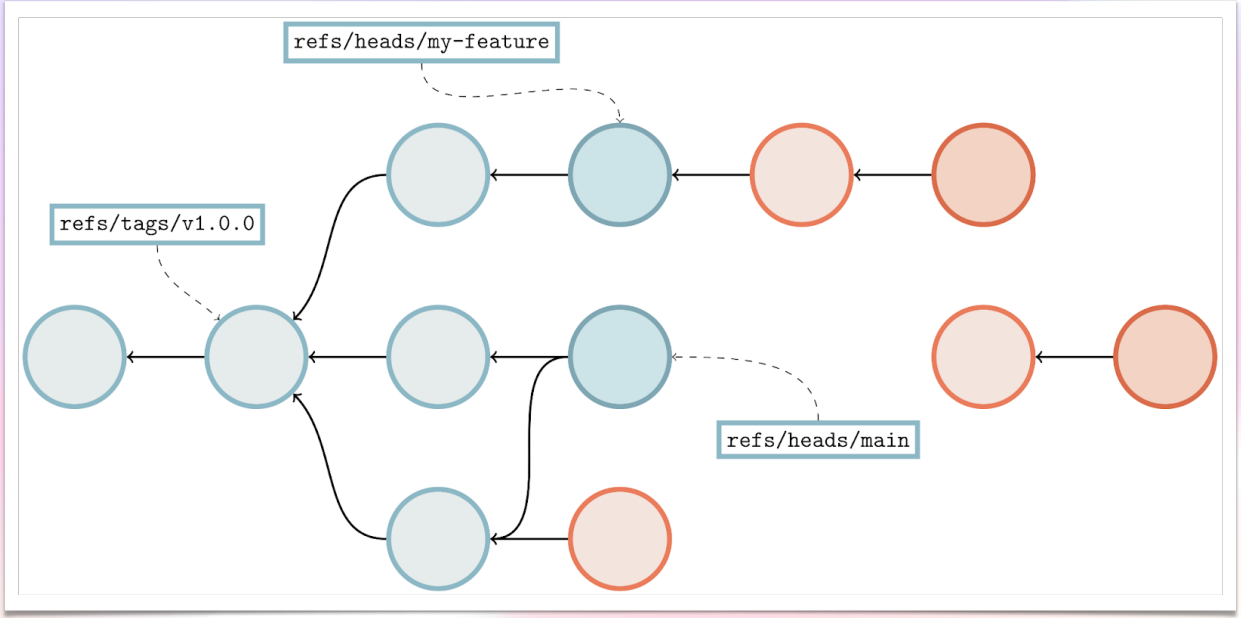

github.nw_repack (overall sum, vs. last week)
this week
15d 17h
Days
Tue 16
Wed 17
Thu 18
Fri 19
Sat 20
Mar 21
Mon 22
Tue 23
Wed 24
Thu 25
Fri 26
Sat 27
Mar 28
Mon 29
03/30 10:00
Wed 31
April
Fri 2
Sat 3
Apr 4
Mon 5

5.67 CPU-days

saved every hour on GitHub.com

Cruft packs

Unreachable objects



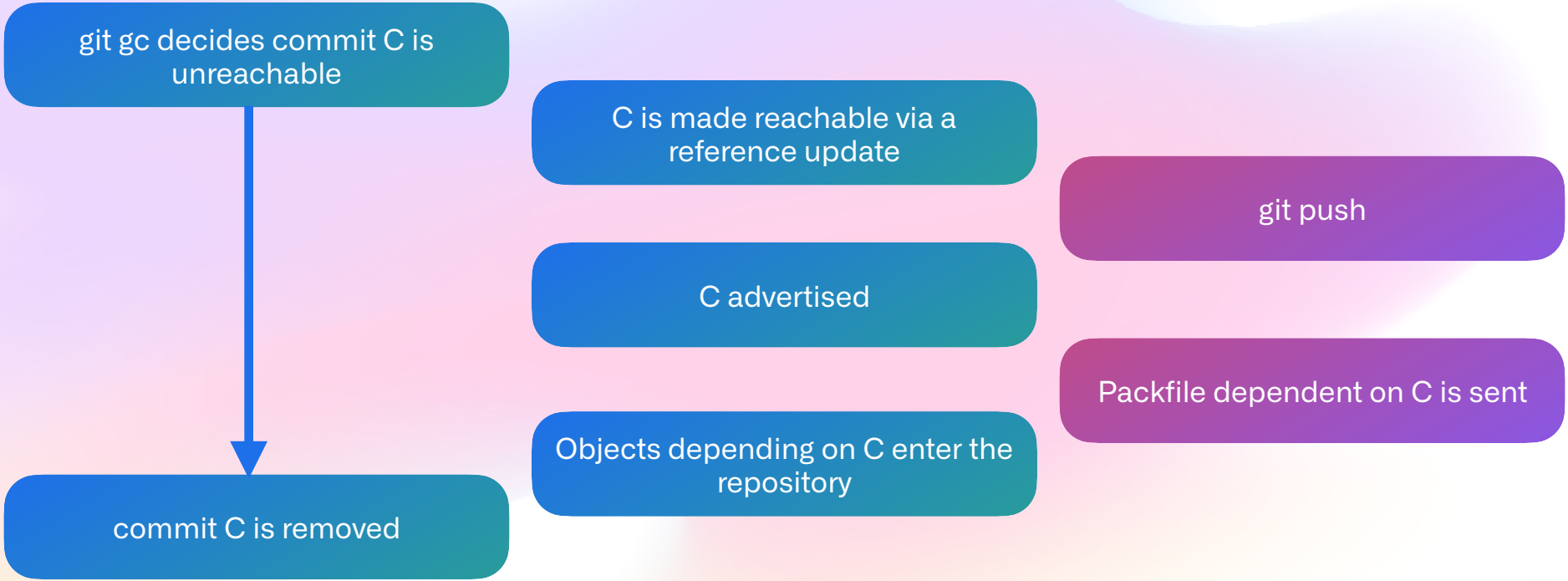
Unreachable objects at GitHub

- Lots of unreachable objects in repositories on GitHub
  - Test-merges
  - Force-pushes
  - Branch deletions
- Usually let these grow without bound

Pruning unreachable objects

- Unreachable objects left alone by default
- Occasionally users request manual cleanup, and we run `.gc <repo>` in chat
- This causes us to run something along the lines of:
  `git repack -Adn --unpack-unreachable=5.minutes.ago`
  - Moves reachable objects into one pack
  - Removes unreachable objects (older than 5 minutes)
  - Loosens remaining unreachable objects

Pruning raciness



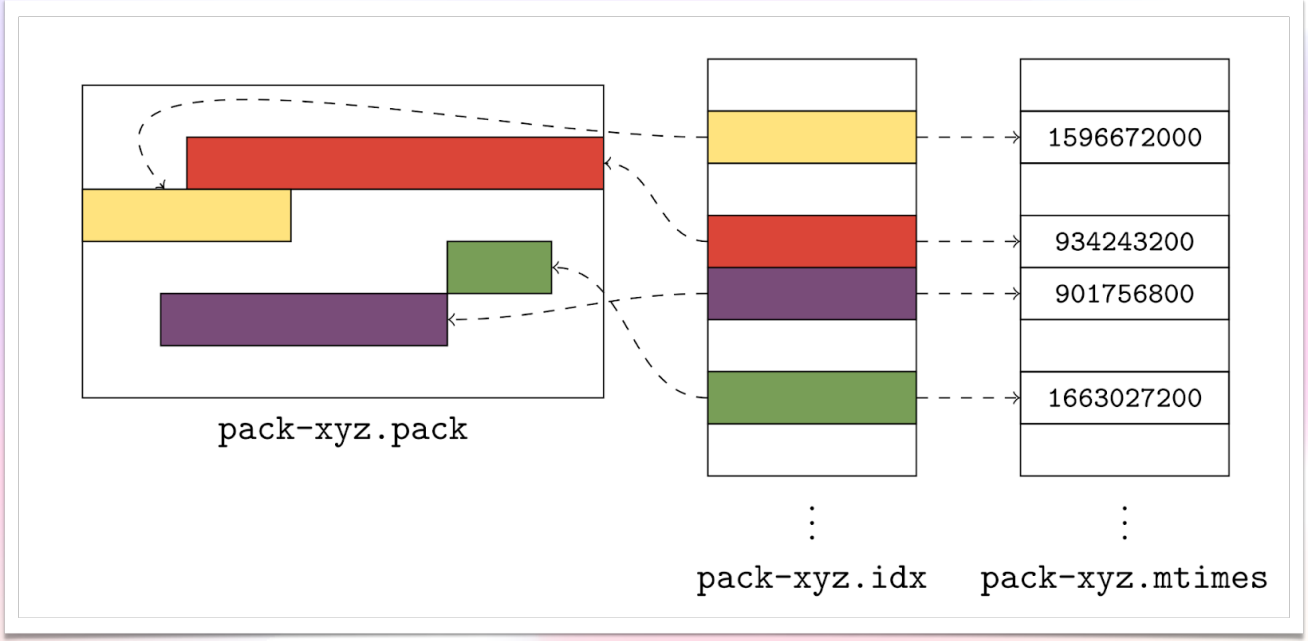
Pruning unreachable objects

- Unreachable objects which are too recent to be pruned are stored loose
- The `mtime` of each loose object file tracks the object's "age"
- Writing the object sets the `mtime` to be "now".

Pruning unreachable objects

- Storing unreachable objects loose can result in creating many files, especially in large/active repositories with many unreachable objects
- Has a handful of other drawbacks
  - Pairs of unreachable objects do not share their contents
  - Having too many files can lead to performance problems, including inode exhaustion
  - Any operation which scans loose objects slows, eventually becoming unusable

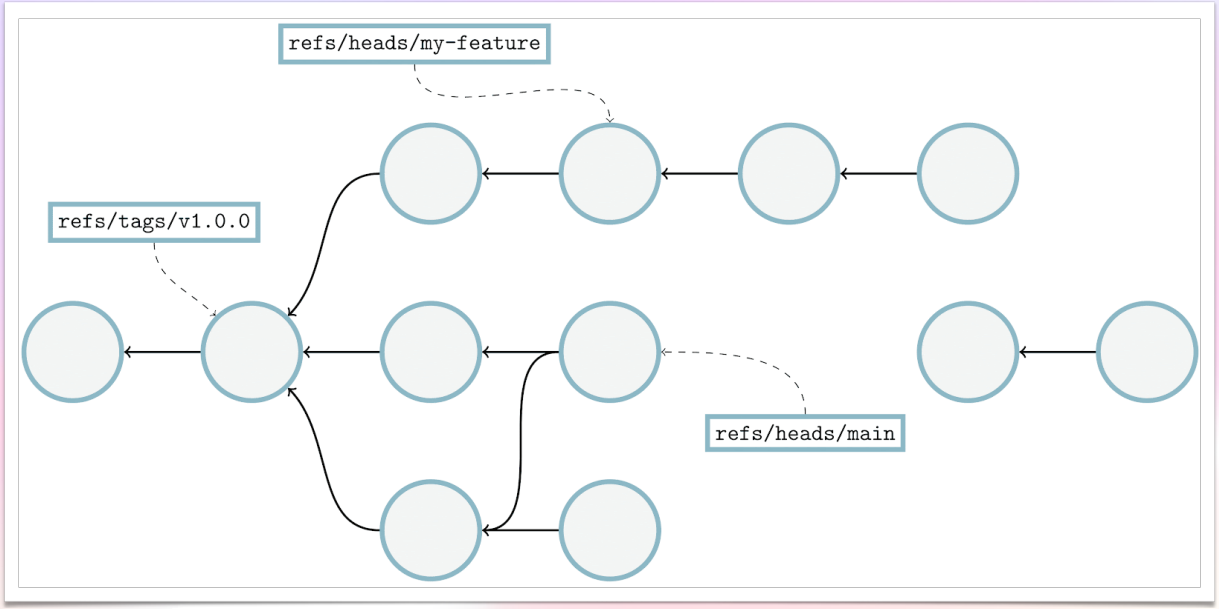
Cruft packs


1596672000
934243200
901756800
1663027200
pack-xyz.pack
pack-xyz.idx
pack-xyz.mtimes

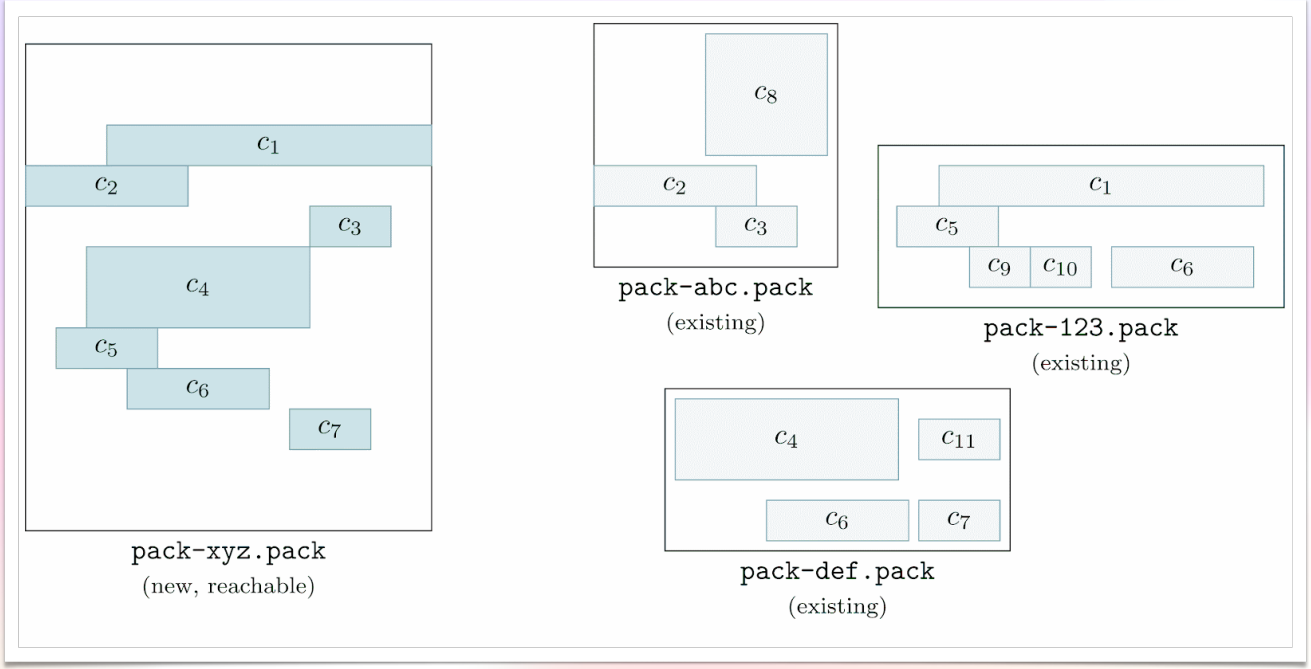
Generating cruft packs

- Two cases:
  - With object expiration
    - First mark all reachable objects, pack those separately
    - Then examine remaining unreachable objects, pack those separately along with their mtimes
  - Without object expiration
    - Same as above, but only pack recent unreachable objects
    - Before packing, traverse unreachable objects to rescue any stale objects that are reachable from recent objects

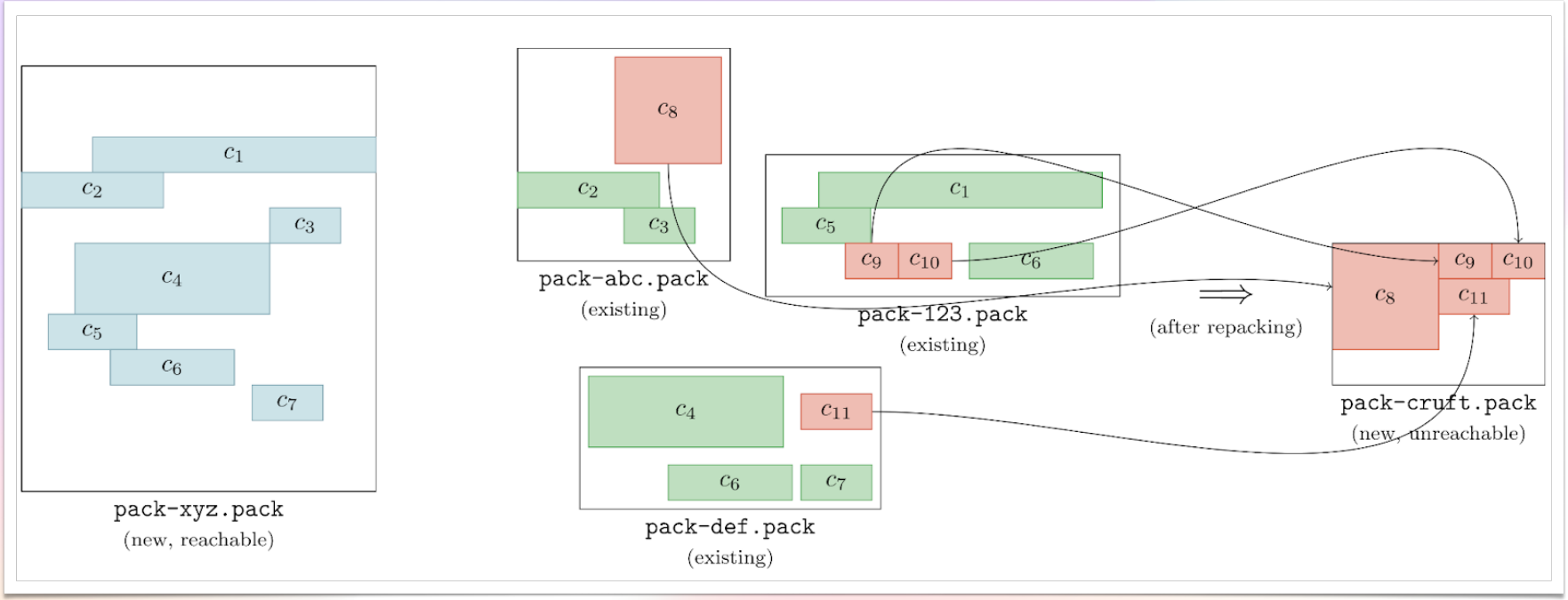
Generating cruft packs (without object expiration)



Generating cruft packs (without object expiration)

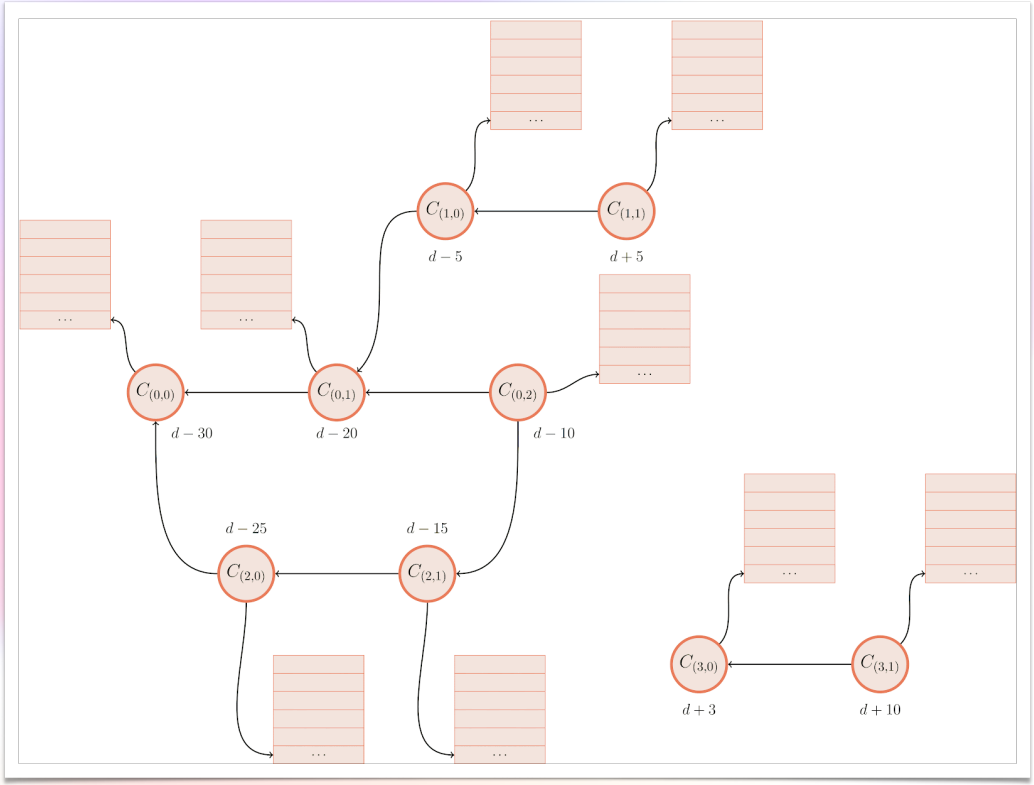


Generating cruft packs (without object expiration)



Generating cruft packs (with object expiration)

- Same procedure, except want to keep unreachable clusters of objects around
- Easy if all connected clusters will/won't be pruned
- But tricky if some objects in a cluster are pruned and others aren't
- Solution: rescuing pass to save unreachable but reachable-from-recent objects



Cruft packs at GitHub

- Typically use geometric repacking eight out of every nine maintenance runs
- Ninth maintenance job collects repositories into a reachable and cruft pack
- Normal maintenance jobs does not prune objects
  - (e.g., `git repack --cruft -dn --write-bitmap-index`)
- GitHub Support can respond to requests by running `.gc <repo>` in chat
  - (e.g., `git repack --cruft --cruft-expiration=1.minute.ago -dn --write-bitmap-index`)

Limbo repositories

- But we still have the advertise-then-prune race from earlier
- Idea: don't eliminate this race entirely, but instead make it easy to recover from
  - Joint work with Michael Haggerty and Torsten Walter
- In particular, move all unreachable objects to a "limbo" repository instead of deleting
- Then git fsck the repository to make sure no races occurred (ie., that the repository is non-corrupt)
- Restore objects from the limbo repository
- Then remove the limbo repository

Limbo repositories

- Limbo repository is “just another cruft pack” with a couple of tweaks
  - Cruft pack excludes all objects in the new reachable and cruft packs of the main repository
  - And does not prune, so all objects are picked up
- This is every object that would be pruned during GC
- Pack is written to a separate repository with experimental `--expire-to` option
- RFC patches on the mailing list

Recap

Recap

- Git is a reliable, fast, secure implementation that can be relied upon at GitHub scale.
- GitHub uses features from upstream Git, including partial clone, commit-graph, and the MIDX.
- GitHub contributes tools that it writes back to the ecosystem, including MIDX bitmaps, cruft packs, and more.
- The same tools that power GitHub can (and do!) run on your laptop.