Scaling Git

Taylor Blau (GitHub, Inc.)

Git Merge 2024

$ whoami

Taylor Blau

@ttaylorr

Staff Software Engineer
GitHub, Inc.

Agenda

Legacy repository maintenance

Geometric repacking / MIDX bitmaps

(c.f., *Git at GitHub Scale*, Git Merge 2022)

New things

- Multi-pack verbatim reuse
- Boundary-based bitmap traversal
- Pseudo-merge reachability bitmaps
- Multiple cruft packs
- Incremental MIDXs

Legacy repository maintenance

Background

- Each new push to a repository on GitHub results in a new packfile in `$GIT_DIR/objects/pack`.
- Every ~20 pushes, repository “maintenance” runs in the background.
- Runs `git repack -adkn` to repack the repository.

Why?

- Faster object lookups (O(log N) within a single pack, but O(N) across all packs in worst-case).
- Keep reachability bitmaps up-to-date for fast fetches/clones.
- Compact loose objects and references.
- Enable verbatim pack reuse optimization.

Problems

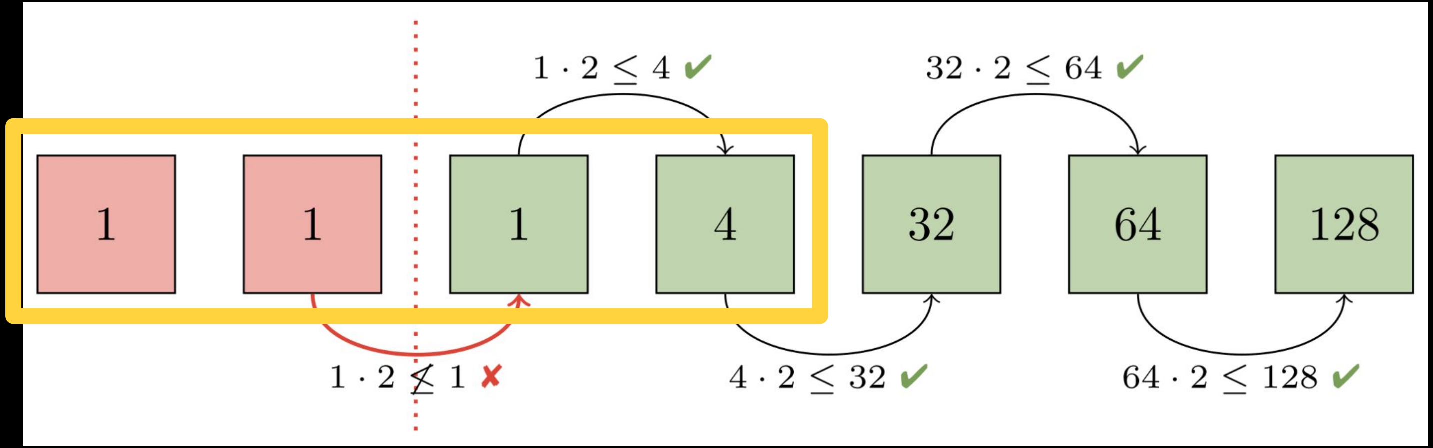
- Generates a single pack for all objects in a repository.
  - Can be slow / memory-intensive, especially in large repositories.
- Often ran into (generous) self-imposed timeouts.
- Failing to run maintenance frequently can significantly degrade repository performance.

Geometric repacking & multi-pack bitmaps

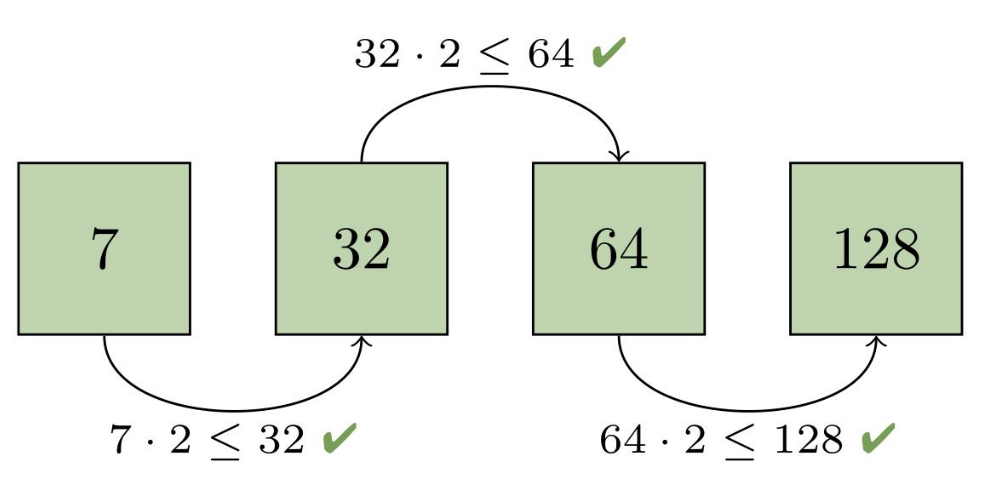
Geometric repacking

- Idea: ensure each pack contains at least twice as many objects as next-largest pack.
- Maintenance runs generally operate on recent history, avoiding expensive repacks.

Geometric repacking



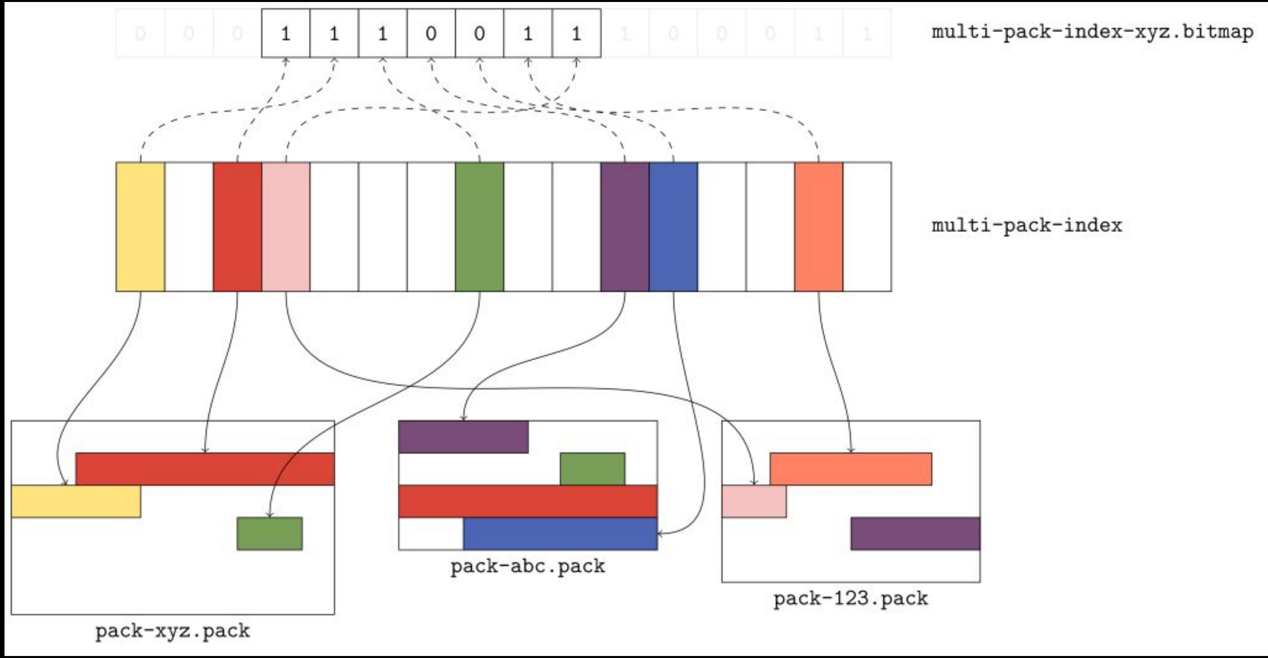
Geometric repacking



Reachability bitmaps

- Reachability bitmaps still a critical optimization.
- But which pack do we use to generate the bitmap?
  - Single-pack bitmaps can only refer to objects in one pack.
  - Can’t generate bitmaps for “new” parts of the repository based on an older pack.
- Idea: construct a “pseudo-pack” based on the multi-pack index (MIDX) which refers to all packs.

Multi-pack reachability bitmaps



Current maintenance approach

- Result: two-tiered repository maintenance routine.
  - *N* fast maintenance operations (do a geometric repack, update the MIDX).
  - 1 slow maintenance operation (generate a single pack, destroy geometric progression).

- Skipping over some details (single-, and multi-pack reverse indexes, cruft packs, etc.)
  - For more details, c.f., *Git at GitHub Scale*.

Problems

- “Fast”-tier maintenance operations still need to update their bitmaps, which requires rewriting the MIDX, which is *O(# objects)*.
- “Slow”-tier maintenance operations are likely intractable for the world’s largest repositories.
- Could we only do “fast” operations?
  - Missed delta opportunities
  - Can’t do verbatim pack reuse
  - etc.

Maintenance for any repository

New things

Multi-pack reuse
Extending verbatim pack reuse to enable storing multiple packs at rest.

Bitmap improvements
Faster bitmap traversal and reads for repositories with many references.

Multi-cruft pack support
Quickly mark objects unreachable for repositories with many such objects.

Incremental MIDX bitmaps
Fast, incremental bitmap updates that don’t require O(N) time/memory.

Multi-pack reuse

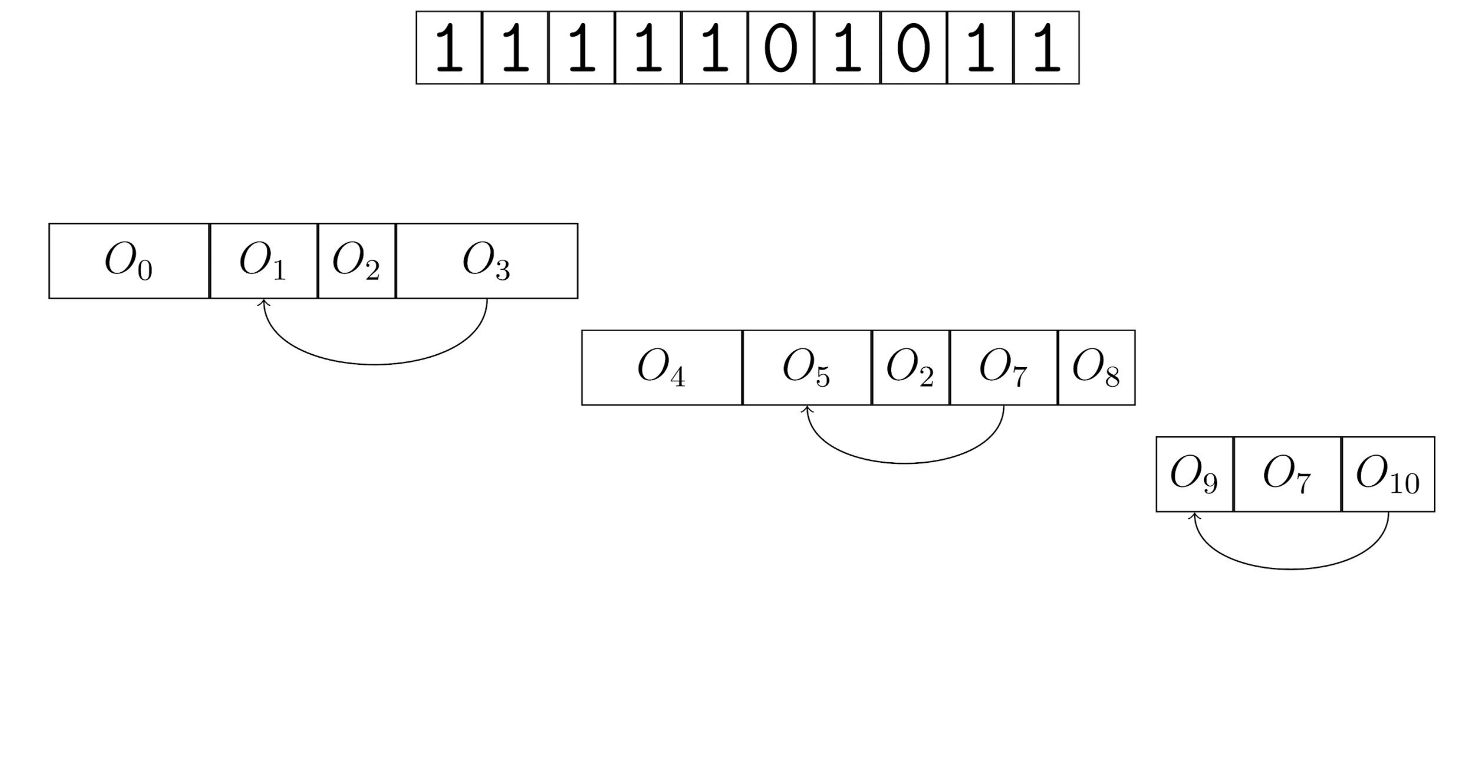
- When generating a pack (e.g., to fulfill a fetch/clone request), Git either:
  - Writes an object based on an existing copy.
  - Writes a delta based on an existing base.
  - Writes a section verbatim from an existing pack.

- Verbatim reuse occurs when the request wants a pack which contains a section similar to an existing pack.

Multi-pack reuse

- When this is the case, Git tries to stream bytes directly from a source pack to fulfill part of the fetch/clone request.
- Doing so avoids per-object bookkeeping, so is generally faster.
- ...but did not support verbatim reuse from multiple source packs.

Multi-pack reuse



Multi-pack reuse

- Copy bytes for a given object verbatim from source pack(s) to destination, iff:
  - The destination pack should include that object.
  - The source object is either a delta of an object we reused earlier, or not stored as a delta.
- Break cross-pack deltas.
- Patch `OFS_DELTA`s when there are >0 non-reused bytes between delta/base objects.

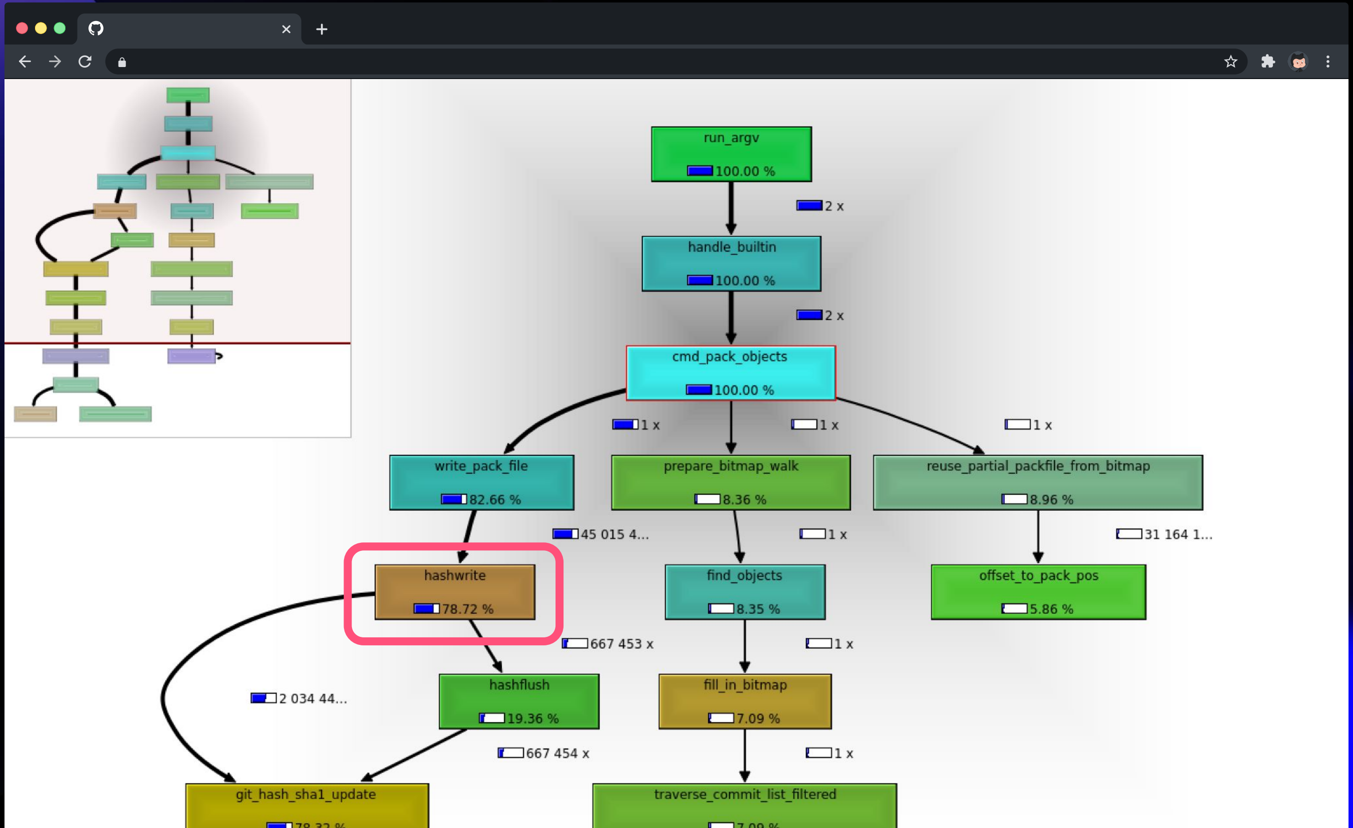
```
$ hyperfine -L v single,multi -n '{v}-pack reuse'
    'git.compile -c pack.allowPackReuse={v} pack-objects --revs --stdout
       --use-bitmap-index --delta-base-offset <in >/dev/null'
```

```
$ hyperfine -L v single,multi -n '{v}-pack reuse'
    'git.compile -c pack.allowPackReuse={v} pack-objects --revs --stdout
       --use-bitmap-index --delta-base-offset <in >/dev/null'

Benchmark 1: single-pack reuse
  Time (mean ± σ):      6.094 s ±  0.023 s    [User: 43.723 s, System: 0.358 s]
  Range (min … max):    6.063 s …  6.126 s    10 runs

Benchmark 2: multi-pack reuse
  Time (mean ± σ):     906.5 ms ±   3.2 ms    [User: 1081.5 ms, System: 30.9 ms]
  Range (min … max):   903.5 ms … 912.7 ms    10 runs

Summary
  multi-pack reuse ran
    6.72 ± 0.03 times faster than single-pack reuse
```

Non-collision detecting SHA-1

- Git uses a collision detecting SHA-1 by default.
- But noticed something peculiar when starting to use multi-pack reuse within GitHub's infrastructure...

kcachegrind of linux.git clone



Non-collision detecting SHA-1

- Git spends ~78% of CPU instructions (!) in `hashwrite()` to generate a checksum which is not used for cryptographic purposes.
- Could we use a faster, non-collision detecting SHA-1 for non-cryptographic uses only?
  - Yes, lots of subtlety discussed here, but ultimately safe.

```
$ git for-each-ref --format='%(objectname)' refs/{heads,tags} >in
$ hyperfine -L v slow,fast -n '{v} SHA-1\
    'git.{v} pack-objects --revs --stdout --all-progress --use-bitmap-index
       --delta-base-offset >/dev/null <in'
```

```
$ git for-each-ref --format='%(objectname)' refs/{heads,tags} >in
$ hyperfine -L v slow,fast -n '{v} SHA-1\
    'git.{v} pack-objects --revs --stdout --all-progress --use-bitmap-index
       --delta-base-offset >/dev/null <in'

Benchmark 1: slow SHA-1
  Time (mean ± σ):     17.414 s ±  0.118 s    [User: 17.175 s, System: 0.239 s]
  Range (min … max):   17.337 s … 17.712 s    10 runs

Benchmark 2: fast SHA-1
  Time (mean ± σ):     10.056 s ±  0.062 s    [User: 9.831 s, System: 0.225 s]
  Range (min … max):    9.955 s … 10.122 s    10 runs

Summary
  fast SHA-1 implementation ran
    1.73 ± 0.02 times faster than slow SHA-1
```

Bitmap improvements

- Ideally have coverage for all branches/tags within a repository.

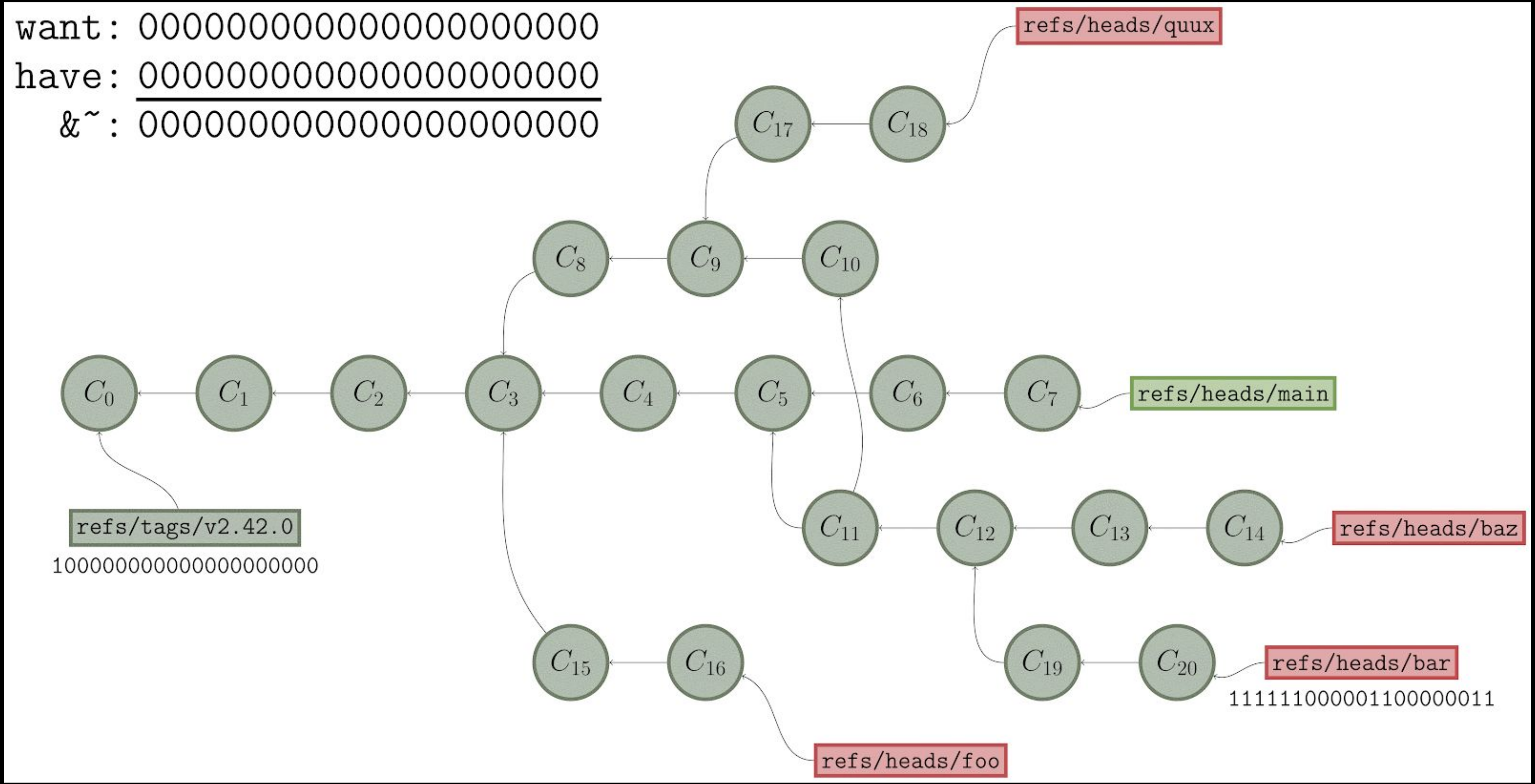
- But having a bitmap for each reference can be expensive
  - Requires lots of memory
  - Cache-inefficient, lots of time spent decompressing `EWAH` bitmaps, `XOR`-ing, etc.

- Two improvements to bitmap reads
  - Boundary-based bitmap traversal
  - Pseudo-merge reachability bitmaps

Boundary-based bitmap traversals

- Existing bitmap traversal routine:
  - Build up a complete bitmap of UNINTERESTING objects, using existing bitmaps when possible
  - Build up a bitmap of interesting objects, using existing bitmaps where possible, stopping when we “run into” any object(s) in the UNINTERESTING bitmap.
- “Demo”

Classic bitmap traversal



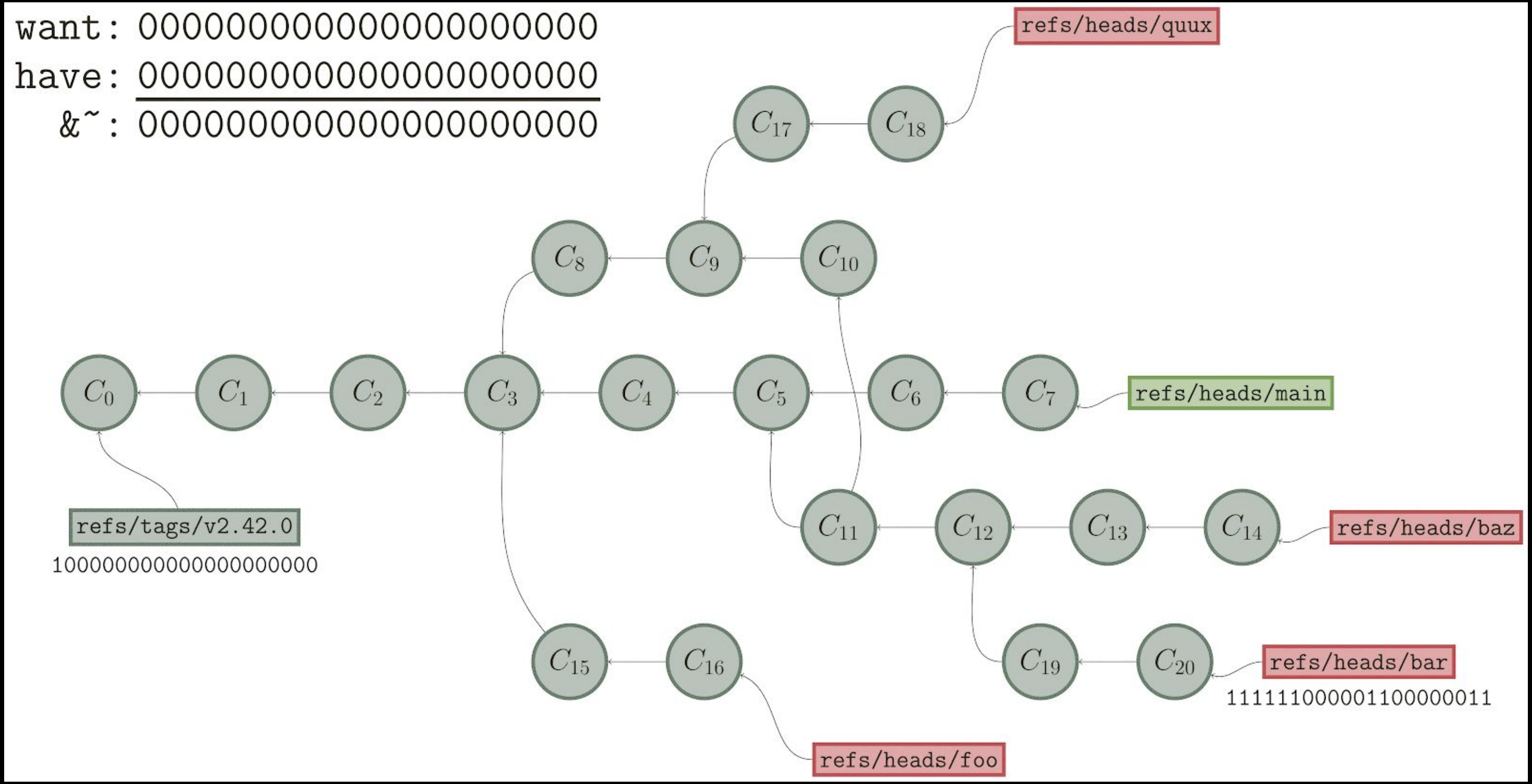
Boundary-based bitmap traversals

- With poor bitmap coverage, existing traversal can degenerate into a full object walk.

- Idea: represent the `UNINTERESTING` side of the query by the *boundary* between interesting and uninteresting objects.
    - For our purposes, *boundary* means the first commit reachable from interesting side that is also reachable from uninteresting side.

- “Demo”

Boundary-based bitmap traversal



```
$ ours="$(git branch --show-current)"
$ argv="--count --objects $ours --not --exclude=$ours --branches"
$ hyperfine \
    -n 'classic bitmap traversal' "git rev-list --use-bitmap-index $argv" \
    -n 'boundary bitmap traversal' "git.compile rev-list --use-bitmap-index $argv"
```

```
$ ours="$(git branch --show-current)"
$ argv="--count --objects $ours --not --exclude=$ours --branches"
$ hyperfine \
    -n 'classic bitmap traversal' "git rev-list --use-bitmap-index $argv" \
    -n 'boundary bitmap traversal' "git.compile rev-list --use-bitmap-index $argv"

Benchmark 1: classic bitmap traversal
  Time (mean ± σ):      82.6 ms ±   9.2 ms    [User: 63.6 ms, System: 19.0 ms]
  Range (min … max):    73.8 ms … 105.4 ms    28 runs

Benchmark 2: boundary bitmap traversal
  Time (mean ± σ):      19.8 ms ±   3.1 ms    [User: 13.0 ms, System: 6.8 ms]
  Range (min … max):    17.7 ms …  38.6 ms    158 runs

Summary
  'boundary bitmap traversal' ran
    4.17 ± 0.57 times faster than classic bitmap traversal'
```

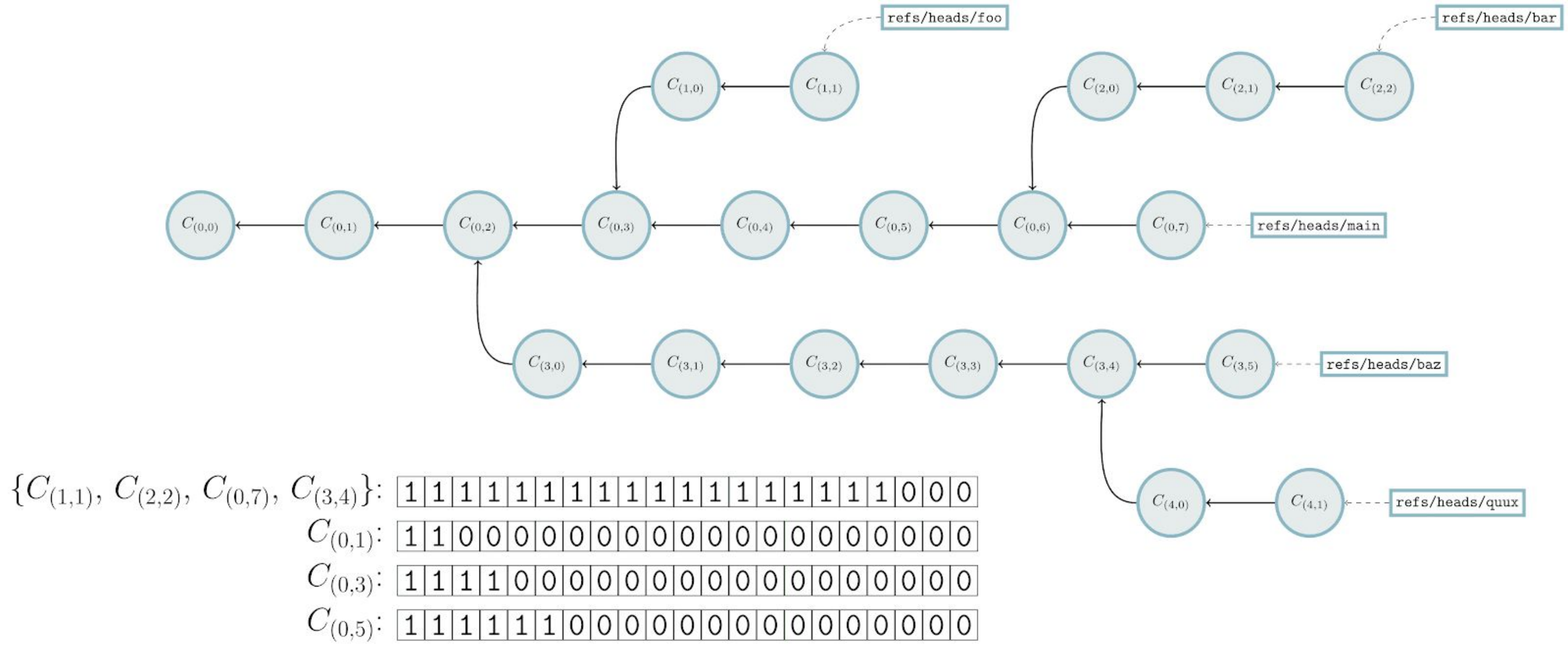
Pseudo-merge bitmaps

- Another aspect of poor bitmap coverage: lots of references limits bitmap selection.

- Suppose a user tells us they already have objects reachable from branches `A`, `B`, and `C`.
    - Ideally we have bitmaps for `A`, `B`, and `C`.
    - Storing individual bitmaps for every branch can be expensive.
    - What if we stored a single bitmap for the conceptual “merge” between `A`, `B`, and `C`?

- “Demo”

Pseudo-merge bitmaps



```
$ hyperfine -L v ,.compile 'git{v} rev-list --all --objects --count
    --use-bitmap-index'
```

```
$ hyperfine -L v ,.compile 'git{v} rev-list --all --objects --count
    --use-bitmap-index'

Benchmark 1: git rev-list --all --objects --count --use-bitmap-index
  Time (mean ± σ):     16.129 s ±  0.079 s    [User: 15.681 s, System: 0.446 s]
  Range (min … max):   16.029 s … 16.243 s    10 runs

Benchmark 2: git.compile rev-list --all --objects --count --use-bitmap-index
  Time (mean ± σ):     874.9 ms ±  20.4 ms    [User: 611.4 ms, System: 263.3 ms]
  Range (min … max):   847.1 ms … 904.3 ms    10 runs

Summary
  git.compile rev-list --all --objects --count --use-bitmap-index ran
   18.43 ± 0.44 times faster than git rev-list --all --objects --count
--use-bitmap-index
```

Multi-cruft pack support

- Cruft packs store unreachable objects with their last-modified time in a corresponding `*.mtimes` file.
  - Used to record last-modified times for unreachable objects which are too recent to prune instead of exploding as loose.

- Requires significant number of I/O-cycles to update the set of unreachable objects for large repositories.

- Solution: allow storing multiple cruft packs, use most recent mtime to break ties.

Incremental MIDX/bitmaps

- Lots of optimizations discussed so far, but...
- Updating the MIDX (& bitmaps) is still O(# objects)
- Want to get to a place where:
    - Bitmaps can be updated independently of pack generation
    - Updating bitmaps does not require rewriting existing bitmaps
    - IOW: updating bitmaps should be proportional to O(# new objects)

Incremental MIDX/bitmaps

- Idea: store the multi-pack indexes in a incremental chain
- Each layer of the chain contains a distinct set of packs/objects from previous layers
- “Object order” for bitmap generation is concatenated across multiple MIDX layers
  - Safe to do, since each layer stores a distinct set of objects

Incremental MIDX/bitmaps

- Still in development.

- Three-phase approach:
  - Phase one: support for incremental MIDXs, no bitmaps
  - Phase two: support for incremental MIDXs with bitmaps.
  - Phase three: new repacking strategy.

- Phase one is merged, phase two is in review. Phase three is still in-design.

Putting it all together

- Pre-2020 maintenance routines scale like O(# objects in repository)
- Current maintenance routines scale (mostly) like O(# new objects), but still require expensive maintenance at the end of long cycles.
- Four groups of work that will enable us to remove O(# objects) steps(s)
  - Multi-pack reuse ⇒ Break repository into multiple packs long-term without sacrificing performance.
  - Multi-cruft pack support ⇒ Cheap updates to the set of unreachable objects, regardless of size.
  - Bitmap improvements ⇒ Fast repository traversal, even with large numbers of references.
  - Incremental MIDX bitmaps ⇒ Cheap updates to reachability bitmaps, working only in recent parts of the repository.

Putting it all together

- Git repository maintenance that can scale to the world's largest repositories (and beyond).

- ...powered by tools and techniques developed at GitHub, which are shared with the open-source project.

- The same tools powering GitHub can (and do!) run on your laptop all the time.

Thank you