commit-graph

Git Commit-Graph Design Notes ==========

Git walks the commit graph for many reasons, including:

1. Listing and filtering commit history.
2. Computing merge bases.

These operations can become slow as the commit count grows. The merge base calculation shows up in many user-facing commands, such as ’merge-base’ or ’status’ and can take minutes to compute depending on history shape.

There are two main costs here:

1. Decompressing and parsing commits.
2. Walking the entire graph to satisfy topological order constraints.

The commit-graph file is a supplemental data structure that accelerates commit graph walks. If a user downgrades or disables the ’core.commitGraph’ config setting, then the existing object database is sufficient. The file is stored as "commit-graph" either in the .git/objects/info directory or in the info directory of an alternate.

The commit-graph file stores the commit graph structure along with some extra metadata to speed up graph walks. By listing commit OIDs in lexicographic order, we can identify an integer position for each commit and refer to the parents of a commit using those integer positions. We use binary search to find initial commits and then use the integer positions for fast lookups during the walk.

A consumer may load the following info for a commit from the graph:

1. The commit OID.
2. The list of parents, along with their integer position.
3. The commit date.
4. The root tree OID.
5. The generation number (see definition below).

Values 1-4 satisfy the requirements of parse_commit_gently().

There are two definitions of generation number:

1. Corrected committer dates (generation number v2)
2. Topological levels (generation number v1)

Define "corrected committer date" of a commit recursively as follows:

- A commit with no parents (a root commit) has corrected committer date equal to its committer date.
- A commit with at least one parent has corrected committer date equal to the maximum of its committer date and one more than the largest corrected committer date among its parents.
- As a special case, a root commit with timestamp zero has corrected commit date of 1, to be able to distinguish it from GENERATION_NUMBER_ZERO (that is, an uncomputed corrected commit date).

Define the "topological level" of a commit recursively as follows:

- A commit with no parents (a root commit) has topological level of one.
- A commit with at least one parent has topological level one more than the largest topological level among its parents.

Equivalently, the topological level of a commit A is one more than the length of a longest path from A to a root commit. The recursive definition is easier to use for computation and observing the following property:

```
If A and B are commits with generation numbers N and M, respectively,
and N <= M, then A cannot reach B. That is, we know without searching
that B is not an ancestor of A because it is further from a root commit
than A.

Conversely, when checking if A is an ancestor of B, then we only need
to walk commits until all commits on the walk boundary have generation
number at most N. If we walk commits using a priority queue seeded by
generation numbers, then we always expand the boundary commit with highest
generation number and can easily detect the stopping condition.
```

The property applies to both versions of generation number, that is both corrected committer dates and topological levels.

This property can be used to significantly reduce the time it takes to walk commits and determine topological relationships. Without generation numbers, the general heuristic is the following:

```
If A and B are commits with commit time X and Y, respectively, and
X < Y, then A _probably_ cannot reach B.
```

In absence of corrected commit dates (for example, old versions of Git or mixed generation graph chains), this heuristic is currently used whenever the computation is allowed to violate topological relationships due to clock skew (such as "git log" with default order), but is not used when the topological order is required (such as merge base calculations, "git log --graph").

In practice, we expect some commits to be created recently and not stored in the commit-graph. We can treat these commits as having "infinite" generation number and walk until reaching commits with known generation number.

We use the macro GENERATION_NUMBER_INFINITY to mark commits not in the commit-graph file. If a commit-graph file was written by a version of Git that did not compute generation numbers, then those commits will have generation number represented by the macro GENERATION_NUMBER_ZERO = 0.

Since the commit-graph file is closed under reachability, we can guarantee the following weaker condition on all commits:

```
If A and B are commits with generation numbers N and M, respectively,
and N < M, then A cannot reach B.
```

Note how the strict inequality differs from the inequality when we have fully-computed generation numbers. Using strict inequality may result in walking a few extra commits, but the simplicity in dealing with commits with generation number **_INFINITY or** _ZERO is valuable.

We use the macro GENERATION_NUMBER_V1_MAX = 0x3FFFFFFF for commits whose topological levels (generation number v1) are computed to be at least this value. We limit at this value since it is the largest value that can be stored in the commit-graph file using the 30 bits available to topological levels. This presents another case where a commit can have generation number equal to that of a parent.

Design Details ----------

- The commit-graph file is stored in a file named ’commit-graph’ in the .git/objects/info directory. This could be stored in the info directory of an alternate.
- The core.commitGraph config setting must be on to consume graph files.
- The file format includes parameters for the object ID hash function, so a future change of hash algorithm does not require a change in format.
- Commit grafts and replace objects can change the shape of the commit history. The latter can also be enabled/disabled on the fly using `--no-replace-objects`. This leads to difficulty storing both possible interpretations of a commit id, especially when computing generation numbers. The commit-graph will not be read or written when replace-objects or grafts are present.

- Shallow clones create grafts of commits by dropping their parents. This leads the commit-graph to think those commits have generation number 1. If and when those commits are made unshallow, those generation numbers become invalid. Since shallow clones are intended to restrict the commit history to a very small set of commits, the commit-graph feature is less helpful for these clones, anyway. The commit-graph will not be read or written when shallow commits are present.

Commit-Graphs Chains ----------

Typically, repos grow with near-constant velocity (commits per day). Over time, the number of commits added by a fetch operation is much smaller than the number of commits in the full history. By creating a "chain" of commit-graphs, we enable fast writes of new commit data without rewriting the entire commit history — at least, most of the time.

File Layout ~~~~~~~~~~~

A commit-graph chain uses multiple files, and we use a fixed naming convention to organize these files. Each commit-graph file has a name `$OBJDIR/info/commit-graphs/graph-{hash}.graph` where `{hash}` is the hex- valued hash stored in the footer of that file (which is a hash of the file's contents before that hash). For a chain of commit-graph files, a plain-text file at `$OBJDIR/info/commit-graphs/commit-graph-chain` contains the hashes for the files in order from "lowest" to "highest".

For example, if the `commit-graph-chain` file contains the lines

```
    {hash0}
    {hash1}
    {hash2}
```

then the commit-graph chain looks like the following diagram:

```
+-----------------------+
|  graph-{hash2}.graph  |
+-----------------------+
     |
+-----------------------+
|                       |
|  graph-{hash1}.graph  |
|                       |
+-----------------------+
     |
+-----------------------+
|                       |
|                       |
```

```
|                       |
|  graph-{hash0}.graph  |
|                       |
|                       |
|                       |
+-----------------------+
```

Let X0 be the number of commits in `graph-{hash0}.graph`, X1 be the number of commits in `graph-{hash1}.graph`, and X2 be the number of commits in `graph-{hash2}.graph`. If a commit appears in position i in `graph-{hash2}.graph`, then we interpret this as being the commit in position (X0 + X1 + i), and that will be used as its "graph position". The commits in `graph-{hash2}.graph` use these positions to refer to their parents, which may be in `graph-{hash1}.graph` or `graph-{hash0}.graph`. We can navigate to an arbitrary commit in position j by checking its containment in the intervals [0, X0), [X0, X0 + X1), [X0 + X1, X0 + X1
X2).

Each commit-graph file (except the base, `graph-{hash0}.graph`) contains data specifying the hashes of all files in the lower layers. In the above example, `graph-{hash1}.graph` contains `{hash0}` while `graph-{hash2}.graph` contains `{hash0}` and `{hash1}`.

Merging commit-graph files ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

If we only added a new commit-graph file on every write, we would run into a linear search problem through many commit-graph files. Instead, we use a merge strategy to decide when the stack should collapse some number of levels.

The diagram below shows such a collapse. As a set of new commits are added, it is determined by the merge strategy that the files should collapse to `graph-{hash1}`. Thus, the new commits, the commits in `graph-{hash2}` and the commits in `graph-{hash1}` should be combined into a new `graph-{hash3}` file.

```
                +---------------------+
                |                     |
                |    (new commits)    |
                |                     |
                +---------------------+
                |                     |
  +-------------------------+  +---------------------+
  |  graph-{hash2}          |->|                     |
  +-------------------------+  +---------------------+
       |                  |                     |
  +-------------------------+  +---------------------+
  |                         |  |                     |
```

```
|  graph-{hash1}        |->|                     |
|                       |  |                     |
+-----------------------+  +---------------------+
     |                  tmp_graphXXX
+-----------------------+
|                       |
|                       |
|                       |
|  graph-{hash0}        |
|                       |
|                       |
|                       |
+-----------------------+
```

During this process, the commits to write are combined, sorted and we write the contents to a temporary file, all while holding a `commit-graph-chain.lock` lock-file. When the file is flushed, we rename it to `graph-{hash3}` according to the computed `{hash3}`. Finally, we write the new chain data to `commit-graph-chain.lock`:

```
    {hash3}
    {hash0}
```

We then close the lock-file.

Merge Strategy ~~~~~~~~~~~~~~~

When writing a set of commits that do not exist in the commit-graph stack of height N, we default to creating a new file at level N + 1. We then decide to merge with the Nth level if one of two conditions hold:

```
1. `--size-multiple=<X>` is specified or X = 2, and the number of commits in
   level N is less than X times the number of commits in level N + 1.


2. `--max-commits=<C>` is specified with non-zero C and the number of commits
   in level N + 1 is more than C commits.
```

This decision cascades down the levels: when we merge a level we create a new set of commits that then compares to the next level.

The first condition bounds the number of levels to be logarithmic in the total number of commits. The second condition bounds the total number of commits in a `graph-{hashN}` file and not in the `commit-graph` file, preventing significant performance issues when the stack merges and another process only partially reads the previous stack.

The merge strategy values (2 for the size multiple, 64,000 for the maximum number of commits) could be extracted into config settings for full flexibility.

Handling Mixed Generation Number Chains
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

With the introduction of generation number v2 and generation data chunk, the following scenario is possible:

1. "New" Git writes a commit-graph with the corrected commit dates.
2. "Old" Git writes a split commit-graph on top without corrected commit dates.

A naive approach of using the newest available generation number from each layer would lead to violated expectations: the lower layer would use corrected commit dates which are much larger than the topological levels of the higher layer. For this reason, Git inspects the topmost layer to see if the layer is missing corrected commit dates. In such a case Git only uses topological level for generation numbers.

When writing a new layer in split commit-graph, we write corrected commit dates if the topmost layer has corrected commit dates written. This guarantees that if a layer has corrected commit dates, all lower layers must have corrected commit dates as well.

When merging layers, we do not consider whether the merged layers had corrected commit dates. Instead, the new layer will have corrected commit dates if the layer below the new layer has corrected commit dates.

While writing or merging layers, if the new layer is the only layer, it will have corrected commit dates when written by compatible versions of Git. Thus, rewriting split commit-graph as a single file (`--split=replace`) creates a single layer with corrected commit dates.

Deleting graph-{hash} files ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

After a new tip file is written, some `graph-{hash}` files may no longer be part of a chain. It is important to remove these files from disk, eventually. The main reason to delay removal is that another process could read the `commit-graph-chain` file before it is rewritten, but then look for the `graph-{hash}` files after they are deleted.

To allow holding old split commit-graphs for a while after they are unreferenced, we update the modified times of the files when they become unreferenced. Then, we scan the `$OBJDIR/info/commit-graphs/` directory for `graph-{hash}` files whose modified times are older than a given expiry window. This window defaults to zero, but can be changed using command-line arguments or a config setting.

Chains across multiple object directories
~~~~~~

In a repo with alternates, we look for the `commit-graph-chain` file starting in the local object directory and then in each alternate. The first file that exists defines our chain. As we look for the `graph-{hash}` files for each `{hash}` in the chain file, we follow the same pattern for the host directories.

This allows commit-graphs to be split across multiple forks in a fork network. The typical case is a large "base" repo with many smaller forks.

As the base repo advances, it will likely update and merge its commit-graph chain more frequently than the forks. If a fork updates their commit-graph after the base repo, then it should "reparent" the commit-graph chain onto the new chain in the base repo. When reading each `graph-{hash}` file, we track the object directory containing it. During a write of a new commit-graph file, we check for any changes in the source object directory and read the `commit-graph-chain` file for that source and create a new file based on those files. During this "reparent" operation, we necessarily need to collapse all levels in the fork, as all of the files are invalid against the new base file.

It is crucial to be careful when cleaning up "unreferenced" `graph-{hash}.graph` files in this scenario. It falls to the user to define the proper settings for their custom environment:

```
1. When merging levels in the base repo, the unreferenced files may still be
   referenced by chains from fork repos.

2. The expiry time should be set to a length of time such that every fork has
   time to recompute their commit-graph chain to "reparent" onto the new base
   file(s).

3. If the commit-graph chain is updated in the base, the fork will not have
   access to the new chain until its chain is updated to reference those files.
   (This may change in the future [5].)
```

Related Links ---------- [0] https://bugs.chromium.org/p/git/issues/detail?id=8 Chromium work item for: Serialized Commit Graph

[1] https://lore.kernel.org/git/20110713070517.GC18566@sigill.intra.peff.net/ An abandoned patch that introduced generation numbers.

[2] https://lore.kernel.org/git/20170908033403.q7e6dj7benasrjes@sigill.intra.peff.net/ Discussion about generation numbers on commits and how they interact with fsck.

[3] https://lore.kernel.org/git/20170908034739.4op3w4f2ma5s65ku@sigill.intra.peff.net/ More discussion about generation numbers and not storing them inside commit objects. A valuable quote:

```
"I think we should be moving more in the direction of keeping
 repo-local caches for optimizations. Reachability bitmaps have been
 a big performance win. I think we should be doing the same with our
 properties of commits. Not just generation numbers, but making it
 cheap to access the graph structure without zlib-inflating whole
 commit objects (i.e., packv4 or something like the "metapacks" I
 proposed a few years ago)."
```

[4] https://lore.kernel.org/git/20180108154822.54829-1-git@jeffhostetler.com/T/#u A patch to remove the ahead-behind calculation from ’status’.

[5] https://lore.kernel.org/git/f27db281-abad-5043-6d71-cbb083b1c877@gmail.com/ A discussion of a "two-dimensional graph position" that can allow reading multiple commit-graph chains at the same time.

□□□ #2
□ □□□ □□□ 2026-10-31 20:16:24 UTC
□ □□□ □□□ 2026-10-31 20:16:26 UTC