Version control post-Git

Pierre-Étienne Meunier (Coturnix, Pijul)

February 4th, 2023

FOSDEM 2024

Plan

Version control

Our solution

Implementation

Post-git, really?

What is version control?

- One or more coauthors edit a tree of documents concurrently
- Asynchronous edits: coauthors can choose when they want to “sync” or “merge”
- Edits may *conflict*
- Review a project’s history

A solved problem?

Our tools (Git, Hg, SVN, CVS...):

- Aren't used by non-coders, despite their maturity (30 years+)
- Are distributed, yet most of the time used with a global central server:

  All paths may not lead to Chrome, but can the same be said for GitHub?
- Require strong work discipline and planning
- Waste significant human worktime at a global scale

Improvements have been proposed (Darcs) but don't really scale.

Is there a quick fix?

- Leaky abstractions: if Merkle trees are the core mechanism, they can't be hidden from the user.[1]
- Strict ordering of snapshots is the main feature, yet the most used Git commands (rebase, rerere, cherry-pick...) are "fixes" around that "feature".

[1] Credit: Raphaël Gomès, Mercurial core team

Some symptoms that it may not be a solved problem

- Inflation of commands and options:
  `https://git-man-page-generator.lokaltog.net`
- Inflation of UIs: even “big tech” is now investing in Git/Mercurial UIs.
- Inflation of forges: how many started in the last year alone? (vs how many text editors? window managers?)

Our demands

- Associative merges:

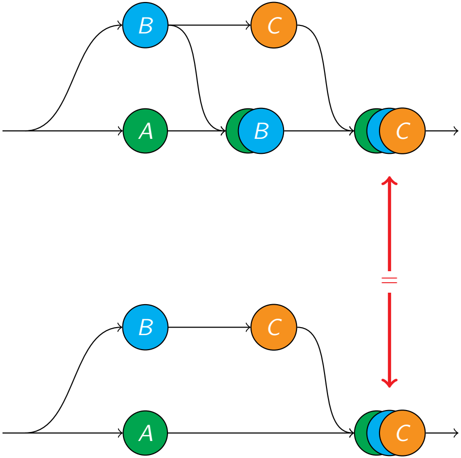
  Changes A and B together are the same as A, followed by B.

- Commutative merges:

  If A and B can *be produced* independently, *their order does not matter.*

- Branches (or maybe not: more on that later)
- Low algorithmic complexity, and ideally fast implementations

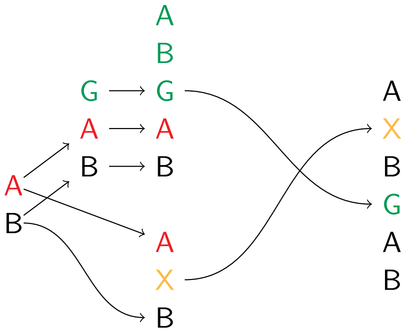
Associative merges, a.k.a "one-by-one review"



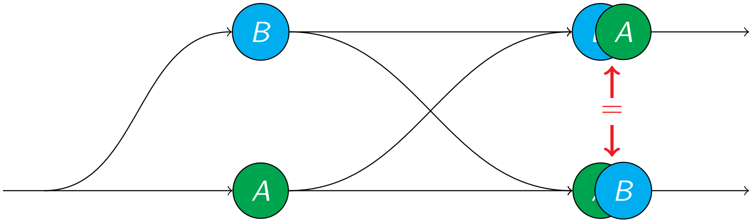
So you think you know Git merge?

3-way merge (Git, Hg, SVN, CVS...) is not associative

Workflow: review your PRs, then merge and then review them again



Commutative merges



Git and SVN are *never* commutative, why would we want this?

- *Unapplying* old changes, even after others have been applied.
- *Cherry-picking*.
- *Partial clones*: pull the patches related to a subproject, or merge repos transparently.

States vs changes

- Git, Hg, SVN, CVS... store *states*, and compute *changes* when needed (3-way merge).
- What if we did the *opposite*?
- What if we stored *both*?

A change-based idea: Operational Transforms

$$\begin{tikzcd} \text{abc} \arrow[r, "T2 = \text{del}(2, \text{``c''})"] \arrow[d, "T1 = \text{ins}(0, \text{``x''})"'] & \text{ab} \arrow[d, "T1"] \\ \text{xabc} \arrow[r, "T2' = \text{del}({\color{red}3}, \text{``c''})"'] & \text{xab} \end{tikzcd}$$

- *Darcs* does this, and uses it to detect conflicts
- Quadratic explosion of cases
- A nightmare to implement

A hybrid (state/change) approach: CRDTs

- General principle: design a structure where all operations have the properties we want
- Natural examples: increment-only counters, insert-only sets...
- More subtle: tombstones, Lamport clocks...
- Useless: a full Git repository (not just `HEAD`)

Plan

Version control

Our solution

Implementation

Post-git, really?

Conflicts

- Where we need a good tool the most
- The exact definition depends on the tool
- *Example:* Alice and Bob write to the same file at the same place
- *Example:* Alice renames a file from f to g while Bob renames f to h
- *Example:* Alice renames a function f while Bob adds a call to f

Using category theory

For any two patches f and g, we want a unique state P such that:

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "g"'] & Y \arrow[d] \\ Z \arrow[r] & P \end{tikzcd}$$

Started by Samuel Mimram and Cinzia Di Giusto

Using category theory

For any two patches f and g, we want a unique state P such that:
For any state Q accessible by Alice and Bob after f and g, respectively

$$
\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"'] & Y \arrow[d] \arrow[ddr, bend left, "\forall"] & \\
Z \arrow[r] \arrow[drr, bend right, "\forall"'] & P & \\
& & Q
\end{tikzcd}
$$

Started by Samuel Mimram and Cinzia Di Giusto

Using category theory

For any two patches f and g, we want a unique state P such that:
For any state Q accessible by Alice and Bob after f and g, respectively
There is a patch from P to Q.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"'] & Y \arrow[d] \arrow[ddr, bend left, "\forall"] & \\
Z \arrow[r] \arrow[drr, bend right, "\forall"'] & P \arrow[dr, dashed, "\exists"] & \\
 & & Q
\end{tikzcd}$$

If P exists, we call P the *pushout* of f and g.

Started by Samuel Mimram and Cinzia Di Giusto

Problem: the pushout doesn't always exist

- Equivalent to saying that conflicts happen.
- How to generalise the representation of states (X, Y, Z) so that all pairs of changes (f and g) have a pushout?

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "g"'] & Y \arrow[d] \\ Z \arrow[r] & P \end{tikzcd}$$

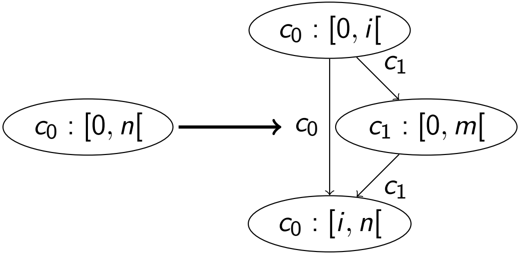
Solution: States are directed graphs, where:

- Vertices are bytes (or byte intervals).
- Edges represent the union of all known orders between bytes.

Adding some bytes

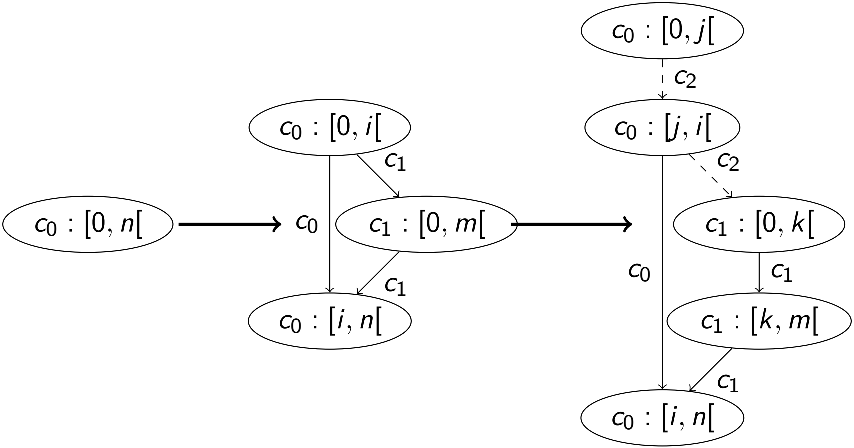
- Vertices are labelled by a change number c_0 and an interval (such as $[0, n[$) in that change.
- Edges are labelled by the change that introduced them.

Here, c_1 adds m bytes between positions $i - 1$ and i of c_0:



Deleting bytes

Deleting bytes j to i from c_0, and 0 to k from c_1:



That's all we need!

Two kinds of changes:

- Add a vertex, in a *context* (parents and children)
- Change an edge's label

Our definition of conflicts

- *Alive* vertices are vertices whose incoming edges are all alive.
- *Dead* vertices are vertices whose incoming edges are all dead.
- Other vertices are called *zombies*.

A graph has *no conflict* if and only if it has no zombie
and all its alive vertices are totally ordered.

Notes

- Changes are partially ordered by their dependencies on other changes.
- Cherry-picking is the same as applying a patch.
- No `git rerere`: conflicts are solved by changes, which can be cherry-picked.
- Partial clones/monorepos/submodules: easy as long as "wide" patches are disallowed.
- Large files: the description of operations (insertions/deletions) is not even stored in the graph.

Plan

Version control

Our solution

Implementation

Post-git, really?

Working with large graphs on disk

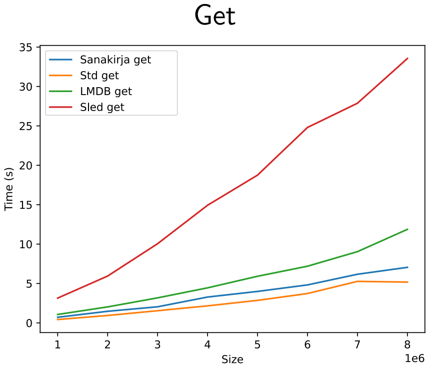
- We can't load the entire graph each time.
- Store edges in a key-value store.
- Transactions: passive crash-safety.
- Branches: efficiently forkable store.

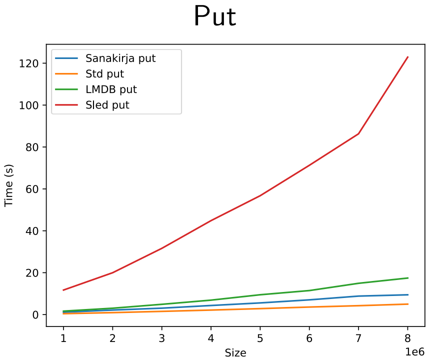
Introducing *Sanakirja*, an on-disk transactional KV store

- ACID block allocator in a file
- Crash-safety using referential transparency and copy-on-write.
- Forkable in $O(\log n)$, where n is the total size.
- Written in Rust, allowing direct pointers to generic types stored in the file.
- Generic underlying storage layer: we've used it on memory-mapped files, zstd-compressed files, Cloudflare KV...
- But: tricky API, conflicting with most aspects of the Rust memory model (not completely avoidable).

Sanakirja is the fastest we've tested

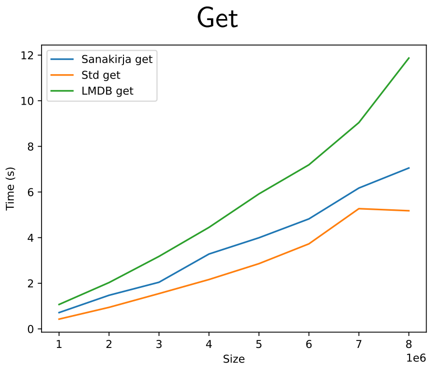
- Performance of retrieval (get) and insertion (put) into a B tree.
- Not specific to Pijul.

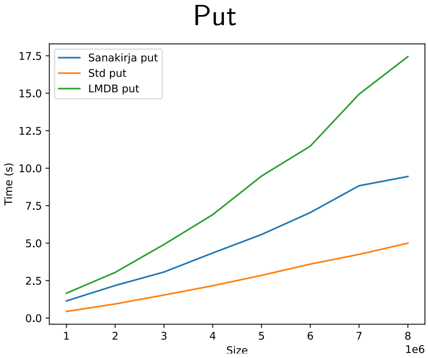




Sanakirja is the fastest we've tested

- Performance of retrieval (get) and insertion (put) into a B tree.
- Not specific to Pijul.





Modular databases

- Sanakirja is actually just a transactional block allocator with reference-counting included.
- I have built on-disk R trees, Patricia trees (text search!), Ropes.
- Composite types: Pijul stores branches as (roughly) a `BTree<String, BTree<Vertex, Edge>>`.

  I have a prototype text editor with forkable files, its type is `BTree<String, (Rope, BTree<Vertex, Edge>)>`.

Interested in datastructures and performance challenges? **Join us!**

Plan

Version control

Our solution

Implementation

Post-git, really?

Things we get for free

- Superfast `pijul credit`[2]: info readily available in the graph
- Have your bugfixes on your main branch.
- Submodules for free: changes on unrelated projects are commutative!
- Signing + identity: your identity is your public key. Patches signed by default, identity (email/name/...) changes for free.
- Free cherry-picking: just apply that patch, no need to change its identity.
- Almost free scalability, no Rube Goldberg machine needed.

[2] Stop blaming your coauthors!

Commutative state identifiers

- We want to check repo states equality, even with different orders.
- We want to compute each state identifier in constant time from the previous state id and a patch.
- We want states to be hard to forge.

Solution: **discrete log on elliptic curves!**

Turn each patch identity h into an integer, and have the state with patches $h_0, h_1, \ldots, h_n$ be identified by $e^{h_0 \cdot h_1 \cdot \ldots \cdot h_n}$.

Towards a hybrid state/patch system

- In Git/SVN/CVS/Hg, commits are *states*, not changes, even though patches can be applied and recomputed.
- Darcs only has changes, and recomputes states as needed.
- Pijul has both: a data structure modelling the current state, but it was found from the patches and is therefore completely transparent.

Towards a hybrid state/patch system: ongoing projects

- **Lightweight tags** to add super fast history browsing, while retaining all the good properties of patches.

  Current tags: Sanakirja, but using a compressed file as a backend rather than the raw disk.

- **Patch groups**, i.e. keywords to describe features, allowing patches on the same branch to be handled (pushed) independently, even when interspersed with others.

- **Cues** to avoid half-merged states when merging a series of patches.

Help us!

- This is currently a large project with a small team, but proper maths can make that work.
- Bootstrapped (used for itself) since 2017.
- Documentation, accessibility, UI, bikeshedding...
- "Good first bugs" tags on nest.pijul.com/pijul/pijul to get acquainted with our codebase.
- `https://pijul.zulipchat.com`

Conclusion

- Open Source version control based on algorithms and theorems.
- Scalable to monorepos and large files.
- Potentially usable by non-coders: parliaments, artists, lawyers, Sonic Pi composers, LEGO builders...
- Repo hosting service available: nest.pijul.com

Acknowledgements: Florent Becker, Tankf33der, Rohan Hart, Chris Bailey, Angus Finch...