code basicstyle=, frame=single, rulecolor=, breaklines=true, showstringspaces=false, columns=fullflexible, keepspaces=true, framesep=6pt, aboveskip=1em, belowskip=1em, xleftmargin=0pt

out basicstyle=, frame=tb, rulecolor=, breaklines=true, showstringspaces=false, columns=fullflexible, keepspaces=true, framesep=4pt, aboveskip=1em, belowskip=1em

Content-Addressable Storage
Fondasi Penyimpanan pada Sistem Kontrol Versi

Muhammad Riduwan Khafidi
Program Studi Teknik Rekayasa Perangkat Lunak
Politeknik Negeri Batam
4342611034@students.polibatam.ac.id

November 5, 2026

Abstrak

Content-addressable storage adalah model penyimpanan di mana alamat setiap objek ditentukan oleh isi objek itu sendiri, bukan oleh lokasi fisik atau nomor urut. Model ini menjadi fondasi Git, Mercurial, dan sistem kontrol versi modern karena memungkinkan deduplikasi otomatis, immutability, dan verifikasi integritas bawaan. Makalah ini membedah mekanisme content-addressable storage dalam sistem kontrol versi dari empat perspektif: hash sebagai alamat, empat tipe objek dasar, struktur Merkle yang menjamin immutability, serta siklus hidup satu commit dari pembuatan hingga penyimpanan. Setiap perspektif dilengkapi analogi konkret, contoh berangka, dan sembilan visualisasi TikZ yang menyoroti hubungan antar komponen. Melalui kerangka ini, pembaca memperoleh mental model untuk membaca implementasi VCS mana pun, dari Git hingga sistem kustom seperti libvctrl.

1 Pendahuluan

1.1 Analogi Pembuka

Bayangkan sebuah perpustakaan di mana setiap buku tidak diberi nomor rak, tidak diberi kode katalog, dan tidak diberi label tanggal masuk. Setiap buku justru diidentifikasi oleh isinya sendiri: judul, penulis, dan seluruh teksnya diolah menjadi sidik jari unik. Jika dua buku identik datang, pustakawan tidak menyimpan dua salinan; ia menyimpan satu, karena keduanya memiliki sidik jari yang sama. Jika isi sebuah buku berubah sedikit saja, sidik jarinya berubah total, sehingga buku lama dan buku baru menempati alamat yang berbeda.

Perpustakaan itu adalah content-addressable storage. Setiap buku adalah objek. Sidik jari adalah hash kriptografis. Alamatnya bukan nomor urut, melainkan hash dari isi. Sistem kontrol versi modern seperti Git dan libvctrl menggunakan model ini untuk menyimpan seluruh riwayat proyek: setiap file, setiap direktori, dan setiap commit memiliki alamat yang ditentukan oleh isinya, bukan oleh kapan atau di mana ia disimpan.

1.2 Mengapa Topik Ini Penting

Seorang pengembang yang memahami content-addressable storage dapat menjelaskan mengapa Git tidak menggandakan file saat berpindah branch, mengapa commit yang isinya sama selalu memiliki hash yang sama, dan mengapa sebuah file yang sudah dihapus dari working directory tetap ada di riwayat selama masih dirujuk oleh commit. Tanpa pemahaman ini, pengembang cenderung memperlakukan Git sebagai kotak hitam, memakai perintah tanpa mengerti biaya dan jaminan yang melekat padanya.

Selain itu, model ini adalah jawaban paling bersih untuk dua masalah lama dalam penyimpanan data: deduplikasi dan integritas. Sistem berkas konvensional menyimpan dua file identik sebagai dua salinan, dan tidak memiliki cara bawaan untuk mendeteksi korupsi. Content-addressable storage menyimpan satu salinan untuk setiap isi unik, dan mendeteksi korupsi secara otomatis karena hash dari isi yang rusak tidak akan cocok dengan alamat yang tercatat.

1.3 Peta Pembahasan

Makalah ini dibagi menjadi empat bagian konsep. Bagian kedua membahas hash sebagai alamat, termasuk properti kriptografis yang membuatnya layak dipercaya. Bagian ketiga membahas empat tipe objek dasar pada object store: `blob`, `tree`, `commit`, dan `tag`. Bagian keempat membahas struktur Merkle yang menghubungkan objek-objek tersebut dan immutability sebagai sifat bawaan. Bagian kelima menyandingkan model content-addressable dengan model address-based konvensional. Bagian keenam membawa seluruh konsep ke studi kasus delapan langkah menyimpan satu commit dari nol. Bagian ketujuh dan kedelapan merangkum kesalahan umum dan lembar rujukan cepat. Bagian kesembilan menutup dengan tiga prinsip.

2 Hash sebagai Alamat Konten

2.1 Definisi dan Analogi

Hash kriptografis adalah fungsi yang menerima input dengan panjang berapa pun dan menghasilkan output dengan panjang tetap, dengan jaminan bahwa input yang berbeda menghasilkan output yang berbeda hampir sepanjang waktu. Pada content-addressable storage, output inilah yang menjadi alamat.

Analogi paling dekat adalah cap jari manusia. Dua orang bisa memiliki nama yang sama, tetapi tidak bisa memiliki cap jari yang sama. Nama adalah alamat yang bisa bertabrakan; cap jari adalah alamat yang hampir tidak mungkin bertabrakan. Content-addressable storage memakai cap jari, bukan nama.

2.2 Mengapa Hash, Bukan Nomor Urut

Sistem berkas konvensional memberi alamat pada file berdasarkan nomor inode, path, atau offset. Alamat itu stabil selama file tidak dipindahkan, tetapi tidak memberi informasi apa pun

tentang isi. Sistem kontrol versi membutuhkan tiga hal yang tidak bisa diberikan oleh alamat konvensional.

Pertama, deduplikasi. Jika dua commit menyimpan file yang identik, sistem tidak perlu menyimpannya dua kali. Dengan alamat berbasis isi, keduanya memiliki alamat yang sama, dan penyimpanan kedua tidak melakukan apa-apa.

Kedua, verifikasi integritas. Jika file di disk rusak, sistem bisa menghitung ulang hash dan membandingkannya dengan alamat. Ketidakcocokan menandakan korupsi. Model address-based tidak memiliki cara ini.

Ketiga, immutability. Objek yang alamatnya ditentukan oleh isi tidak bisa diubah tanpa mengubah alamatnya. Sistem bisa mengasumsikan bahwa objek dengan alamat tertentu selalu memiliki isi yang sama, di mana pun dan kapan pun.

Gambar **??** menyandingkan kedua model.

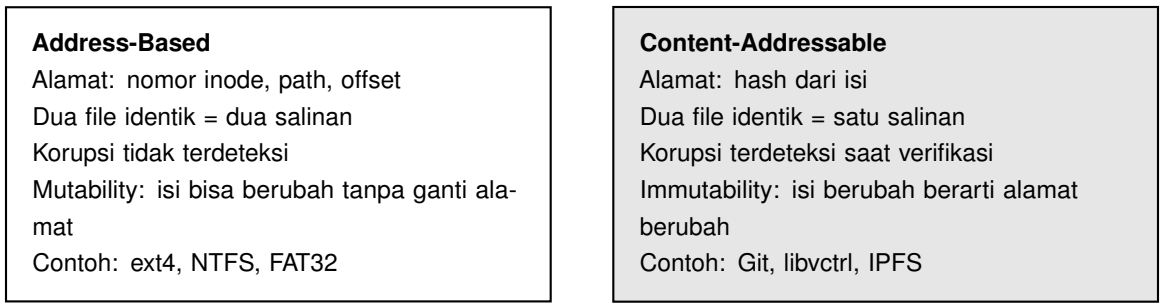


Figure 1. Perbandingan alamat berbasis posisi dengan alamat berbasis konten.

2.3 Properti Kriptografis yang Wajib Dipenuhi

Tidak semua fungsi hash layak menjadi alamat. Ada tiga properti yang wajib dipenuhi.

Pertama, determinisme. Input yang sama harus selalu menghasilkan output yang sama. Jika hash bisa berubah antar pemanggilan, alamat tidak lagi stabil, dan seluruh model runtuh.

Kedua, ketahanan tabrakan. Dua input berbeda harus menghasilkan output yang berbeda, kecuali dengan probabilitas yang sangat kecil. Untuk SHA-256, probabilitas tabrakan acak sekitar satu banding dua pangkat seratus dua puluh delapan. Angka ini jauh lebih kecil daripada probabilitas hard disk gagal secara fisik pada saat yang sama.

Ketiga, ketahanan preimage. Dari hash, tidak mungkin menghitung input aslinya dalam waktu yang wajar. Sifat ini melindungi objek dari pemalsuan: penyerang tidak bisa membuat objek dengan hash tertentu tanpa memiliki isi yang sesuai.

Ketiga properti ini adalah alasan mengapa SHA-256 dan SHA-512 dipakai, bukan fungsi hash non-kriptografis seperti CRC32 atau MD5 (yang sudah dianggap rusak untuk keperluan ini). Tabel **??** merangkumnya.

Table 1. Properti kriptografis yang wajib dimiliki fungsi hash alamat.

Properti	Arti	Akibat jika gagal
Deterministik	input sama, output sama	alamat tidak stabil, tidak bisa dipakai
Collision-resistant	input beda, output beda	dua objek bisa bertabrakan alamatnya
Avalanche	ubah satu bit, output berubah total	perubahan kecil tidak terdeteksi
Preimage-resistant	tidak bisa balik dari hash ke input	objek bisa dipalsukan

2.4 Contoh Berangka

Mari hitung hash dari dua string yang berbeda satu karakter. Perintahnya memakai `sha256sum` yang universal.

[style=code] echo -n "hello" | sha256sum echo -n "hellp" | sha256sum

Keluaran kedua perintah:

[style=out] 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824 - ea09ae9cc6768c50fcee903ed054556e5bfc8347907f12598aa241203221d2b0 -

Dua input yang berbeda satu karakter menghasilkan dua hash yang sama sekali berbeda. Inilah yang disebut efek avalanche: perubahan kecil menyebar ke seluruh output. Untuk content-addressable storage, sifat ini berarti bahwa setiap perubahan pada file menghasilkan alamat baru, sehingga versi lama tetap bisa diakses melalui alamat lamanya.

3 Object Store: Empat Tipe Dasar

3.1 Mengapa Empat Tipe, Bukan Satu

Jika seluruh data disimpan sebagai byte stream tanpa tipe, sistem tidak bisa membedakan file dari direktori, atau direktori dari commit. Setiap operasi harus menebak tipe dari isinya, yang lambat dan rawan salah. Object store modern memecah data menjadi empat tipe yang masing-masing memiliki peran jelas: `blob`, `tree`, `commit`, dan `tag`. Gambar **??** menunjukkan hierarkinya.

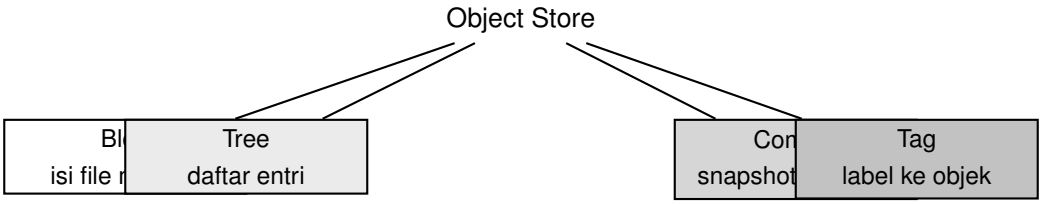


Figure 2. Empat tipe objek pada content-addressable object store.

3.2 Blob: Isi File Mentah

`blob` adalah singkatan dari *binary large object*. Isinya adalah byte mentah dari sebuah file, tanpa nama, tanpa path, tanpa metadata apa pun. `blob` tidak tahu namanya sendiri; ia hanya tahu isinya.

Pemisahan nama dari isi ini adalah kunci deduplikasi. Jika file `notes.txt` dan `backup.txt` memiliki isi yang sama persis, keduanya merujuk ke `blob` yang sama. Sistem menyimpan satu `blob`, bukan dua.

Karena `blob` tidak menyimpan nama, cara sistem menemukan `blob` adalah melalui `tree`.

3.3 Tree: Daftar Entri Direktori

`tree` adalah satu tingkat direktori. Isinya adalah daftar entri, di mana setiap entri memiliki tiga komponen: nama, mode, dan hash dari objek target. Nama adalah nama file atau subdirektori. Mode adalah bit permission yang menyatakan apakah entri itu file biasa, file yang bisa dieksekusi, symlink, subdirektori, atau submodule. Hash adalah alamat dari `blob` (untuk file) atau `tree` (untuk subdirektori).

`tree` adalah tempat nama dan isi dipertemukan kembali. Sebuah `blob` bisa dirujuk oleh banyak `tree` dengan nama yang berbeda, dan sebuah `tree` bisa merujuk banyak `blob` dengan nama yang berbeda.

3.4 Commit: Snapshot dengan Sejarah

`commit` adalah snapshot dari seluruh `tree` di root proyek pada satu titik waktu, ditambah satu atau lebih `parent` yang menunjuk ke commit sebelumnya. Sebuah `commit` menyimpan hash dari root `tree`, daftar hash `parent`, penulis, committer, timestamp, pesan, dan metadata opsional seperti encoding.

Perbedaan antara `tree` dan `commit` adalah perbedaan antara "bagaimana isi direktori saat ini" dan "mengapa isi itu seperti ini". `tree` hanya menyimpan keadaan; `commit` menyimpan keadaan plus alasannya.

3.5 Tag: Label yang Bisa Ditandatangani

`tag` adalah label yang menunjuk ke objek lain, biasanya `commit`. Ada dua jenis: lightweight dan annotated. Lightweight hanyalah alias, sedangkan annotated adalah objek tersendiri yang menyimpan nama tag, hash target, tagger, pesan, dan tanda tangan kriptografis opsional.

Annotated `tag` adalah satu-satunya objek yang dirancang untuk ditandatangani. Tanda tangannya membuktikan bahwa tag itu dibuat oleh pemegang kunci privat, bukan oleh siapa pun yang bisa menulis ke repositori.

3.6 Ringkasan Tipe Objek

Table 2. Ringkasan empat tipe objek pada object store.

Tipe	Isi utama	Peran
`blob`	byte mentah dari satu file	menyimpan isi tanpa nama
`tree`	daftar (nama, mode, hash)	memetakan nama ke isi
`commit`	tree + parent + metadata	merekam snapshot dan sejarah
`tag`	target + tagger + message	label permanen yang bisa ditandatangani

4 Struktur Merkle dan Immutability

4.1 Merkle: Hash dari Hash

Setiap objek pada object store memiliki hash. Yang membuat struktur ini kuat adalah kenyataan bahwa `tree` menyimpan hash dari anak-anaknya, dan `commit` menyimpan hash dari root `tree`. Artinya, hash sebuah `commit` secara transitif mencakup seluruh isi repositori pada saat commit itu dibuat. Ubah satu byte pada satu file, dan hash dari `blob`, `tree`, `commit`, dan semua `commit` turunan berubah.

Struktur ini disebut pohon Merkle. Gambar **??** menunjukkan bahwa perubahan pada satu `blob` merambat naik sampai ke `commit` teratas.

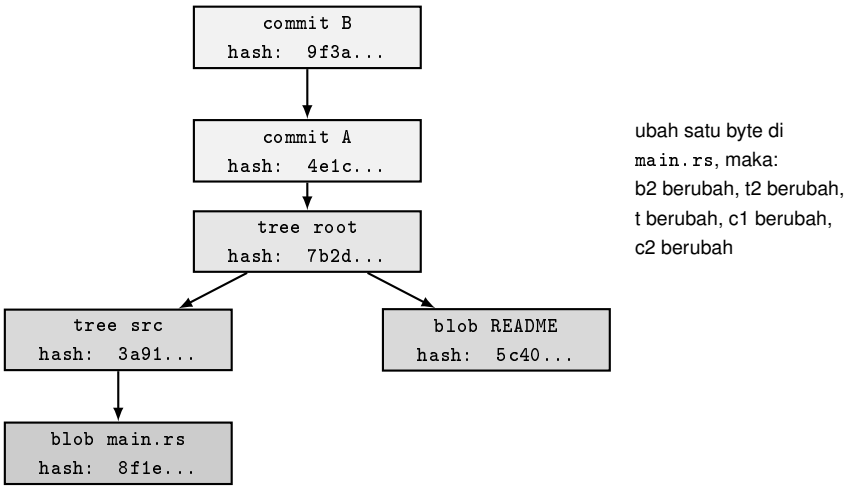


Figure 3. Pohon Merkle pada object store. Perubahan pada satu blob merambat sampai ke commit teratas.

4.2 Mengapa Immutability Terjadi dengan Sendirinya

Immutability bukan fitur yang ditambahkan ke sistem; ia adalah konsekuensi langsung dari alamat berbasis hash. Jika objek dengan alamat `H` bisa diubah isinya, maka hash dari isi yang baru tidak akan sama dengan `H`, dan alamat itu tidak lagi merujuk objek tersebut. Dengan kata lain, "mengubah objek" berarti "membuat objek baru dengan alamat baru"; objek lama tetap

ada dengan alamat lamanya.

Sifat ini membawa tiga konsekuensi praktis. Pertama, riwayat tidak bisa ditulis ulang tanpa mengubah semua hash setelah titik perubahan. Kedua, dua pengguna yang memiliki hash yang sama bisa yakin bahwa mereka memiliki isi yang sama. Ketiga, commit bisa di-verifikasi secara offline; pembaca tidak perlu server untuk memastikan bahwa commit yang ia terima benar-benar berasal dari penulis yang diklaim.

Gambar **??** menyandingkan sifat mutable dan immutable pada sistem penyimpanan.

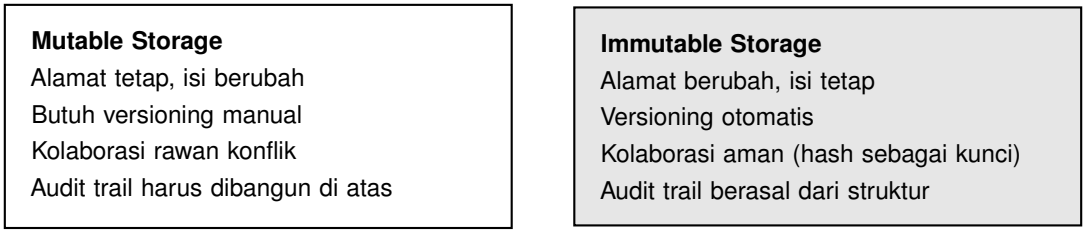


Figure 4. Perbandingan mutable dan immutable storage.

4.3 Siklus Hidup Satu Objek

Proses penyimpanan satu objek pada content-addressable storage mengikuti lima langkah yang tetap. Gambar **??** menunjukkan siklusnya.

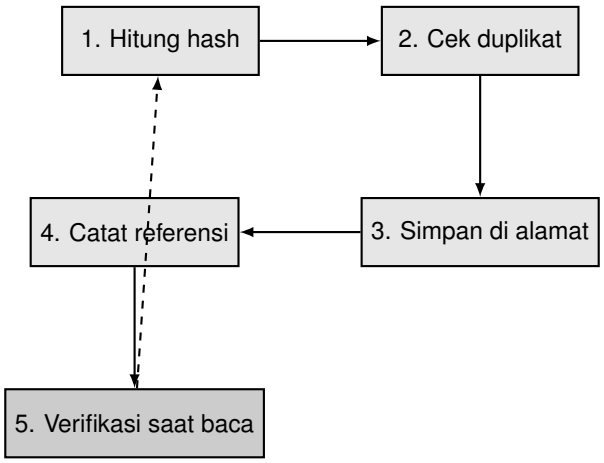


Figure 5. Siklus lima tahap penyimpanan objek pada content-addressable storage.

Garis putus-putus dari verifikasi kembali ke hash menandakan bahwa pembacaan objek berikutnya akan mengulang siklus yang sama. Verifikasi integritas tidak terjadi sekali; ia terjadi setiap kali objek dibaca.

5 Perbandingan Model Penyimpanan

Setelah memahami hash, empat tipe objek, dan struktur Merkle, perbedaan antara content-addressable storage dan address-based storage dapat diringkas dalam satu tabel. Tabel **??**

menyandingkannya pada tujuh aspek.

Table 3. Perbandingan address-based dan content-addressable storage.

Aspek	Address-Based	Content-Addressable
Alamat	nomor inode, path, offset	hash dari isi
Duplikasi	setiap file adalah salinan baru	hanya satu salinan per isi unik
Integritas	tidak terdeteksi tanpa tool eksternal	terdeteksi saat baca
Mutability	isi berubah tanpa ganti alamat	isi berubah berarti alamat berubah
Kolaborasi	butuh mekanisme lock eksternal	aman karena hash sebagai kunci
Pemulihan setelah korupsi	dari backup eksternal	dari replika lain dengan hash sama
Contoh sistem	ext4, NTFS, FAT32	Git, Mercurial, libvctrl, IPFS

Dua paragraf penutup dari perbandingan ini. Pertama, content-addressable storage bukan pengganti sistem berkas; ia adalah lapisan di atas sistem berkas. Objek disimpan sebagai file biasa dengan nama hash, di dalam direktori biasa. Kedua, model ini memindahkan sebagian besar kerumitan dari kode aplikasi ke struktur penyimpanan itu sendiri. Aplikasi tidak perlu memverifikasi integritas, tidak perlu menangani duplikasi, dan tidak perlu menulis log audit; semuanya berasal dari struktur alamat.

6 Studi Kasus: Menyimpan Satu Commit dari Nol

6.1 Langkah 1: Tentukan Isi File

Misalkan sebuah proyek memiliki satu file, `notes.txt`, dengan isi dua baris. Di content-addressable storage, isi dihitung sebagai byte mentah, bukan sebagai baris.

[style=code] printf "hello" > notes.txt

6.2 Langkah 2: Hitung Hash Blob

Isi file dihash. Hasilnya adalah alamat dari `blob`.

[style=code] sha256sum notes.txt

[style=out] 8d9e3f4c1b8a7d2e6f5c4b3a2e1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b6a5f4e3d notes.txt

Alamat ini adalah identitas dari isi `notes.txt`, bukan identitas dari file bernama `notes.txt`. Jika ada file lain bernama `backup.txt` dengan isi yang sama, keduanya akan memiliki alamat yang sama.

6.3 Langkah 3: Cek Duplikat Sebelum Menyimpan

Sebelum menyimpan, sistem memeriksa apakah alamat sudah ada. Jika sudah, tidak ada yang perlu disimpan.

[style=code] test -f .git/objects/8d/9e3f4c1b8a7d2e6f5c4b3a2e1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b6a5f4e3d
echo "already exists" || echo "must store"

[style=out] must store

Pada percobaan pertama, alamat belum ada, sehingga `blob` disimpan. Pada percobaan berikutnya dengan isi yang sama, alamat sudah ada, dan sistem tidak menyimpan apa pun.

6.4 Langkah 4: Simpan Blob di Alamat Hash

Blob disimpan sebagai file dengan nama hash, di dalam direktori dua karakter pertama. Konvensi ini mencegah direktori berisi puluhan ribu file.

[style=code] mkdir -p .git/objects/8d cp notes.txt .git/objects/8d/9e3f4c1b8a7d2e6f5c4b3a2e1d0c9b8a7f6e5d4c

Setelah langkah ini, `blob` sudah ada di object store. Tetapi `blob` tidak tahu namanya sendiri. Nama `notes.txt` belum tersimpan di mana pun.

6.5 Langkah 5: Bangun Tree yang Memetakan Nama ke Blob

`tree` adalah daftar entri. Setiap entri mencatat nama, mode, dan hash. Untuk proyek dengan satu file, `tree` berisi satu entri.

[style=code] printf "100644 blob 8d9e3f4c1b8a7d2e6f5c4b3a2e1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b6a5f4e3d.txt"
> /tmp/tree.txt

Baris ini menyatakan: entri bernama `notes.txt`, dengan mode `100644` (file biasa yang tidak bisa dieksekusi), menunjuk ke `blob` dengan hash yang sudah dihitung di langkah dua.

6.6 Langkah 6: Hash Tree

Hash dari isi `tree` menjadi alamat `tree`. Langkah ini menghasilkan hash yang mencakup hash semua `blob` di dalamnya.

[style=code] sha256sum /tmp/tree.txt

[style=out] a1b2c3d4e5f60718293a4b5c6d7e8f90a1b2c3d4e5f60718293a4b5c6d7e8f90
/tmp/tree.txt

Dua properti penting. Pertama, hash ini bukan hash dari `blob`; ia adalah hash dari daftar entri yang di dalamnya tercantum hash `blob`. Kedua, jika isi `tree` diubah urutannya, hash-nya juga berubah; itulah sebabnya `tree` selalu diurutkan sebelum dihash.

6.7 Langkah 7: Bangun Commit

Commit menyimpan hash dari `tree`, daftar parent (kosong untuk commit pertama), penulis, committer, timestamp, dan pesan.

[style=code] printf "tree a1b2c3d4e5f60718293a4b5c6d7e8f90a1b2c3d4e5f60718293a4b5c6d7e8f90" printf "author Alice <alice@example.com> 1700000000 +0000" printf "committer Alice <alice@example.com> 1700000000 +0000" printf "" printf "first commit" > /tmp/commit.txt

6.8 Langkah 8: Hash Commit dan Update Ref

Hash dari isi commit menjadi alamat commit. Langkah terakhir adalah menulis alamat ini ke `refs/heads/main`, sehingga commit bisa ditemukan melalui nama branch.

[style=code] sha256sum /tmp/commit.txt printf "> .git/refs/heads/main

Setelah langkah ini, seluruh isi commit sudah tersimpan di object store. Untuk membaca kembali `notes.txt` pada commit tersebut, sistem cukup membaca `refs/heads/main`, membaca commit di alamat itu, membaca `tree` di alamat yang tercantum di commit, dan membaca `blob` di alamat yang tercantum di `tree`.

6.9 Ringkasan Studi Kasus

Table 4. Ringkasan delapan langkah menyimpan satu commit.

Langkah	Operasi	Hasil
1	tentukan isi file	`notes.txt` dengan dua baris
2	hash `blob`	alamat `blob` (`8d9e...`)
3	cek duplikat	`blob` belum ada, harus disimpan
4	simpan `blob`	file di `.git/objects/8d/9e...`
5	bangun `tree`	satu entri (nama, mode, hash)
6	hash `tree`	alamat `tree` (`a1b2...`)
7	bangun `commit`	tree + author + committer + pesan
8	hash `commit` dan update ref	`refs/heads/main` menunjuk `f7e6...`

7 Kesalahan Umum dan Cara Menghindarinya

Table 5. Kesalahan umum pada implementasi content-addressable storage.

Kesalahan	Akibat	Perbaikan
hash tanpa encoding yang konsisten	hash berbeda untuk isi sama	tentukan encoding sebelum hash
cek duplikat dihilangkan	storage membengkak	periksa alamat sebelum menulis
`tree` tidak diurutkan	hash berbeda untuk isi sama	urutkan entri sebelum hash
`commit` tanpa parent	semua commit menjadi orphan	sertakan hash parent
`ref` ditulis langsung ke target	ref korup jika write terputus	tulis ke temp lalu rename atomik
`blob` disimpan tanpa hash	tidak bisa ditemukan kembali	hitung hash dulu, baru simpan
hash disimpan sebagai string hex	storage lebih besar, lookup lambat	simpan sebagai byte biner
pesan commit tanpa newline	parser gagal	tambahkan `\n` di akhir
timestamp lokal bukan UTC	riwayat tidak konsisten	simpan dalam UTC
hapus `blob` yang masih dirujuk	commit lama menjadi korup	gunakan ref-count atau GC

Dua kesalahan yang paling sering muncul adalah tidak mengurutkan entri `tree` sebelum hash dan menyimpan hash sebagai string hex. Yang pertama membuat dua `tree` dengan isi sama tetapi urutan berbeda memiliki hash yang berbeda, sehingga deduplikasi gagal. Yang kedua membuat setiap lookup dan perbandingan memakan empat kali lipat ruang; untuk dua puluh juta objek, selisih ini menjadi ratusan megabyte.

8 Lembar Rujukan Cepat

Table 6. Rujukan cepat perintah dan operasi pada content-addressable storage.

Kebutuhan	Perintah atau operasi
hash file	`sha256sum file`
hash stdin	`echo -n "x" | sha256sum`
verifikasi hash	`sha256sum -c checksum.txt`
cek `blob` ada	`test -f .git/objects/ab/cdef...`
simpan `blob`	`mkdir -p .git/objects/ab` lalu copy file
baca `blob`	`cat .git/objects/ab/cdef...`
baca `tree`	`git cat-file -p <tree-hash>`
baca `commit`	`git cat-file -p <commit-hash>`
update ref	`git update-ref refs/heads/main <hash>`
list semua ref	`git show-ref`
baca HEAD	`cat .git/HEAD`
root `tree` dari `commit`	`git rev-parse <hash>^tree`
`tree` dari path	`git ls-tree <tree-hash> path/`
diff dua `tree`	`git diff-tree <tree1> <tree2>`
verifikasi integritas	`git fsck`
pack objek	`git gc`
prune unreachable	`git prune`
lihat history	`git log -graph -oneline`
trace perubahan baris	`git blame file`
cek hash commit saat ini	`git rev-parse HEAD`

9 Kesimpulan

Content-addressable storage adalah model penyimpanan di mana alamat objek ditentukan oleh isinya. Hash kriptografis memastikan bahwa setiap perubahan sekecil apa pun menghasilkan alamat yang berbeda. Empat tipe objek, yaitu `blob`, `tree`, `commit`, dan `tag`, bekerja sama dalam struktur Merkle di mana hash setiap level mencakup hash level di bawahnya. Immutability bukan fitur tambahan; ia adalah konsekuensi dari alamat berbasis isi.

Tiga prinsip penutup:

1. Alamat adalah isi. Jika dua objek harus dianggap sama, mereka harus memiliki alamat yang sama. Jika mereka harus berbeda, alamatnya harus berbeda. Aturan ini menghilangkan hampir semua kerumitan yang biasanya muncul pada sistem deduplikasi manual.
2. Urutan adalah bagian dari isi. Dua `tree` dengan entri yang sama tetapi urutan berbeda adalah dua objek yang berbeda. Selalu urutkan sebelum menghitung hash, dan dokumentasikan aturan pengurutannya.
3. Immutability adalah fondasi, bukan kemewahan. Sistem yang mengizinkan objek berubah tanpa mengganti alamatnya kehilangan semua jaminan yang ditawarkan content-addressable storage: verifikasi integritas, deduplikasi otomatis, dan audit trail yang bisa dipercaya. Jangan pernah menulis ulang objek setelah alamatnya dipublikasikan.

Dengan kerangka ini, pembaca siap membaca implementasi VCS mana pun, atau membangun VCS kecil sendiri di atas model yang sudah terbukti.

Referensi

References

[1] Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Apress. https://git-scm.com/book/

[2] Loeliger, J., & McCullough, M. (2012). *Version Control with Git* (2nd ed.). O'Reilly Media.

[3] Git Project. (2026). *Git Internals — Git Objects*. Git Documentation. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[4] Merkle, R. C. (1988). A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87* (pp. 369–378). Springer.

[5] NIST. (2015). *Secure Hash Standard (SHS)* (FIPS PUB 180-4). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.180-4

[6] Eastlake, D., & Hansen, T. (2011). *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)* (RFC 6234). IETF.

[7] Khafidi, M. R. (2026). *libvctrl: A Precision Toolkit for Building Custom Version Control Systems*. GitHub. https://github.com/mroczect/libvctrl