Matematični pogled na Git

Martin Vuk
Fakulteta za računalništvo in informatiko
Univerza v Ljubljani

15. januar 2026

Povzetek

Git je program, ki omogoča vodenje zgodovine različic datotek v neki mapi(direktoriju). V glavnem se uporablja za upravljanje z izvorno kodo pri razvoju računalniških programov. Mnogi med nami pa ga uporabljajo tudi pri pisanju besedil v LaTeX-u. Poleg tega, da Git hrani zgodovino sprememb, tudi olajša združevanje sprememb, ko več ljudi hkrati ureja iste datoteke. Ogledali si bomo, kako Git deluje. Opisali bomo, kako Git uporabi *zgoščevalne funkcije*, *Merklejeva drevesa* in *usmerjene aciklične grafe*, da shrani zgodovino različic in olajša hkratno urejanje vsebine. Matematični model, ki ga Git uporablja, je v resnici zelo preprost in njegovo razumevanje nas lahko reši marsikatere zagate, ki nastane med njegovo uporabo.

Abstract

Git is a version control system that allows tracking changes in files within a directory. It is mainly used for source code management in software development. However, many of us also use it for writing texts in LaTeX. Besides tracking history, Git facilitates merging changes when multiple people edit the same files simultaneously. We will look at how Git works. We will describe how Git uses *hash functions*, *Merkle trees*, and *directed acyclic graphs* to store version history and facilitate concurrent content editing. The mathematical model used by Git is actually very simple, and understanding it can save us many headaches during its use.

Ključne besede: Git, sistem za nadzor različic, zgoščevalna funkcija, usmerjen aciklični graf, rojstnodnevni problem.

Keywords: Git, Version Control System, Hash Function, Directed Acyclic Graph, Birthday Problem.

Math. Subj. Class. (2020) 68P05, 68P20, 05C20, 60C05.

1 Kaj je Git?

Git je kot *časovni stroj* za datoteke. Uporabniku omogoča, da vidi *pretekle različice* datotek, spreminja datoteke, brez skrbi, da bi kaj pokvaril in jih deli z drugimi. Poleg časovnega stroja je Git tudi *razpršeno skladišče datotek*. Omogoča, da datoteke hkrati ureja več uporabnikov na različnih računalnikih in kasneje spremembe združi.

Git hrani vsebino mape z datotekami in celotno zgodovino različic datotek iz preteklosti. Za vsako različico hrani Git zapis o avtorju, datumu in opis sprememb, ki so nastale v primerjavi s predhodno različico. Vse te imformacije dajejo podroben pregled nad zgodovino sprememb.

Sisteme, ki omogočajo hranjenje preteklih različic datotek, imenujemo sistemi za nadzor različic (angl. version control system (VCS)) ali *sistemi za upravljanje z izvorno kodo* (angl. Source Code Management (SCM)). Poleg nadzora različic Git omogoča hkratno spreminjanje datotek več uporabnikov na različnih računalnikih. Zato je Git distribuiran sistem za nadzor različic (angl. Distributed Version Control System (DVCS)).

Opomba 1 *Ljudje pogosto mešajo Git in GitHub, ki pa nista eno in isto. Git je program, ki si ga lahko vsakdo namesti in poganja na svojem računalniku. Program Git je ustvaril Linus Torvalds, da bi lažje upravljal z izvorno kodo za jedro operacijskega sistema Linux. GitHub je javno spletišče, ki je namenjeno skladiščenju Git repozitorijev.*

V nadaljevanju si bomo ogledali, kako Git uporablja zgoščevalno funkcijo (angl. hash function) in posplošitev Merklejevih dreves za hranjenje posnetkov vsebine mape. Kako zgodovino sprememb predstavimo z usmerjenim acikličnim grafom, v katerem so vozlišča različice, povezave pa povežejo različice z njihovimi neposrednimi predhodniki in kako preproste reference (kazalci) na vsebino omogočajo bliskovito preklaplanje med različicami in preprečijo popolno zmešnjavo, ko več ljudi hkrati spreminja iste datoteke?

2 Podatkovno skladišče

Ko ustvarimo nov Git repozitorij, Git ustvari podmapo z imenom `.git` z vsemi podatki, ki jih potrebuje. V mapi `.git` se hranijo različne stvari:

- vsebina datotek, ki smo jih dodali v repozitorij,
- drevesna struktura korenske mape, ki jo hranimo v repozitoriju,
- posnetki stanja v različnih trenutkih s podatki o avtoju, datumu in opisu sprememb,
- kazalci na posamezne posnetke stanja.

Git repozitorij je vsaka mapa, ki vsebuje podmapo `.git` z zgoraj navedenimi podatki. Kako Git hrani podatke bomo spoznali v nadaljevanju, podrobnosti pa si lahko preberete v knjigi Pro Git [2].

3 Zgoščevalna funkcija

Git ne shranjuje datotek z običajnimi imeni, ampak za ime uporabi 160 bitno število (40 mestno število v 16-tiškem zapisu), ki ga izračuna iz vsebine datoteke. Git za izračun imena uporabi *zgoščevalno funkcijo*. Naj bo B množica vseh možnih podatkovnih nizov(besedil), n-bitna zgoščevalna funkcija je funkcija

$$H : B \to \{0, 1, \ldots, 2^n - 1\},$$

ki vsakemu besedilu b priredi n-bitno vrednost $H(b)$. Vrednosti zgoščevalne funkcije $H(b)$ pravimo *zgostitev* vsebine b (angl. hash). Git hrani datoteke pod imeni, ki so enaka zgostitvi vsebine. Kaj pa če imata dve različni vsebini isto zgostitev? Funkcija H ni injektivna, saj je množica nizov, bistveno večja od množice zgostitev. To pomeni, da imata lahko dve različni datoteki enako zgostitev. Če se to zgodi, rečemo, da pride do *trka zgostitve*. V primeru trka zgostitve bi Git shranil le eno datoteko, za drugo pa bi predpostavil da je že shranjena. Zato je funkcija H izbrana tako, da sprememba enega samega bita v besedilu $b \in B$ spremeni vrednost $H(b)$ in je porazdelitev vrednosti $H(b)$ čim bližje enakomerni porazdelitvi. To pomeni, da so vse vrednosti $H(b)$ približno enako verjetne. Na ta način zmanjšamo verjetnost trka (glej poglavje 9). Verjetnost trka je izjemno majhna, zato Git lahko predpostavi, da je niz b enolično določen z njegovo zgostitvijo $H(b)$. Git uporablja 160 bitno zgoščevalno funkcijo *SHA1*, ki se je uporabljala v kriptografiji[1].

Ko datoteko z vsebino b zabeležimo v Git repozitorij, Git izračuna zgostitev vsebine $H(b)$ in jo shrani v datoteko z imenom $H(b)$ v `.git/objects`[2]. Vsebina b je tako vedno dostopna pod imenom, ki je enako njeni zgostitvi $H(b)$. Tako dobimo vsebinsko naslovljivo shrambo objektov, ki je ena od bistvenih značilnosti Gita. Ta način shranjevanja omogoča, da lahko vedno preverimo, če ima shranjenjena vsebina isto zgostitev, kot je njeno ime. Lahko tudi shranimo več različic iste datoteke, saj ima vsaka različica drugačno zgostitev. Zgostitev služi tudi kot kontrola, če je prišlo do kvaritve podatkov, ki so shranjeni v Git repozitoriju.

[1] Leta 2017 so raziskovalci iz CWI Amsterdam in Google Research našli prvi praktični primer dveh različnih pdf datotek, ki imata isto SHA1 zgostitev[4]. Opisan napad so poimenovali *SHAttered*. Git je zato z verzijo `v2.13.0` začel uporabljati verzijo SHA1, ki je odporna proti napadu *SHAttered*. Kljub temu razvijalci Gita načrtujejo, da bodo SHA1 postopoma nadomestili s 256 bitno zgoščevalno funkcijo SHA-256.

[2] V resnici Git shrani vsebino v datoteko z imenom $h_3h_4 \ldots h_{40}$ v mapi h_1h_2, kjer je $h_1h_2h_3 \ldots h_{40}$ zapis $H(b)$ v 16-tiškem sistemu. Datoteka, katere vsebina ima zgostitev $H(b)$ enako `8dd6d4bdaeff93016bd49474b54a911131759648` bo shranjena v `.git/objects/8d/d6d4bdaeff93016bd49474b54a911131759648`. Zavoljo preglednosti bomo v nadaljevanju večrat napačno zatrjevali, da je ime datoteke enako zgostitvi njene vsebine.

4 Datotečna drevesa

V vsebinsko naslovljivo shrambo objektov lahko shranimo vsebino datotek in njihovih prejšnjih različic. A kako ohranimo informacijo o imenu datotek in drevesni strukturi mape? Git za to ustvari nov tip objekta *drevo* (angl. *tree*), ki hrani preprost seznam imen datotek in naslovov na vsebino datotek v mapi. Naslov na vsebino datoteke b je seveda zgostitev vsebine $H(b)$.

```
100644 blob 33476f4951afc28d5ac2dc0d42d82f17ac817de2    bla.txt
100644 blob 2ce22b4dc77442103f095503f1205937c1b0fcfc    blabla.txt
040000 tree ae247f2a35aadade5863aec2475cf13020304b06    podmapa
```

Slika 1: Vsebina mape v Gitu je preprost seznam datotek in podmap ter zgostitev njihove vsebine. Številke na začetku določajo dovoljenja za datoteke po sistemu Posix.

Drevo je preprost seznam v tekstovni datoteki, za katerega lahko prav tako izračunamo zgostitev. Zgostitev datotečnega drevesa natanko določa tako imena kot tudi vsebino datotek, ki so vsebovane v mapi. Če se katerakoli datoteka ali njeno ime v mapi spremeni, se bo spremnila tudi njena zgostitev in posledično zgostitev za drevo. Poleg posameznih datotek, lahko drevo vsebuje tudi poddrevesa. Tako lahko rekurzivno ustvarimo drevesno podatkovno strukturo, ki zajema mapo z datotekami in podmapami v poljubni globini.

Poglejmo si primer. Denimo, da imamo v korenski mapi naslednje datoteke in podmape.

```
├── bla.txt (vsebina: bla)
├── blabla.txt (vsebina: blabla)
└── podmapa
    └── bla.txt (vsebina: bla)
```

Slika 2: Struktura datotek in podmap, ki jo bomo hranili v Gitu.

Git bo shranil naslednje objekte v vsebinsko naslovljivo shrambo:

- vsebino datoteke `bla.txt`

  ```
  bla
  ```

  v `.git/objects/bc/c1382241e267cf790ca6b3afe9fde6dcf1072f`

- vsebino datoteke `blabal.txt`

  ```
  blabla
  ```

  v `.git/objects/2c/e22b4dc77442103f095503f1205937c1b0fcfc`

- seznam datotek v mapi `podmapa`

  ```
  100644 blob bcc1382241e267cf790ca6b3afe9fde6dcf1072f    bla.txt
  ```

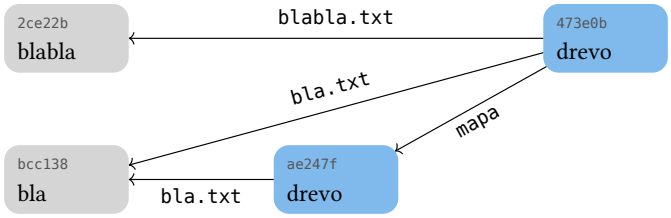
  v `.git/objects/ae/247f2a35aadade5863aec2475cf13020304b06`

- seznam datotek v korenski mapi

  ```
  100644 blob 33476f4951afc28d5ac2dc0d42d82f17ac817de2   bla.txt
  100644 blob 2ce22b4dc77442103f095503f1205937c1b0fcfc   blabla.txt
  040000 tree ae247f2a35aadade5863aec2475cf13020304b06   podmapa
  ```

  v `.git/objects/47/3e0bbfc9de64fdca00e611e5666788ddf664ca`

Z uporabo zgostitve kot kazalca na vsebino, Git vsebino mape postavi v podatkovno strukturo, ki jo matematično lahko opišemo z *usmerjenim grafom*. Če je vsebina datotek enaka(npr. `bla.txt` in `mapa/bla.txt`), Git shrani le eno kopijo, ki je dostopna v datoteki z imenom enakim zgostitvi vsebine. Zato datotečno drevo v Gitu ni nujno predstavljeno kot drevo, ampak kot *usmerjen (aciklični) graf*[3].



Slika 3: Primer datotečnega grafa povezanega z zgostitvami. Zaradi preglednosti bomo v slikah izpisali le prvih 6 znakov zgostitve.

Posledično lahko vsebino celotne mape opišemo z eno samo zgostitvijo. Če spremenimo vsebino, ime ali lokacijo datoteke, bo sprememba vplivala na zgostitev spremenjene vsebine in sprememba bo splavala na površje do zgostitve za korensko mapo. Zgostitev služi tako kot identifikator vsebine, kot tudi kot kontrolna vsota, ki omogoča detekcijo sprememb.

Opomba 2 *Podatkovna struktura objektov v Gitu je podobna Merklejevim drevesom[3]. Postopek graditve datotečnega drevesa v Gitu je soroden veriženju blokov, ki se uporablja v kriptovalutah.*

Ponovimo, kar smo spoznali o Gitu. Git hrani vsebino datotek in datotečno strukturo v *vsebinsko naslovljivi shrambi* (v mapi `.git/objects`). To pomeni, da je referenca na posamezen objekt v Gitu preprosto zgostitev njegove vsebine in da lahko do določene vsebine dostopamo le, če poznamo njeno zgostitev. Po drugi strani je vsebina za vse praktične primere določena s svojo zgostitvijo. Tako lahko enostavno preverimo verodostojnost vsebine, ki je shranjena v Gitu.

5 Zgodovinski graf sprememb

V prejšnjem poglavju smo videli, kako Git hrani vsebino celotne mape in kako zgostitev korenske mape določa vsebino vseh shranjenih datotek. Zgodovinsko drevo sprememb je preprosta razširitev omenjene podatkovne strukture.

5.1 Posnetki stanja

Osnovna enota v Gitu je *vnos* (angl. *commit*). Vnos je posnetek stanja zabeleženih datotek v trenutku, ko je bil ustvarjen. Poleg vsebine datotek vsak vnos vsebuje še metapodatke o avtorju, datumu vnosa in opisom sprememb. Podobno kot objekt tipa *drevo*, je tudi vnos objekt v vsebinsko naslovljivi shrambi in ima določeno *zgostitev vnosa*. Zgostitev vnosa je natanko določena z vsebino shranjenih datotek in metapodatkov vnosa.

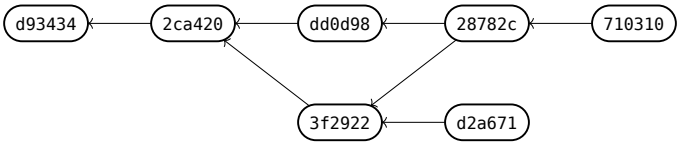
```
tree 65c47feec7465e80492620a48206793e078702e0
parent 16f2994757f1213935b8edb9ae7fee3a8e9ec98d
author MV <mv@example.com> 1765235698 +0100
committer MV <mv@example.com> 1765235698 +0100

Dodaj bla
```

Slika 4: Vnos v Gitu je shranjen v podatkovno shrambo pod imenom, ki je zgostitev vsebine vnosa: `.git/objects/8d/d6d4bdaeff93016bd49474b54a911131759648`.

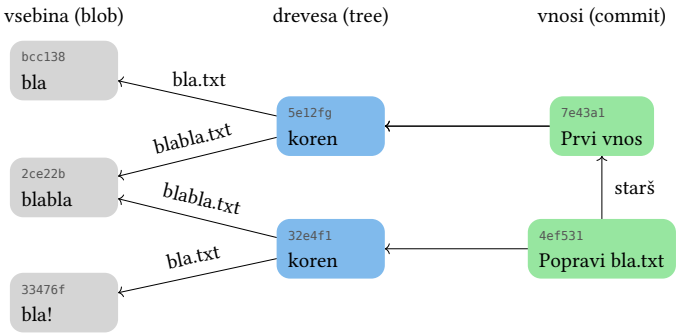
[3]Teoretično bi lahko dosegli, da bi bili v grafu tudi cikli, a je to zelo malo verjetno in zato to možnost ignoriramo.

Vsak vnos je povezan s točno določenim posnetekom vsebine korenskega datotečnega drevesa, ki ga identificira zgostitev. Poleg tega so posamezni vnosi povezani v *usmerjen acikličen graf*, ki predstavlja zgodovino sprememb. Vsak vnos je *vozlišče* v grafu in izhaja iz enega ali več starševskih vnosov. Izjema je prvi vnos. Povezave v grafu povezujejo vnose z njihovimi starši.



Slika 5: Vnosi v Gitu kot usmerjen graf. Vsak vnos(razen prvega) ima povezavo na vnose iz katerih izhaja.

Tudi vnose hrani Git v vsebinsko naslovljivi shrambi pod imenom, ki je enako zgostitvi vnosa. V shrambi imamo tri vrste objektov: vsebina datotek (blob), datotečna drevesa (tree) in vnose (commit). Vsi objekti so dostopni, če poznamo njihovo zgostitev in so med seboj povezani v usmerjen aciklični graf. Zgostitve objektov "na vrhu"natanko določajo vsebino vseh objektov pod njimi. Na vrhu grafa so vnosi, ki vsebujejo reference na druge vnose in na posnetke korenske mape. Posnetek korenske mape vsebuje reference na vsebino datotek in posnetke podmap.



Slika 6: Vsebinsko naslovljiva shramba objektov v Gitu. Naslovi so zgostitve vsebine. Shramba vsebuje dva vnosa. V prvem vnosu smo dodali dve datoteki `bla.txt` in `blabla.txt`, v drugem vnosu pa smo spremenili le vsebino datotoeke `bla.txt`.

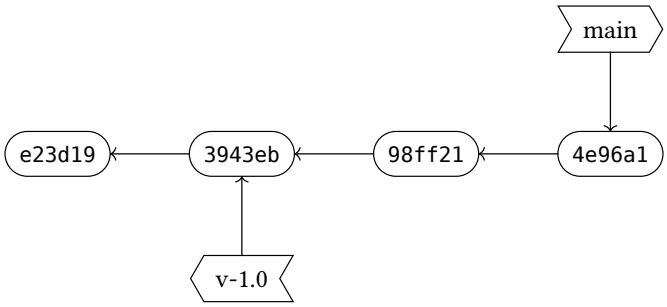
6 Kazalci: veje in značke

Poleg objektov kot so *vnosi*, *posnetki map* in *posnetki datotek* pozna git še reference. Reference so preproste datoteke, ki vsebujejo zgostitev za posamezen vnos. Referenc git ne hrani v skladišču objektov, temveč posebej v mapi `.git/refs`.

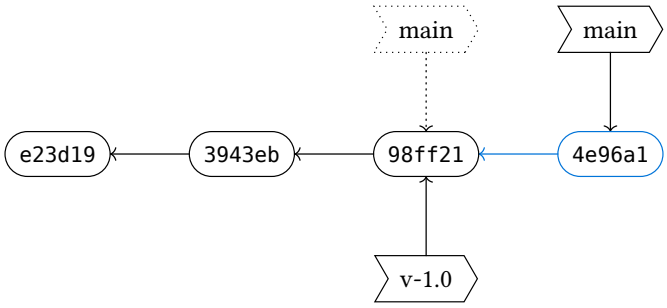
Git pozna dve vrste referenc. *Veja* (angl. *branch*) je posebne vrste referenca, ki se premika, ko dodajamo nove vnose. Vsakič ko ustvarimo nov vnos, se trenutno aktivna veja premakne na novo ustvarjeni vnos. Veje uporabljamo za vzdrževanje vzporednih razvojnih linij, ki so med sabo neodvisne. *Značka* (angl. *tag*) je referenca, ki je statična. Za razliko od veje, se oznaka nikoli ne premika samodejno. Zato se uporablja predvsem za označevanje pomembnih mejnikov v zgodovini na primer verzij posameznih izdaj.

HEAD je posebna referenca, ki kaže na trenutno aktiven vnos. Vnos, na katerega kaže *HEAD* bo starševski vnos naslednjeg vnosa, ki ga bomo dodali. Ko spreminjamo datoteke Git najprej postavi spremenjene datoteke v *čakalnico* (angl. *staging area*), ki se imenuje tudi *indeks* (angl. *index*). Šele ko ustvarimo vnos, Git indeks trajno shrani.

Veje in značke nimajo v Gitu nobenega posebnega pomena, razen tega, da so reference na vnose. Pomen posamenznih vej je stvar dogovora med uporabniki. Tako se pogosto uporablja različne veje za različne namene: `main` ali `master` je navadno glavna veja razvoja, veje z imeni `stable`, `production`, `development` in podobno označujejo različne stopnje razvoja programske opreme, veje s predpono `feature`



Slika 7: Veja `main` in značka `v-1.0` sta preprosta kazalca na posamezen vnos.



Slika 8: Ko ustvarimo nov vnos, se aktivna veja `main` premakne naprej, značka `v-1.0` pa ostane tam, kjer je bila.

označujejo razvoj novih funkcionalnosti. Vse te pomene damo vejam ljudje, ki sodelujemo v nekem Git repozitoriju. Za Git so vse veje in značke zgolj preprosti kazalci na določen vnos.

7 Povzetek

Povzemimo sedaj, kaj smo spoznali o podatkovnem modelu Gita. Git hrani zgodovino sprememb v *vsebinsko naslovljivi shrambi objektov*, ki hrani tri vrste objektov:

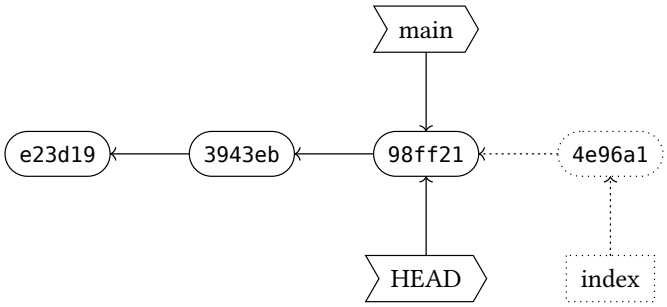
- `blob`: vsebina datotek,
- `tree`: imena vsebovanih datotek in podmap skupaj z njhovimi zgostitvami,
- `commit`: posnetek stanja projekta v nekem trenutku z metapodatki o avtorju, času in sporočilom.

Naslovi objektov so *zgostitve* vsebine objekta, zato je zagotovljena verodostojnost shranjenih podatkov. Vnosi so povezani v *usmerjen aciklični graf*, ki opiše zgodovino sprememb. Vsak vnos je določen z *zgostitvijo vnosa* (angl. *commit hash*), ki je 40-mestna heksadecimalna vrednost, izračunana s SHA1. Zgostitev vnosa je določena na podlagi vsebine vseh datotek, kot tudi metapodatkov vnosa.

Izven shrambe objektov hrani Git še reference na posamezne vnose. Poznamo dve vrsti referenc:

- *Veja* (angl. *branch*) je premična referenca, ki kaže na določen vnos v zgodovini in se samodejno premakne naprej, ko dodajamo nove vnose.
- *Oznaka* (angl. *tag*) je statična referenca, ki trajno kaže na določen vnos.
- *HEAD* je posebna oznaka, ki kaže na trenutno aktiven vnos v delovni kopiji.

Omenimo še dva pojma, ki jih uporabljamo pri delu z Gitom:



Slika 9: *HEAD* je referenca na trenutno aktiven vnos. *Index* vsebuje spremembe, ki bodo zabeležene v naslednjem vnosu.

- *Delovna kopija* (angl. *workout copy*) je mapa v kateri urejamo datoteke, ki jih nato vnesemo v Git. V delovni kopiji imajo na začetku datoteke isto vsebino kot je vsebina trenutno aktivnega vnosa (`HEAD`). Spremembe, ki jih naaredimo na delovni kopiji lahko zabeležimo v nov vnos.
- *Oddaljen repozitorij* (angl. *remote*) je povezava(url) na drug repozitorij (ponavadi na drugem računalniku), s katerim lahko izmenjujemo vsebino.

8 Git ukazi kot operacije na grafu

Gitov podatkovni model omogoča, da je večina operacij v Gitu obrnljivih. To pomeni, da lahko repozitorij povrnemo v prejšnje stanje. Večina operacij le dodaja nove vnose in starih ne briše[4]. Zato so stare različice datotek vedno na voljo. Git uporabniku daje samozavest, da brez strahu spreminja vsebino, saj se lahko vedno vrne v času nazaj. Kot bi imel časovni stroj.

Opremljeni z razumevanjem podatkovnega modela Gita, lažje razumemo posamezne operacije, ki jih Git omogoča. Ukazov ne bom prevajal, ampak jih bom navedel kot jih pozna program `git`.

```
git checkout referenca
```

spremeni datoteke v delovni kopiji tako, da se ujemajo z vnosom, na katerega kaže `referenca`. Poleg tega prestavi oznako `HEAD` na isti vnos. Če je referenca veja, jo nastavi, kot aktivno vejo. Če je referenca oznaka ali zgostitev vnosa, priedmo v stanje brez aktivne veje (angl. *deteached HEAD*).

```
git commit -m "Sporočilo za vnos"
```

ustvari nov vnos, ki kaže na stanje v čakalnici (angl. staging area ali index). V zgodovinskem grafu ustvari novo vozlišče, ki je povezano s prejšnjim vnosom. Poleg tega prestavi aktivno vejo in oznako `HEAD` na novo ustvarjeni vnos.

```
git add bla.txt
```

doda vsebino spremenjene datoteke `bla.txt` v čakalnico. Ukaz ne spreminja zgodovinskega grafa, pač pa doda novo vsebino in datotečna drevesa, ki vsebujejo spremembe v shrambo objektov. Vsebina čakalnice bo zabeležena v naslednjem vnosu.

```
git pull
```

pobere vsebino(objekte in reference) iz oddaljenega repozitorija in uskladi lokalno vejo z oddaljeno. Shrambi objektov se preprosto doda nove objekte, ki so v oddaljeni veji. Če je lokalna veja prednik oddaljene, se lokalna veja enostavno prestavi, da kaže na isti vnos, kot oddaljena veja. V nasprotnem primeru, mora uporabnik posredovati in razrešiti morebitne konflikte.

```
git push
```

[4]Nekatere operacije vnose tudi brišejo (npr. `git rebase`). Takim operacijam rečemo, da spreminjajo zgodovino. Uporabniki morajo biti pri njihovi uporabi posebej pazljivi, da česa trajno ne zamočijo.

potisne novo vsebino na oddaljeni repozitorij. Push deluje obratno kot `pull`. Ukaz je uspešno izveden le, če je oddaljena veja predhodnica lokalne veje.

`git fetch`

pobere novo vsebino (vnose, veje in oznake) iz oddaljenega repozitorija. Pri tem ne more priti do konfliktov, ker git preprosto doda nove objekte v shrambo in obstoječih objektov nikakor ne spreminja. Oddaljenim vejam in oznakam preprosto doda predpono z imenom oddaljenega repozitorija.

`git reset referenca`

spremeni kam kaže trenutno izbrana veja. Trenutno izbrano vejo prestavi na isti vnos, na katerega kaže dana `referenca`. Ukaz ne spremeni zgodovinskega drevesa, ampak le to, na kateri vnos kaže trenutno izbrana veja.

`git merge referenca`

ustvari nov vnos, ki združi dve ločeni veji v eno (trenutno izbrano in referenco). Nov vnos ima dva starša: vnos na katerega kaže trenutna veja in vnos, na katerega kaže `referenca`. Če pride do konfliktov, jih mora uporabnik sam razrešiti, preden se ustvari nov vnos.

`git rebase referenca`

prestavi vnose v trenutno izbrani veji tako, da so potomci vnosa, na katerega kaže `referenca`. Med ukazi, ki smo jih spoznali, je ta ukaz edini, ki lahko povzroči izgubo podatkov. Običajno ukazi le dodajajo nove vnose in prestavljajo reference. Zato je večina ukazov v Gitu varna, v smislu, da jih lahko kasneje prekličemo in pridemo nazaj na prejšnje stanje. Ukaz `rebase` pa spremeni zgodovino in ga ne moremo preklicati, saj trenutne vnose nadomesti z novimi in stare vnose pobriše[5].

9 Trki zgostitev in rojstnodnevni paradoks

Git hrani datoteke pod imeni, ki so enaka zgostitvi vsebine. Če bi imeli dve datoteki z različno vsebino isto zgostitev, bi Git shranil le eno datoteko in bi prišlo do izgube podatkov. Git se zanaša na to, da je verjetnost za to izjemno majhna. Kako bi ocenili to verjetnost?

Koliko datotek bi morali shraniti v Git, da bi z znatno verjetnostjo prišlo do trka? Vprašanje je povezano z rojstnodnevnim problemom. Kako velika naj bo skupina ljudi, da bo vsaj 50% verjetnost, da imata dve osebi na isti dan rojstni dan? Velikost skupine je presenetljivo majhna(23), zato rojstnodnevnei problem imenujemo tudi rojstnodnevni paradoks. Vprašanje zastavimo matematično. Naključno izberemo $n < d$ števil iz množice $\{1, 2, \ldots, h\}$, tako da je vsaka izbira enakomerno porazdeljena. Kolikšna je verjetnost $p(n, h)$, da bosta vsaj dve števili enaki? Verjetnost $p(n, h)$ izračunamo elementarno z verjetnostjo nasprotnega dogodka:

$$1 - p(n,h) = \frac{h \cdot (h-1) \cdots (h-n+1)}{h^n} = \prod_{k=1}^{n-1} \left(1 - \frac{k}{h}\right). \tag{1}$$

Če izraz logaritmiramo, dobimo

$$\log(1 - p(n,h)) = \sum_{k=1}^{n-1} \log(1 - \frac{k}{h}) < -\sum_{k=1}^{n-1} \frac{k}{h} = \frac{-(n(n-1))}{2h}. \tag{2}$$

Res! Logaritem je konveksna funkcija, zato so vrednosti manjše od vrednosti na tangenti $\log(1 - \frac{k}{h}) = \log(1-x) < x = \frac{k}{h}$.

Od tod izpeljemo oceno za $p(n, h)$

$$p(n,h) > 1 - e^{\frac{-(n(n-1))}{2h}} \approx 1 - e^{-\frac{n^2}{2h}}. \tag{3}$$

Za vrednosti $1 \ll n \ll h$ je $1 - e^{-\frac{n^2}{2h}}$ tudi dobra aproksimacija za $p(n, h)$.

[5]Obstaja enostaven način, kako `rebase` izvedemo tako, da ga lahko kasneje prekličemo. Na vnos, ki ga želimo prestaviti z `rebase`, preprosto postavimo novo vejo ali oznako. To povzroči, da se stari vnosi ne pobrišejo, ko se izvede ukaz `rebase`.

Da bi odgovorili kako odporna je zgoščevalna funkcija na morebitne trke, moramo rešiti obratno nalogo: največ koliko števil $n(p, d)$ lahko izberemo, da bo verjetnost pojava dveh enakih števil manjša od $p \in [0, 1]$? Natančen odgovor na to vprašanje ni tako preprost [1]. Lahko pa uporabimo oceno (3) in čez palec ocenimo vrednost $n(p, h)$:

$$\begin{aligned} -n^2 &\approx \log(1-p) \Rightarrow \\ n(p,h) &\approx \sqrt{2h \log(\tfrac{1}{1-p})} \approx \sqrt{2h}. \end{aligned} \tag{4}$$

Funkcija $\sqrt{\log(\frac{1}{1-p})}$ zelo počasi narašča, ko se p približuje 1, zato jo lahko zanemarimo. Če je zgoščevalna funkcija 160 bitna, kot na primer SHA1, je $n \approx \sqrt{2^{160}} \approx 2^{80}$. Znatna verjetnost, da pride do trka zgostitev, bi se pojavila, ko bi shranili 2^{80} različnih verzij datotek v Git. Raziskovalci, ki so razvili napad *SHAttered*, so se posebej potrudili in so potrebovali "zgolj" približno 2^{63} primerov, da so prišli do trka.

10 Zaključek

Spoznali smo, kako deluje Git in s katerimi matematičnimi pojmi lahko opišemo njegov podatkovni model. Upam, da boste s tem znanjem bolj samozavestno uporabljali Git. Opis dela z Gitom presega namen tega dokumenta, zato vas raje usmerim na uradno dokumentacijo:

`https://git-scm.com/cheat-sheet`

Pri pisanju tega članka sem sevada uporabljal Git. V javno dostopnem repozitoriju [5] si lahko ogledate celotno zgodovino nastajanja tega članka. Pri pripravi dokumenta sem uporabil Gemini 3, a sem vse odgovore skrbno preveril in uredil po svoje.

Literatura

[1] David Brink. A (probably) exact solution to the Birthday Problem. *The Ramanujan Journal*, 28(2):223–238, June 2012. `doi:10.1007/s11139-011-9343-9`.

[2] Scott Chacon and Ben Straub. 10.2 Git Internals - Git Objects. In *Pro Git*. URL: `https://git-scm.com/book/en/v2/Git-Internals-Git-Objects`.

[3] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer. `doi:10.1007/3-540-48184-2_32`.

[4] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The First Collision for Full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.

[5] Martin Vuk. git-intro. URL: `https://git.fri.uni-lj.si/martin.vuk/git-intro`.