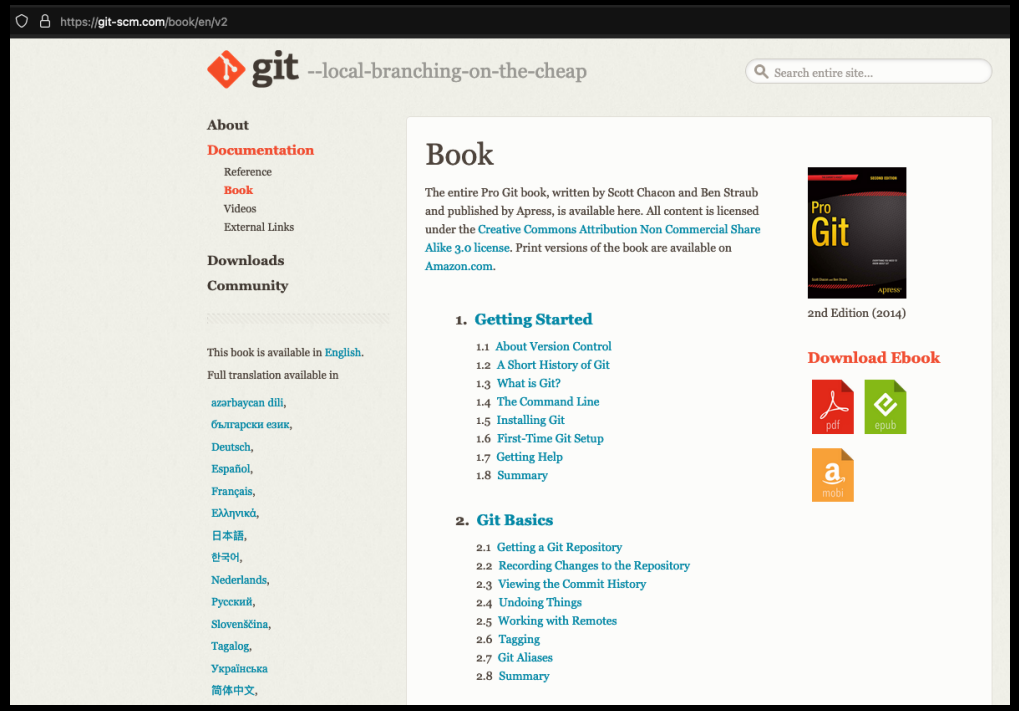
Under huven på Git

En djupdykning i hur Git faktiskt fungerar

Anders Sigfridsson, November 2021

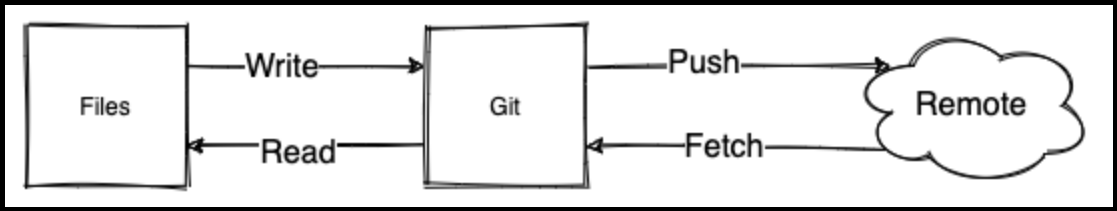
Pro Git (git-scm.com)

Vi kommer utgå från exempel som finns i sektionen "10. Git Internals" på git-scm.com/book



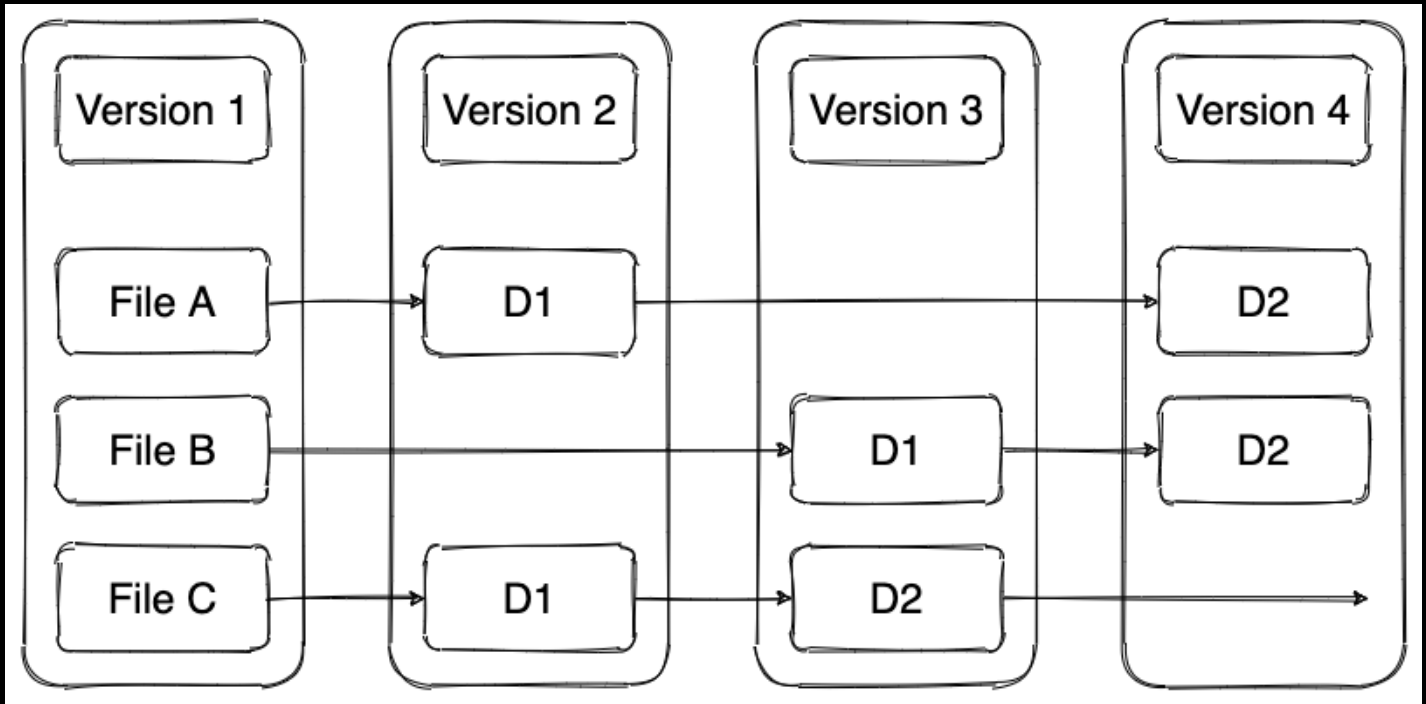
Git är en databas

- Data lagras i en filstruktur och associeras med en unik nyckel
- Användaren skriver och läser text från databasen
- Data skickas och hämtas mellan en lokal och en eller fler avlägsen databas



Delta-baserad versionshantering

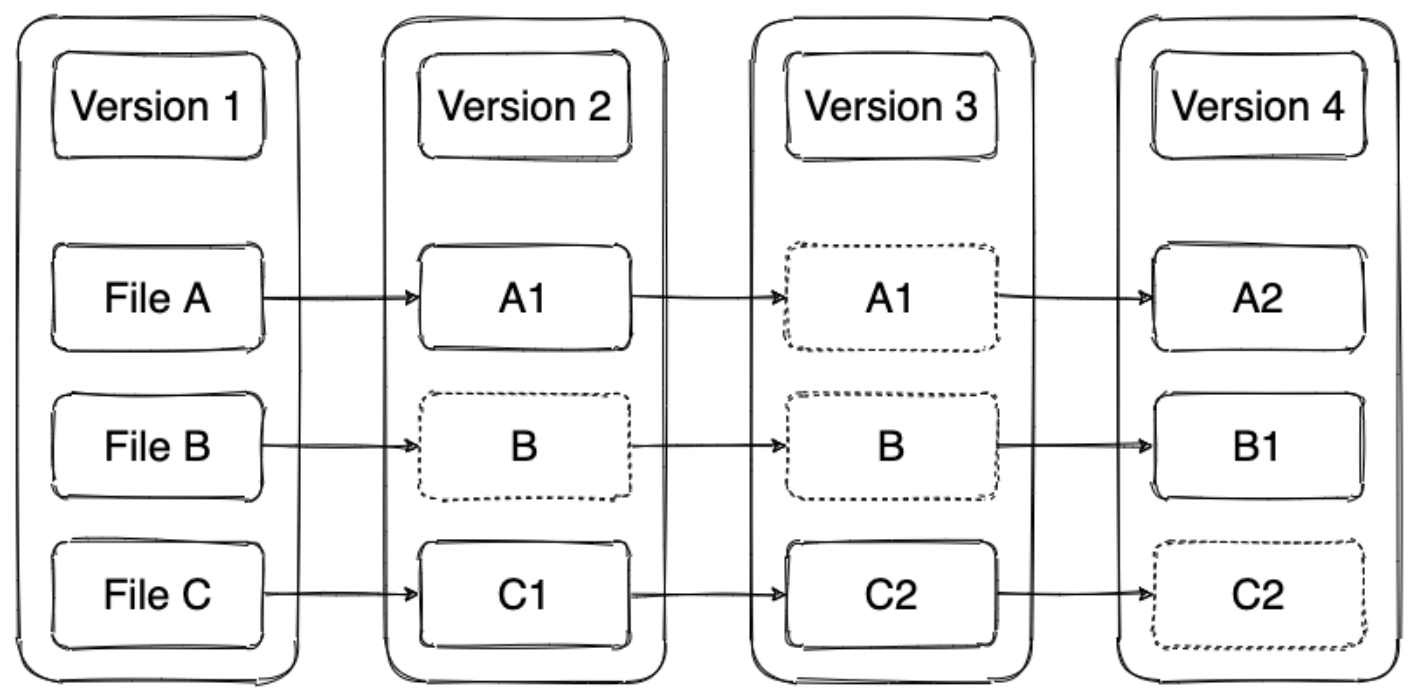
I t ex Subversion lagras varje fil en gång och därefter är versioner bara skillnader (deltan)



Anpassad från: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Versioner genom ögonblicksbilder

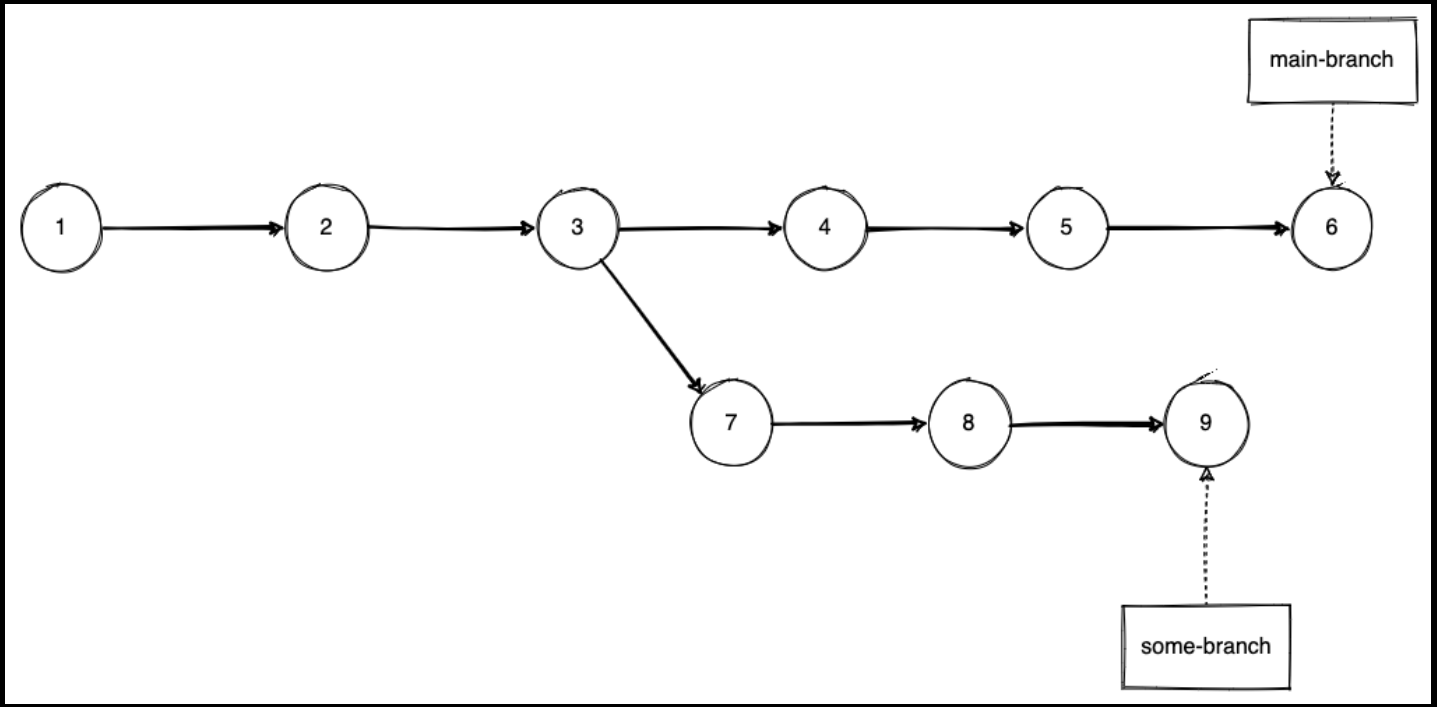
I Git är varje version en komplett filstruktur



Anpassad från: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

En graf med ögonblicksbilder

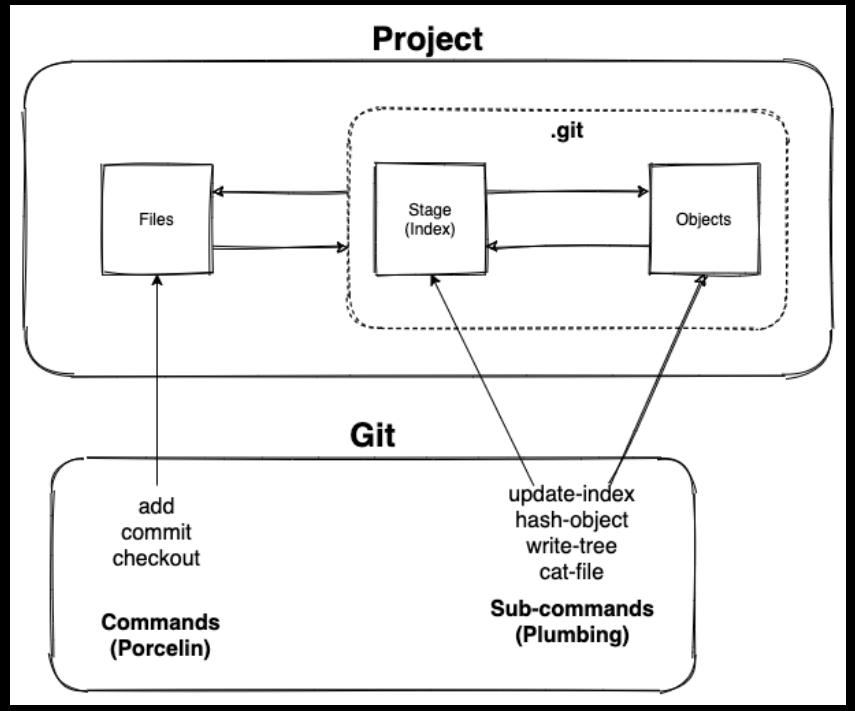
Från användarens synvinkel ser historiken i Git ut som en serie med ögonblicksbilder.



Filer och historik

Porslin och rörmokeri

Förutom vanliga kommandon finns det ett antal specifika sub-kommandon som opererar på interna data-strukturer

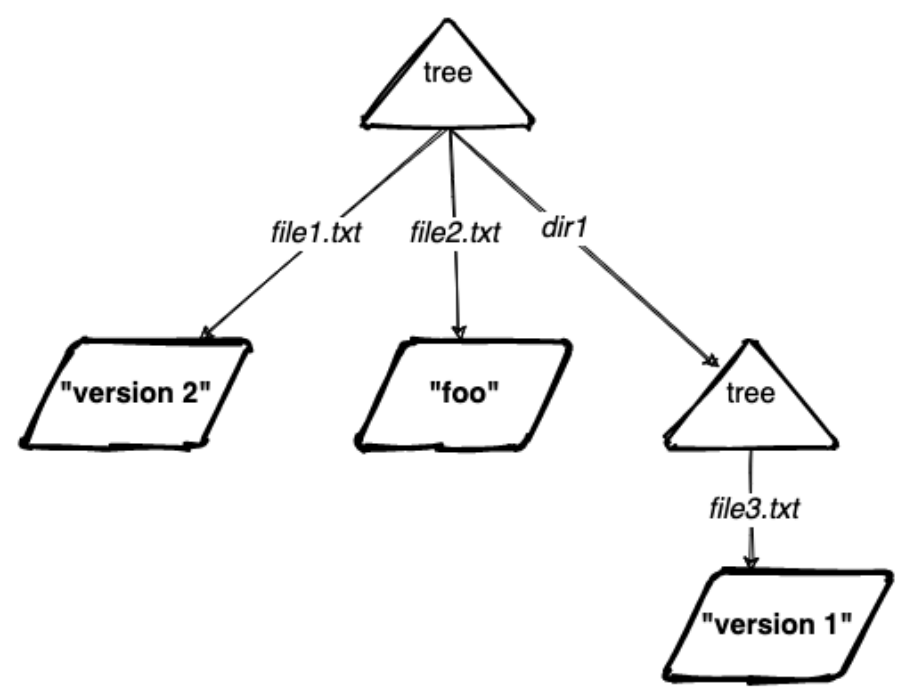


I katalogen `.git`

Namn	Innehåll
HEAD	pekar (indirekt) på den ögonblicksbild som är i fokus
config	projekt-specifik konfiguration (inklusive `remotes`)
index	en magisk binär fil (dyker upp om en stund)
objects/	allt projektets innehåll kommer lagras här
refs/	innehåller pekare till saker i `objects`

Data-klumpar och träd

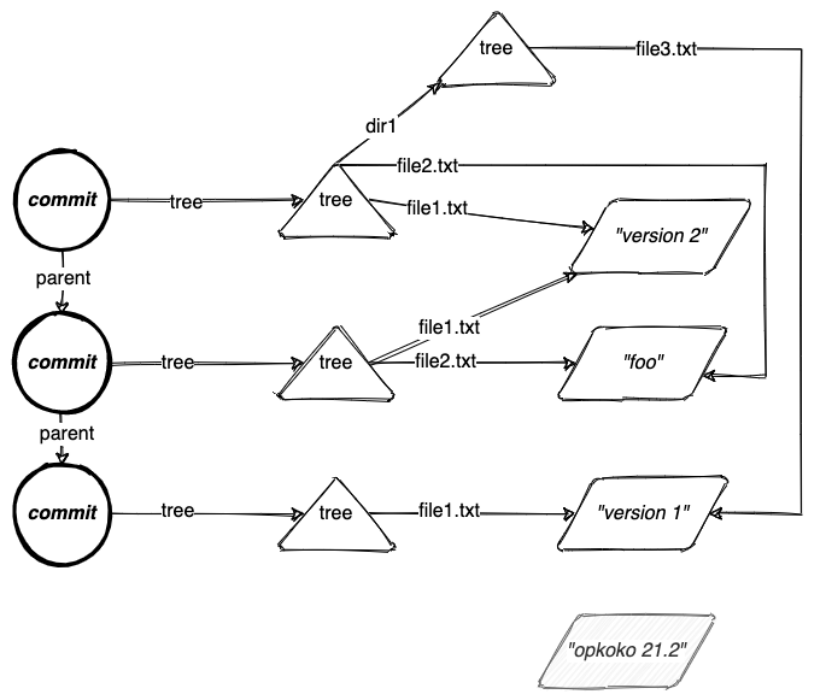
Allt innehåll i Git lagras som data-klumpar (`blobs`) och träd (`trees`)



Anpassad från: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

Förbindelser (`commits`)

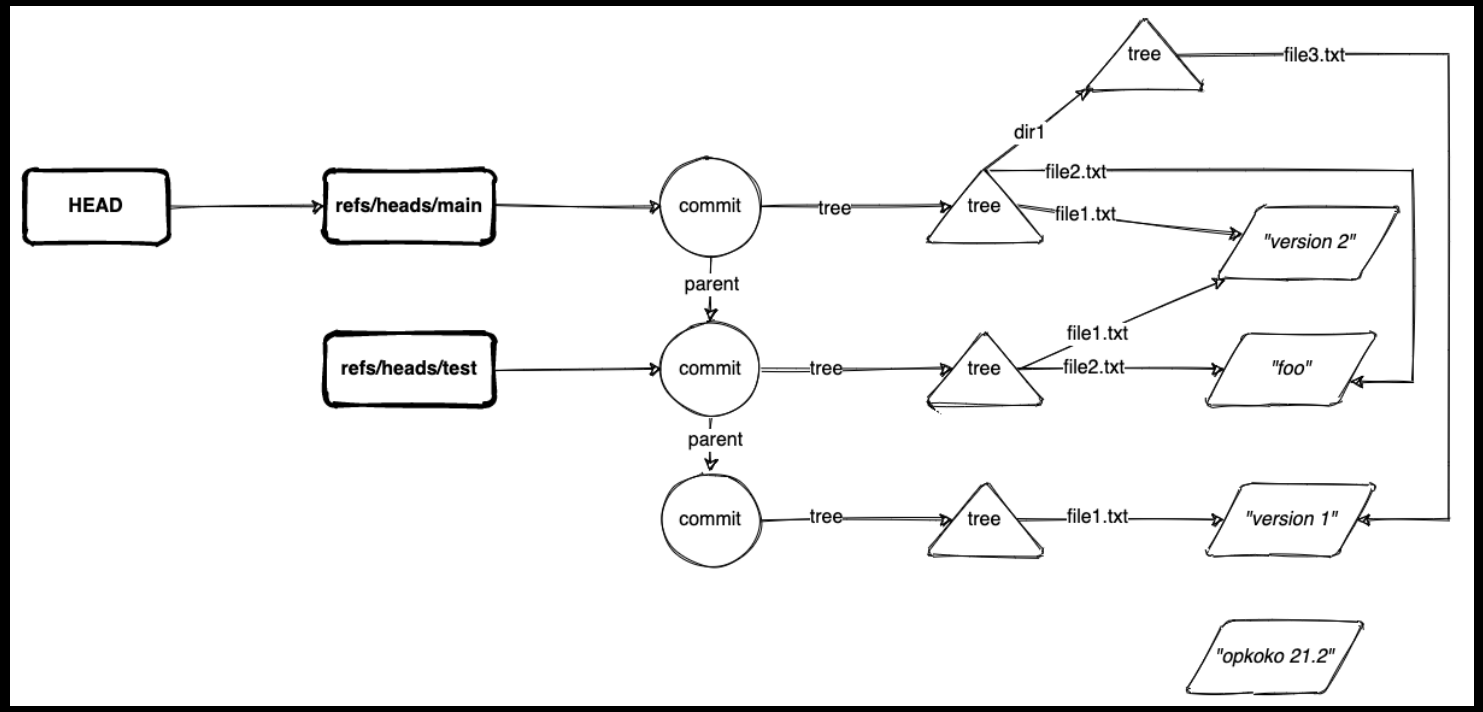
Historiken i Git får vi genom förbindelse-objekt (`commits`)



Anpassad från: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

Referenser och huvudet (`HEAD`)

Grenar i Git är pekare till sha1-summan av senast sparade förbindelsen som representerar en historik



Anpassad från: https://git-scm.com/book/en/v2/Git-Internals-Git-References

Porslin och rörmokeri igen

Vid `add`:

- spara dataklump (`hash-object`)
- uppdatera index-filen som är prepareringsytan (`update-index`)

Vid `commit`:

- skapa nytt träd från prepareringsytan (`write-tree`)
- skapa förbindelse som pekar på det nya trädet och eventuell föregående förbindelse (`commit-tree`)

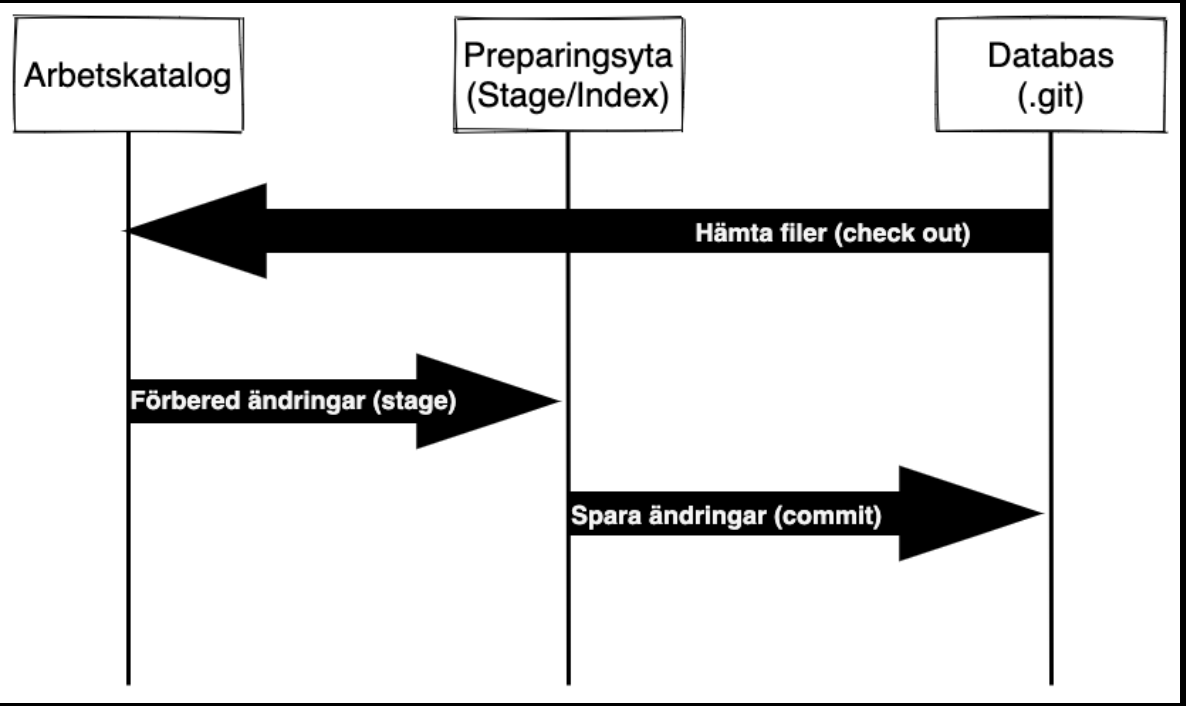
Lärdomar

- Git bygger upp en graf med dataklumpar, träd och förbindelser
- Git skiljer på innehållet och strukturen
- Grenar (och taggar) är inte objekt

Prepareringsytan

De 3 tillstånden

Från användarens synvinkel rör sig data mellan 3 olika ytor:



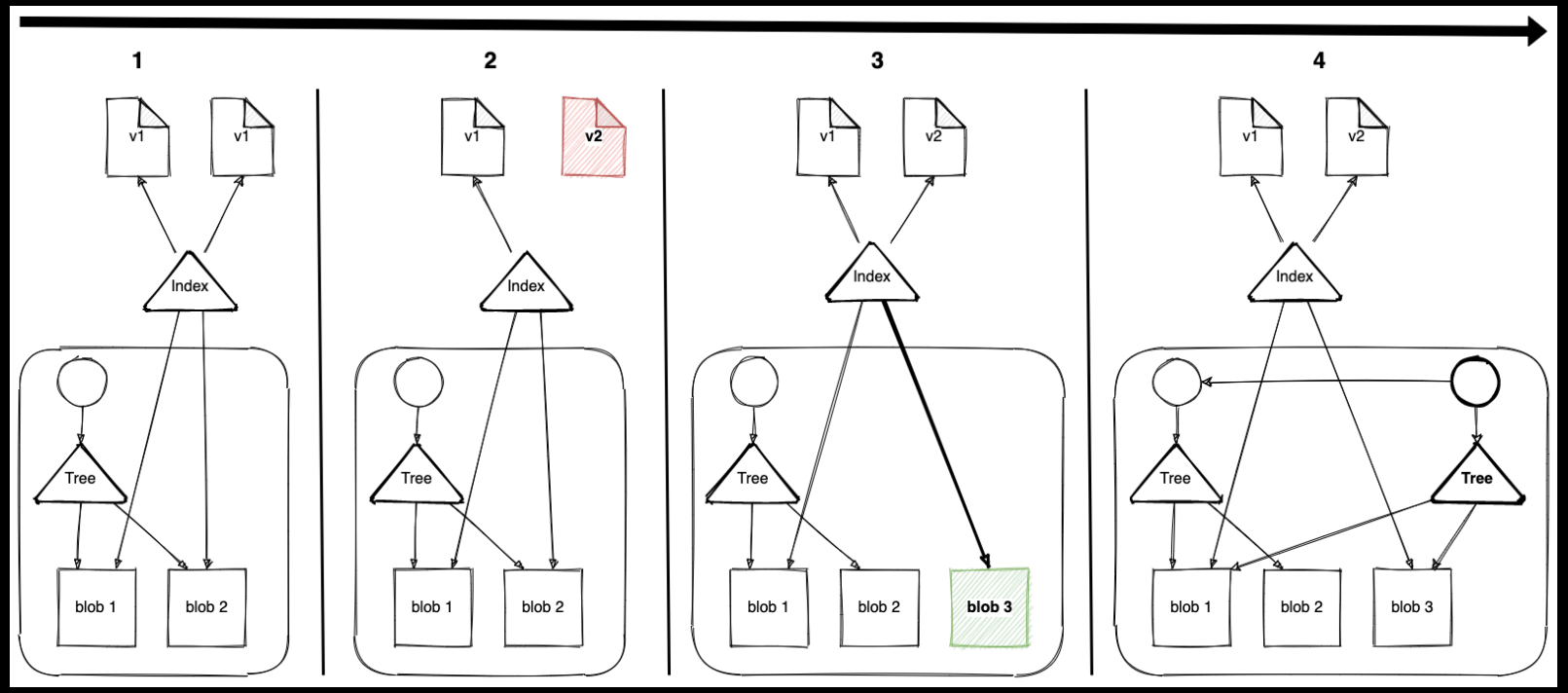
Anpassad från: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Prepareringsytan ("index" eller "staging area")

- Prepareringsytan är en magisk fil i .git-katalogen som:
  - lagrar information för nästa träd (`tree`) som ska skrivas till databasen
  - håller reda på hur tillståndet i filytan och i databasen förhåller sig till varandra

Ett smutsigt index

När en fil ändras i arbetsytan eller lagts till i databasen och ingen ny förbindelse (`commit`) har gjorts har vi ett smutsigt index



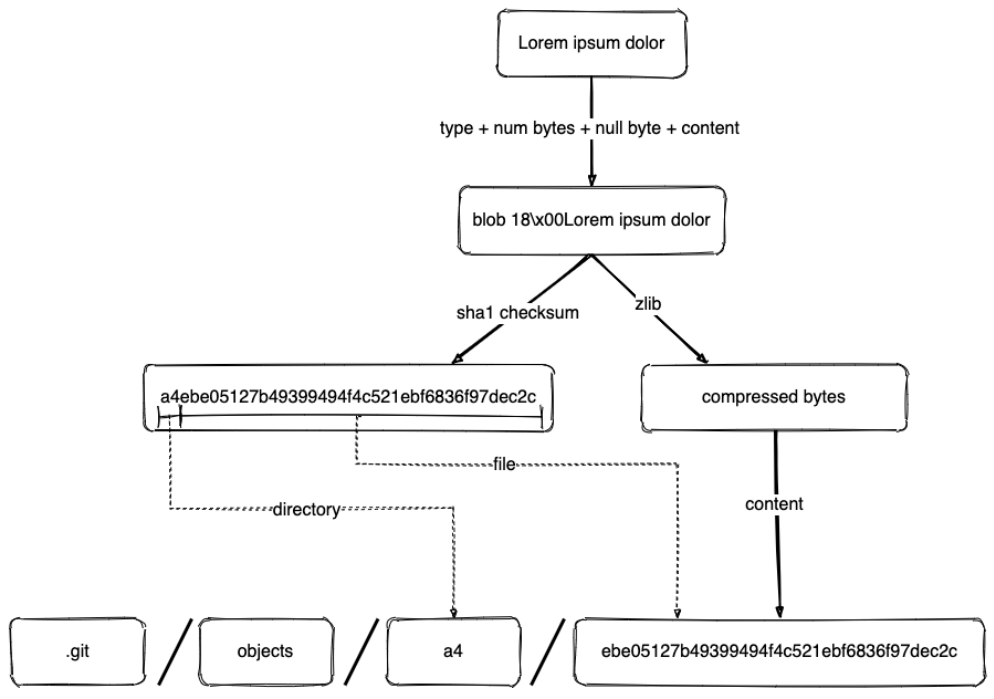
Lärdomar

- Filer lagras i Git redan vid `add`, men träd och förbindelser mellanlagras i ett index (prepareringsytan)
- Med `diff` visas skillnaden mellan arbetsytan och prepareringsytan; med `diff --staged` (för gammalt `--cached`) visas skillnaderna mellan prepareringsytan och databasen

Optimerad lagring

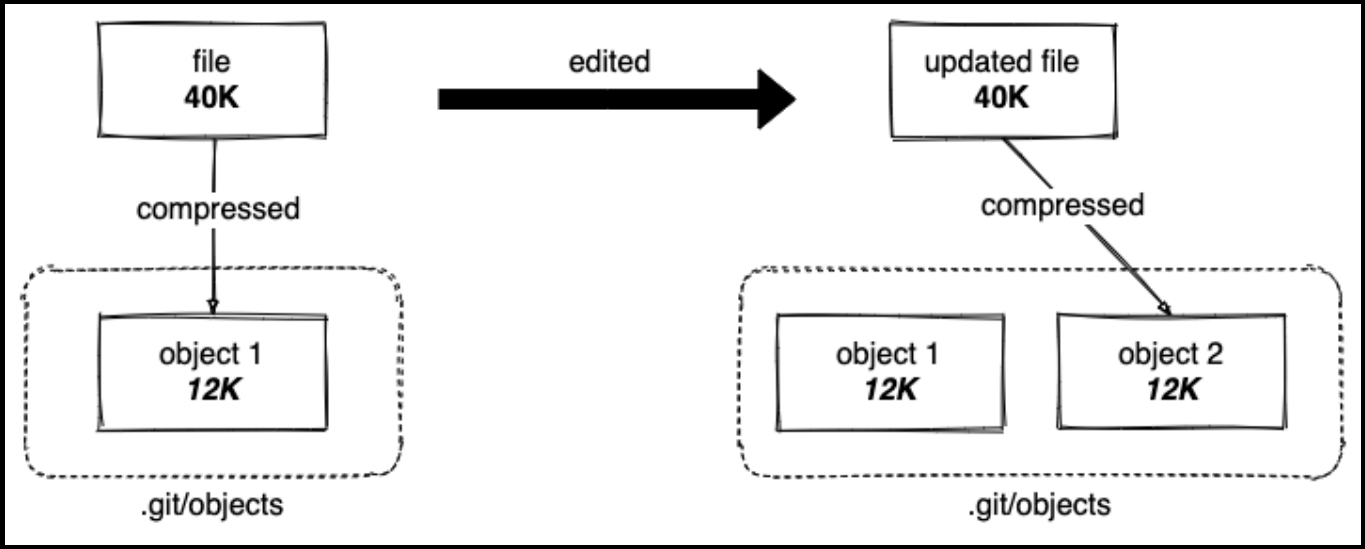
Hur objekten lagras

Alla objekt skrivs i filer som är komprimerade med `zlib` och har namn baserat på `sha1` av innehållet.



Dilemat med små ändringar

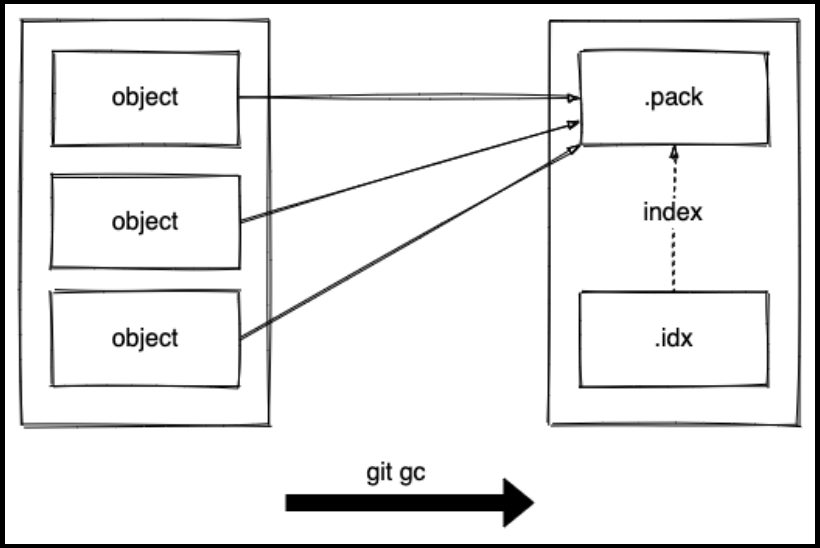
Eftersom varje fil sparas i sin helhet (om än komprimerad) kommer även en liten ändring av en fil kräva onödigt mycket diskutrymme



Lösa objekt och paketeringsfiler

Vid `git gc` så paketeras lösa objekt-filer i `objects/packs`:

- `.idx` - en fil som indexerar innehållet i pack-filen
- `.pack` - binär fil med alla objekten



Lärdomar

- Konceptuellt sett lagrar Git ögonblicksbilder men detta optimeras med delta-baserad paketering
- Git städar regelbundet genom att flytta lösa objekt till paketeringsfiler (men bara de som kan knytas till en förbindelse!)

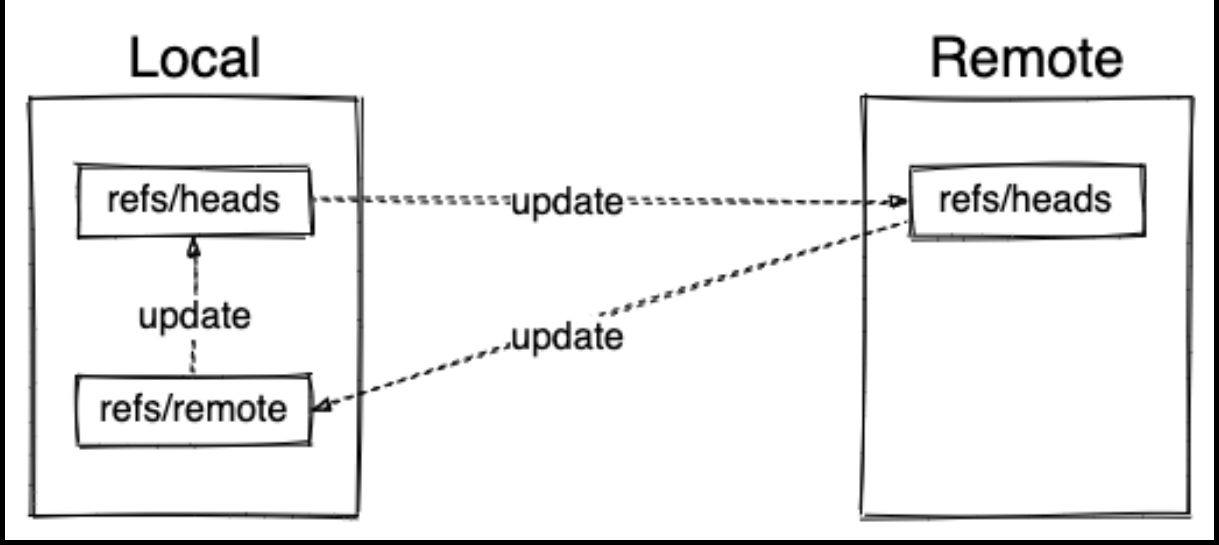
Lokala och fjärran databaser

Konsekvenser av hur Git lagrar data

- Eftersom filnamnet inte är en del av en dataklump kan vilka filer som helst i databasen komprimeras i pack-filer
- Det spelar ingen roll *hur* ett objekt skapas och det är enkelt att avgöra om ett visst objekt finns i databasen
- En förbindelse är en unik identifierare av *hela* historiken från den tidpunkten

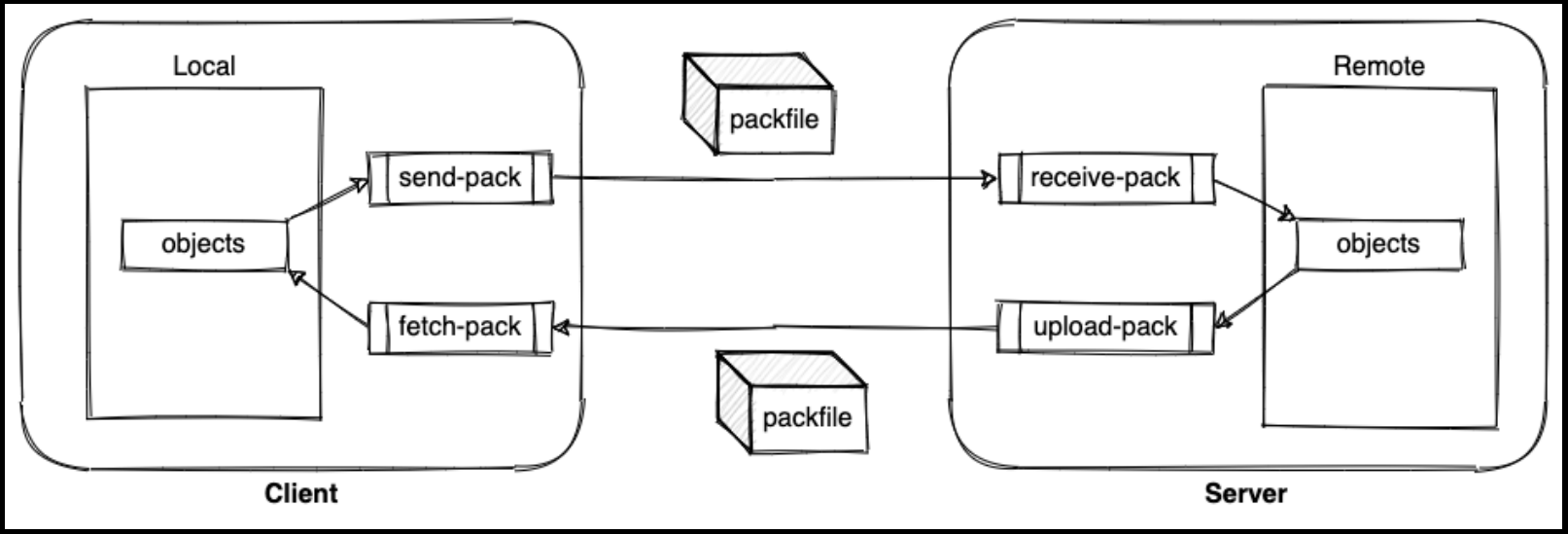
Referenser till fjärran saker

- När vi lägger till andra databaser (`repos`) sparas information om dem i `.git/config`
- Under `refs/remotes` lagras referenser som håller reda på det senast kända tillståndet i fjärr-databasen



Hur objekten skickas och hämtas

När vi talar om för Git att vi vill skicka eller hämta referenser (med en `refspec`) så räknar den ut vad som behöver överföras och skapar en paketeringsfil med detta.

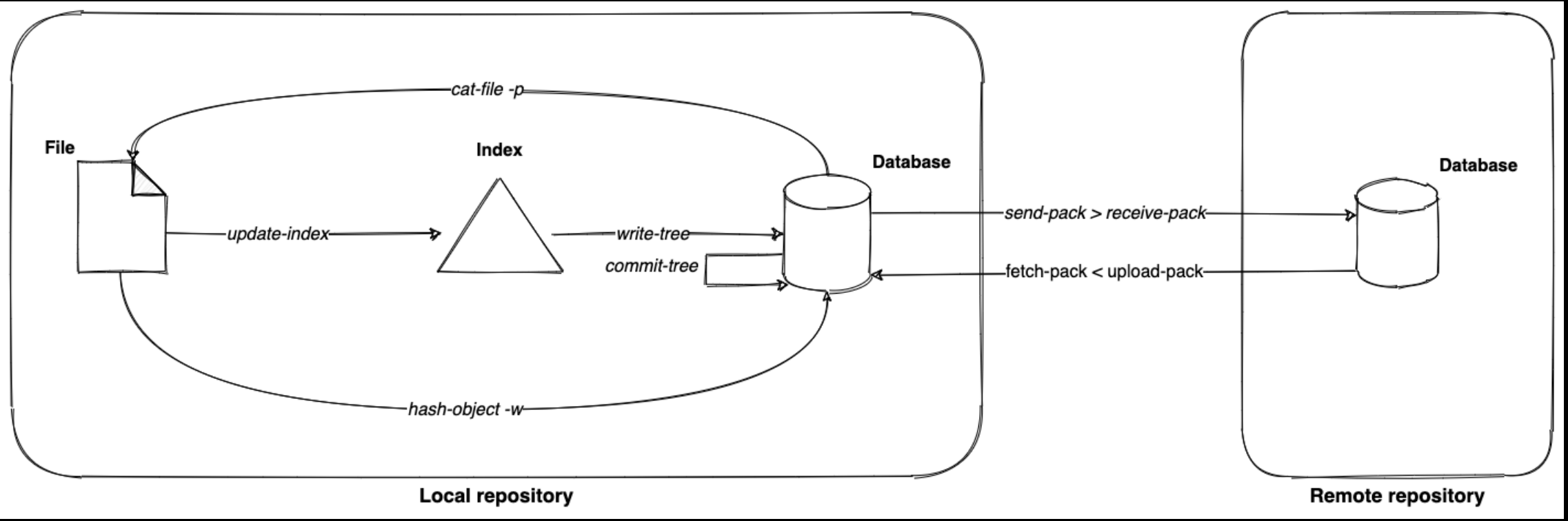


Lärdomar

- Objekt överförs mellan databaser
- Konfiguration och index överförs **inte** mellan databaser
- Referenser **synkroniseras** mellan databaser

Summering och avslut

En mer detaljerad bild



Lärdomar

- Användaren lägger in data i Git ("databasen") och därifrån hämtas den till arbetsytan
- All data sparas i sin helhet i databasen*, men filnamn- och struktur samt historik är meta-data

*Konceptuellt sett åtminstone

Jobba effektivt med Git

Tänk att ni rör er mellan olika tillstånd av *hela* projektet (alltså de faktiskta filerna ni ser) och att vad ni vill göra är att få detta tillstånd som ni vill ha det - Git är bara ett verktyg för att göra detta!

Vidare läsning

- "ProGit" av Scott Chacon & Ben Straub - https://git-scm.com/book/en/v2
- "Version Control with Git" (2nd edition) av Jon Loeliger & Matthew McCullough
- "Advanced Git: Graphs, Hashes, and Compression, Oh My!" av Matthew MCullough - https://www.youtube.com/watch?v=ig5E8CcdM9g

Slut

Bonusmaterial

Skapa objekt med Python

```
import os
import zlib
from hashlib import sha1

content = 'Lorem ipsum dolor\n'
header = f'blob {len(content.encode('utf-8'))}\0'

obj = header + content
sha = sha1(obj.encode('utf-8')).hexdigest()
compressed_obj = zlib.compress(obj.encode('utf-8'))

path = f'.git/objects/{sha[0:2]}/{sha[2:]}'
os.makedirs(os.path.dirname(path))
with open(path, 'wb') as f:
    f.write(compressed_obj)
```

Märken (`tags`)

- Märken (`refs/tags`) är referenser som pekar till commits (oftast)
- Det är som en gren (`refs/heads`) men flyttas inte automatiskt
- Det finns två typer:
  - lättviktigt märke (`lightweight tag`) - endast en referens
  - kommenterat märke (`annotated tag`) - ett eget objekt

Hur Git håller reda på konflikter

- `.git/MERGE_HEAD` innehåller sha1-summan av förbindelsen som mergas in
- I förberedelseytan (`index`) lägger Git 3 versioner av en fil som innehåller konflikter:
  - 1 - sammanslagningsbasen (`merge base`)
  - 2 - "vår" version
  - 3 - "deras" version
- En version av filen med alla konflikter markerade lagras i filytan (finns *inte* i förberedelseytan eller databasen)

Hitta borttappad data

- Det är väldigt ovanligt att data faktiskt försvinner under normalt arbete i Git eftersom objekt bara städas vid `gc`
- Två bra sätt att hitta förbindelser som blivit dolda:
  - `refspec` - visar lokal historik över alla åtgärder som skedd med referenser
  - `fsck --full` - visar bl a objekt som hamnat utanför grafen (`dangling objects`)