La terminologia per i *merge*

La terminologia più usata fa riferimento alla coppia di programmi POSIX:

`diff` calcola la differenza fra due revisioni (R_0, R_1), calcolata per righe, cercando di minimizzare il numero di inserimenti e cancellazioni (ricerca della sottosequenza più lunga)

L'intersezione è: `a b c d f g j z`

Le differenze sono:

```
e h i q k r x y
+ - + - + + + +
```

Il diff è diviso in **hunk** ("fette"): `e, h/i, q/krxy`

`patch` è il programma che permette di applicare il diff non solo a R_0 per ottenere R_1, ma anche a un qualunque R'_0 "vicino" a R_0 (che diventerà un R'_1), applicando alcune euristiche per ogni *hunk*.

R_0

```
a
b
c
d
f
g
h
j
q
z
```

R_1

```
a
b
c
d
e
f
g
i
j
k
r
x
y
z
```

```
4a5
> e

7c8
< h
---
> i

9c10,13
< q
---
> k
> r
> x
> y
```

diff

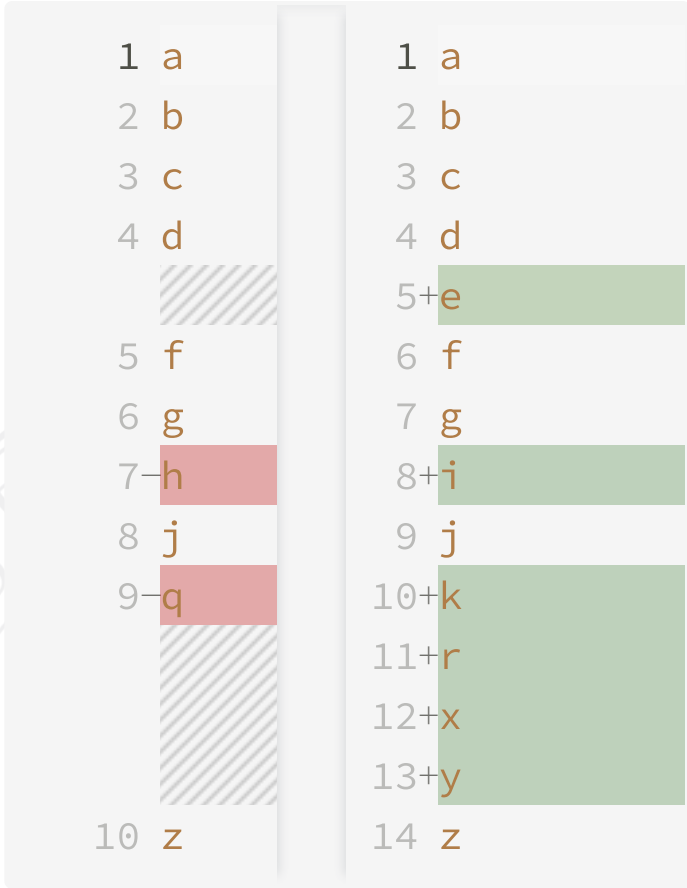
```
a
b
c
d
f
g
h
j
q
z
```

```
a
b
c
d
e
f
g
i
j
k
r
x
y
z
```

```
4a5
> e

7c8
< h
---
> i

9c10,13
< q
---
> k
> r
> x
> y
```



https://git-scm.com/docs/merge-strategies

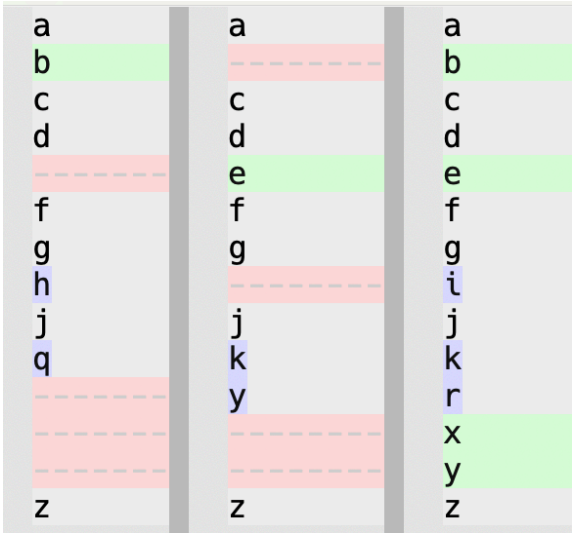
3-way merge

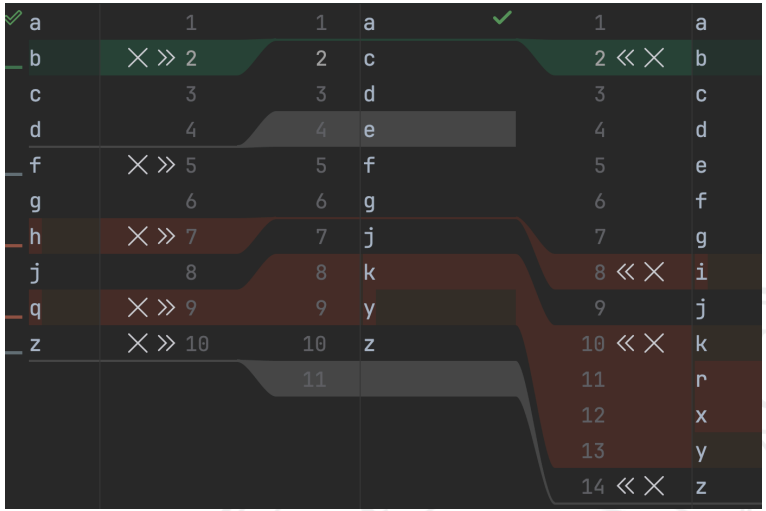
Quando, come nel caso di lavoro parallelo sullo stesso *artifact*, le due revisioni non hanno una chiara gerarchia temporale ma hanno un antenato comune (per esempio la revisione da cui entrambi sono partiti) si può *facilitare* il lavoro di merge.

Siano A' e A'' due revisioni, con antenato comune A

- *hunk* uguale nelle tre revisioni: inalterato
- *hunk* uguale in due delle tre revisioni
  - A' e A'' uguali: merge $\Rightarrow A'$
  - A e A' uguali: merge $\Rightarrow A''$
  - (A e A'' uguali: merge $\Rightarrow A'$)
- *hunk* diverso nelle tre revisioni deve essere valutato a mano

Esempio

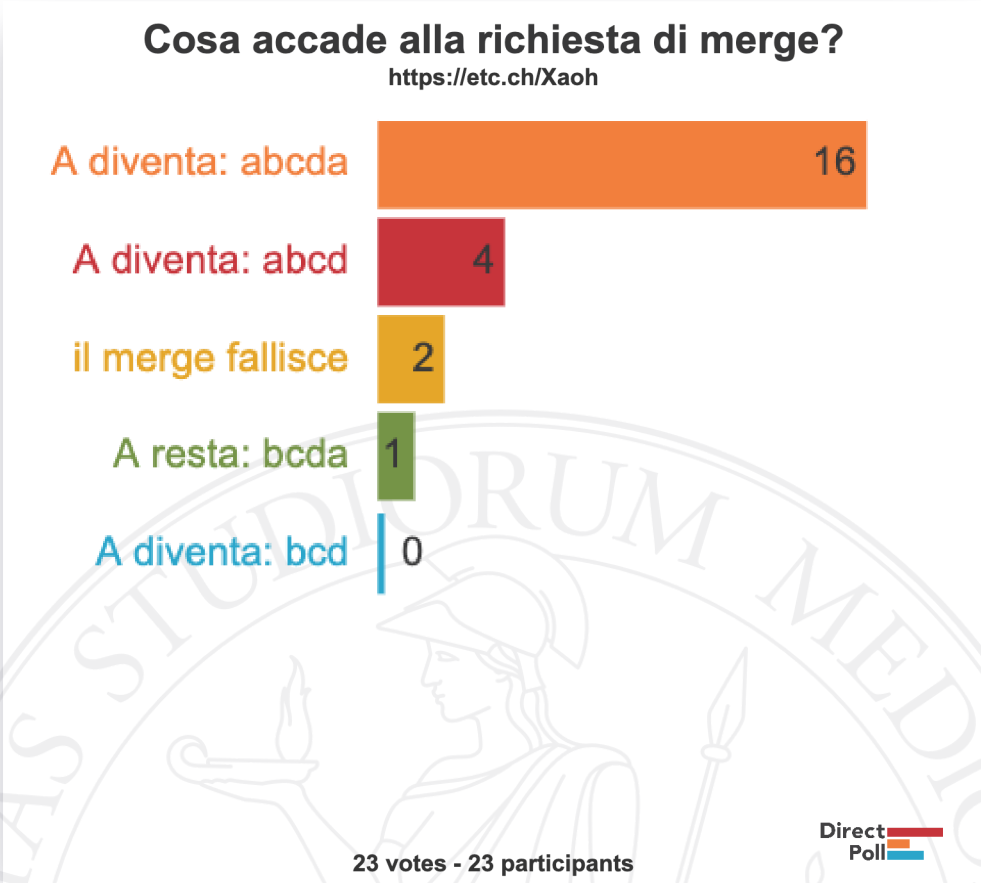





```
a
b
c
d
f
g
<<<<<<< carlo
h
j
q
=======
i
j
k
r
x
y
>>>>>>> mattia
z
```

Domanda

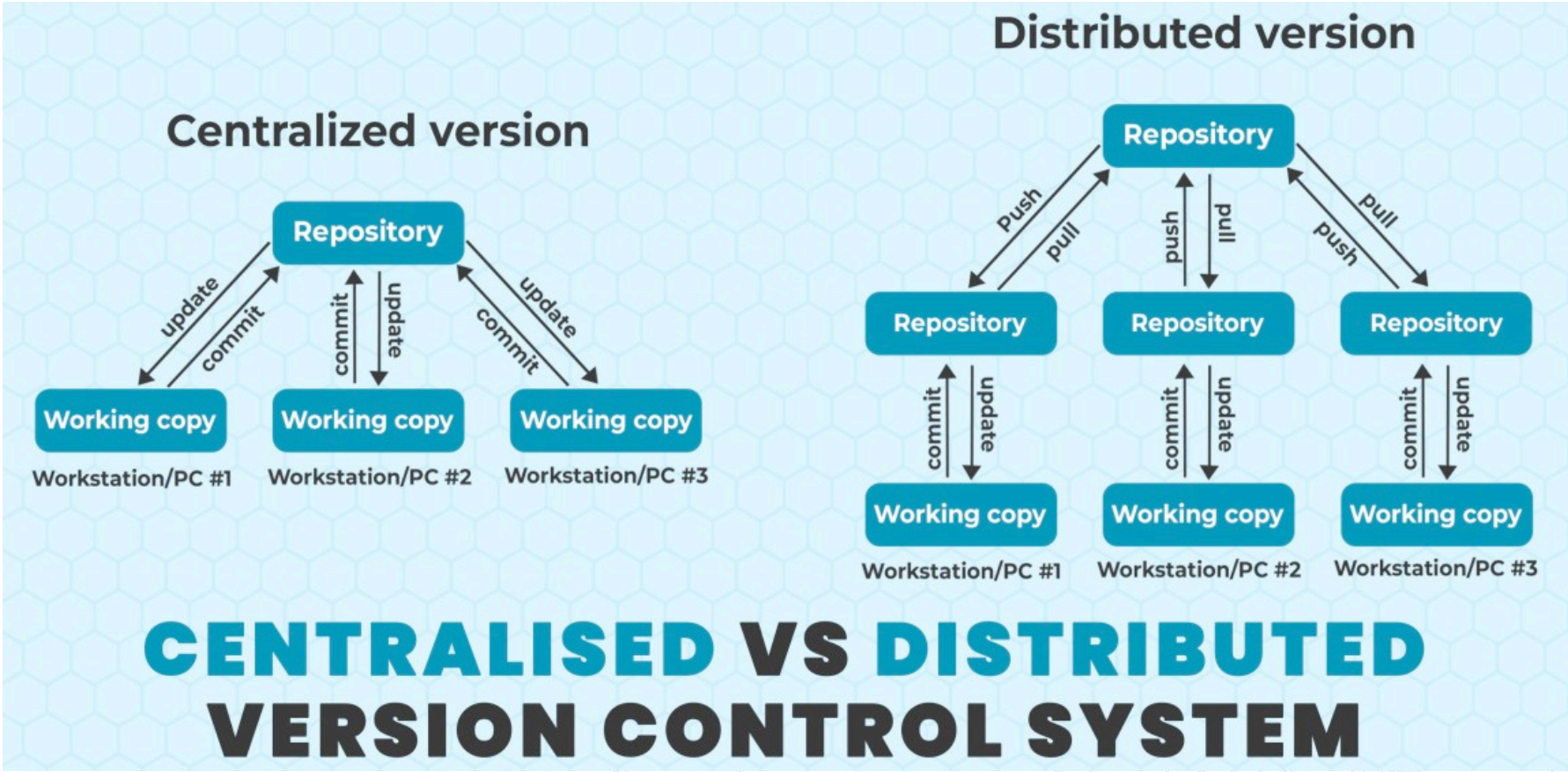
- Il file A inizialmente contiene tre linee: **bcd**
- Mattia sul suo ramo aggiunge una linea **a** in fondo e fa commit
- anche Carlo sul suo ramo aggiunge una linea **a** in fondo e fa commit
- poi Mattia ci ripensa e fa un commit in cui cancella la linea **a** dal fondo
- Mattia fa un ulteriore commit in cui mette la riga **a** all'inizio
- Carlo fa merge con il ramo di Mattia



Git merge strategies

- resolve -> recursive -> ort (2021)
- octopus
- ours
- subtree

Versioning distribuito



UNIVERSITÀ DEGLI STUDI
DI MILANO

Git

https://www.youtube.com/embed/4XpnKHJAok8 - https://gist.github.com/dukeofgaming/2150263

Linus presenta git a Google

Versioning distribuito

- Internals
- Architettura
- Alcuni comandi un po' più avanzati: History rewriting
- Alcuni workflow

https://youtu.be/fWMKue-WBok

GIT Internals

GIT INTERNALS (INTRO)

WHAT HAPPENS WHEN WE GIT COMMIT?

WHAT IS STORED AT EACH COMMIT?

WHAT DOES GIT INIT ACTUALLY DO?

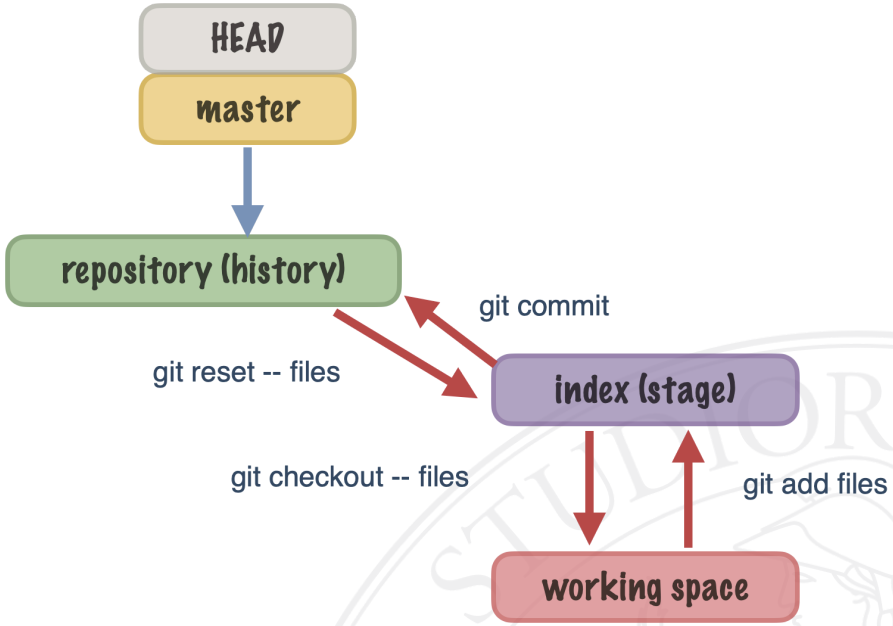
WHY DO WE CARE?

WE ARE PROS

IT'S USEFUL

IT'S COOL ^^

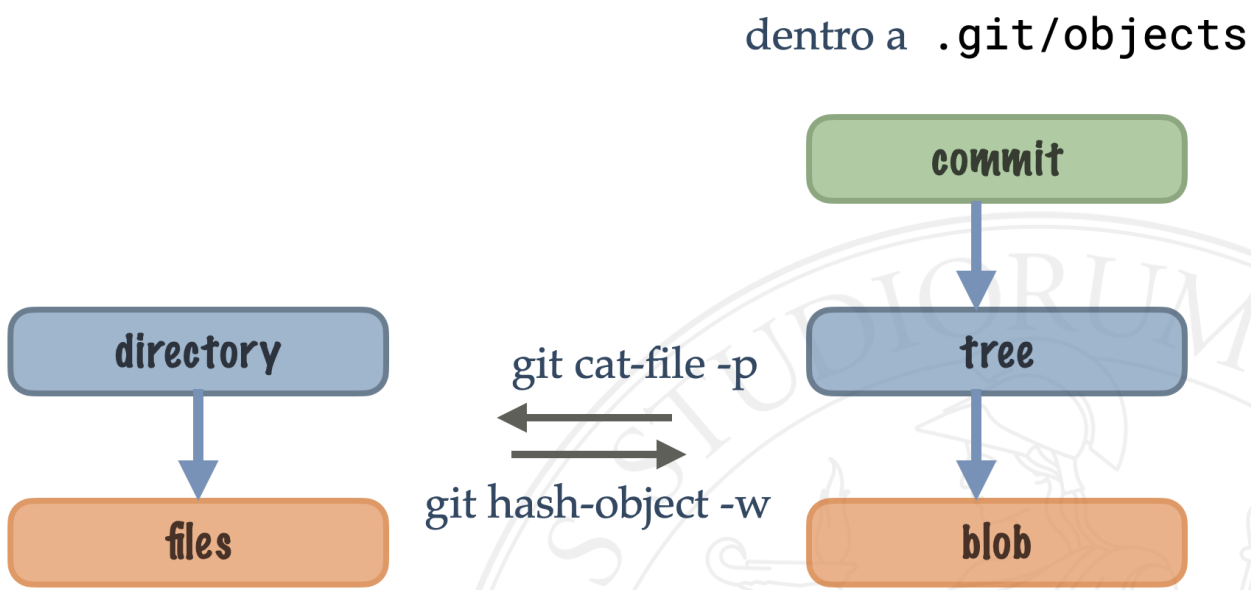
index



- contiene tutte le informazioni per potere fare il commit
- ogni volta che viene fatto un git -add vengono creati gli object relativi a tali snapshot

http://arokem.github.io/2013-09-16-ISI/lessons/git-notebook/git-for-scientists.slides.html
http://eagain.net/articles/git-for-computer-scientists

Working space vs .git/…



Minimal requirement

```
git init
```

```
.git
├── HEAD
├── config
├── description
├── hooks
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

```
mkdir -p .git/refs
mkdir -p .git/objects
printf "ref: refs/heads/main" > .git/HEAD
```

Formato degli objects

```
function putRawObject(content: Buffer, type: string, objDir: string): string {
  const size = content.length.toString();
  const header = Buffer.from(`${type} ${size}\0`);
  const store = Buffer.concat([header, content]);

  const sha1 = crypto.createHash('sha1').update(store).digest('hex');
  const directory = path.join(objDir, sha1.substring(0, 2));
  const objectPath = path.join(directory, sha1.substring(2));

  if (!fs.existsSync(objectPath)) {
    const compressed = zlib.deflateSync(store);

    fs.mkdirSync(directory, { recursive: true });
    fs.writeFileSync(objectPath, compressed);
  }
  return sha1;
}
```

Simuliamo *a mano*

A↩
B↩
C

```
$ cont="A\nB\nC"
$ printf "blob %d\0$cont" $(printf $cont | wc -c) | pigz -z | od -x

0000000      5e78    ca4b    4fc9    3052    7065    72e4    72e2    0006
0000020      0d16    cf02

$ printf "blob %d\0$cont" $(printf $cont | wc -c) | shasum

870951ade2b2b7af4a1912009fe0e2b91ffd088f
```

```
$ printf $cont > A
$ git add A
$ cat .git/objects/87/0951ade2b2b7af4a1912009fe0e2b91ffd088f | pigz -d | od -a

0000000    b   l   o   b  sp   5 nul   A  nl   B  nl   C
```

```
$ git cat-file -t 87095

blob
```

```
$ git cat-file -s 87095

5
```

```
$ git cat-file -p 87095

A
B
C
```