Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Dipartimento di Informatica · Scienza e Ingegneria · DISI
Corso di Laurea in Ingegneria Informatica

DATABASE A GRAFI PER SISTEMA DI CONTROLLO DELLE VERSIONI

Relatore:
Prof. Paolo Ciaccia

Presentata da:
Riccardo Barbieri

Anno Accademico 2021/2022

Elenco delle figure

Indice

Introduzione

Un VCS (Version Control System) è un tipo di sistema utile a gestire le versioni di un progetto. Nell'ambito dello sviluppo di progetti software, la funzionalità principale di un sistema di questo tipo è la possibilità di tenere traccia delle modifiche apportate al codice sorgente, ai file di documentazione e ad altri contenuti di un progetto.
Senza l'utilizzo di un VCS, gli sviluppatori di un progetto potrebbero semplicemente mantenere diverse copie dei contenuti, classificandole a seconda della versione e delle modifiche apportate da una versione all'altra; un approccio simile tuttavia diventa ingestibile per progetti di grandi dimensioni, in quanto si verrebbero a creare molteplici copie di file uguali o molto simili tra di loro.

In questo lavoro di tesi verrà illustrato e discusso il processo di sviluppo di un rudimentale VCS, chiamato GGit (ispirato a Git), basato su un database a grafi e implementato come una applicazione CLI in Python 3.10.

I database a grafi sono database che fanno uso di strutture a grafi per immagazzinare informazioni. Questo tipo di struttura è composta da due elementi fondamentali, nodi (o vertici) e archi, applicando all'ambito dei database, i nodi sono utilizzati per rappresentare le entità, mentre gli archi rappresentano le relazioni tra le entità.

Una delle proprietà più vantaggiose dei database a grafi utilizzati per questo progetto, ovvero LPG o labeled-property graph, consiste nel fatto che il tempo di attraversamento di un grafo è costante[1].

1 | Scelte progettuali

Dato che Git è il sistema di versioning più utilizzato dagli sviluppatori di software[2], si è deciso di ispirarsi al suo modello di funzionamento per la realizzazione del primo VCS a grafi, chiamato `GGit`.

1.1 Strutture dati di GGit

Git è nato come un semplice content-addressable filesystem, ovvero un sistema di storage basato suuna relazione chiave-valore, nel quale è possibile archiviare qualsiasi tipo di contenuto digitale.Essenzialmente, l'azione di base che Git esegue è quella di memorizzare i contenuti di un file associandoli ad un hash calcolato su di essi: questa è la chiave che può essere utilizzata per accedere ai dati salvati.

L'hash di ogni oggetto (nomenclatura utilizzata per file e altre entità descritte in seguito), è calcolato usando l'algoritmo SHA-1, una funzione di hashing che genera un hash di 20 byte. Pur essendo un algoritmo di hashing crittografico, SHA-1 non è sicuro, in quanto è stato provato che è possibile generare collisioni[3] (ovvero due input che generano lo stesso hash). Tuttavia, per l'utilizzo al quale è destinato all'interno di un VCS (cioè identificare univocamente il contenuto di un file) la complessità di SHA-1 è sufficiente, in quanto, nel peggior caso trovato, è stato necessario effettuare 2^{61} operazioni per ottenere una collisione[4].

Git si basa su 5 tipi di oggetti:

- blob: oggetti che rappresentano un file;
- tree: oggetti che rappresentano una directory;
- commit: oggetti che pongono un particolare tree (la root directory) nella storia della repository, associando un commit padre, un timestamp e un messaggio che descrive le modifiche apportate;
- tag: un container che riferisce un altro oggetto, usato prevalentemente per dare un nome a un commit;
- packfile: una versione compressa di una serie di oggetti.

Il contenuto che verrà dato come input alla funzione di hash sarà una stringa che racchiude le caratteristiche dello specifico oggetto, fattore che aiuta a ridurre ulteriormente la probabilità di collisioni.

Si è deciso di strutturare il modello del dominio di GGit considerando i primi tre tipi di oggetti: blob, tree e commit. Di seguito viene illustrato come vengono calcolati gli hash di ogni tipo di oggetto.

1.1.1 Blob

Un file è rappresentato da un blob, per calcolarne l'hash si considera la caratteristica principale, ovvero il suo contenuto, che viene considerato in byte sia per file in puro testo ASCII che binari. Viene poi calcolata la lunghezza in byte del contenuto e il valore risultante viene concatenato con la stringa `"blob "`, seguita da un carattere NULL il tutto seguito dal contenuto del file:

```
blob <file_length>\0<file_content>
```

1.1.2 Tree

Un tree è essenzialmente una collezione di blob e tree, la funzione di un tree è associare a blob e tree un nome e una stringa che esprime i permessi sul file. I permessi sono espressi in modo analogo alla modalità utilizzata dai sistemi UNIX, ovvero tre caratteri ottali che indicano i permessi di lettura, scrittura e esecuzione per il proprietario, il gruppo e gli altri utenti, una cifra ottale che rappresenterebbe i bit setuid, setgid e sticky (ignorati da Git) e altri tre caratteri ottali che indicano la POSIX mode[5] dell'oggetto:

- 04 per directory;
- 10 per file regolari;
- 12 per link simbolico;
- 16 per Git link o sottomoduli, non presente tra le classiche POSIX modes (non supportati da GGit).

L'hash di un tree viene calcolato in modo analogo a quello di un blob:

```
tree <tree_length>\0<tree_content>
```

dove `tree_length` è la lunghezza in byte del `tree_content`, che è una lista di oggetti, in ordine alfabetico per nome, nel seguente formato:

```
<object_mode> <object_name>\0<object_hash>
```

La stringa `object_name` è il nome del file o directory e object_hash l'hash dell'oggetto corrispondente. La `object_mode` è la stringa di sei caratteri ottali che identifica il tipo e i permessi dell'oggetto descritta in precedenza, questa stringa in GGit può assumere i seguenti valori:

- 100644 per file regolari;
- 100755 per file eseguibili;
- 040000 per directory (altri tree);
- 120000 per link simbolici,

nel caso in cui la stringa abbia zeri iniziali, questi vengono troncati (regola applicabile soltanto nel caso delle directory).

1.1.3 Commit

La funzione principale di un commit è quella di associare lo stato corrente dell'area di commit a un punto nella storia della repository e facilitare la reperibilità di un particolare stato della repository. L'area di commit, in GGit chiamata `stash`, è essenzialmente una lista di file, questi file sono quelli che verranno aggiunti al prossimo commit. Per calcolare l'hash di un commit si procede in modo analogo a quello degli altri tipi di oggetti:

```
commit <commit_length>\0<commit_content>
```

dove `commit_length` è la lunghezza in byte del `commit_content`, che è una stringa formattata come segue:

```
tree <tree_hash>
parent <parent_hash>
author <author_name> <<author_email>> <author_timestamp> <timezone>
committer <committer_name> <<committer_email>> <committer_timestamp> <timezone>

<commit_message>
```

Il `tree_hash` è l'hash del tree che rappresenta la root directory dell'area di lavoro, il `parent_hash` è l'hash del commit padre, ovvero il commit dal quale è stato creato il commit corrente, se il commit è il primo della storia della repository, questo campo è omesso. Le due linee successive contengono le informazioni che identificano rispettivamente l'utente che ha implementato le modifiche (author) e l'utente che ha

creato il commit (committer). A oggi GGit non supporta la possibilità di applicare patch a commit passati come Git[6], è però possibile, al momento della creazione del commit, specificare un autore diverso dal committer, per dare credito all'effettivo autore del codice e non solo all'utente che lo ha accettato all'interno della repository. Infine l'ultima riga è il messaggio associato al commit, stringa che idealmente descrive le modifiche che vengono implementate con il commit.

1.2 Database

I tipi di oggetti in gioco hanno un alto livello di interconnessione tra di loro: ogni commit è relativo a un tree, un tree può contenere molteplici blob e tree e un tree diverso può contenere un sottoinsieme di blob e tree di un altro tree. Data questa caratteristica, si è scelto di utilizzare un database a grafi.

La gamma di diversi tipi di database a grafi è ampia (anche se non come nel caso dei database relazionali), si è scelto di utilizzare per questo progetto Neo4j, un database a grafi scalabile orizzontalmente del quale esiste una community edition totalmente opensource[7]. Neo4j è inoltre il database a grafi più utilizzato[8], con una vasta community di sviluppatori, il che aiuta in fase di sviluppo per trovare soluzioni ad eventuali problemi.

Un'altra caratteristica che ha portato alla scelta di questo particolare database sono i tool a supporto dello sviluppo, come ad esempio Neo4j Desktop o Browser, che espongono una interfaccia grafica intuitiva per eseguire query e visualizzare rappresentazioni grafiche dei grafi risultanti; è disponibili inoltre Neo4j AuraDB che permette di effettuare il deployment di un database a grafi in cloud, permettendo quindi di creare repository condivise senza necessità di gestire infrastrutture server.

1.2.1 Sviluppo del modello di dati

Il modello di dati è stato sviluppato in modo da poter rappresentare in modo completo tutti i tipi di oggetti presenti in GGit e grazie alla caratteristica schema-less dei database a grafi, sarà possibile aggiungere nuovi tipi di oggetti o caratteristiche al modello del dominio senza intaccare maggiormente la struttura preesistente.

Anche se il database scelto è schema-less, è comunque buona pratica definire e documentare in modo preciso il modello dei dati, così da evitare in futuro di fare scelte incompatibili con il modello preesistente. Per sviluppare il modello dei dati è possibile sfruttare il modello "whiteboard", che permette di rappresentare le entità in gioco in modo grafico per esprimere ad alto livello le entità e le relazioni in gioco per poi formalizzare il modello senza dover convertire le decisioni prese in tabelle.

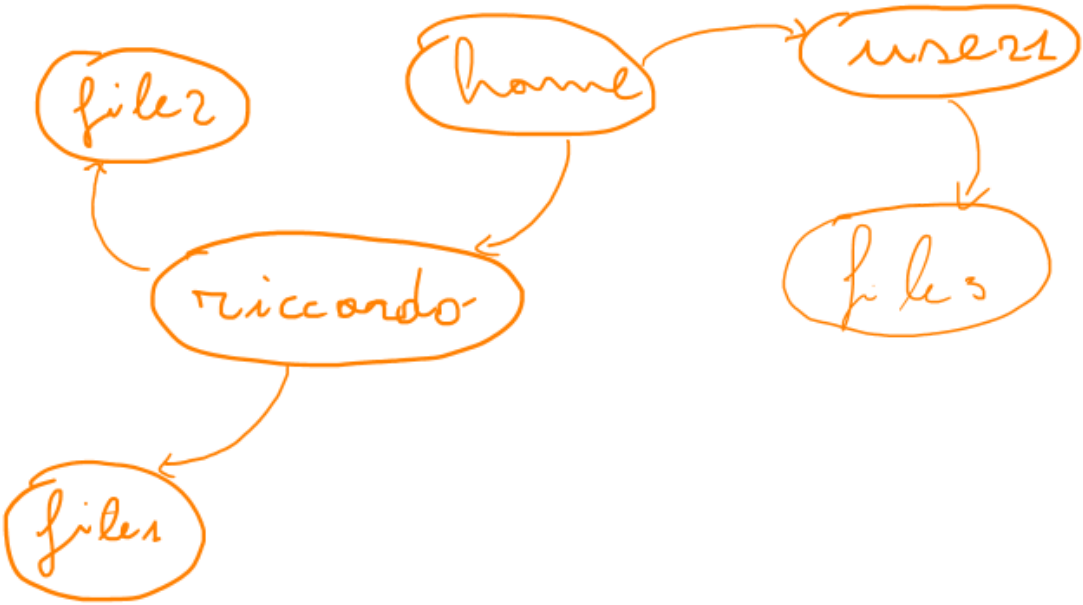

Figura 1.1: Whiteboard sketch della struttura di un'ipotetica repository

Dopo aver creato il cosiddetto "whiteboard model" si procede alla formalizzazione delle entità e relazioni individuate, etichettando con label le entità e relazioni e associandovi tutte le proprietà.

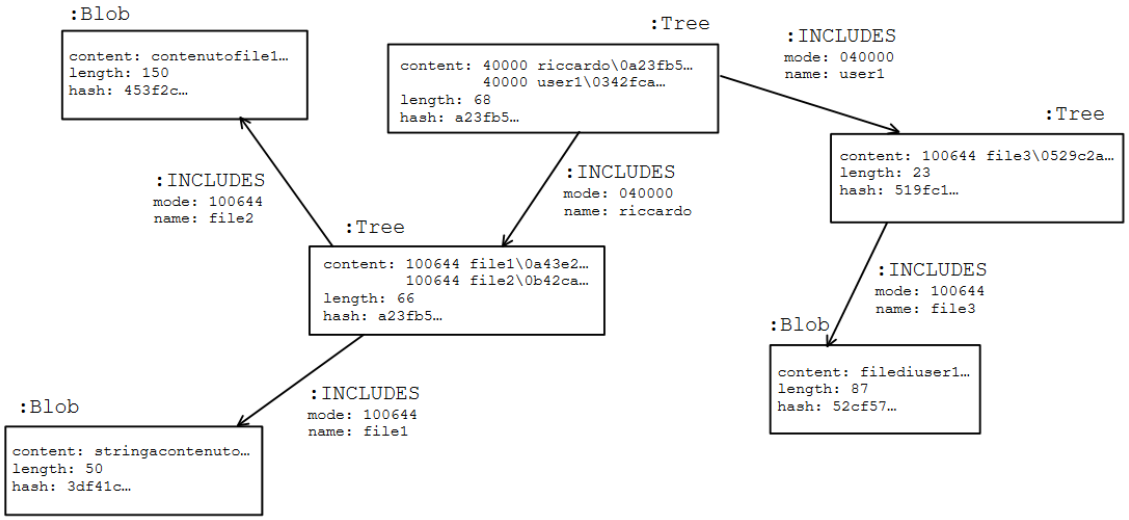

Figura 1.2: Formalizzazione dello sketch con proprietà e label

Come si nota dalla figura precedente si è deciso di assegnare i label (case sensitive) `:Tree` e `:Blob` alle rispettive entità, si individuano inoltre le seguenti proprietà per entrambi i tipi di oggetti:

- `hash`: identificatore univoco dell'oggetto, calcolato con l'algoritmo SHA-1;
- `size`: dimensione in byte del contenuto;

- `content`: contenuto dell'oggetto, come descritti sopra.

Per le relazioni, in questo sottoinsieme del modello se ne individua soltanto una: il label `:CONTAINS` indica che oggetto ne contiene un altro. Le relazioni sono ordinate e nel caso di `:CONTAINS` il nodo di partenza è sempre un `:Tree` e il nodo di arrivo può essere sia un `:Tree` che un `:Blob`. Si denotano inoltre due proprietà per questa relazione:

- `name`: nome del file o della cartella;
- `mode`: GGit mode dell'oggetto, può assumere il valore "040000" se il nodo di arrivo ha il label `:Tree` e i valori "100644", "100755", "120000" se il nodo di arrivo ha il label `:Blob`.

È stata fatta la scelta di associare le proprietà `name` e `mode` alla relazione e non al nodo in quanto rispecchia correttamente il modello dei dati definito: un oggetto `blob`, così come un `tree`, non ha un nome o una mode associati fino al momento in cui viene inserito all'interno di un `tree`.

Sotto viene mostrato il whiteboard sketch relativo al modello dei `commit` e degli `user`, e come possono essere collegati tra di loro:

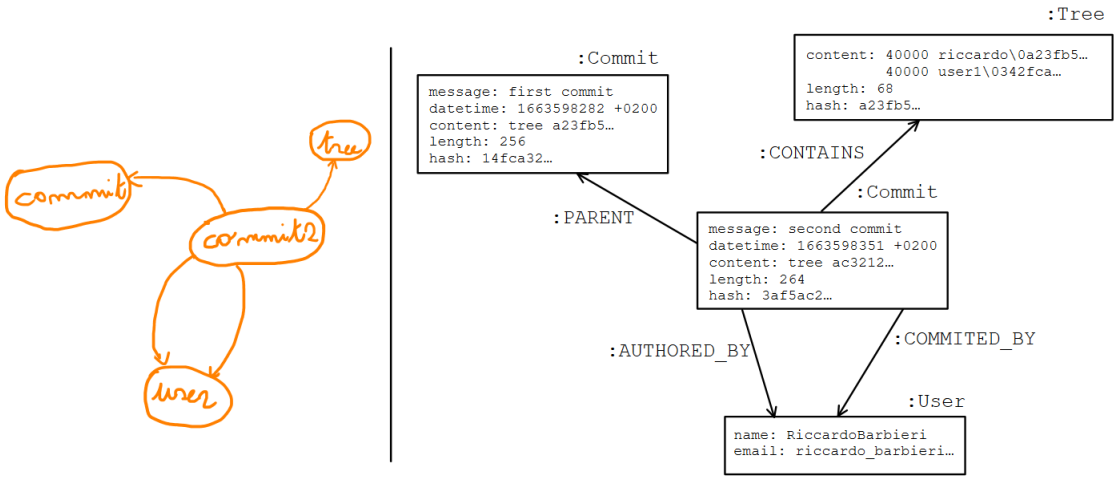

Figura 1.3: Whiteboard sketch del modello dei commit e degli user

In questa sezione del modello si individuano due nuove entità, `:Commit`, che è identificato dalle seguenti proprietà:

- `message`: messaggio del commit;
- `datetime`: momento di creazione del commit (UNIX timestamp) e fuso orario (in ore dal UTC);
- `content`: stringa che rappresenta il commit (quella che viene data come input all'algoritmo di hashing).

- `length`: lunghezza del content.
- `hash`: identificatore univoco del commit, calcolato come descritto sopra;

Si identificano inoltre nuove relazioni: `:COMMITTED_BY` e `:AUTHORED_BY` che indicano rispettivamente il committer e l'autore di un commit; `:PARENT` che parte da un commit e punta al commit padre e infine `:CONTAINS` che parte da un commit e punta ad un oggetto `tree` che rappresenta lo stato della repository al momento del commit. Si denota infine l'entità `:User`, identificata dalle seguenti proprietà:

- `name`: nome dell'utente;
- `email`: email dell'utente;

Come dimostrato, il modello whiteboard è estremamente utile per formalizzare le entità e relazioni in gioco e per individuare le proprietà da associare ad esse, data la natura ad alto livello di questo approccio è molto semplice passare da un idea, o uno sketch, a una definizione formale del modello dei dati.

1.3 Interfaccia utente

1.4 Impostazioni di configurazione

1.5 Manager

1.5.1 Difference

1.5.2 Stash

1.5.3 Neo4j

Bibliografia

[1] Christian Theil Have e Lars Juhl Jensen. «Are graph databases ready for bioinformatics?» In: *Bioinformatics* 29.24 (ott. 2013), pp. 3107–3108. DOI: 10.1093/bioinformatics/btt549. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3842757/.

[2] Stackoverflow. *Developer Survey*. Mag. 2022. URL: https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems.

[3] Marc Stevens. *Attacks on Hash Functions and Applications*. 19 Giu. 2012.

[4] Aron Toponce. *The Reality of SHA1*. 6 Mar. 2014. URL: https://pthree.org/2014/03/06/the-reality-of-sha1/.

[5] Git. *Git Documentation*. 1 Mag. 2021. Cap. The Git index file. URL: https://git-scm.com/docs/index-format.

[6] Git. en. 2ª ed. Berlin, Germany: APress, nov. 2014. Cap. 2.3. URL: https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History.

[7] *Neo4j*. URL: https://github.com/neo4j/neo4j.

[8] *DB-Engines Ranking of Graph DBMS*. 2022. URL: https://db-engines.com/en/ranking/graph+dbms.