Università degli Studi di Catania
Dipartimento di Matematica e Informatica
Corso di Basi di Dati

Gico

Santo Cariotti

2 Aprile 2021

Indice

1. Introduzione

Gico è un software che tramite un'API RESTful permette la gestione di commit Git a scopo statistico. Scritto in Rust[1] il backend si interfaccia ad un database relazionale chiamato PostgreSQL[2].

1.1 Database

Un database è una collezione di dati organizzati. Un DBMS è un software capace di gestire suddetta mole di dati interagendo con l'utente eseguendo i comandi più tipici dell'algebra relazionale, qui chiamate *query*.

1.1.1 SQL

In realtà il DBMS PostgreSQL fa riverimento al linguaggio SQL, il quale è suddiviso in diversi sottolinguaggi che, per l'appunto, suddividono il loro ruolo:

- **DDL**: il Data Description Language è quello responsabile della gestione delle tabelle e degli indici (`CREATE`, `ALTER`, `DROP`).
- **DCL**: il Data Control Language è quello responsabile della gestione delle autorizzazioni. Esso gestisce, in particolare, quello che un utente può o non può fare all'interno della base di dati (`GRANT`, `REVOKE`).
- **DML**: il Data Manipulation Language è quello responsabile della modifica dei dati (`INSERT`, `DELETE` e `UPDATE`).
- **QL**: il Query Language è quello che si occupa delle query, ovvero, *interrogazioni*, *proiezioni* e *giunzioni* (`SELECT`).

1.1.2 ACID

Il DBMS PostgreSQL permette il set di proprietà dette **ACID**:

- **Atomicity**: una transazione deve eseguire tutte le operazioni di cui è composta, oppure nessuna; non può permettersi di lasciare operazioni indietro o smarrite.

[1] https://www.rust-lang.com/
[2] https://www.postgresql.org/

- **Consistency**: una transazione deve mantenere i dati coerenti con le specifiche imposte (come ad esempio chiavi primarie o esterne).
- **Isolation**: le transazioni eseguite in concorrenza non corrompono il database; il loro funzionamento deve essere identico a delle transazioni eseguite in modo seriale.
- **Durability**: una volta completata una transazione, sarà per sempre valida, anche in caso di errore nel sistema.

1.1.3 Perché non MySQL?

Durante il corso di *Basi di dati*[3] abbiamo lavorato con il DBMS MySQL[4]. Nel progetto Gico, però, ho deciso di utilizzare PostgreSQL perché quest'ultimo ha una migliore gestione dei triggers e il supporto a una quantità maggiore di query (esempio `INTERSECT` o `EXCEPT`) e tipi di dati.

1.2 Backend

Per la scrittura del codice lato backend è stato impiegato il framework web Actix[5]. La sourcecode[6] è totalmente opensource, distribuita sotto licenza `GPL v3`.
I dettagli tecnici esulano il corso di Basi di Dati, quindi li ometterò e lascerò al README file presente nella sourcecode su GitHub.

1.3 Frontend

La sourcecode del frontend[7] è anch'esso opensource. Scritto con il framework Vue[8].
Anche questa parte esula il corso di Basi di Dati, quindi ometterò la descrizione in questo documento e rimando al README presente su GitHub.

[3] http://web.dmi.unict.it/corsi/l-31/insegnamenti/?cod=16709
[4] https://www.mysql.com/
[5] https://actix.rs/
[6] https://github.com/gico-net/server
[7] https://github.com/gico-net/client
[8] https://vuejs.org/

2. Analisi

Dalla creazione di Git sono stati creati svariati client web, tra cui il più famoso GitHub. Qualunque sviluppatore in erba - o una persona che si definisce tale - ha un account GitHub o passa il tempo a smanettare la sezione *Explore* per scoprire nuovi progetti. In questo caso io consiglio di guardare le repository *awesome-**stack*** per scoprire progetti, più o meno famosi, che usano quel determinato stack. Gli sviluppatori più smanettoni hanno un'istanza di un client opensource instanziata sulla propria macchina (che sia GitLab, Gitea, CGit o qualsiasi altro).
Ma rimaniamo in tema GitHub, dato che, anche se è proprietario, e questo turba alcune persone, resta il miglior client web Git per pubblicizzare il proprio progetto. In continua evoluzione, da qualche mese oramai, la lista dei *contributors* si trova in bella vista nella sidebar laterale. È una buona prassi, da parte di tutti, "stalkerare" i contributors per vedere se hanno altri progetti interessanti a cui hanno contribuito.

Qui nasce proprio Gico, un software dove poter caricare le repositories delle altre persone e salvare tutti i commit in un database centralizzato. In questo modo si possono filtrare i commit per autore e vedere a quale repositories ne fanno parte. Si "stalkera" quindi nel medesimo modo in cui si fa nei social network, ma in modo non ossessivo e per una buona causa: la curiosità di trovare altri progetti interessanti da contribuire/usare.
Per come è progettato, Git salva, all'interno dei commit, l'email e il nome dell'autore e del committer. Gico si occuperà di esaminare il log della repository salvando il commit e alcuni dei suoi dati utili a noi.

2.1 Glossario

Qui di seguito una lista delle parole chiavi impiegate all'interno del database.
Come si può notare non c'è il termine "Utente" proprio perché non esiste, all'interno di Git, una concezione tale.

Termine	Descrizione	Sinonimi	Termini collegati
Repository	Codice sorgente	Sourcecode, Progetto	
Branch	Flusso di modifiche	Diramazione	Repository
Commit	Modifica alla sourcecode	Modifica	Branch, Repository
Autore	L'autore delle modifiche	Programmatore	Commit
Committer	L'autore del commit	Code reviewer	Commit

3. Schema concettuale

Qui di seguto è riportato uno schema concettuale del database mediante modello Entità-Relazione.

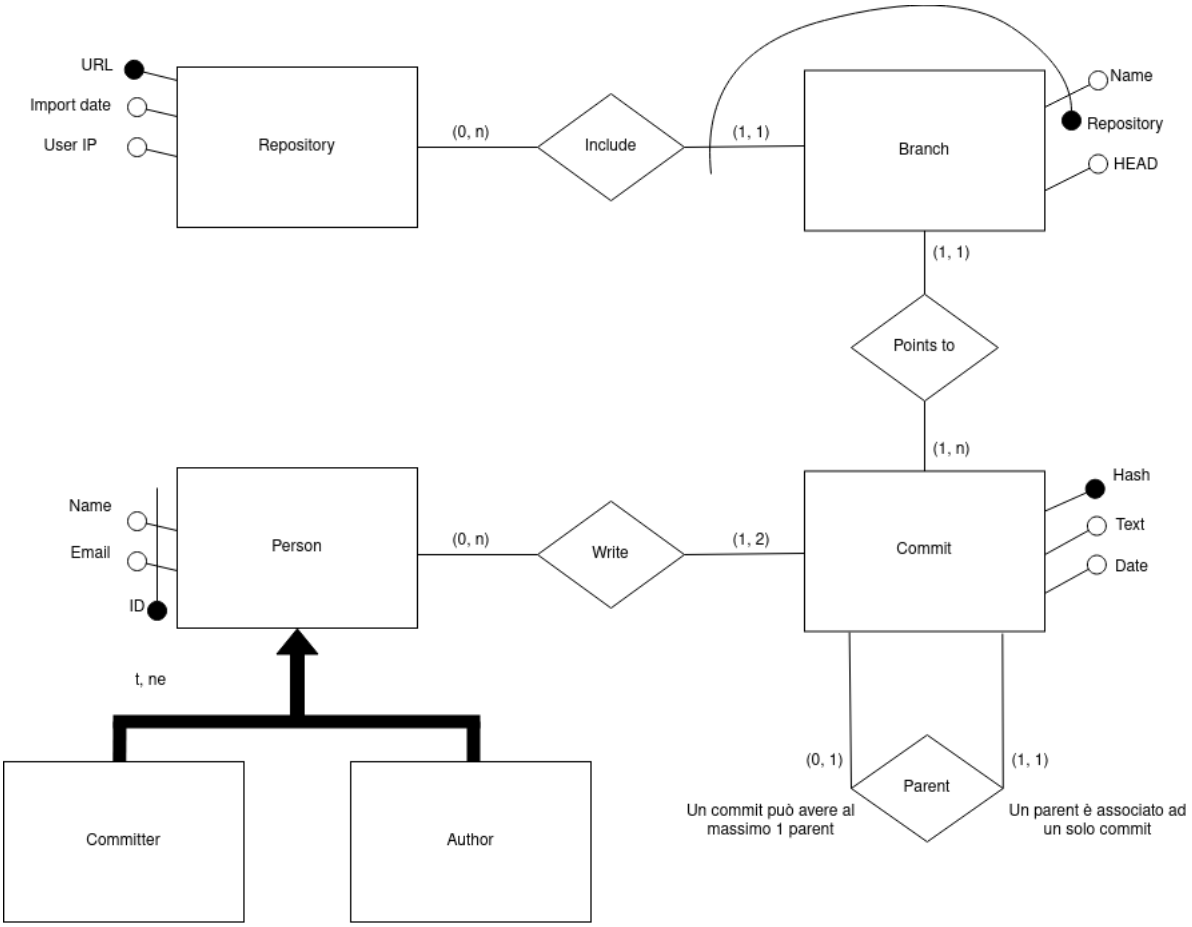

Figure 3.1: Modello E/R

4. Schema logico

Qui di seguito è riportato uno schema logico del database mediante modello Relazionale.

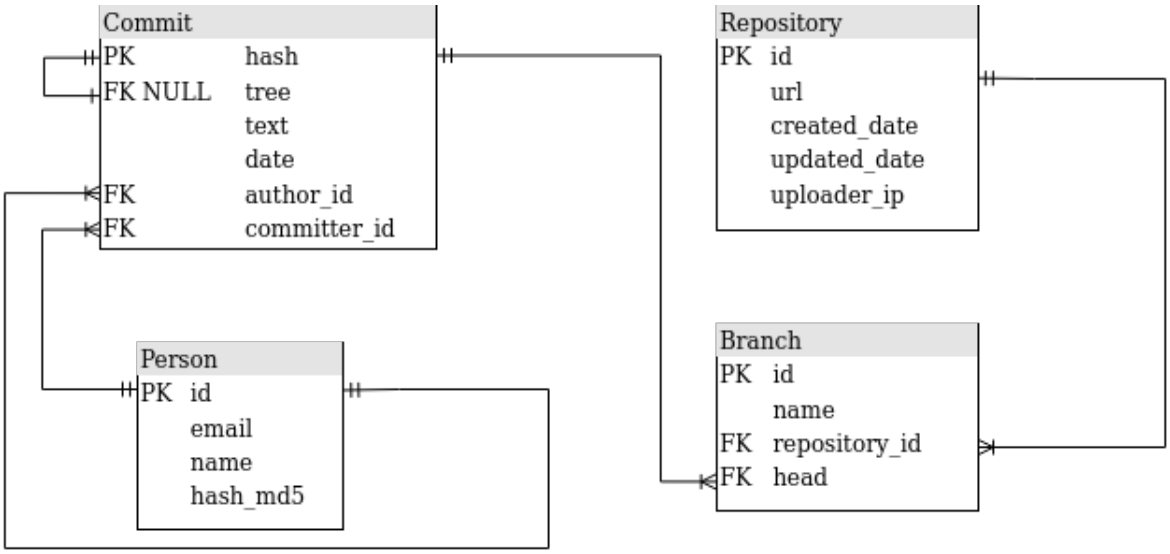

Figure 4.1: Modello relazionale

Come si può ben vedere sono state fatte delle modifiche:

- Repository: aggiunte le date di creazione e modifica;
- Branch: ha un suo id come PK, in modo da accedervi più facilmente tramite backend. Questo perché i branch permettono anche lo slash nel nome, cosa che andrebbe a creare errore se passato nell'endpoint;
- Commit: ha un campo `tree` che si riferisce al parent: può ovviamente essere null. Dato che il commit può essere scritto da massimo 2 persone, ho preferito aggiungere i campi `author_id` e `committer_id`;
- Person: ha un campo `hash_md5` che calcola l'MD5[1] dell'email una volta sola: usato per il Gravatar[2];

[1] https://en.wikipedia.org/wiki/MD5
[2] https://en.gravatar.com/

4.1 Considerazioni

Il numero di elementi presenti nella tabella `Commit` saranno molti: molti di più rispetto a quelli presenti in `Person`. Questo perché si presuppone che aggiunta una `Repository` verranno aggiunti i commit di un determinato `Branch` che, a meno di repositories particolari, non avranno un numero di contributors molto maggiore rispetto a quello dei commits.

Esempio Il progetto Rust[3] ha — mentre scrivo questo documento — **139,832** commits e **3,203** contributors.

Se seguiamo il progetto citato sopra e lanciamo una query d'esempio:

Query 1 Selezionare hash, testo, data, nome ed email di autore e committente.

Per eseguire la *Query 1* dovremmo realizzare 2 join a `Person` per ogni record. Se togliessimo le due foreign key avremmo un risparmio considerevole: non faccio il calcolo perché è facile da vedere.

Insert 1 Inserire un nuovo commit.

Per eseguire l'operazione *Insert 1* bisognerebbe cercare su `Person` se esiste già un elemento associato a quell'email e quel nome: se esiste lo legge, altrimenti lo crea. Cercare su `Person` se esiste già un elemento associato a quell'email e quel nome: se esiste lo legge, altrimenti lo crea. Infine scrivere all'interno di `Commit`. La foreign key `tree` è irrilevante perché fa riferimento ad un hash di un commit già esistente, quindi non c'è bisogno di controllo.

Se volessimo sapere a quale repository si riferisce un commit bisognerebbe cercare il `Branch` che ha un `head` che è uguale all'`hash` e in caso negativo procedere ricorsivamente per `tree`. Se siamo fortunati becchiamo il `Commit` che è `head` del `Branch` e ce ne usciamo con 1 giunzione, altrimenti il costo in SQL sarebbe quello di una chiamata ricorsiva tale che, partendo da un commit `c1` vada a cercare il commit appartenente al `tree` e così via. Ora, eseguire una query del genere 1 volta per apertura della pagina del commit[4] non è un problema; il problema è quando vogliamo vedere la repository associata ad ognuno dei commit. Certo, si potrebbe risolvere lato codice, ma questo è un progetto per il corso di Basi di Dati, quindi dobbiamo risolverlo tramite chiamata SQL.

4.2 Schema logico finale

In Figura 4.2 vi è una versione revisionata del modello Relazionale.

Possiamo notare svariate modifiche apportate:

- I `Commit` non puntano direttamente a 2 record di `Person`: le foreign key fanno riferimento solo alle emails. Questo perché Git considera diverse due persone con stessa email ma nome diverso. A noi interessa sapere l'hash MD5 per avere il Gravatar associato all'email. Se poi cambia nome, l'immagine dovrà essere comunque sempre la stessa. Quando non ci interessa avere l'avatar, inoltre, non dobbiamo fare giunzioni alla nuova tabella `Email`.

[3] https://github.com/rust-lang/rust

[4] Ad esempio con endpoint */api/commit/2cde51fbd0f310c8a2c5f977e665c0ac3945b46d/*

Se ci serve avere l'hash potremmo prima fare una `SELECT` delle email associate all'hash e utilizzare un hash per associare, lato client, l'hash alla singola email. Quindi non fare giunzioni. Ma questo lato client, quindi la tabella resta per il funzionamento tramite SQL.

- Al `Commit` è associata la url della `Repository`. Dato che la PK della `Repository` è l'`id`, il campo `url` è diventanto `UNIQUE`. Cosa che a noi sta più che bene, dato che non vogliamo `Repository` duplicate.
- Anche il campo `hash_md5` è `UNIQUE`. In realtà è una cosa extra, dato che, se dovessimo riscontrare un'errore del genere le alternative sono due: o in realtà l'email esiste già nel database oppure abbiamo trovato un'altra parola che ha come risultato dell'hash MD5, una già esistente.

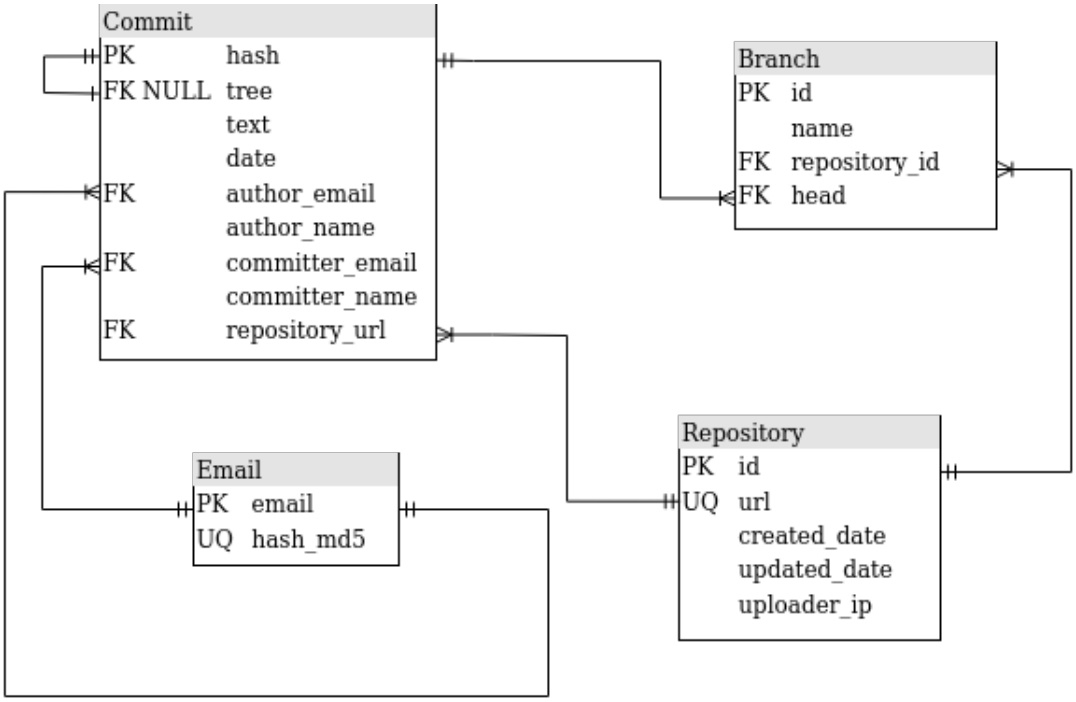

Figure 4.2: Modello relazionale finale

5. Schema fisico

Qui di seguito è riportato il `dump` in SQL per la creazione delle tabelle all'interno di PostgreSQL

```
1  CREATE TABLE "repository" (
2      id uuid PRIMARY KEY NOT NULL,
3      url varchar(255) UNIQUE NOT NULL,
4      created_at timestamp NOT NULL DEFAULT NOW(),
5      updated_at timestamp NOT NULL DEFAULT NOW(),
6      uploader_ip varchar(21) NOT NULL
7  );
8
9  CREATE TABLE "email"(
10     email varchar(120) PRIMARY KEY NOT NULL,
11     hash_md5 varchar(32) UNIQUE NOT NULL
12 );
13
14 CREATE TABLE "commit" (
15     hash varchar(40) PRIMARY KEY NOT NULL,
16     tree varchar(40) REFERENCES commit(hash) NULL,
17     text text NOT NULL,
18     date timestamptz NOT NULL,
19     author_email varchar(120) REFERENCES email(email) NOT NULL,
20     author_name varchar(120) NOT NULL,
21     committer_email varchar(120) REFERENCES email(email) NOT NULL,
22     committer_name varchar(120) NOT NULL,
23     repository_url varchar(256) REFERENCES repository(url) NOT NULL
24 );
25
26 CREATE TABLE "branch" (
27     id uuid PRIMARY KEY NOT NULL,
28     name varchar(120) NOT NULL,
29     repository_id uuid REFERENCES repository(id) NOT NULL,
30     head varchar(40) REFERENCES commit(hash) NULL
31 );
```

5.1 Trigger

Qui di seguito si viene implementato un trigger che aggiorna automaticamente la data di `updated_at` ogni qual volta si viene modificata una `Repository`.

```
1  CREATE OR REPLACE FUNCTION update_repository_last_updates() RETURNS TRIGGER AS $$
2  BEGIN
3      NEW."updated_at" = NOW();
```

```
4       RETURN NEW;
5   END;
6   $$ LANGUAGE PLPGSQL;
7
8   CREATE TRIGGER update_repository_t BEFORE UPDATE ON repository
9   FOR EACH ROW
10  EXECUTE PROCEDURE update_repository_last_updates();
```

5.2 Top authors

Lato client si sarebbero potuti esaminare i dati presi dai commit ma, per velocizzare il tutto lato client, ho preferito inserire una query lato server con un'endpoint per trovare la TOP 7 degli sviluppatori con più commit di cui sono autori. *Ho preferito non considerare i committers perché, se presi da GitHub, automaticamente il committer sarà sempre noreply@github.com*.

```
1   SELECT COUNT(hash) as num, author_email, author_name
2   FROM commit
3   GROUP BY author_email, author_name
4   ORDER BY COUNT(hash) DESC
```

6. Conclusioni

In conclusione posso dire che la webapp non è completa affatto. Come lei, qualsiasi altro software FOSS: esso può migliorare e mutare. Non posso dire con certezza che sarà mantenuto con costanza, ma ci proverò.

Una cosa che vorrei introdurre sono i *commenti*: essi potranno riferirsi alla repository o al singolo commit.

Ci sono svariate altre modifiche per l'app backend e frontend che si potrebbero e dovrebbero fare, ma preferisco non includerle in questo documento. Ho citato prima i commenti perché sono una funzionalità lato backend che si interlacciano al database.

Il trigger mostrato nella `Sezione 5.1` si riferisce al linguaggio SQL usato da PostgreSQL, lo si può reinterpretare senza la divisione con la procedura per dar luogo a un SQL più simile a quello visto durante il corso di Basi di Dati.

Le immagini sono state realizzate con un programma di terze parti[1]. Questo documento è stato realizzato in LaTeX mediante Gummi[2].

[1] https://app.diagrams.net/
[2] https://github.com/alexandervdm/gummi