Postgres: Primi passi tramite ORM

Come cercare di rendersi la vita semplice quando si parla di database

LINUX DAY · TORINO 2023

VOLCANIC MINDS

Di cosa parleremo

- Database, in particolare di PostgreSQL
- Cosa sono e a cosa servono gli ORM
- Come fargli fare amicizia
- Un paio di esempi
- Considerazioni finali
- Qualche link e contatti

Che cos'è un database

Un ***database*** rappresenta una raccolta sistematica di dati archiviati elettronicamente, accessibili tramite un sistema informatico.

Comunemente, il termine ***database*** viene utilizzato per riferirsi all'insieme organizzato di dati che al sistema di gestione dei dati stessi (DBMS).

Tipologie di database

I database ***relazionali*** sono organizzati per tabelle, ogni tabella ha delle righe (record) e ogni record può avere delle relazioni con altre tabelle. Si possono eseguire le JOIN. Si basano sul modello E-R.

I database ***non relazionali*** (aka NoSQL) sono organizzati in strutture ottimizzate per specifici scopi e presentano schemi flessibili. Non abbiamo righe e tabelle ma altre soluzioni come: coppie di chiavi/valori, documenti, ecc..

Database relazionali

Introdotti negli anni settanta, rimangono una scelta affidabile per numerose applicazioni.

Alcuni punti di forza includono: i dati strutturati, la capacità di definire delle chiavi primarie, esterne, naturali, surrogate per creare solide relazioni fra le tabelle.

Offrono la possibilità di sfruttare un linguaggio (SQL) per eseguire operazioni di manipolazione e ricerca, anche avanzata, delle informazioni contenute.

Esempi di RDB sono PostgreSQL, MariaDB, MySQL, ..

Database non relazionali

- I database **orientati ai documenti** rappresentano i dati tramite documenti (es JSON) con dei schemi dinamici, rendendo più immediata la gestione del dato (MongoDB, ..)
- I database **a grafo** hanno il loro punto di forza proprio nella definizione di archi e nodi. Non seguono un modello E-R ma, per l'attraversamento, sfruttano un meccanismo a puntatore (es: Neo4j, OrientDB, ..)

- Altri DB interessanti: key-value (RocksDB, Redis, ..), a colonne (Apache Cassandra, ..), a oggetti (ObjectDB, ..)

Un veloce confronto

Relazionali

- Strutture tabellari rigide, dati organizzati per righe, colonne e tramite relazioni
- Possono implementare sia dati primitivi sia dati complessi
- Hanno proprietà ACID: atomicità, coerenza, isolamento e durabilità
- Migliori prestazioni agendo su indici, join e modifiche alle strutture
- La scalabilità è più difficile (verticale)

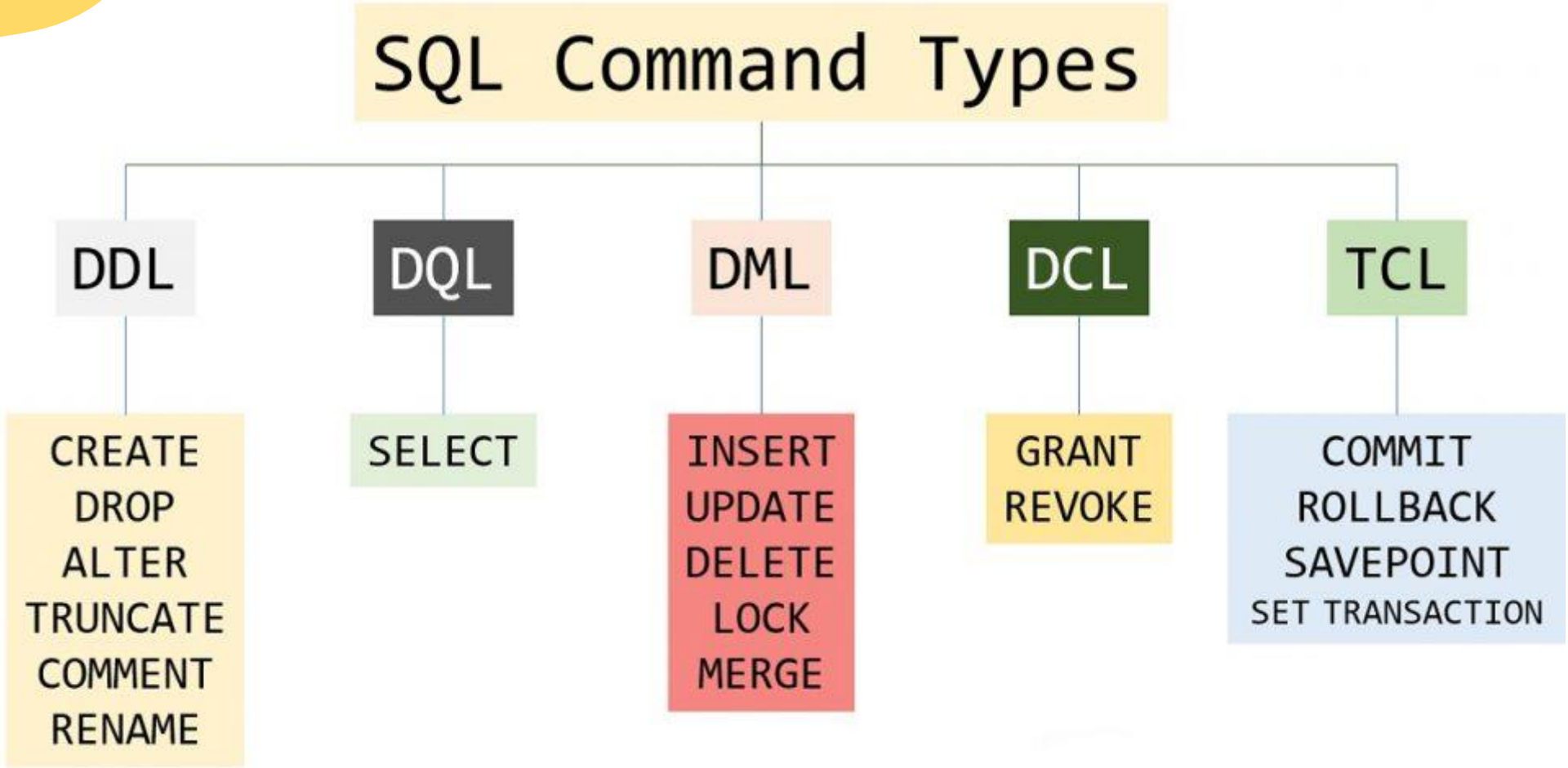
Non relazionali

- Strutture più fluide adatte agli scopi
- Non hanno le relazioni ma offrono alternative interessante
- Ci son diversi modelli quali chiave-valore, documenti, grafo
- Non sempre si han tutte le proprietà ACID (ma non è detto, es Mongo è ACID-compliant)
- Sono partizionabili e offrono una scalabilità orizzontale

Structured Query Language

Tramite SQL possiamo eseguire diverse operazioni, quali:

- Creare e modificare schemi di database (**DDL**, Data Definition Language)
- Inserire, modificare, eliminare dati memorizzati (**DML**, Data Manipulation Language)
- Interrogare i dati memorizzati (**DQL**, Data Query Language)
- Creare e gestire strumenti di controllo e accesso ai dati (DCL, Data Control Language)
- Utilizzare le transazioni efficacemente (**TCL**, Transaction Control Language)



VOLCANIC MINDS

La storia di PostgreSQL

Il progetto inizialmente si chiama INGRES (acronimo di INteractive Graphics REtrieval System). Da qui nascono sistemi come Sybase, SQL Server ed altri.

Postgres nasce per offrire anche la possibilità di definire nuovi tipi di dato e relazioni. Dopo alcuni anni viene abbandonato. Grazie al tipo di licenza (BSD), venne ripreso nel 94 e gli venne aggiunto l'interprete SQL.

Nel corso del tempo, ha cambiato diverse volte nome: INGRES, poi in Postgres, nuovamente in Postgres95 e infine in PostgreSQL.

Da allora PG ha aggiunto moltissime caratteristiche.
https://www.postgresql.org/about/featurematrix/

Le qualità di PostgreSQL

- Non è solo un RDBMS, ma è un O(R)DBMS
- Possibilità di sfruttare i JSON per i dati non strutturati
- Gestione dati geografici (tramite PostGIS)
- Meccanismo di LISTEN/NOTIFY (da non usare come un Message Broker - Pub/Sub System)
- Ottime performance soprattutto su grandi mole di dati
- Rilasciato sotto licenza OpenSource, simile alla BSD e MIT (PostgreSQL License)

Cosa sono gli ORM

Gli ORM (Object-Relational Mapping) offrono una interfaccia conveniente e veloce fra il mondo della programmazione orientata ad oggetti e il mondo dei database.

In pratica, hanno la capacità di convertire gli oggetti in righe/colonne e viceversa. Inoltre, offrono metodi utili per il recupero e la gestione del dato e delle strutture.

Le qualità degli ORM

- **Astrazione** del database: il mapping delle tabelle e dei campi e le relazioni fra essi diventa dichiarativo.
- **Portabilità** del codice: proprio per la capacità di essere "indipendenti" dal DB aumentano la capacità di riutilizzare il codice.
- **Velocità** di sviluppo: riduce il tempo necessario per la gestione e la manipolazione del dato.

Le qualità degli ORM

- **Semplicità** (iniziale) immediata: permettono di creare e mantenere le strutture, avviare progetti in tempi brevi senza sacrificare funzionalità.

- **Curva di apprendimento**: se si ha la fortuna di trovare un ORM che soddisfi le esigenze applicative e di database.

- **Accesso** allo strato base del DB: permettono spesso di eseguire comandi raw direttamente al database.

Quindi non si usa più l'SQL?

Non proprio, con gli ORM abbiamo comunque dei limiti e degli svantaggi, per esempio:

- Possono aggiungere complessità al codice
- Possono richiedere tempo per l'apprendimento, soprattutto in scenari complessi
- Soffrono di problemi di performance e di overhead, possono generare query non performanti
- Possono non supportare tutte le funzionalità (soprattutto quelle avanzate) rendendo necessario l'uso di query specifiche

Quando dovrei usarli?

Dipende.

Sono una **soluzione potente**, ma è essenziale considerare le specifiche esigenze del progetto, il proprio livello di competenza e l'uso futuro per determinare se rappresentano la scelta giusta.

Diventa **cruciale** l'analisi di tutti questi fattori per determinare se l'adozione di un ORM si trasformi in una scelta vincente.

Quali ORM per PostgreSQL

- Sequelize (JavaScript)
- TypeORM (JavaScript, TypeScript)
- DjangoORM (Python)
- SQLAlchemy (Python)
- Eloquent (PHP)
- Hibernate (Java)
- Prisma ORM(TypeScript)
- Knex.js (JavaScript)
- Entity Framework (C#)
- NHibernate (C#)
- Active Record (Rails)
- Doctrine (PHP)

Se ne avete altri da consigliare ..

Introduzione a TypeORM

TypeORM è un'ottima libreria che si integra velocemente con TypeScript e Javascript, ed è quindi adatto per progetti basati su Node.js.

Fra i vantaggi:

- Consente di gestire i dati in modo intuitivo
- Utilizza i decoratori per definire (e semplificare) le entità
- Permette la sincronizzazione delle strutture
- Permette di definire logiche personalizzate per la migrazione dei dati
- E' possibile definire diverse relazioni (one-to-one, one-to-many, many-to-many) rendendole anche lazy o eager.
- Si possono usare più sorgenti (DataSource) insieme
- Offre differenti modalità: Active Record, Entity Manager o tramite Repository
- Gode di una buona community e una popolarità al pari di Sequelize, Knex e Prisma

TypeORM: database supportati

VOLCANIC MINDS

TypeORM: active record

```ts
@Entity()
export class User extends BaseEntity {

    @PrimaryGeneratedColumn()
    id: number


    @Column()
    firstName: string


    @Column()
    lastName: string

    @Column()
    isActive: boolean
}
```

```ts
    // how to save AR entity
    const user = new User()
    user.firstName = "Timber"
user.lastName = "Saw"
user.isActive = true
await user.save()

// how to remove AR entity
await user.remove()

// how to load AR entities
const users = await User.find({ skip: 2, take: 5 })
const users = await User.find({ isActive: true })
const users = await User.find({ firstName: "Timber", lastName:
    "Saw" })
```

TypeORM: entity manager

```
@Entity()
export class User extends BaseEntity {

    @PrimaryGeneratedColumn()
    id: number


    @Column()
    firstName: string


    @Column()
    lastName: string
```

```
    import { DataSource } from "typeorm"
import { User } from "./entity/User"

const myDataSource = new DataSource(/*...*/)
const user = await myDataSource.manager.findOneBy(User, { id: 1 })

user.name = "Umed"
await myDataSource.manager.save(user)

    await myDataSource.manager.transaction(async (manager) => {
    // your stuff here
})
```

```
()
    isActive:
const       = await                .       .       (`SELECT * FROM USERS`)
```

TypeORM: data mapper

```
@Entity()
export class User extends BaseEntity {

    @PrimaryGeneratedColumn()
    id: number


    @Column()
    firstName: string


    @Column()
    lastName: string
```

```
    import { User } from "./entity/User"

const userRepository = dataSource.getRepository(User)
const user = await userRepository.findOneBy({
    id: 1,
})

user.name = "Umed"
await userRepository.save(user)
```

TypeORM: view entity

```
@ViewEntity({
    expression: (dataSource: DataSource) =>
        dataSource.createQueryBuilder().select("post.id", "id")
            .addSelect("post.name", "name").addSelect("category.name", "catName")
        .from(Post, "post").leftJoin(Category, "category", "category.id = post.categoryId"),
})
export class PostCategory {
    @ViewColumn()
    id: number

    @ViewColumn()
    name: string
```

```
const postCategories = await dataSource.manager.find(PostCategory)
const postCategory = await dataSource.manager.findOneBy(PostCategory, { id: 1 })
```

TypeORM: relations

```
@Entity()
export class User extends BaseEntity {

  @PrimaryGeneratedColumn()
  id: number

    @OneToOne(() => Profile)
  @JoinColumn()
  profile: Profile

  @OneToMany(() => Photo, (photo) => photo.user)
  photos: Photo[]
```

```
  // Eager relations are loaded automatically

@ManyToMany(
        (type) => Category,
        (category) => category.questions, { eager: true })
@JoinTable()
categories: Category[]


// Lazy relations are loaded once you access them

@ManyToMany((type) => Question, (question) =>
```

questions < []>

(() =>)
()

VOLCANIC MINDS

Esempio (TypeORM + Express + PG)

```
    import * as express from "express"
import { Request, Response } from "express"
import { User } from "./entity/User"
import { DataSource } from "typeorm"

const myDataSource = new DataSource({ /* connection */ }).initialize()

    const app = express()
app.use(express.json())

app.get("/users", async function (req: Request, res: Response) {
    const users = await myDataSource.getRepository(User).find()
    res.json(users)
}
```

Altro di interessante?

- Custom repositories
- Active record custom functions
- Transactions
- Multiple data sources, databases, schemas
- Subscriber and listener
- Migrations
- Hidden columns (es password)
- Caching queries (QueryBuilder e EntityManager)

Uno stack backend

Partendo dalle nostre esigenze, abbiamo iniziato a sviluppare un backend basato su Fastify e TypeORM. L'obiettivo principale è fornire delle funzionalità pratiche: routing, validazione dati, gestione entità, meccanismi di auth/auth, ruoli, middlewares, hooks, Swagger/OAS ...

Progetto di esempio
https://github.com/volcanicminds/volcanic-backend-sample

NPM (licenza MIT)
https://www.npmjs.com/package/@volcanicminds/backend
https://www.npmjs.com/package/@volcanicminds/typeorm

VOLCANIC MINDS

Grazie della pazienza!

Risorse

- https://typeorm.io
- https://www.postgresql.org
- https://www.postgresql.org/about/featurematrix
- https://fastify.dev
- https://it.wikipedia.org/wiki/PostgreSQL
- https://github.com/volcanicminds/volcanic-database-typeorm
- https://github.com/volcanicminds/volcanic-backend

Link e contatti, scriviamoci!

Scarica la presentazione da qui

Restiamo in contatto?

davide.morra@volcanicminds.com
https://www.linkedin.com/in/dmorra

Per conoscerci meglio:
https://volcanicminds.com

VOLCANIC MINDS