Text2SQL

Lorenzo Ventrone
ventrone.1802393@studenti.uniroma1.it

1 Obiettivo del progetto

Il progetto consiste nello sviluppo di un'applicazione *full-stack* che consente di interrogare un database di film utilizzando domande in linguaggio naturale. L'interazione è limitata a una serie di domande predefinite, e di conseguenza, anche le informazioni restituite sono circoscritte.

L'intero sistema è containerizzato tramite Docker, al fine di garantire portabilità e isolamento dell'ambiente di esecuzione.

2 Organizzazione del Database

A partire dai dati forniti, è stato progettato un diagramma Entità-Relazione (ER) in grado di rappresentare al meglio le entità e le relazioni esistenti.

Il modello ER progettato prevede la seguente struttura di entità e relazioni:

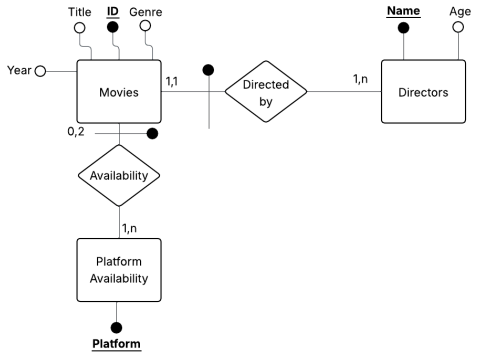

Figure 1: Diagramma ER del database.

Tabella	Attributo	Tipo / Chiave
Movies	id	INT, PK
	title	VARCHAR
	director	VARCHAR, FK
	year	INT
	genre	VARCHAR
Directors	name	VARCHAR, PK
	age	INT
Platform Availability	movie_id	INT, FK
	platform	VARCHAR, PK

Table 1: Struttura delle tabelle del database.

La tabella 1 è il risultato di ristrutturazione e traduzione del diagramma ER 1. Per quanto riguarda la tabella **Directors**, l'attributo `name` rappresenta la chiave primaria, identificando univocamente ogni regista.

La tabella **Movies** utilizza come chiave primaria un campo `id`, ovvero un intero autoincrementale che garantisce l'unicità di ogni film. L'attributo `director` è invece una chiave esterna che si collega alla tabella `Directors` derivante dall'accorpamento.

Infine, la tabella **Platform Availability** rappresenta l'associazione tra film e piattaforme su cui sono disponibili. È definita da una chiave composta: `movie_id` (foreign key riferita a `Movies`), attributo ereditato dall'accorpamento, e `platform`. Se un film non è disponibile su alcuna piattaforma, non sarà presente in questa tabella.

3 Organizzazione del codice

L'applicazione è organizzata in: database, descritto in precedenza, backend che garantisce la creazione e comunicazione con il database e inoltre contiene tutta la logica per la formulazione delle query a partire dal linguaggio naturale, ed infine il frontend che, attraverso un'interfaccia web, accetta richieste e gestisce le comunicazioni con il backend.

3.1 Backend

Il backend si occupa della gestione delle connessioni con il database e il frontend, utilizzando la libreria Python `mariadb` per le operazioni sul database, e `Uvicorn` insieme a `FastAPI` per l'esposizione degli endpoint. Per garantire chiarezza e modularità al progetto, sono state implementate due classi principali, `DatabaseManager` e `QueryHandler`, oltre al modulo `backend`.

3.1.1 Classe DatabaseManager

La classe `DatabaseManager` è deputata all'instaurazione delle connessioni e del successivo inserimento dei dati contenuti nel file `.tsv` qualora il database risulti vuoto, oppure dell'inserimento o aggiornamento dei record es-

istenti. In fase di aggiornamento, viene verificata la presenza della chiave primaria e, in caso di differenze rispetto ai dati esistenti, vengono modificati solo i campi necessari. Inoltre gestisce l'esecuzione delle query, ponendo attenzione anche alla sicurezza, grazie all'uso di segnaposti che limitano i tentativi di SQL-injection. Sono inoltre previsti metodi per la chiusura della connessione e per la pulizia completa del database.

3.1.2 Classe QueryHandler

La classe `QueryHandler` svolge la funzione principale del progetto, ovvero la conversione di domande in linguaggio naturale in query SQL eseguibili. Richiamando internamente la classe `DatabaseManager`, la `QueryHandler` utilizza un dizionario `query_mapping` che associa espressioni regolari alle relative query SQL. Attraverso i metodi `match_query` ed `execute_query`, è possibile riconoscere la domanda dell'utente, selezionare la query corrispondente ed eseguirla sul database. I risultati sono poi riformattati in JSON, come richiesto dalle specifiche.

3.1.3 Modulo backend

Il modulo `backend` integra le due classi principali e definisce gli endpoint di comunicazione con il frontend. In fase di avvio, viene instaurata la connessione al database e, se necessario, viene effettuato il popolamento iniziale dei dati. Sono stati implementati tre endpoint principali: `schema_summary`, che restituisce lo schema del database; `search`, che riceve una domanda in linguaggio naturale ed esegue la relativa interrogazione; e `add_data`, che consente l'inserimento di nuove righe nel database. La validazione dei dati in ingresso e in uscita è garantita attraverso l'uso di modelli `Pydantic`.

3.2 Frontend

Il frontend è responsabile dell'interfaccia utente e della comunicazione con il backend, utilizzando FastAPI per la gestione delle route e Jinja2 come motore di template per la generazione dinamica delle pagine HTML. Per garantire il corretto funzionamento sia in locale che in ambiente Docker, sono stati configurati percorsi dinamici per i template e l'indirizzo del backend viene gestito tramite variabili d'ambiente.

La pagina principale consente all'utente di eseguire tre operazioni: `search` invia una domanda in linguaggio naturale per interrogare il database, `add_data` aggiunge una nuova riga oppure aggiorna i dati esistenti, e `show_schema` permette la visualizzazione dello schema corrente del database. L'interfaccia utente è stata progettata per essere semplice ed intuitiva, con una chiara separazione tra i diversi tipi di operazioni. I risultati delle interrogazioni, i messaggi di successo o di errore, e la visualizzazione dello schema sono gestiti dinamicamente all'interno della stessa pagina, migliorando l'esperienza utente.

La comunicazione con il backend avviene attraverso richieste HTTP asincrone. I dati ricevuti vengono interpretati e resi disponibili nella pagina HTML, seguendo una logica che gestisce in modo distinto le risposte vuote, gli errori e i risultati validi. In particolare, in caso di ricerca senza risultati, viene mostrato un messaggio dedicato senza generare errori di sistema, migliorando così la chiarezza del feedback offerto all'utente.

3.3 Docker-Compose

La gestione e il coordinamento dei servizi sono stati realizzati tramite un file `docker-compose.yaml`, che consente l'orchestrazione dell'intero sistema.

Il servizio `db` utilizza l'immagine di MariaDB, esponendo la porta `3306` del container sulla `3307` dell'host, e inizializza automaticamente il database grazie a uno script SQL montato all'avvio.

Il servizio `backend` viene buildato a partire dal contesto `./backend`, si avvia solamente dopo che il database risulta disponibile e comunica con esso sfruttando variabili d'ambiente per la configurazione.

Infine, il servizio `frontend`, costruito a partire da `./frontend`, si avvia dopo il backend e gestisce la comunicazione con esso tramite la variabile d'ambiente `BACKEND_URL`.

Il servizio `db` dispone di un *healthcheck* per garantire la corretta disponibilità del database prima dell'avvio del `backend`.

4 Conclusione

Il progetto ha permesso di integrare efficacemente linguaggio naturale, database relazionali e tecnologie web, realizzando un sistema completo, modulare e facilmente distribuibile, grazie all'uso di Docker che garantisce la portabilità dell'applicazione. Documentazione completa disponibile nella repository GitHub.