Report Tematica Ricerca Semantica

Cristian Bassotto

1 Requisiti e Sfide

L'architettura del sistema è guidata da due requisiti fondamentali:

- Scalabilità nella ricerca: Un sistema avanzato di ricerca su larga scala deve sostenere un elevato throughput di query di ricerca con bassa latenza. La ricerca vettoriale è un'operazione computazionalmente costosa (approssimabile a $O(N)$ o $O(N \log k)$ per la ricerca esatta) e non deve degradare linearmente all'aumentare del dataset.
- Gestione degli aggiornamenti: Le operazioni di aggiunta, modifica ed eliminazione di tickets non devono bloccare la richiesta API. L'indicizzazione è un processo I/O e CPU-bound lento, che deve essere disaccoppiato dalla risposta al client.

2 Architettura e Flussi

Il sistema va quindi suddiviso in tre componenti: un **Database SQL** (Source of Truth) per i dati testuali (es. SQLite); un **servizio API** (es. Flask) per la gestione delle richieste HTTP e l'orchestrazione generale; e un **servizio di ricerca** che codifica i testi (e le query) attraverso un modello di embedding e gestisce la ricerca in un indice vettoriale (FAISS) aggiornato.

2.1 Flusso di Ricerca - Sincrono

Il flusso (Figura 1) è un'operazione sincrona ottimizzata per la latenza.

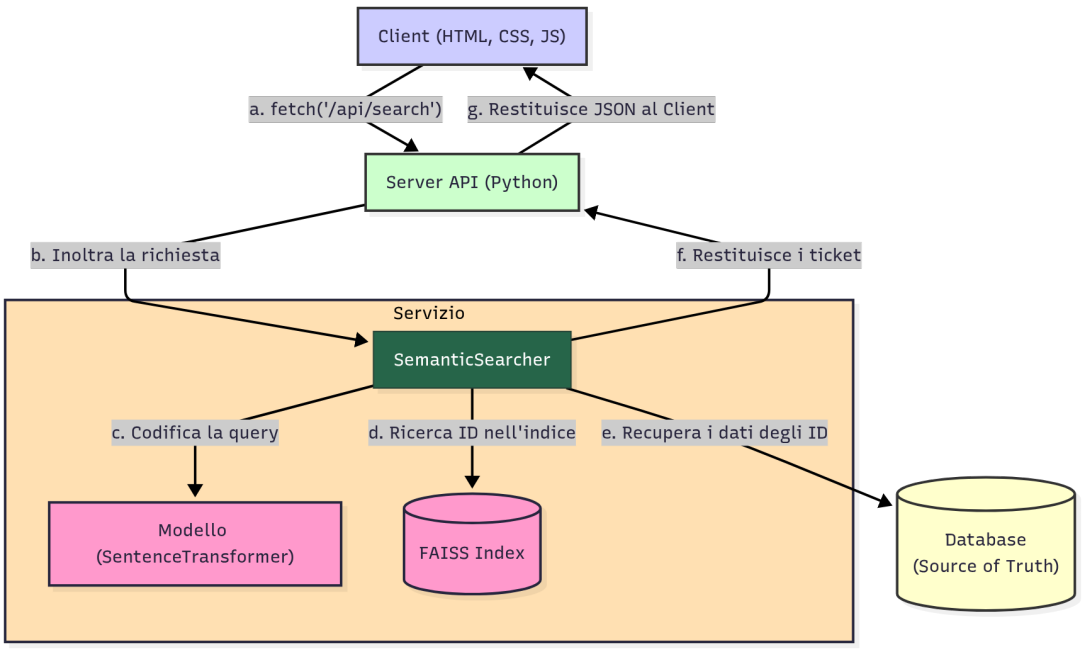

Figura 1: Diagramma di flusso per l'operazione di ricerca.

La richiesta API viene inoltrata al `Searcher`, che esegue l'encoding della query e la ricerca sull'indice FAISS. Questa operazione restituisce solo una lista con gli ID dei ticket. Il recupero dei dati testuali (l'arricchimento) avviene in un secondo momento, interrogando il database SQL tramite chiave primaria. La scalabilità è data dal fatto che l'operazione costosa (la ricerca) è gestita da un indice in-memory ad alte prestazioni (FAISS), mentre l'accesso al disco (DB) è un'efficiente query.

2.2 Flusso di Aggiornamento - Asincrono

Per garantire la reattività dell'API, le operazioni di scrittura (Figura 2) sono gestite in modo asincrono.

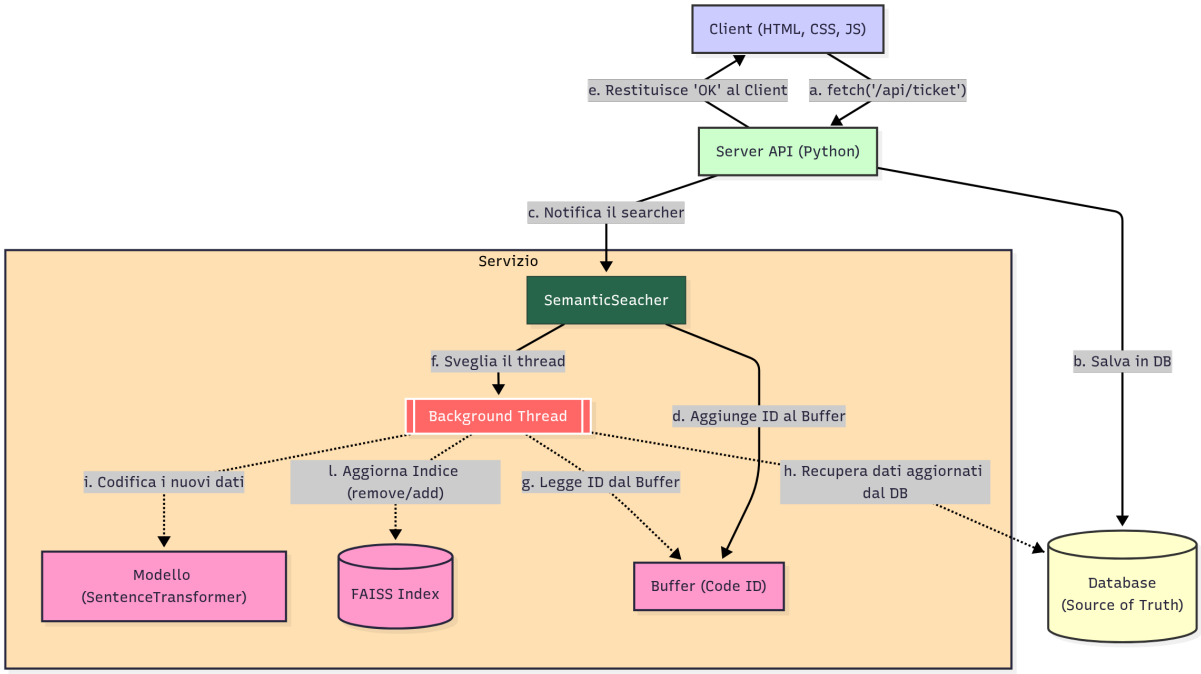

Figura 2: Diagramma di flusso per le operazioni di aggiornamento.

Il disaccoppiamento è fondamentale: la richiesta API esegue l'operazione di scrittura esclusivamente sul database e restituisce immediatamente al client. L'operazione lenta, la re-indicizzazione, viene delegata. L'API notifica il `Searcher` dell'ID del ticket modificato, che viene aggiunto a un buffer. Un **Thread** dedicato in background processa questo buffer in batch di lunghezza massima prefissata (per evitare anche un uso eccessivo di memoria), ricalcolando gli embedding per gli ID "sporchi" e aggiornando l'indice FAISS.

2.3 Scalabilità

Questo design garantisce un alto throughput di scrittura. Il sistema può ingerire un elevato numero di aggiornamenti poiché l'endpoint API è bloccato solo per la durata della transazione SQL, non per il costoso ricalcolo dell'embedding. Il compromesso di questo approccio è la consistenza. Esiste una latenza di indicizzazione (configurabile) tra la modifica sul DB e la sua visibilità nelle ricerche.

L'architettura attuale è **thread-safe** e scalabile verticalmente, ma non orizzontalmente. Lo stato (l'indice FAISS) è istanziato all'interno del processo API, impedendo la replica (es. con Gunicorn), che creerebbe istanze di indice separate e inconsistenti. Per abilitare la scalabilità orizzontale, il `Searcher` stateful deve essere estratto in un **microservizio dedicato**. Questo renderebbe l'API stateless e replicabile, ma richiederebbe anche la migrazione da SQLite a un

DB client-server (es. PostgreSQL) e l'uso di una message queue per le notifiche di aggiornamento asincrone.

3 Implementazione

L'intera architettura è costruita attorno alla scelta di un modello di embedding (Sentence Transformer). Questa decisione influenza direttamente ogni altro aspetto del sistema.

3.1 Preprocessing e Arricchimento

A differenza degli approcci lessicali (come TF-IDF), i modelli di embedding richiedono un preprocessing minimo. Il modello gestisce autonomamente l'ambiguità linguistica, i sinonimi e gli acronimi, poiché è addestrato a comprendere il significato semantico.

Pertanto, il preprocessing si limita alla pulizia dei dati:

- Rimozione di tag HTML e normalizzazione degli spazi bianchi (es. '\n', '\t') in un singolo spazio.
- Gestione dei valori nulli (es. 'NaN').

Non eseguiamo operazioni come la rimozione di stopwords, la lemmatizzazione o il case-folding (conversione in minuscolo). Mantenere punteggiatura e maiuscole è fondamentale, poiché forniscono al modello un contesto semantico più ricco per l'analisi della frase.

Per affrontare la criticità dei **ticket troppo brevi** (che mancano di contesto), si adotta una strategia di **arricchimento** del contesto. I diversi campi del ticket (titolo, categoria, contenuto) vengono concatenati in una singola stringa, utilizzando parole chiave per dare una struttura al testo che il modello possa interpretare:

Title: {short_description} | Category: {category} {subcategory} | Software: {software/system} | Content: {content}

Per i **ticket troppo lunghi** (che superano il limite di token del modello, es. 128 per il modello multilingua scelto o 512 per i modelli monolingua), l'input viene troncato. Si accetta questo troncamento assumendo che la parte finale del campo 'content' sia meno rilevante per il significato generale del ticket. Un'alternativa, più complessa, sarebbe la suddivisione del testo in *chunk*.

3.2 Modello Semantico, Indicizzazione e Similarità

La scelta del **modello semantico** è ricaduta su un modello di embedding (Sentence Transformer) per il suo bilanciamento ottimale tra prestazioni e velocità. Questi modelli mappano il testo in uno spazio vettoriale denso dove la distanza riflette la similarità semantica, risolvendo nativamente i problemi di **sinonimi, acronimi e ambiguità linguistica**, a differenza di TF-IDF che si basa sulla corrispondenza esatta delle parole. Per questo prototipo è stato scelto il modello `paraphrase-multilingual-mpnet-base-v2`, un modello multilingua robusto, sebbene un'alternativa più valida sia `multi-qa-mpnet-base-dot-v1`, un modello monolingua, ma più accurato e con limite di token in input maggiore. Sebbene un Large Language Model (LLM) possa offrire una comprensione contestuale superiore, il suo utilizzo per calcolare l'embedding di ogni singola query in tempo reale sarebbe proibitivo in termini di latenza e costi computazionali, rendendolo inadatto per un'applicazione di ricerca a bassa latenza.

Per **l'indicizzazione e la ricerca**, la scansione lineare di migliaia di vettori sarebbe troppo lenta. Come descritto in precedenza si utilizza FAISS (Facebook AI Similarity Search), una libreria ottimizzata per la ricerca di similarità ad alta velocità. L'indice FAISS memorizza i vettori e permette di trovare i K-Nearest Neighbors (K-NN) di un dato vettore-query in modo estremamente efficiente.

La **valutazione della similarità** tra la query e i ticket avviene tramite **Cosine Similarity**. Per ottimizzare ulteriormente la ricerca, i vettori (sia dei ticket che della query) vengono normalizzati durante l'encoding. In questo modo, la cosine similarity (che è un prodotto scalare normalizzato) diventa matematicamente equivalente a un semplice prodotto scalare, che FAISS può calcolare in modo ancora più rapido.

3.3 Gestione Aggiornamenti e Scalabilità su Grandi Volumi

Come descritto nella sezione sull'architettura, **aggiornamenti e nuovi ticket** sono gestiti in modo asincrono. Le operazioni di scrittura si traducono in un'aggiunta o rimozione dei vettori corrispondenti dall'indice FAISS.

Questo approccio è efficiente per la **scalabilità su grandi volumi**. Per il dataset attuale (27k ticket), l'indice FAISS esegue una ricerca esatta (K-NN). Se il dataset crescesse a milioni di ticket, si passerebbe a un indice Approximate K-NN (ANN) (es. 'IndexIVFPQ'). Questo tipo di indice è ordini di grandezza più veloce nella ricerca, a costo di una minima perdita di precisione.

Tuttavia, gli indici ANN soffrono di degrado delle prestazioni se si aggiungono e rimuovono vettori dinamicamente. La "best practice" in un sistema su larga scala sarebbe quindi quella di eseguire un ri-addestramento completo dell'indice FAISS in modo asincrono (es. ogni notte), per ricostruire gli indici sui dati aggiornati e garantire che la qualità della ricerca rimanga ottimale.

3.4 Metriche di Valutazione

Il dataset fornito non include "ground truth". Di conseguenza, non è possibile calcolare metriche offline tradizionali come precisione e recall.

La valutazione deve quindi basarsi sul **feedback utente**, che può essere raccolto in due modi:

1. **Feedback esplicito:** Aggiungendo un'interfaccia (es. un bottone "mi piace/non mi piace") su ogni ticket restituito.
2. **Feedback implicito:** Analizzando il comportamento dell'utente (es. *click-through rate*), ovvero quali ticket l'utente sceglie di aprire data una specifica query.

Questi dati di feedback possono essere utilizzati per il **fine-tuning** del modello di embedding tramite tecniche come il RHLF, specializzando ulteriormente il modello sul dominio specifico dei ticket IT.

3.5 Deployment e Testing

Per garantire la portabilità, l'applicazione è stata containerizzata utilizzando **Docker**. Questo permette di astrarre le dipendenze e di eseguire l'intero stack applicativo in qualsiasi ambiente in modo coerente. È stata inoltre implementata una **suite di test** di base per validare gli endpoint del server API, le operazioni sul database e la logica di indicizzazione. Sebbene questi test verifichino il corretto funzionamento dei flussi principali, la copertura è ancora parziale e dovrebbe essere notevolmente ampliata in un contesto di produzione per includere casi limite, test di carico e una validazione più robusta dei risultati di ricerca.