ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in Ingegneria e Scienze Informatiche

Generazione automatica di query su sistemi multistore tramite Large Language Models

Relatore:
Prof. Matteo Golfarelli

Candidato:
Mattia Rocchi

Correlatore:
Dott. Manuele Pasini

Sessione Luglio 2026
Anno Accademico 2025/2026

Indice

1 Introduzione

1.1 Scenario applicativo e motivazioni

L'agricoltura di precisione è un paradigma gestionale che integra tecnologie dell'informazione, reti di sensori e sistemi di geolocalizzazione per monitorare, analizzare e gestire la variabilità spaziale e temporale delle colture e dei suoli, ottimizzando l'impiego di input produttivi mediante un processo decisionale basato su dati ad alta risoluzione. Negli ultimi anni, questo settore ha subito un'ulteriore accelerazione grazie alla diffusione di dispositivi IoT [1] e piattaforme di monitoraggio in tempo reale, che consentono di raccogliere volumi crescenti di dati eterogenei sull'ambiente agricolo. Tuttavia, la stessa eterogeneità che rende questi dati preziosi ne complica la gestione: informazioni strutturali e relazionali (appezzamenti, dispositivi) coesistono con serie temporali di misurazioni continue (temperatura, umidità, ecc.), che richiedono paradigmi di interrogazione e storage profondamente diversi.

Per affrontare questa eterogeneità, una parte dei sistemi reali adotta architetture *polyglot persistence* [7], in cui più database specializzati operano in parallelo, collegati tramite logica applicativa esterna. Un'alternativa strutturalmente più integrata è rappresentata dai *sistemi multistore* [2], nei quali, modelli di dati eterogenei coesistono all'interno di una singola infrastruttura che espone un'interfaccia di interrogazione unificata. Il presente lavoro si basa su un sistema di quest'ultima categoria: un'istanza PostgreSQL estesa con Apache AGE per la gestione del grafo tramite il linguaggio Cypher e con TimescaleDB per le serie temporali. Tale sistema consente di interrogare congiuntamente dati strutturali e misurazioni IoT attraverso *query ibride* combinando sintassi Cypher e SQL in un unica espressione.

Proprio la necessità di formulare query ibride costituisce la principale barriera all'accesso per gli utenti operativi del settore agricolo, chiamati a padroneggiare due domini linguistici distinti: Cypher per la navigazione delle relazioni del grafo e SQL per l'aggregazione delle misurazioni temporali. A ciò si aggiungono le peculiarità dell'interprete ibrido di Apache AGE, come la gestione esplicita dei tipi (i risultati di `cypher()` sono di tipo `agtype` e vanno convertiti) e le regole sugli alias nelle sottoquery [9]. Questa complessità tecnica preclude di fatto l'accesso diretto ai dati a utenti non specializzati.

1.2 Obiettivi della tesi

Questa tesi esplora il potenziale dei Large Language Models (LLM) come interfaccia in linguaggio naturale per l'interrogazione di sistemi multistore basati su PostgreSQL. In particolare, il lavoro analizza la capacità dei modelli di tradurre richieste espresse in linguaggio naturale in query ibride Cypher+SQL, garantendone la correttezza sintattica e la coerenza semantica rispetto alle sorgenti dati eterogenee. A tal fine, viene progettato e sviluppato un framework automatizzato end-to-end che supporta l'intero processo di interrogazione: dalla costruzione e ottimizzazione dei prompt, alla generazione delle query, fino alla loro esecuzione e valutazione. Il framework è concepito per gestire in modo integrato la complessità derivante dall'interazione tra diversi modelli di dati e linguaggi di query. La validazione sperimentale è condotta su un benchmark di quesiti in ambito agricolo, rappresentativo di scenari applicativi concreti. L'analisi su vari modelli LLM mira ad individuare pattern ricorrenti di errore e valutare l'efficacia di tecniche di prompting. I risultati evidenziano opportunità e limiti dell'approccio proposto, contribuendo a ridurre il divario tra la complessità tecnica dei sistemi multistore e le esigenze degli utenti finali, e aprendo nuove prospettive per l'interazione intelligente con dati eterogenei.

2 Gestione dei dati

La gestione di dati eterogenei rappresenta una sfida centrale nei sistemi informativi moderni, particolarmente rilevante nei contesti applicativi caratterizzati dalla convergenza di informazioni strutturali, relazionali e temporali. Nel dominio dell'agricoltura di precisione, questa eterogeneità si manifesta in modo esemplare: da un lato vi sono entità relativamente stabili come appezzamenti, dispositivi e relazioni spaziali; dall'altro vi sono flussi continui di misurazioni prodotte da reti di sensori IoT, caratterizzati da volumi elevati e pattern di interrogazione distinti. Per comprendere il contesto del sistema su cui si basa la presente tesi, è necessario analizzare i principali modelli di dati che ne costituiscono il fondamento teorico e le architetture che ne consentono l'integrazione.

2.1 Il modello relazionale

Il paradigma dominante per la gestione di dati strutturati è rappresentato dal modello relazionale, il quale offre un formalismo matematico rigoroso per la rappresentazione e la manipolazione dell'informazione.

2.1.1 Principi fondamentali, chiavi e integrità referenziale

Il modello relazionale si fonda su vari principi fondamentali che ne definiscono la struttura e il comportamento[3] tra i quali:

I. i valori atomici in ogni posizione della relazione che eliminano la complessità della gestione di campi multivalore.; II. I principi di forma normale che prevengono le rindondanze a livello strutturale e di relazioni.; III. le relazioni derivabili da altre relazioni che garantiscono completezza espressiva.; IV. la chiusura algebrica sotto operazioni relazionali che assicura che ogni risultato di una query sia anch'esso una relazione valida.; V. l'indipendenza logica dei dati che separa la struttura concettuale da quella fisica.

Ogni relazione è un insieme di tuple omogenee che rappresentano fatti semanticamente significativi dal punto di vista del dominio applicativo, con ogni attributo che assume valori atomici e ogni tupla che descrive un'istanza completa del concetto modellato [3].

Determinati attributi della tupla fungono da identificatori univoci dell'entità o delle sue relazioni e sono detti chiavi. Il sistema delle chiavi rappresenta il meccanismo fondamentale per codificare la semantica del dominio all'interno della struttura tabulare. Le chiavi primarie identificano univocamente le tuple all'interno di una relazione, mentre le chiavi esterne stabiliscono relazioni logiche tra tabelle distinte. Il vincolo di integrità referenziale impone che ogni valore di chiave esterna presente in una relazione "figlia" corrisponda esattamente a un valore di chiave primaria nella relazione "padre", preservando così la consistenza semantica del database attraverso tutte le operazioni di modifica [3]. Questo meccanismo consente di modellare relazioni complesse uno a molti e molti a molti attraverso una normalizzazione rigorosa che elimina ridondanze e anomalie di aggiornamento.

2.1.2 Operazioni relazionali e limiti su dati fortemente connessi

L'algebra relazionale fornisce le operazioni primitive che costituiscono la base teorica di SQL: selezione (σ), che filtra tuple secondo un predicato; proiezione (π), che estrae attributi specifici; join naturale ($\bowtie$), che combina tuple di due relazioni sulla base di attributi comuni; unione ($\cup$), differenza ($-$) e prodotto cartesiano ($\times$) [3]. SQL estende questo formalismo con operatori di raggruppamento `(GROUP BY)`, funzioni di aggregazione `(SUM, COUNT, AVG)` e ordinamento `(ORDER BY)`, supportati da un sistema di ottimizzazione basato su costi. L'ottimizzatore è il componente del DBMS responsabile di selezionare, tra i possibili piani di esecuzione equivalenti di una query, quello con il costo stimato minore in termini di Input/Output e CPU. Per farlo, sfrutta statistiche sui dati, indici e regole di riordinamento delle operazioni algebriche per minimizzare il costo complessivo dell'esecuzione.

Nonostante la sua solidità, il modello relazionale mostra limiti quando si devono esprimere interrogazioni che attraversano relazioni connesse in modo ricorsivo o multi-hop. Nei database relazionali, la necessità di eseguire join multipli per ricostruire percorsi in un grafo rende le query complesse e spesso inefficienti, soprattutto quando la profondità delle relazioni non è nota a priori [4]. Inoltre, la rigidità dello schema-on-write del modello relazionale, approccio in cui la struttura dei dati (tabelle, colonne, tipi) deve essere definita prima di qualsiasi inserimento, non si adatta a domini dove la struttura dei dati evolve frequentemente o dove le relazioni stesse costituiscono l'oggetto primario dell'interesse.

2.2 Il modello a grafo

Quando le relazioni tra entità costituiscono l'aspetto dominante dell'informazione, il modello a grafo offre un paradigma espressivamente più naturale. Estensioni come Apache AGE, utilizzata nel sistema sperimentale della presente tesi, implementano questo modello all'interno di PostgreSQL.

2.2.1 Proprietà fondamentali del property graph

Il modello a grafo è definito come una struttura composta da nodi (entità) e archi (relazioni), dove entrambi possono avere proprietà (coppie chiave-valore) ed etichette che ne definiscono il tipo [4]. A differenza del modello relazionale, dove le connessioni sono realizzate tramite chiavi esterne e join, nel grafo le relazioni sono cittadini di prima classe: ogni arco ha un tipo, una direzione e può trasportare informazioni proprie. Questa caratteristica rende le traversate di grafo molto più dirette ed efficienti, soprattutto per percorsi di profondità arbitraria.

I nodi rappresentano entità del dominio, mentre gli archi codificano relazioni semanticamente significative. Le etichette multiple consentono ruoli polimorfi per la stessa entità, mentre le proprietà chiave-valore offrono flessibilità descrittiva.

2.2.2 Linguaggi di interrogazione e valutazione del modello a grafo

Cypher è il linguaggio dichiarativo più diffuso per i database a grafo, basato su pattern matching: un paradigma in cui l'interrogazione non specifica come navigare il grafo, ma descrive la forma topologica dei sottografi di interesse (nodi, archi, etichette, proprietà), lasciando al motore il compito di individuare tutte le occorrenze corrispondenti [5]. La sintassi fondamentale utilizza `MATCH` per descrivere pattern di nodi e archi, `WHERE` per filtrare, `RETURN` per proiettare i risultati.

Un pattern tipo: `(n:Appezzamento)-[r:CONTIENE]->(d:Dispositivo)` identifica tutti gli archi di tipo CONTIENE tra nodi etichettati Appezzamento e Dispositivo. Cypher supporta traversate a lunghezza variabile, come `(a)-[*1..3]-(d)`, per esplorare percorsi di profondità 1-3 [5] rendendo questo tipo di interrogazione molto più conciso ed efficiente rispetto a sequenze di join in SQL [4].

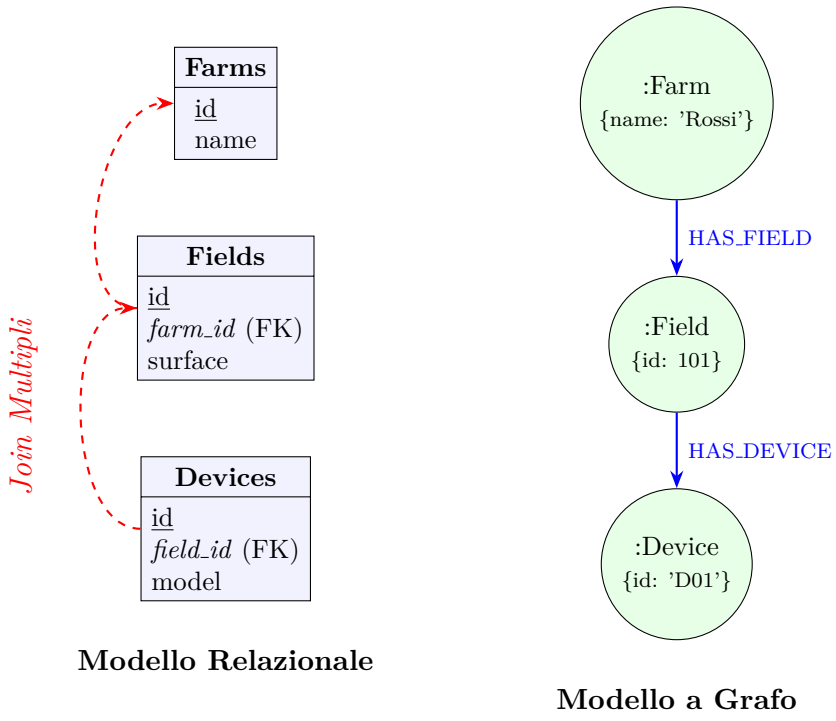

Figura 1: Confronto tra la navigazione tabulare tramite chiavi esterne e la navigazione nativa su grafo

Il modello a grafo eccelle quando l'analisi è incentrata sulle connessioni, ma non è nativamente progettato per serie temporali ad alta frequenza o aggregazioni statistiche su grandi insiemi tabulari, dove i database relazionali (con indici clusterizzati e partizionamento) mantengono una superiorità prestazionale.

2.3 Limiti dei sistemi separati e soluzione proposta

L'adozione di database specializzati rappresenta la risposta tradizionale all'eterogeneità informativa. Tuttavia, come osservato precedentemente, queste architetture introducono complessità significative, in particolare quando l'integrazione tra i diversi sistemi non è gestita in modo trasparente.

2.3.1 Polyglot persistence: specializzazione e integrazione applicativa

La cosiddetta *polyglot persistence* è una strategia architetturale che, invece di affidarsi a un singolo database management system, prevede l'utilizzo di tanti database quanti necessari per soddisfare tutti i requisiti di una moderna infrastruttura di gestione dati [6]. Questa soluzione risulta particolarmente indicata quando si deve garantire la retrocompatibilità con un'applicazione legacy: il nuovo database può funzionare insieme a quello preesistente, permettendo di soddisfare nuovi requisiti senza compromettere le funzionalità esistenti.

L'integrazione tra questi motori eterogenei viene generalmente demandata a un apposito *integration layer*. Questo strato software si occupa di decomporre le query in sottocomponenti, reindirizzarle al database appropriato e ricombinare i risultati ottenuti. Idealmente, dovrebbe anche offrire molteplici metodi di accesso, essere in grado di gestire i diversi linguaggi di interrogazione e tradurli quando necessario, oltre a garantire la coerenza tra i database sincronizzando le operazioni di aggiunta, modifica o cancellazione [6].

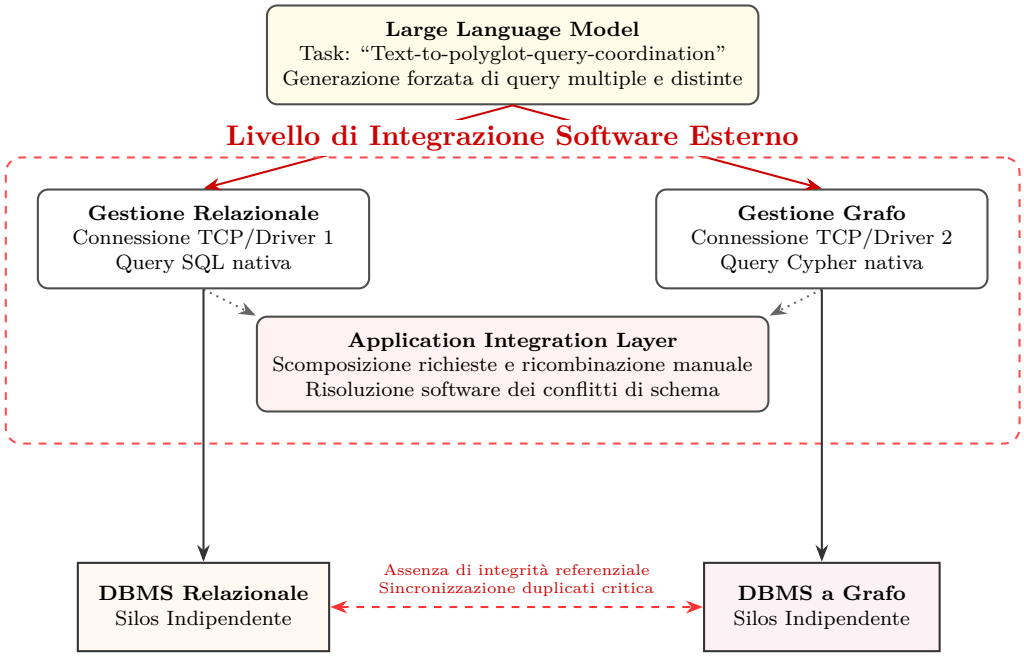

Figura 2: Rappresentazione della polyglot persistence: L'LLM deve produrre output multipli per un integration layer applicativo esterno che si fa carico di traduzione, multi-connessione e merge manuale dei dati, operando su DBMS isolati privi di consistenza nativa condivisa.

2.3.2 Sfide e implicazioni per l'interrogazione automatica

Nonostante sia flessibile e consolidata, la polyglot persistence presenta severe limitazioni [6]:

- **accesso non uniforme**: non esiste un'interfaccia o un linguaggio di query unico. L'accesso ai diversi sistemi non è unificato e richiede la conoscenza di tutti i metodi necessari.
- **consistenza**: garantire la consistenza cross-database è una sfida importante, poiché l'integrità referenziale deve essere assicurata attraverso i diversi sistemi. Nel caso in cui i dati siano duplicati (e quindi presenti in rappresentazioni diverse in più database contemporaneamente), i duplicati devono essere aggiornati o cancellati all'unisono.
- **interoperabilità**: i database sottostanti sono sviluppati indipendentemente l'uno dall'altro. Nuove versioni potrebbero non essere interope-

rabili con il livello di integrazione, e l'amministratore deve monitorare frequenti aggiornamenti.

- **ridondanza logica**: può essere evitata solo con un progetto che assegna sottoinsiemi di dati disgiunti ai diversi database, il che potrebbe entrare in conflitto con alcuni requisiti di accesso degli utenti.

- **sicurezza**: il controllo degli accessi deve essere applicato sia dal livello di integrazione sia da tutti i database connessi, che devono essere configurati per consentire solo accessi ristretti.

Per approcci moderni basati su Large Language Models, questa complessità si traduce in difficoltà moltiplicativa. Oltre alla corretta generazione di query per ciascun motore, sarebbe necessario orchestrare l'esecuzione multi-sistema, gestire dialetti diversi, combinare risultati eterogenei e risolvere conflitti di schema. L'estensione del text-to-SQL tradizionale diventerebbe "text-to-polyglot-query-coordination", un problema di complessità notevolmente superiore rispetto alla gestione di una singola interfaccia unificata.

2.4 Il modello multistore

Le architetture multistore affrontano la frammentazione infrastrutturale, offrendo un'interfaccia unificata che astrae la molteplicità dei motori sottostanti.

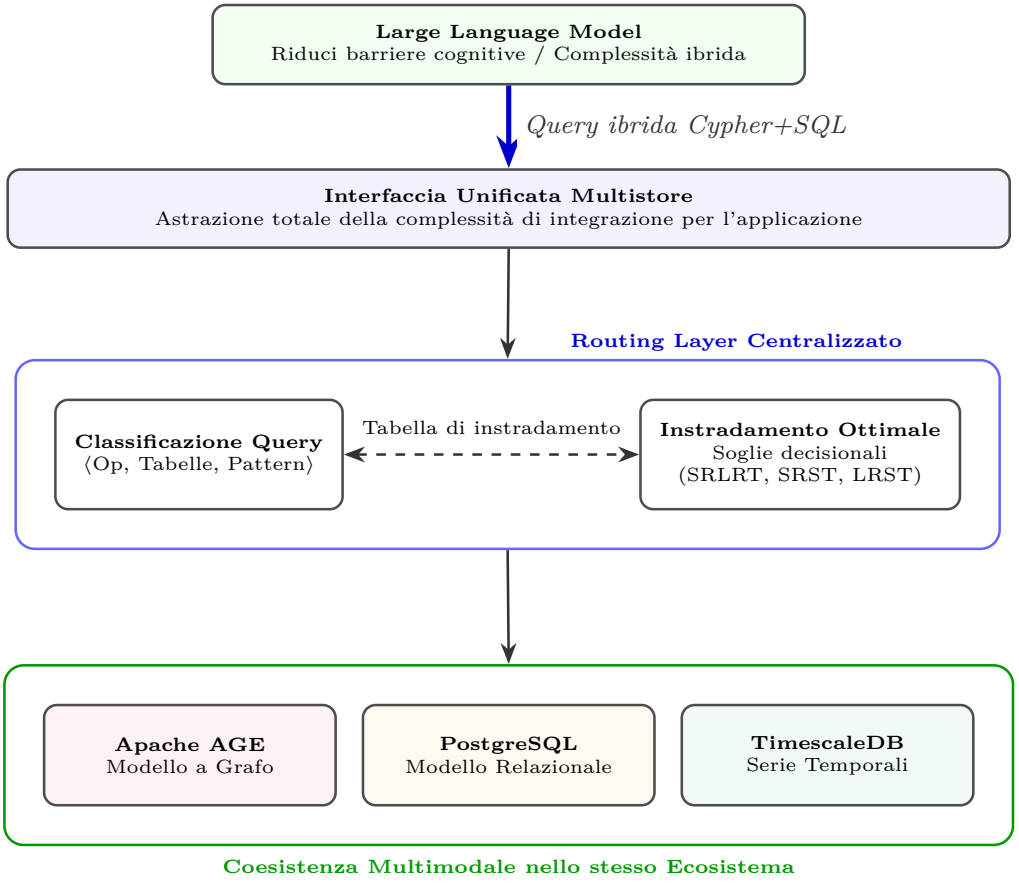

Figura 3: Architettura concettuale del multistore l'interfaccia unificata riceve la query ibrida generata dall'LLM, astraendone la complessità, mentre il routing layer centralizzato analizza il pattern di accesso per coordinare l'esecuzione sui motori eterogenei coesistenti.

2.4.1 Integrazione multimodale nei sistemi multistore

Un multistore è un sistema che integra più DBMS eterogenei attraverso un *routing layer* centralizzato, classificando le interrogazioni in base al pattern di accesso e instradandole verso il motore ottimale per quella specifica classe di query [2]. A differenza della polyglot persistence, dove l'applicazione deve gestire esplicitamente i diversi motori, il multistore astrae la complessità di integrazione, offrendo all'utente un'interfaccia unificata.

Il sistema Icarus, ad esempio, classifica le query in *query class* definite dalla tripletta: operazione, tabelle coinvolte, pattern di accesso. Per ogni classe esso costruisce una tabella di instradamento probabilistica che assegna percentuali di esecuzione ai vari motori, bilanciando throughput e latenza. Soglie come la Short-Running-to-Long-Running Threshold (SRLRT) distinguono query Online Transaction Processing (OLTP) da Online Analytical Processing (OLAP), mentre Short/Long Running Similarity Thresholds (SRST/LRST) determinano quali motori sono "sufficientemente simili" al migliore per quella classe [2].

L'architettura multistore fornisce il fondamento concettuale per il sistema sperimentale di questa tesi. L'integrazione multimodale di PostgreSQL con le estensioni Apache AGE (per il modello a grafo) e TimescaleDB (per le serie temporali) realizza una forma di coesistenza multimodale all'interno dello stesso ecosistema, consentendo l'espressione di query ibride Cypher+SQL che attraversano simultaneamente domini informativi distinti. È proprio questa capacità di interrogazione integrata che rende il problema dell'automazione tramite Large Language Models non solo tecnicamente realizzabile, ma strategicamente rilevante per superare le difficoltà nell'apprendere la complessa sintassi delle query, come verrà mostrato nella presente tesi.

3 Il sistema di riferimento

Il sistema multistore su cui si basa la sperimentazione di questa tesi è un'istanza PostgreSQL estesa con Apache AGE per la gestione del grafo e TimescaleDB per le serie temporali. Il presente capitolo descrive in dettaglio ciascuna di queste tecnologie e l'ambiente di sviluppo utilizzato.

3.1 PostgreSQL come base estensibile

PostgreSQL è un sistema di gestione di database relazionale open-source. Il suo principale punto di forza risiede in un'architettura facilmente estensibile, progettata per consentire l'aggiunta dinamica di nuovi tipi di dato, operatori e funzioni personalizzate, senza dover mai modificare o ricompilare il codice sorgente principale.

3.1.1 Architettura a estensioni.

L'architettura intrisicamente estensibile di PostgreSQL [8] è oggi realizzata attraverso il meccanismo delle estensioni(`CREATE EXTENSION`), le quali permettono di caricare dinamicamente codice compilato che si integra perfettamente con il pianificatore di query e il gestore dello storage.

Grazie a questa architettura, PostgreSQL può fungere da piattaforma unificante per diversi modelli di dati. Il pianificatore basato su costi, ovvero il componente responsabile di selezionare il piano con il costo stimato minore in termini di Input/Output e CPU, è in grado di ottimizzare query ibride che attraversano questi diversi modelli, producendo un unico piano di esecuzione. Per il problema della generazione automatica di query, ciò si traduce in un vantaggio fondamentale: l' LLM deve produrre una singola stringa sintatticamente valida per l'interprete ibrido di AGE, senza dover gestire connessioni o protocolli distinti.

3.2 Apache AGE: il modulo grafo

Apache AGE (*A Graph Extension*) è un'estensione di PostgreSQL che implementa il modello *Property Graph* direttamente sopra le strutture di storage relazionali esistenti [9]. Questo paradigma modella la realtà attraverso nodi e relazioni, consentendo a entrambi di possedere proprietà descrittive espresse sotto forma di coppie chiave-valore. Il modulo consente di eseguire query in linguaggio Cypher all'interno di query SQL standard, trattando il grafo come un oggetto di prima classe perfettamente integrato nell'ecosistema di PostgreSQL.

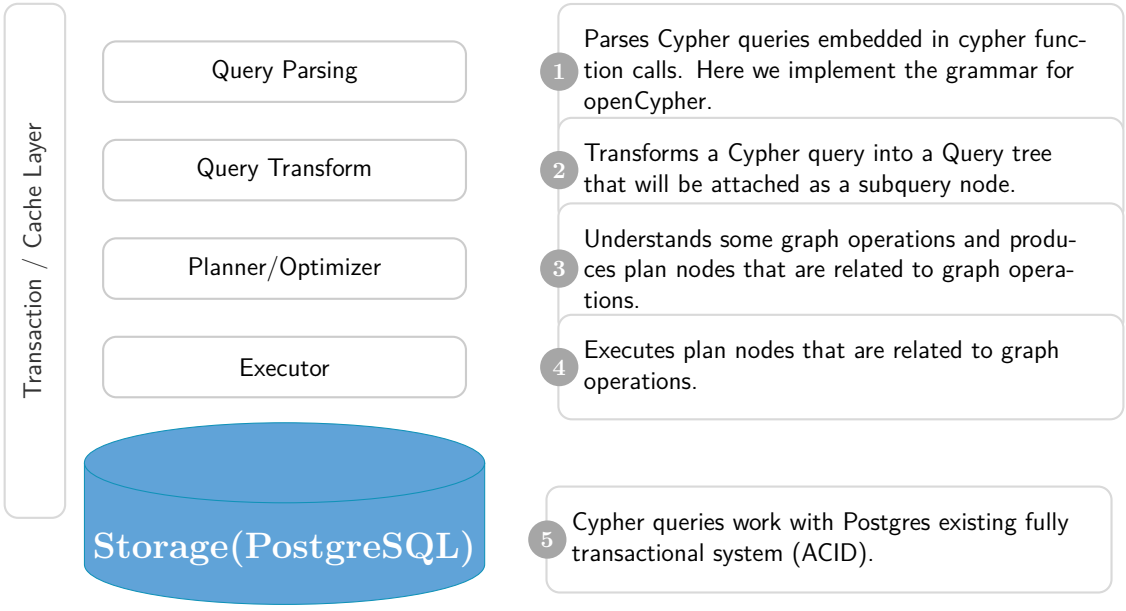

Figura 4: Architettura di Apache AGE integrata in PostgreSQL.

3.2.1 Property graph, tipi e query ibride

AGE mappa nodi e archi in tabelle PostgreSQL dedicate, utilizzando il tipo di dato `agtype` per memorizzare le proprietà e rappresentare in modo uniforme tutti i valori restituiti dalle query Cypher. Il tipo `agtype` è definito come un superset esteso di JSON, ispirato a `jsonb` ma ottimizzato per supportare in modo nativo i costrutti del property graph: i *vertex* (le entità o i nodi del dominio), gli *edge* (gli archi direzionali che codificano le relazioni) e i *path* (le sequenze concatenate di nodi e archi). In particolare, `agtype` può rappresentare scalari come numeri e booleani, collezioni quali array e oggetti, nonché le strutture topologiche complesse del grafo stesso, mantenendo un modello omogeneo per l'intero risultato di una query.

L'architettura illustrata in Figura 4 mostra come il parser di AGE riconosca le chiamate alla funzione `cypher()`, le trasformi in un *query tree* (la rappresentazione logica e gerarchica dell'istruzione) e le affidi direttamente al pianificatore e ottimizzatore di PostgreSQL. I nodi del piano relativi alle operazioni sul grafo vengono poi eseguiti dal motore, che sfrutta il sistema transazionale ACID del database sottostante. In questo modo, le operazioni Cypher partecipano allo stesso modello di concorrenza e durabilità delle normali query SQL, senza la necessità di introdurre un motore di computazione separato.

La funzione `cypher(nome_grafo, query)` accetta una stringa contenente l'interrogazione Cypher e restituisce un insieme di righe in cui ogni colonna è implicitamente di tipo `agtype`. Per integrare i risultati Cypher all'interno di query SQL ibride, ad esempio *join* con tabelle relazionali/hypertable, è obbligatorio specificare uno schema di output tramite la clausola `AS`, che associa a ciascuna colonna proiettata un alias e un tipo nativo PostgreSQL compatibile.

Esempio di query Cypher semplice

```
SELECT *
    FROM cypher('demo_graph', $$
    RETURN 10, 'hello'
$$) AS (value_int INTEGER, value_text TEXT);
```

Figura 5: Esempio di query Cypher semplice.

In questa figura 5, la query Cypher restituisce due valori scalari che, grazie alla clausola `AS (...)` dichiarata a valle della funzione, vengono automaticamente convertiti da `agtype` ai tipi SQL nativi `INTEGER` e `TEXT`. Lo stesso meccanismo si applica a valori derivati da nodi e archi del grafo: è sufficiente proiettare in Cypher le proprietà di interesse e dichiararne il tipo SQL corrispondente.

Nel sistema sperimentale della presente tesi, tale schema posizionale viene sfruttato per costruire query ibride capaci di combinare la navigazione sul grafo e le aggregazioni temporali. Un caso tipico, particolarmente rilevante nel dominio dell'agricoltura di precisione, consiste nell'utilizzare la sintassi Cypher per individuare i dispositivi associati a una specifica azienda agricola e, successivamente, nel correlare tali sensori con le misurazioni registrate nella hypertable time-series. Il frammento seguente, mostrato in Figura 6, rappresenta un esempio di questa implementazione:

Esempio di query ibrida con casting obbligatorio

```
WITH min_temp AS (
        SELECT m.device_id, m.value
        FROM measurements m
        WHERE m.controlled_property = 'temperature'
        ORDER BY m.value ASC
        LIMIT 1
      )
      SELECT * FROM cypher('agri_graph', $$
        MATCH (d:Device)
        RETURN d
      $$) as (d agtype)
      WHERE d::json -> 'properties' ->> 'id' = (
        SELECT device_id FROM min_temp);
```

Figura 6: Esempio di query ibrida con casting obbligatorio

La necessità di dichiarare esplicitamente il tipo di ogni colonna restituita da `cypher()` rappresenta uno degli aspetti più delicati per la costruzione di query ibride tramite Large Language Models. Se lo schema è incompleto e non specifica accuratamente il tipo SQL, il motore non è in grado di effettuare il *casting* (il processo di conversione esplicita dal formato `agtype` a quello relazionale standard) e genera errori di runtime. Inoltre, ogni colonna deve essere dichiarata con un alias consistente e un tipo compatibile con le espressioni SQL successive; in caso contrario, il pianificatore fallirà la risoluzione dei riferimenti logici, producendo un errore bloccante già in fase di pianificazione della query.

3.3 TimescaleDB: gestione delle serie temporali

TimescaleDB estende le capacità native di PostgreSQL introducendo un'architettura di storage specificamente ottimizzata per la gestione di carichi di lavoro legati a serie temporali ad alta frequenza [10]. Nei contesti applicativi caratterizzati da flussi continui di metriche, i database tradizionali mostrano forti limiti prestazionali dovuti alla rapida crescita del volume dei dati. TimescaleDB affronta questo problema trasformando tabelle ordinarie in strutture altamente scalabili, ideate per supportare un ingest continuo ad alta velocità e query estremamente efficienti, mantenendo la totale compatibilità SQL.

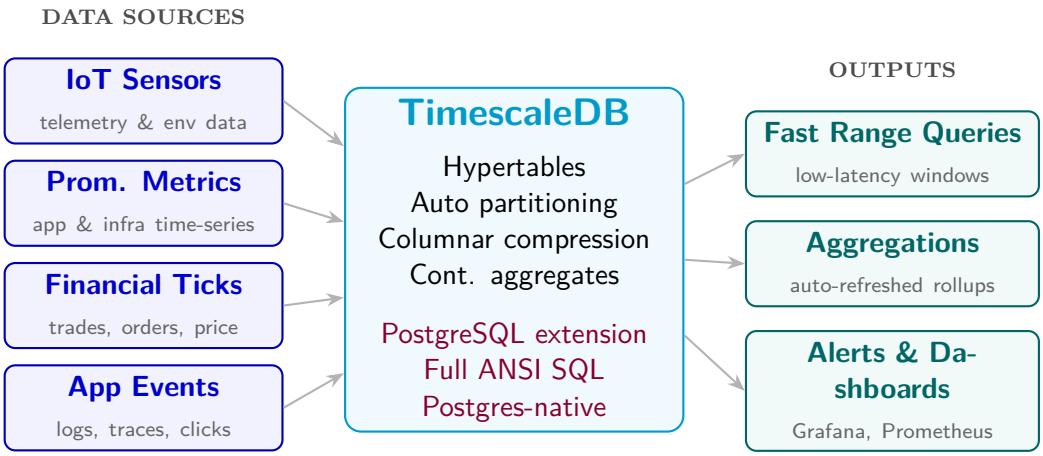

Figura 7: Architettura del flusso di dati con TimescaleDB.

Come evidenziato dal flusso architetturale in Figura 7, questa specializzazione si articola su tre pilastri fondamentali:

- **l'astrazione dell'Hypertable**: L'astrazione centrale del sistema è l'hypertable, una struttura logica che appare all'utente come una singola tabella continua. Internamente, essa partiziona automaticamente i dati in chunk, ovvero tabelle fisiche più piccole che raggruppano i record per finestre temporali. Quando i dati superano la soglia `chunk_time_interval`, viene generato un nuovo chunk per le scritture, mentre i precedenti diventano di sola lettura. Questo previene il table bloat, ossia la frammentazione dello spazio su disco. L'organizzazione in chunk abilita nativamente il chunk pruning, per l'esclusione a priori delle partizioni non pertinenti all'intervallo temporale della query, e il

predicate pushdown, applicando i filtri di ricerca a livello di singola partizione[10].

- **columnar compression**: I chunk storici vengono convertiti utilizzando la columnar compression, una riorganizzazione fisica dello storage che memorizza i dati per colonna anziché per riga, accelerando le letture analitiche. Il sistema riduce lo spazio su disco sfruttando la ridondanza temporale con algoritmi specifici: il delta-of-delta encoding per comprimere i timestamp consecutivi, il run-length encoding per le sequenze di valori identici, e la XOR-based compression per ottimizzare la codifica dei numeri in virgola mobile. La decompressione avviene in modo del tutto trasparente per l'applicazione finale, preservando le capacità di filtraggio tramite gli header dei chunk (min/max/bounds).

- **continuous aggregates**: Per ottimizzare le interrogazioni analitiche su volumi massivi, TimescaleDB ricorre alle continuous aggregates, speciali viste materializzate che pre-calcolano le aggregazioni in modo incrementale dalla hypertable raw. I calcoli vengono eseguiti su specifici time bucket, intervalli temporali regolari come scatti orari o giornalieri, e mantenuti aggiornati in background tramite job schedulati. Questo meccanismo supporta il downsampling, la riduzione della risoluzione temporale raggruppando dati ad alta frequenza, consentendo di popolare le dashboard in tempo reale senza dover ricalcolare l'intero storico grezzo.

4 LLM e in-context Learning

4.1 Generazione di query tramite LLM e In-Context Learning

I LLM basati sull'architettura Transformer hanno dimostrato capacità avanzate non solo nella comprensione semantica, ma anche nella sintesi di linguaggi formali. Grazie all'esposizione massiva a vasti corpus di dati contenenti codice sorgente e documentazione architetturale, i modelli moderni possiedono una notevole competenza nei task di traduzione da linguaggio naturale a linguaggi di interrogazione strutturati, noti in letteratura come *Text-to-SQL* e *Text-to-Cypher*.

Tuttavia, quando si opera su architetture multistore che richiedono la generazione di query ibride complesse, la sola conoscenza latente del modello non è sufficiente. Garantire la correttezza sintattica, il rispetto dei vincoli di *casting* esplicito introdotti da Apache AGE e l'uso corretto delle astrazioni temporali di TimescaleDB richiede un livello di guida contestuale altamente specifico, impossibile da ottenere affidandosi unicamente all'addestramento di base.

4.1.1 In-Context Learning come paradigma di adattamento

Per colmare questo divario, la presente tesi adotta il concetto dell'**In-Context Learning** (ICL), il quale consente agli LLM di eseguire compiti specifici condizionandosi su alcuni esempi dimostrativi forniti in fase di interrogazione, senza richiedere l'aggiornamento dei parametri interni della rete neurale [12].

L'efficienza di questo approccio rappresenta un cambiamento fondamentale rispetto alle tradizionali metodologie di *fine-tuning*, ovvero il riaddestramento dei pesi del modello su dataset etichettati, offrendo un meccanismo molto più flessibile per l'adattamento ai nuovi task. Studi recenti che hanno investigato i meccanismi interni dell'ICL hanno identificato un punto esatto di "riconoscimento del task" all'interno dell'architettura del modello, dimostrando che ottimizzare queste dinamiche può portare a notevoli risparmi computazionali [12].

4.2 Over-prompting e Dynamic Few-Shot Prompting

4.2.1 Over-prompting e Lost in the Middle

Sebbene l'ICL rappresenti uno strumento potente, la sua applicazione pratica espone i modelli a vulnerabilità architetturali significative. Inizialmente, la comunità di ricerca ipotizzava che l'inserimento del maggior numero possibile di esempi dimostrativi all'interno del *prompt*, garantisse sempre un miglioramento delle prestazioni.

Tuttavia, evidenze sperimentali hanno smentito questa ipotesi identificando il fenomeno dell'**over-prompting**: una dinamica in cui l'inclusione di un numero eccessivo di esempi specifici del dominio porta a un degrado non indifferente delle prestazioni dell'LLM [11].

L'effetto deriva da un limite presente nella gestione delle finestre di contesto estese. Gli LLM raggiungono prestazioni ottimali quando le informazioni rilevanti sono posizionate all'inizio o alla fine del contesto di input, ma subiscono un drastico calo di comprensione per le informazioni situate al centro [11]. Questo limite strutturale, noto come *Lost in the Middle*, rende di fatto inefficace un approccio statico che provi a istruire il modello caricando simultaneamente decine di esempi di query; infatti il modello finirebbe per ignorare le regole sintattiche contenute nella parte centrale del testo, producendo interrogazioni non valide, come rappresentato in figura-8.

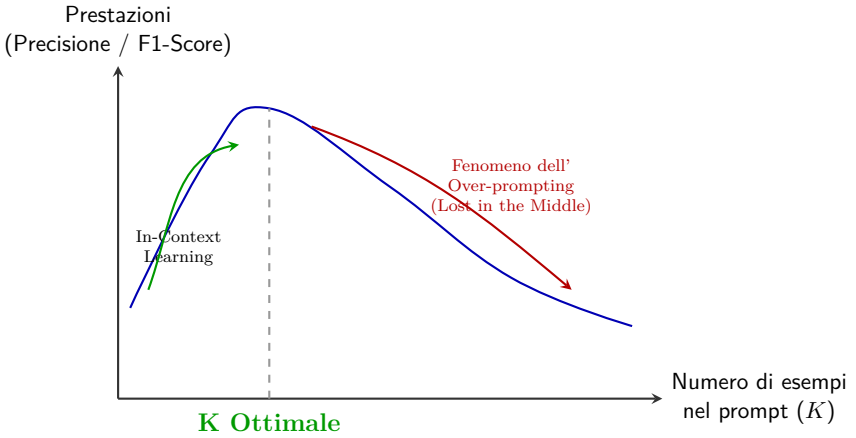

Figura 8: Rappresentazione qualitativa del fenomeno dell'over-prompting.

4.2.2 Dynamic Few-Shot Prompting come soluzione

Per mitigare gli effetti dell'*over-prompting* e aggirare la degradazione dell'attenzione dovuta al *Lost in the Middle*, il framework proposto in questa tesi abbandona l'approccio statico a favore del **"Dynamic" Few-Shot Prompting**. Questa strategia prevede che il sistema estragga in tempo reale unicamente un numero limitato di esempi altamente pertinenti da un *Ground Truth* di riferimento, invece di utilizzare un blocco fisso di istruzioni. Nel sistema sviluppato, tale base di conoscenza è costituita da 20 esempi, progettati per coprire le casistiche fondamentali del dominio dell'agricoltura di precisione 9.

Dal punto di vista architetturale, è necessario tracciare una netta distinzione tra questa metodologia e il paradigma **RAG** (*Retrieval-Augmented Generation*). Infatti Il RAG è progettato per recuperare frammenti testuali contenenti informazioni fattuali per colmare le lacune di conoscenza del modello. Il Dynamic Few-Shot invece ha uno scopo esclusivamente logico-sintattico. Esso non recupera i dati necessari a comporre la risposta, bensì i pattern strutturali di interrogazione. Gli esempi estratti forniscono al modello dei pattern sintattici, mostrandogli come annidare correttamente la funzione cypher() all'interno di una query SQL di aggregazione.

Figura 9: Flusso architetturale semplificato del Dynamic Few-Shot Prompting.

4.3 Information retrieval e selezione degli esempi

Nel framework proposto, l'efficacia del *Dynamic Few-Shot Prompting* dipende dalla rilevanza degli esempi estratti rispetto alla richiesta dell'utente 10. Questi sistemi di selezione sfruttano due metodologie standard di *Information Retrieval*. La prima **semantic embedding** (ricerca semantica) converte le frasi originali in vettori densi, calcolando la similarità tramite distanza coseno all'interno di uno spazio vettoriale. Questo permette di individuare gli esempi che condividono lo stesso intento logico della richiesta in ingresso, gestendo in modo nativo la variabilità linguistica e la sinonimia [11]. La seconda metodologia si basa sulla **vettori TF-IDF** (ricerca lessicale) o algoritmo *Term Frequency-Inverse Document Frequency* che converte le frasi valutando la frequenza di parole chiave all'interno del corpus, bilanciandola in base alla loro rarità complessiva.

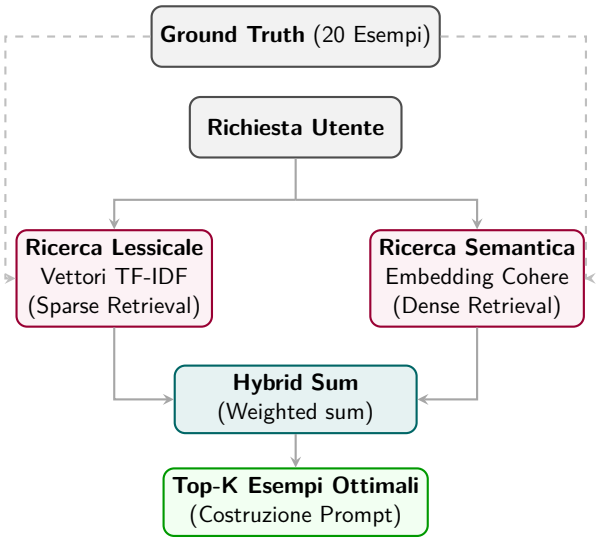

Figura 10: Architettura del motore di Information Retrieval ibrido.

L'approccio basato sui vettori TF-IDF supera l'embedding semantico e il campionamento casuale, poiché privilegia i termini tecnici e le parole chiave specifiche del dominio, a discapito di sinonimi o varianti lessicali [11]. Nel contesto di un sistema multistore, il TF-IDF è importante per intercettare esatte corrispondenze di dominio che l'analisi semantica potrebbe sfumare, come le nomenclature specifiche dei sensori IoT o determinate metriche agricole. Nel framework implementato, l'uso combinato della ricerca semantica e lessicale consente di estrarre gli esempi più significativi, permettendo ai modelli di raggiungere prestazioni superiori con un numero ridotto di esempi e mitigando gli effetti dell'*over-prompting*.

5 Progettazione e sviluppo del framework sperimentale

5.1 Architettura end-to-end della pipeline

Il cuore della sperimentazione è un framework automatizzato, sviluppato in linguaggio Python, capace di orchestrare l'intero processo di traduzione da una richiesta in linguaggio naturale a una query ibrida (Cypher e SQL) da eseguire su un sistema multistore. Il codice sorgente dell'implementazione è disponibile su GitHub [15]. Per gestire l'elevata complessità di questo task, il sistema è stato modellato come una pipeline sequenziale end-to-end, in cui il flusso attraversa moduli incaricati a risolvere un preciso step.

L'esecuzione prende avvio con l'inizializzazione del contesto: il framework carica in memoria le istruzioni base e una sintesi strutturale del database. All'arrivo di una domanda utente, si innesca la fase di Information Retrieval: un motore di ricerca ibrido scandaglia un ground truth (GT) di riferimento per estrarre gli esempi storici più affini alla richiesta. Ottenuti questi pattern, un modulo dedicato assembla il system prompt finale e gestisce la comunicazione con l'endpoint API del LLM. Nella fase conclusiva, la query testuale restituita dal modello viene intercettata, pulita da eventuali caratteri di formattazione e iniettata nel motore PostgreSQL tramite un cursore abilitato per l'estensione Apache AGE. I risultati dell'esecuzione, o le eccezioni sintattiche sollevate, vengono infine serializzati per la successiva fase di analisi. La figura 11 illustra l'architettura complessiva della pipeline.

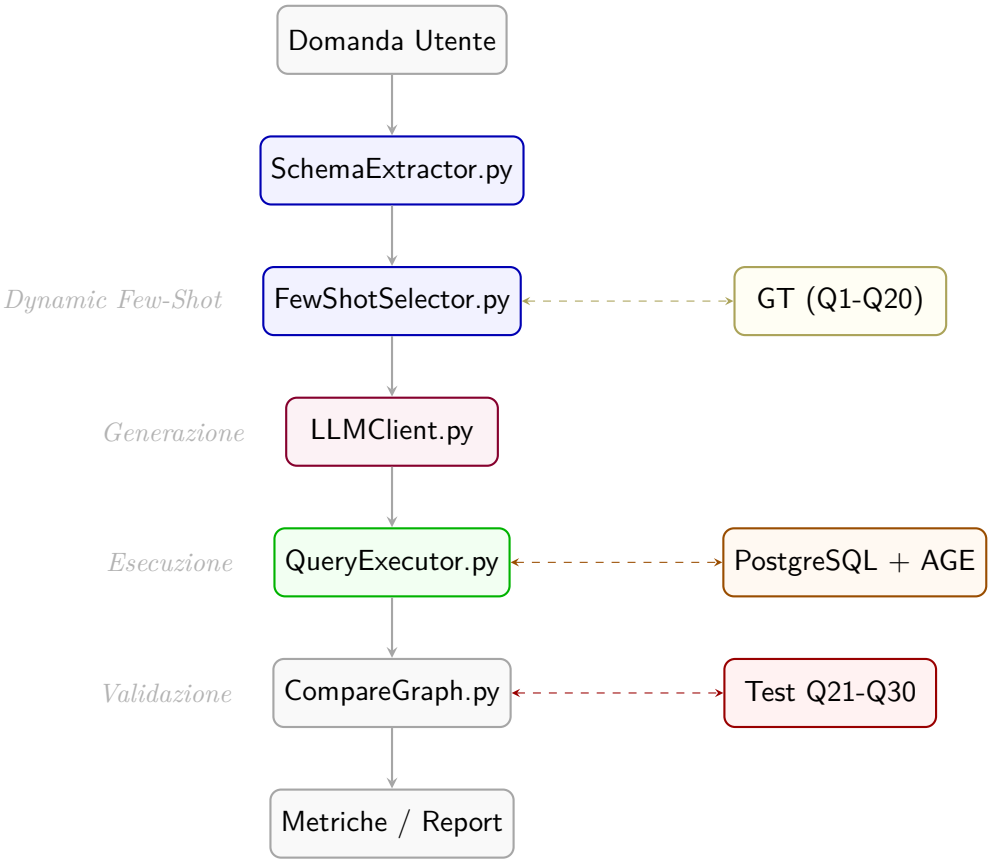

Figura 11: Schema della pipeline end-to-end.

5.2 Inizializzazione del prompt e astrazione del dominio

L'interazione con modelli linguistici su architetture multistore richiede un'attenta fase di Prompt Engineering (ragionamenti e tecniche per migliorare il prompt). Essendo Apache AGE un'estensione restrittiva, che impone casting espliciti e isola gli operatori SQL dai blocchi `MATCH` di Cypher, non è sufficiente delegare al LLM la libera stesura del codice. È stato necessario progettare un prompt capace di tradurre i vincoli hardware e topologici del dominio in rigide regole sintattiche.

5.2.1 Definizione delle regole topologiche

La prima componente del system prompt è costituita dalle istruzioni operative (`system_instructions2.txt`) che istruiscono il modello a compiere deduzioni logiche prima di generare la query. Il cuore del documento è la mappatura tra l'hardware IoT e la profondità nel property graph, già descritta nel capitolo 3. Nel prompt, queste informazioni vengono formalizzate per guidare la costruzione delle clausole `MATCH`. La figura 12 mostra un estratto del file.

`system_instructions2.txt` - Estratto della topologia hardware

```
## TOPOLOGIA HARDWARE

- Misurazioni di umidità ('soilMoisture'): Richiedono un doppio salto
    topologico.
  Il sensore foglia ècollegato a un gateway.
  Path:
      `(p:AgriParcel)<-[:belongsTo]-(d:Device)-[:hasDevice]->(d2:Device)`.
  I dati fisici sono collegati al nodo figlio, `d2`.

- Dispositivi di irrigazione (`dripper`): Richiedono un salto singolo.
  Sono collegati direttamente al lotto.
  Path: `(p:AgriParcel)<-[:belongsTo]-(d:Device)`.
  I dati fisici sono collegati al nodo `d`.
```

Figura 12: Estratto del file `system_instructions2.txt` che definisce le regole topologiche.

La scelta di includere questa mappatura nel prompt deriva da una considerazione fondamentale: la topologia del grafo determina quante clausole `MATCH` sono necessarie per collegare un lotto agricolo a un dispositivo che produce dati. Senza questa specifica, il modello non sarebbe ingrado di determinare la profondità del pattern da esprimere in Cypher. I sensori di umidità (soilMoisture) sono collegati tramite un gateway intermedio, quindi richiedono un doppio salto; gli attuatori di irrigazione (dripper) sono invece direttamente connessi al lotto, quindi richiedono un salto singolo.

5.2.2 Sequenza di ragionamento

Una delle modifiche introdotte è l'aggiunta di una procedura di ragionamento obbligatoria all'interno del prompt, ovvero Chain-of-Thought prompting, questo meccanismo forza il modello a esplicitare i passaggi logici che lo portano alla scelta della struttura della query, prima di scrivere il codice finale. La procedura guida il modello attraverso otto passaggi: I. identificare il tipo di dispositivo; II. determinare il percorso nel grafo; III. scegliere i nodi su cui filtrare; IV. verificare la corrispondenza tra variabili nel `RETURN` e nella clausola `AS`; V. decidere se restituire il path; VI. gestire condizioni multiple; VII. controllare l'uso di `UNION`; VIII. applicare il match testuale flessibile. Questo approccio riduce significativamente gli errori sintattici, in particolare il disallineamento tra le variabili dichiarate nel `RETURN` di Cypher e quelle utilizzate nel `WHERE` esterno, che è una delle cause più frequenti di errore nelle query ibride.

5.2.3 Estrazione e serializzazione dello schema a grafo

Le regole teoriche non bastano se il modello non conosce gli identificatori del database; fornire l'intero schema saturerebbe il contesto. Per risolvere questo problema, è stato implementato il modulo `SchemaExtractor.py`, il cui obiettivo è isolare una frazione rappresentativa della topologia da fornire al modello linguistico. La scelta di non includere l'intero grafo è dettata dalla necessità di rispettare i limiti di contesto del LLM, che non sarebbe in grado di elaborare l'intera struttura del database in una singola richiesta. La figura 13 mostra l'implementazione della funzione extract_schema. La funzione carica il grafo e, per ogni arco, recupera i campi `start_id` e `end_id`. Questi identificativi vengono utilizzati per risalire, tramite un dizionario di supporto, alle etichette dei nodi corrispondenti. La firma relazionale viene quindi

costruita concatenando le due etichette con il tipo di relazione, ottenendo una stringa compatta come (Device)-[:belongsTo]->(AgriParcel). Il risultato viene serializzato in formato `yaml`, che riduce il consumo di token rispetto al JSON.

```
for edge in data.get('edges', []):
    rel_type = edge.get('type') or edge.get('label')
    if rel_type and rel_type != 'hasMeasurement':
        start_label = id_to_label.get(edge.get('start_id'), "UnknownNode")
        end_label = id_to_label.get(edge.get('end_id'), "UnknownNode")

        signature = f"({start_label})-[:{rel_type}]->({end_label})"
        edge_signatures.add(signature)

schema_summary["graph_model"]["relationships"] =
    sorted(list(edge_signatures))
return yaml.dump(schema_summary, sort_keys=False, allow_unicode=True)
```

Figura 13: Estrazione delle firme relazionali e serializzazione `yaml` in SchemaExtractor.py.

La scelta di serializzare in `yaml` anziché in JSON deriva da valutazioni empiriche condotte durante lo sviluppo: il formato `yaml`, essendo privo di parentesi graffe e doppi apici ricorrenti, riduce il consumo di token e risulta più leggibile per il modello Transformer, migliorando la qualità delle query generate.

5.3 Selezione dinamica degli esempi (dynamic few-shot)

Le istruzioni e lo schema forniscono il dominio, ma non insegnano la sintassi ibrida. Questo compito è affidato al Dynamic Few-Shot Prompting, che si basa su un GT curato e su un selettore di esempi. Il GT è composto da 20 interrogazioni (Q1-Q20), suddivise per tipologia nella tabella 1. Questa suddivisione è stata progettata per coprire l'intero spettro delle operazioni multistore: interrogazioni gerarchiche, filtri su proprietà, aggregazioni, match testuali e path. La scelta di 20 esempi rappresenta un compromesso tra completezza e sovraccarico cognitivo del modello: un numero maggiore di esem-

pi avrebbe saturato la finestra di contesto (fenomeno dell'over-prompting), mentre un numero minore non avrebbe coperto tutte le casistiche necessarie.

Tipologia	**Id query**
Navigazione gerarchica	Q1, Q2, Q4, Q12, Q18
Filtri su proprietà	Q3, Q6, Q11, Q17
Aggregazioni e ordinamenti	Q7, Q9, Q10, Q15
Match testuale flessibile	Q5, Q16
Path con OPTIONAL MATCH	Q2, Q16
Join grafo-tabella measurements	Q10, Q11, Q13, Q15, Q17, Q19

Tabella 1: Suddivisione delle 20 query del few-shot (Q1-Q20) per tipologia logica.

5.3.1 Hybrid Information Retrieval: TF-IDF e Cohere

Il modulo `FewShotSelector.py` implementa un motore di ricerca ibrido. All'inizializzazione, il costruttore carica o calcola gli embedding semantici di tutti gli esempi storici tramite la funzione `_batch_embed`, che invia i testi al modello cohere-embed-v3-multilingual (figura 14). Per minimizzare le chiamate API, gli embedding vengono salvati in cache tramite pickle, riducendo i tempi di avvio del sistema da diversi secondi a pochi millisecondi dopo la prima esecuzione.

```
1 def _batch_embed(self, texts: List[str]) -> np.ndarray:
2     try:
3         response = self.client.embed(input=texts, model=self.model_name)
4         return np.array([item.embedding for item in response.data])
5     except Exception as e:
6         print(f"Errore durante il calcolo degli embedding: {e}")
7         return np.zeros((len(texts), 1024))
```

Figura 14: Funzione _batch_embed per il calcolo degli embedding semantici tramite Cohere.

La scelta di un modello di embedding rispetto a un altro (Cohere) [14] e di un approccio ibrido (semantico + lessicale) è motivata dalla natura delle domande in linguaggio naturale. Le query degli utenti possono variare ampiamente nella terminologia: un utente potrebbe chiedere "dispositivi di irrigazione" mentre nel database sono memorizzati come "dripper". L'embedding semantico cattura queste relazioni, mentre il TF-IDF garantisce che i termini tecnici esatti vengano comunque riconosciuti. La combinazione lineare con α=0.5 bilancia equamente i due contributi.

Il cuore del modulo è il metodo `select_top_k`, mostrato in figura 15. La domanda utente viene prima trasformata in embedding e confrontata con tutti gli embedding in cache tramite similarità del coseno, ottenendo un punteggio semantico. In parallelo, la domanda viene trasformata nello spazio TF-IDF e confrontata con la matrice storica, ottenendo un punteggio lessicale. I due punteggi vengono normalizzati tramite Min-Max per portarli nell'intervallo [0,1], rendendoli omogenei e confrontabili. La combinazione lineare con α=0.5 produce un punteggio ibrido, di conseguenza basta variare questo valore per cambiare il peso delle due metriche. La funzione `np.argsort` ordina i punteggi in modo decrescente e seleziona i `k` esempi più rilevanti. Questo valore è stato scelto perché fornisce un numero sufficiente di esempi per guidare il modello senza saturare il contesto (è stato testato a 2,3,5).

```
1  # Normalizzazione Min-Max e combinazione lineare
2  sem_min, sem_max = semantic_scores.min(), semantic_scores.max()
3  if sem_max > sem_min:
4      semantic_scores = (semantic_scores - sem_min) / (sem_max - sem_min)
5
6  user_tfidf = self.tfidf_vectorizer.transform([user_question])
7  lexical_scores = cosine_similarity(user_tfidf,
       self.tfidf_matrix).flatten()
8
9  lex_min, lex_max = lexical_scores.min(), lexical_scores.max()
10 if lex_max > lex_min:
11     lexical_scores = (lexical_scores - lex_min) / (lex_max - lex_min)
12
13 hybrid_scores = (alpha * semantic_scores) + ((1.0 - alpha) *
       lexical_scores)
14 top_indices = np.argsort(hybrid_scores)[::-1][:k]
```

Figura 15: Normalizzazione e fusione dei punteggi in `FewShotSelector.py`.

5.4 Creazione del prompt e interfacciamento LLM

Con lo schema e gli esempi selezionati, il modulo `LLMClient.py` costruisce il prompt finale e gestisce la comunicazione con il modello.

Il framework si appoggia alle API di GitHub Models tramite il client azure.ai.inference [13], astraendo dai singoli vendor. Nel corso dello studio, la pipeline è stata testata su GPT-4o, Codestral e LLaMA. L'astrazione del client ha permesso di cambiare modello semplicemente modificando un parametro, senza alterare il codice.

Il metodo `generate_query` (figure 16) legge le istruzioni di sistema dal file `system_instructions2.txt`, combinandole con lo schema `yaml` e gli esempi estratti per costruire il `SystemMessage`. La domanda utente costituisce lo `UserMessage`. La temperatura è fissata a 0.0 per rendere il modello deterministico: in un task sintattico come la scrittura di codice, qualsiasi deriva stocastica genererebbe eccezioni bloccanti. La risposta viene pulita da eventuali marcatori markdown che renderebbero la query non eseguibile.

```
full_system_content = f"{system_instructions}\n\nDATABASE␣
    SCHEMA\n{schema_context}{examples_text}"

messages: List[ChatRequestMessage] = [
    SystemMessage(content=full_system_content),
    UserMessage(content=user_question)

try:
    response = self.client.complete(
        messages=messages,
        temperature=0.0,
        model=self.model_name
    )
    query = response.choices[0].message.content.strip()

    if query.startswith("‘‘‘"):
        lines = query.splitlines()
        if len(lines) >= 3: query = "\n".join(lines[1:-1])
]
```

Figura 16: Costruzione del `SystemMessage` e `UserMessage` in `LLMClient.py` e chiamata API e pulizia automatica del markdown in `LLMClient.py`.

La figura 17 mostra un esempio di payload JSON effettivamente inviato al modello, con il campo system contenente il prompt completo e il campo user la domanda.

```
1  "model": "gpt-4o",
2  "temperature": 0.0,
3  "messages": [
4    {
5      "role": "system",
6      "content": "Sei un esperto di database multistore che integrano...
           \n\n## CONTESTO\n- Il grafo si chiama 'agri_graph'... \n\n##
           TOPOLOGIA HARDWARE\n- Misurazioni di umidità... \n\n## REGOLE
           CRITICHE\n... \n\n## RICERCA TESTUALE FLESSIBILE\n... \n\n##
           PROCEDURA DI RAGIONAMENTO OBBLIGATORIA\n... \n\n## FORMATO DI
           OUTPUT RICHIESTO\n... \n\nDATABASE SCHEMA\n..."
7    },
8    {
9      "role": "user",
10     "content": "Mostra la topologia di rete (grafo) di tutti gli
           apparecchi nel lotto 'Fondo Errano 2024 T0' che hanno trasmesso
           dati sulla proprietà 'dripper'."
11   }
12 ]
```

Figura 17: Esempio di payload JSON inviato alla API di completamento.

5.5 Esecuzione ibrida e dataset di validazione

La fase di validazione verifica che il modello abbia appreso la sintassi e non si limiti a copiare gli esempi. Sono state isolate 10 interrogazioni inedite (Q21-Q30, tabella 2), progettate per combinare vincoli topologici, temporali e metrici in scenari non visti nel GT. Questa separazione tra esempi di addestramento (Q1-Q20) e test (Q21-Q30) garantisce che la valutazione misuri la capacità di generalizzazione del modello, non la sua capacità di memorizzare gli esempi.

ID	**Descrizione sintetica**	**Tipologia**
Q21	Topologia rete nel lotto T0 con dripper	Navigazione gerarchica
Q22	Nodo con umidità minima storica	Aggregazioni e ordinamenti
Q23	Aziende con sensori soilMoisture < -500	Filtri su proprietà + Navigazione
Q24	Lotti kiwi con dispositivi e sensori	Match testuale + Path
Q25	Dispositivi con >100 eventi dripper	Aggregazioni e ordinamenti
Q26	Albero dispositivi nel lotto T1 con umidità <15	Path + Filtro temporale
Q27	Sensori finali collegati a gateway specifico	Path
Q28	Sensori con media soilMoisture <40	Aggregazioni e ordinamenti
Q29	Dispositivi senza sensori finali in ZESPRI	NOT EXISTS
Q30	Percorso con soilMoisture < 20 OR dripper > 0	Path con OPTIONAL MATCH

Tabella 2: Le 10 interrogazioni del set di test Q21-Q30.

5.5.1 Validazione sul database e salvataggio delle metriche

La query generata viene eseguita dal modulo `QueryExecutor.py`, Figura 18 mostra la funzione `execute_raw`, che esegue la query e recupera righe e colonne. La funzione `parse_agtype` decodifica i valori agtype rimuovendo i marcatori ::vertex, ::edge e ::path e caricando il JSON tramite `json.loads`. Questo passaggio è fondamentale perché i risultati di Apache AGE sono in formato agtype, non JSON puro. Se il caricamento JSON fallisce, il valore originale viene restituito come fallback, garantendo che il sistema non si interrompa in presenza di dati malformati. Se la query fallisce, l'eccezione viene catturata e registrata insieme allo stack di errore, permettendo di distinguere tra query che hanno prodotto un risultato e query che non sono state eseguite per errori sintattici.

```
1  def execute_raw(self, query: str) -> Tuple[List[tuple], List[str]]:
2      if self.cursor is None:
3          raise RuntimeError("Cursor not available")
4
5      query_clean = query.strip()
6      self.cursor.execute(query_clean)
7
8      if self.cursor.description is None:
9          return [], []
10     rows = self.cursor.fetchall()
11     column_names = [desc[0] for desc in self.cursor.description]
12     return rows, column_names
13
14 def parse_agtype(self, value: Any) -> Any:
15     if value is None:
16         return None
17     if isinstance(value, str):
18         clean = value.split('::')[0].strip()
19         try:
20             return json.loads(clean)
21         except (json.JSONDecodeError, TypeError):
22             return value
23     return value
```

Figura 18: Funzioni `execute_raw` e `parse_agtype` in `db_conn.py`.

5.6 Confronto Strutturale tra Grafi e Calcolo delle Metriche

L'ultimo anello della pipeline è il modulo `CompareGraph.py`, che confronta i risultati del GT e quelli generati dal LLM. Un aspetto critico è la gestione del formato `agtype` infatti la funzione `parse_agtype_value` (figura 19) riconosce il suffisso ::path, delegando a `parse_path` che scandisce la stringa carattere per carattere estraendo gli oggetti JSON. Per singoli vertex o edge, la funzione rimuove il marcatore e carica il JSON con `json.loads`. Questo parser è stato implementato manualmente perché la libreria standard di Python non fornisce un supporto nativo per il formato `agtype` di Apache AGE.

```
@staticmethod
def parse_agtype_value(value: Any) -> Any:
    if value is None:
        return None
    if isinstance(value, str):
        stripped = value.strip()
        if stripped.endswith('::path'):
            return CompareGraph.parse_path(stripped)
        clean_value = stripped.split('::')[0].strip()
        try:
            return json.loads(clean_value)
        except (json.JSONDecodeError, ValueError):
            return value
    return value
```

Figura 19: Funzione `parse_agtype_value` in `CompareGraph.py`.

Le funzioni `is_node` e `is_edge`, mostrate in figura 20, determinano il tipo di elemento controllando la presenza di campi specifici: un nodo ha properties con id e label, un arco ha `start_id`, `end_id` e `label`. La funzione `normalize_node` estrae dall'oggetto il campo id dalle properties e la label, restituendo una tupla `(id, label)`. `normalize_edge` segue la stessa logica, estraendo `start_id`, `end_id`, `label` e `id` dell'arco.

```
1  @staticmethod
2  def is_node(item: Any) -> bool:
3      if not isinstance(item, dict):
4          return False
5      properties = item.get('properties', {})
6      has_properties_id = isinstance(properties, dict) and 'id' in
           properties
7      has_label = 'label' in item
8      is_not_edge = 'start_id' not in item and 'end_id' not in item
9      return has_properties_id and has_label and is_not_edge
10
11 @staticmethod
12 def is_edge(item: Any) -> bool:
13     if not isinstance(item, dict):
14         return False
15     has_start = 'start_id' in item
16     has_end = 'end_id' in item
17     has_label = 'label' in item
18     return has_start and has_end and has_label
19
20 @staticmethod
21 def normalize_node(node: Dict) -> Tuple[str, str]:
22     properties = node.get('properties', {})
23     node_id = properties.get('id', '')
24     label = node.get('label', '')
25     return (str(node_id), str(label))
```

Figura 20: Funzioni `is_node`, `is_edge` e `normalize_node` in `CompareGraph.py`.

La funzione `extract_from_item` (figura 21) visita ricorsivamente ogni riga, estraendo nodi e archi. Durante l'estrazione, costruisce una mappa `node_id_map` che associa gli ID interni agli ID semantici, necessaria per risolvere i riferimenti degli archi. Questa visita ricorsiva è necessaria perché i risultati di Cypher possono contenere strutture annidate di profondità arbitraria.

```
1  @staticmethod
2  def extract_from_item(item: Any, nodes: Set, edges: Set, node_id_map:
       Dict):
3      if item is None:
4          return
5      item = CompareGraph.parse_agtype_value(item)
6      if isinstance(item, dict):
7          if CompareGraph.is_edge(item):
8              edge_tuple = CompareGraph.normalize_edge(item)
9              edges.add(edge_tuple)
10         elif CompareGraph.is_node(item):
11             node_tuple = CompareGraph.normalize_node(item)
12             nodes.add(node_tuple)
13             internal_id = str(item.get('id', ''))
14             semantic_id = str(item.get('properties', {}).get('id', ''))
15             if internal_id and semantic_id:
16                 node_id_map[internal_id] = semantic_id
17         else:
18             for value in item.values():
19                 CompareGraph.extract_from_item(value, nodes, edges,
                       node_id_map)
20     elif isinstance(item, list):
21         for element in item:
22             CompareGraph.extract_from_item(element, nodes, edges,
                   node_id_map)
```

Figura 21: Funzione `extract_from_item` per l'estrazione ricorsiva di nodi e archi.

Infine, la funzione compare (figura 22) esegue la differenza insiemistica tra GT e LLM, producendo metriche di missing ed extra per nodi e archi, oltre ai conteggi totali. Questo approccio fornisce una valutazione strutturale indipendente dall'ordine delle righe, a differenza di un confronto posizionale che sarebbe fragile e dipendente dall'ordinamento.

```
1  @staticmethod
2  def compare(results_gt: List[Dict], results_llm: List[Dict], id: str) ->
       ComparisonMetrics:
3      gt_nodes, gt_edges = CompareGraph.extract_graph_elements(results_gt)
4      llm_nodes, llm_edges =
           CompareGraph.extract_graph_elements(results_llm)
5
6      missllm_nodes = gt_nodes - llm_nodes
7      extrallm_nodes = llm_nodes - gt_nodes
8      missllm_edges = gt_edges - llm_edges
9      extrallm_edges = llm_edges - gt_edges
10
11     return ComparisonMetrics(
12         query_id=id,
13         missing_llm={'nodes': list(missllm_nodes), 'edges':
               list(missllm_edges)},
14         extra_llm={'nodes': list(extrallm_nodes), 'edges':
               list(extrallm_edges)},
15         nodes_gt=len(gt_nodes), nodes_llm=len(llm_nodes),
16         edges_gt=len(gt_edges), edges_llm=len(llm_edges)
17     )
```

Figura 22: Funzione di confronto insiemistico tra grafo atteso e generato.

Le metriche così ottenute vengono serializzate in file JSON e costituiscono la base per l'analisi sperimentale del Capitolo 6, dove vengono confrontate le performance dei modelli e discussi i pattern di errore.

6 Valutazione sperimentale e analisi dei risultati

L'infrastruttura di valutazione definita nel Capitolo 5 ha permesso di quantificare empiricamente le capacità dei LLM nel tradurre il linguaggio naturale in query ibride Cypher+SQL. Al fine di valutare la reale flessibilità del framework e la sua resilienza architetturale, l'analisi si è concentrata esclusivamente sul set di test composto dalle interrogazioni `Q21-Q30`. Tali interrogazioni, inedite per il modello al momento della generazione, sono state progettate per stressare la navigazione topologica, i filtri spazio-temporali e le aggregazioni relazionali in scenari complessi.

Il capitolo discute i risultati emersi confrontando tre configurazioni dei parametri, selezionati per coprire lo spettro delle possibili combinazioni: la configurazione ibrida base (K=3, $\alpha = 0.5$), la configurazione ibrida estesa (K=5, $\alpha = 0.5$) e la configurazione puramente semantica (K=3, $\alpha = 1.0$).

6.1 Analisi dei casi di test

Per la valutazione dei singoli casi di test, sono state effettuate due metriche in base alla tipologia di grafico e analisi.

Nei grafici radar (Figg. 23, 26, 29) si è adottata una metrica basata sulla completezza dell'estrazione, che tollera nodi o archi extra, purché tutti gli elementi attesi siano presenti. Questa scelta riflette la natura del problema: nella generazione di query ibride, un risultato che contiene informazioni aggiuntive ma non ne perde è preferibile a uno che ne omette. I radar mostrano quindi il tasso di successo basato sul *recall*.

Nei grafici a barre (Figg. 24, 27, 30) si è invece adottata una metrica più rigorosa: una query viene considerata "corretta" solo se il risultato coincide esattamente con il ground truth. In giallo sono rappresentati i mismatch semantici. Questa metrica, basata sulla *precisione*, fornisce una misura più severa della qualità della query.

Per comprendere la natura dei fallimenti, è stata sviluppata una tassonomia che classifica gli errori in cinque categorie, visualizzata nel terzo grafico di ogni test (Figg. 25, 28, 31).

La categoria più grave è rappresentata dagli **errori sintattici o di dialetto**, in cui la query genera un'eccezione a runtime, impedendo al database di restituire qualsiasi risultato. Rientrano in questa tipologia l'uso di costrutti Cypher non supportati da Apache AGE (come `NOT EXISTS` nella forma compatta) o l'iniezione di testo discorsivo nel payload SQL.

Il secondo pattern è il **mismatch semantico**, in cui la query è sintatticamente valida ma la logica di filtraggio è errata. Il modello seleziona entità non pertinenti, ad esempio confondendo il sensore con il valore massimo anziché quello con il minimo storico.

Il terzo pattern è rappresentato dalle **allucinazioni topologiche**, in cui il modello esegue un over-fetching ingiustificato, introducendo nodi o archi non richiesti (es. estrazione dell'intera gerarchia aziendale quando era richiesto solo un lotto).

Il quarto pattern è il **grafo incompleto**, in cui la query omette intere sezioni topologiche. Su Q30, ad esempio, il modello estrae 69 nodi ma solo 10 archi su 61, indicando una ricostruzione parziale della struttura.

Infine, la criticità più frequente è il **collasso topologico**, in cui il modello estrae correttamente i nodi tramite SQL ma fallisce nel ricostruire i path Cypher, producendo nodi isolati e zero archi. Questo errore si manifesta sistematicamente su Q28 in tutti i modelli e configurazioni, indicando una difficoltà intrinseca del task.

6.1.1 Test con parametri: K=3, $\alpha = 0.5$

Questa configurazione rappresenta il punto di partenza del framework, con un contesto di retrieval di base e un bilanciamento equo tra semantica e lessico.

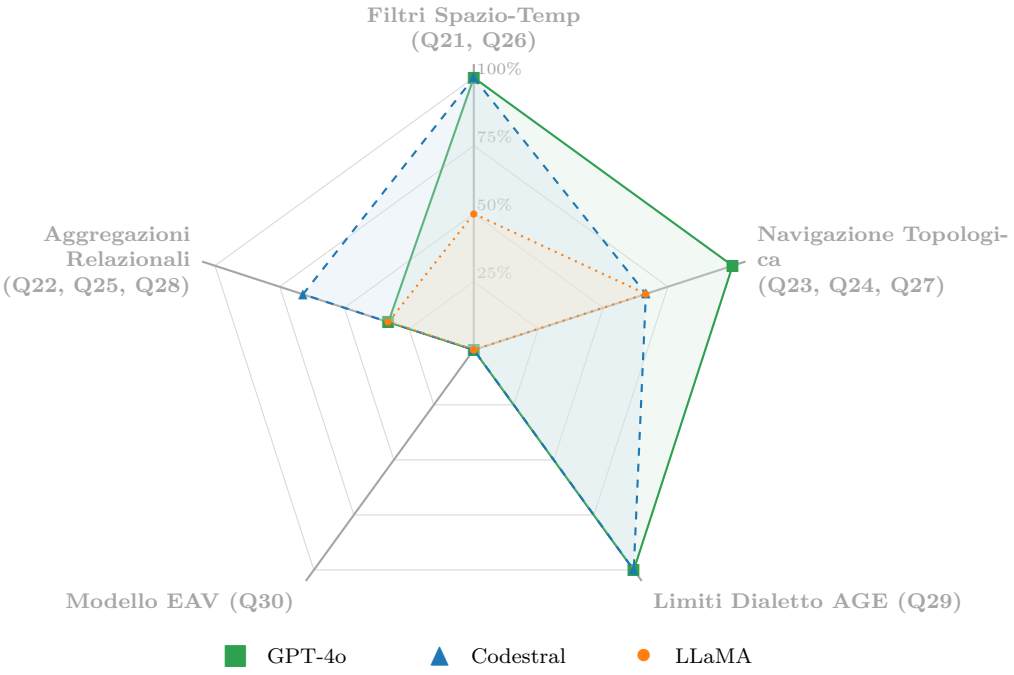

Figura 23: Test $K = 3, \alpha = 0.5$: Radar - Tasso di successo per categoria architetturale (basato su *recall*).

Come mostrato in Figura 23, GPT-4o e Codestral mostrano profili simili, con GPT-4o che eccelle in topologia e Codestral nelle aggregazioni. LLaMA evidenzia un forte over-fetching in topologia (Q24).

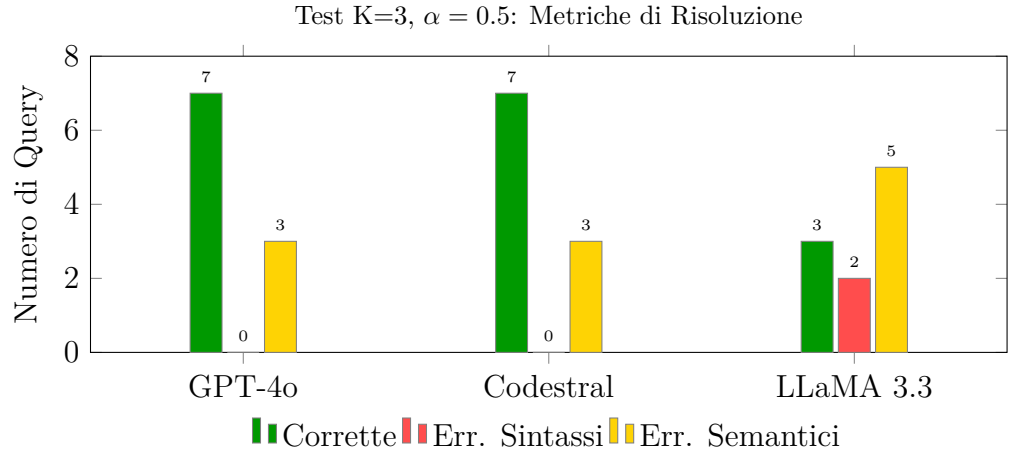

Figura 24: Test K=3, $\alpha = 0.5$: Distribuzione degli esiti (basata su *precisione* esatta).

Da Figura 24 emerge che GPT-4o e Codestral mostrano un'assoluta stabilità sintattica (zero crash), mentre LLaMA-3.3-70B soffre di allucinazioni.

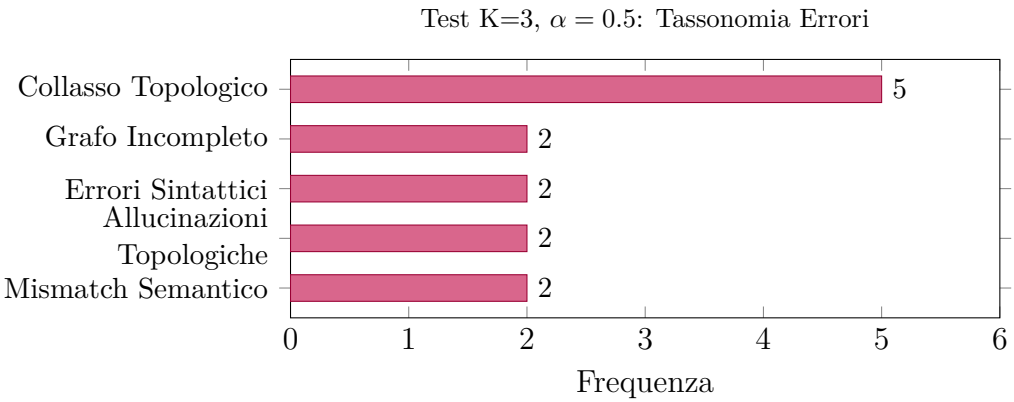

Figura 25: Test K=3, $\alpha = 0.5$: Frequenza aggregata dei fallimenti.

Come evidenziato in Figura 25, il "Collasso Topologico" si conferma la criticità architetturale predominante (5 occorrenze su 13 fallimenti totali), manifestandosi sistematicamente su Q28. GPT-4o e Codestral sono perfettamente equivalenti in termini di query corrette (7/10) e zero errori sintattici. LLaMA ottiene solo 3/10 query corrette, con una distribuzione di errori sbilanciata verso gli errori semantici.

6.1.2 Test con parametri: K=5, $\alpha = 0.5$

Questa configurazione rappresenta il punto ottimale del framework, con un contesto di retrieval esteso e un bilanciamento equo tra semantica e lessico.

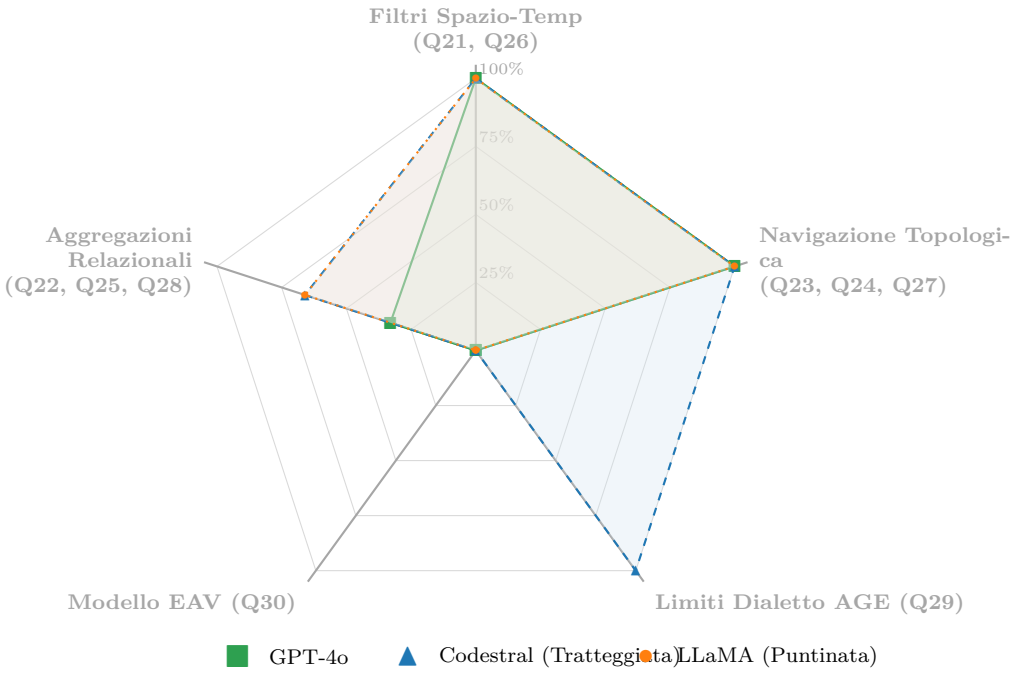

Figura 26: Test $K = 3, \alpha = 1.0$: Radar - Tasso di successo per categoria architetturale (basato su *recall*).

Come mostrato in Figura 29, la convergenza delle prestazioni è massima: GPT-4o e Codestral raggiungono il 100% su tre categorie su cinque. Codestral si distingue superando nettamente GPT-4o sulle aggregazioni relazionali (raggiungendo il 66% di successo contro il 33% del modello OpenAI), pur confermando insieme agli altri modelli l'incapacità di risolvere il Modello EAV (0%).

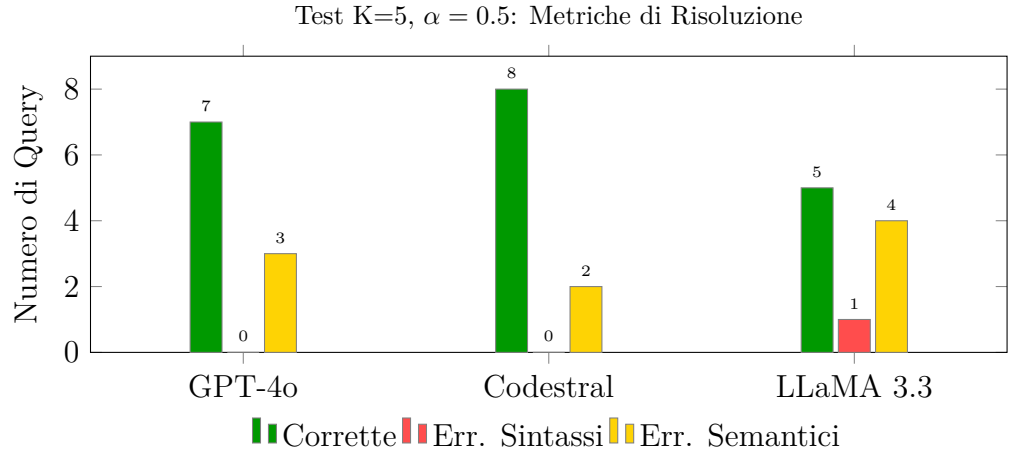

Figura 27: Test K=5, $\alpha = 0.5$: Distribuzione degli esiti (basata su *precisione* esatta).

Da Figura 27 si osserva che GPT-4o e Codestral non generano alcun crash di sistema mentre LLaMA-3.3-70B continua a manifestare problemi di aderenza al formato.

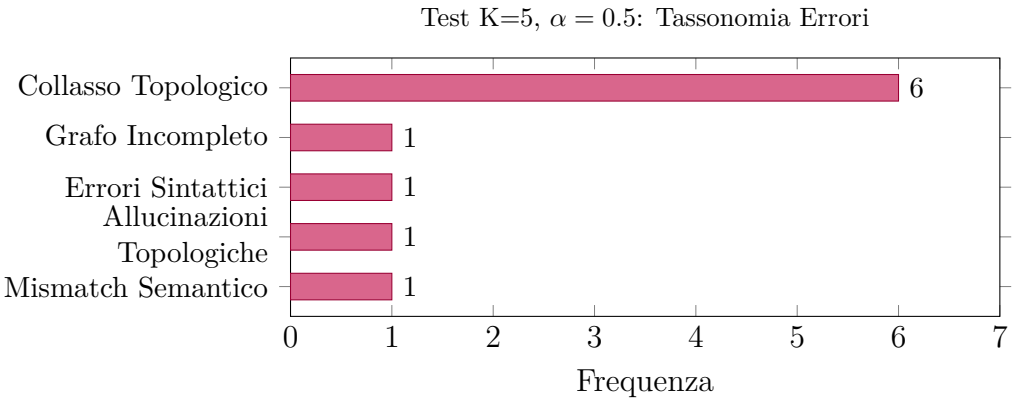

Figura 28: Test K=5, $\alpha = 0.5$: Frequenza aggregata dei fallimenti.

Come evidenziato in Figura 28, il "Collasso Topologico" si aggrava (6 occorrenze su 10 fallimenti totali), manifestandosi in modo sistematico su Q28. Codestral raggiunge 8/10 query corrette (il miglior risultato assoluto), GPT-4o mantiene 7/10 e LLaMA migliora a 5/10.

6.1.3 Test con parametri: K=3, $\alpha = 1.0$

Questa configurazione esplora l'effetto del retrieval puramente semantico, eliminando il contributo lessicale del TF-IDF.

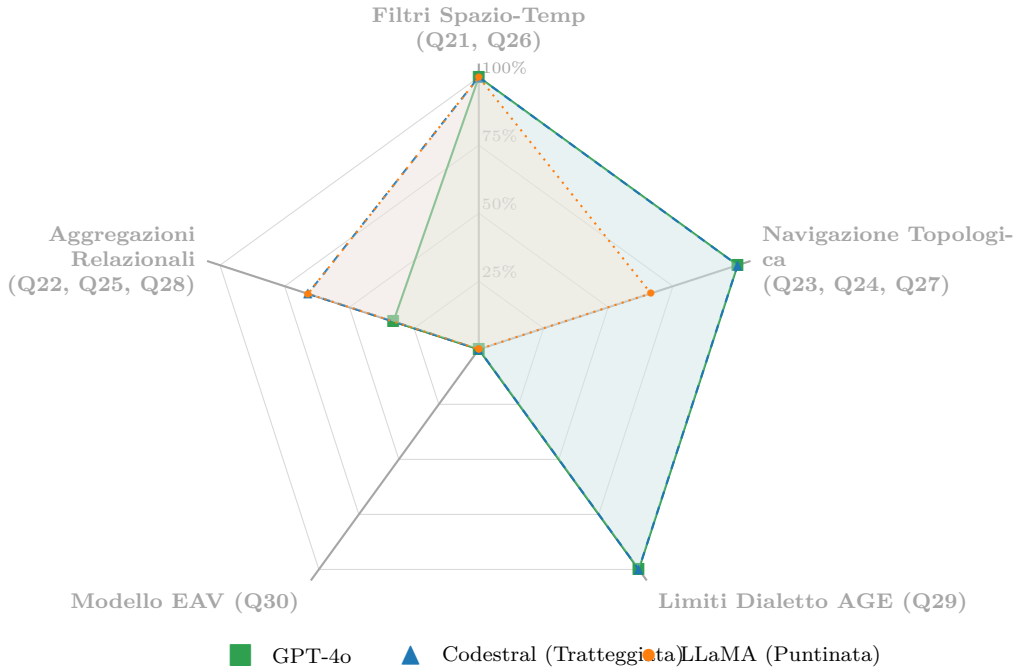

Figura 29: Test $K = 5, \alpha = 0.5$: Radar - Tasso di successo per categoria architetturale (basato su *recall*).

Come mostrato in Figura 26, l'esclusivo affidamento al retrieval semantico penalizza GPT-4o e LLaMA sul dialetto AGE mentre Codestral aggira il limite utilizzando un `OPTIONAL MATCH`.

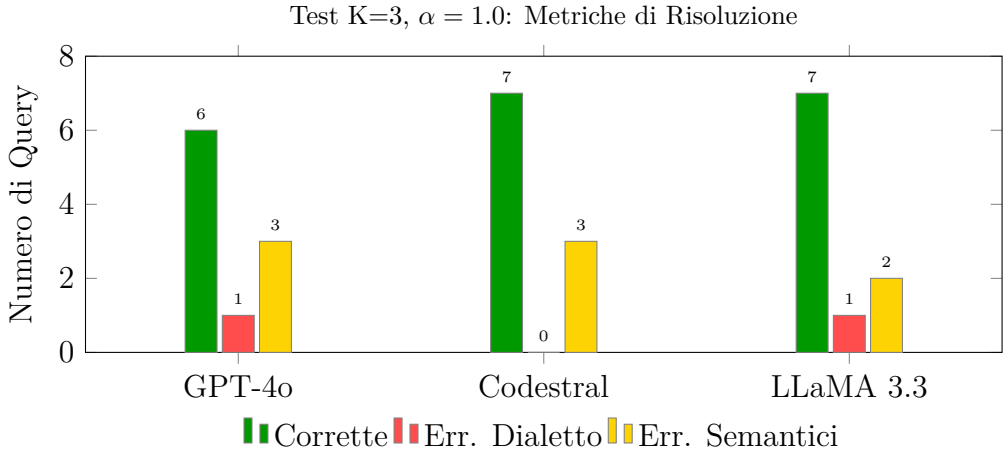

Figura 30: Test K=3, $\alpha = 1.0$: Distribuzione degli esiti (basata su *precisione* esatta).

Da Figura 30 emerge che le alternative Open Weights superano GPT-4o, di fatto Il modello di OpenAI subisce una lieve flessione a causa di un mismatch semantico su Q22.

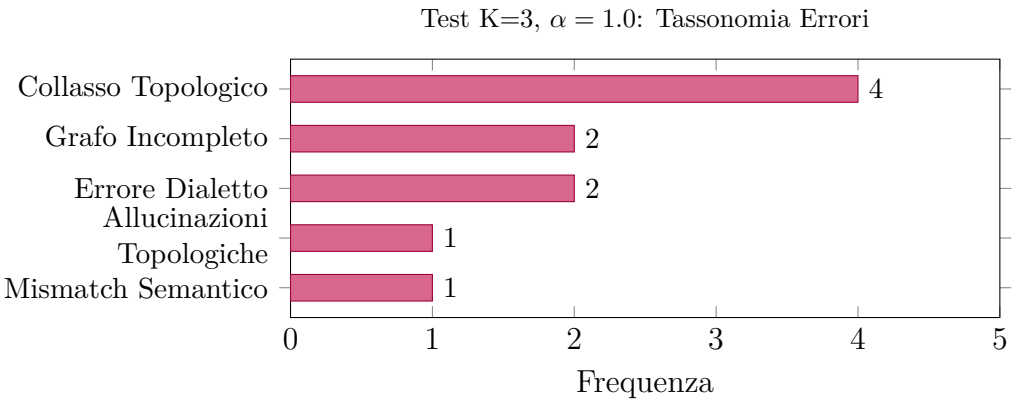

Figura 31: Test K=3, $\alpha = 1.0$: Frequenza aggregata dei fallimenti.

Come evidenziato in Figura 31, eliminando il TF-IDF, GPT-4o e LLaMA subiscono un crollo sul dialetto AGE (Q29), dove generano il costrutto `NOT EXISTS` non supportato da Apache AGE, mentre Codestral, aggira il problema con un `OPTIONAL MATCH`. I modelli open-source superano GPT-4o, ma il crash di GPT-4o è dovuto a un errore di dialetto che in configurazione ibrida

non si sarebbe verificato e inoltre su Q30: GPT-4o e LLaMA estraggono 69 nodi ma solo 10 archi su 61 attesi.

6.2 Confronto e discussione dei parametri di retrieval

L'analisi dei tre test permette di valutare l'impatto effettivo dei parametri K e α sulle performance del framework.

6.2.1 Impatto del numero di esempi (K)

La Tabella 3 mostra le performance aggregate di tutti i modelli al variare del numero di esempi, a parità di bilanciamento semantico-lessicale ($\alpha = 0.5$).

K	Query Corrette	Crash DB	Tasso Successo
3	17/30 (56%)	2	56%
5	**20/30 (66%)**	**1**	**66%**

Tabella 3: Confronto delle performance al variare di K ($\alpha = 0.5$).

L'aumento del numero di esempi da 3 a 5 produce un miglioramento del 10% nel tasso di successo complessivo e una riduzione degli errori fatali. Questo suggerisce che un contesto di retrieval più ampio fornisce ai modelli un vocabolario strutturale sufficiente per generalizzare correttamente, combinando pattern sintattici visti in esempi separati per risolvere query inedite. Tuttavia, come suggerito dalla letteratura [11], un ulteriore incremento a K=7 o K=10 avrebbe rischiato di introdurre rumore e saturare la finestra di contesto, fenomeno noto come over-prompting. Per questo motivo, K=5 è stato identificato come il valore ottimale.

6.2.2 Impatto del bilanciamento semantico-lessicale (α)

La Tabella 4 mostra le performance aggregate di tutti i modelli al variare del bilanciamento, a parità di numero di esempi (K=3).

α	**Query Corrette**	**Crash DB**	**Tasso Successo**
0.5	17/30 (56%)	2	56%
1.0	20/30 (66%)	2	66%

Tabella 4: Confronto delle performance al variare di α (K=3).

I dati mostrano che l'esclusione del TF-IDF ($\alpha = 1.0$) produce un apparente miglioramento del tasso di successo aggregato, che passa dal 56% al 66%. Tuttavia, questa apparente crescita nasconde un problema strutturale: l'assenza di base lessicale porta GPT-4o e LLaMA a generare il costrutto `NOT EXISTS` non supportato da Apache AGE sulla query Q29, causando crash fatali. Nella configurazione ibrida ($\alpha = 0.5$), questo errore non si verifica mai.

Dal punto di vista dell'utente finale, un crash (nessuna risposta) è molto più grave di un errore semantico (risposta parziale ma comunque utile). Il TF-IDF funge quindi da *guardrail sintattico*: garantisce che le query generate siano almeno eseguibili, sacrificando un apparente miglioramento quantitativo in favore della robustezza del sistema. Per questo motivo, la configurazione ibrida ($\alpha = 0.5$) è stata considerata più affidabile e adottata come standard, nonostante un tasso di successo completo inferiore.

6.2.3 Evoluzione delle performance per modello

La Tabella 5 mostra l'evoluzione delle query corrette per ciascun modello al variare dei test.

Modello	**Test 1** (K=3, a=0.5)	**Test 2** (K=5, a=0.5)	**Test 3** (K=3, a=1.0)
GPT-4o	7/10	7/10	6/10
Codestral-2501	7/10	8/10	7/10
LLaMA-3.3-70B	3/10	5/10	7/10

Tabella 5: Evoluzione delle query corrette al variare dei test.

Dalla tabella emergono tre dinamiche principali. L'estensione del contesto (Test 1 $\rightarrow$ Test 2) avvantaggia i modelli che con pochi esempi faticano a mantenere coerenza sintattica, come Codestral (7$\rightarrow$8) e soprattutto LLaMA (3$\rightarrow$5). GPT-4o rimane invece stabile (7$\rightarrow$7), suggerendo che il suo addestramento già fornisce una solida base anche con contesto limitato.

Il passaggio al retrieval puramente semantico (Test 3) produce invece effetti divergenti: GPT-4o peggiora (7$\rightarrow$6) a causa di un errore di dialetto su Q29, mentre LLaMA compie un balzo significativo (3$\rightarrow$7). Questo comportamento suggerisce che i modelli open-source traggono maggiore beneficio dagli esempi semantici quando l'ancoraggio lessicale viene rimosso.

Infine, il Collasso Topologico si conferma l'errore più frequente in tutti i test (5, 6 e 4 occorrenze). La sua persistenza anche nella configurazione ottimale indica che non è un limite del singolo modello, ma una difficoltà intrinseca del task: preservare le relazioni del grafo quando la domanda richiede calcoli SQL complessi seguiti da proiezioni a grafo.

6.3 Riepilogo e prospettive

I risultati presentati in questo capitolo confermano la validità del framework. La configurazione ottimale (K=5, $\alpha = 0.5$) consente a Codestral-2501 di raggiungere l'80% di query corrette con zero errori sintattici, dimostrando che il sistema è in grado di supportare efficacemente la generazione di query ibride. GPT-4o segue con il 70%, mentre LLaMA-3.3-70B ottiene il 50%, evidenziando come la dimensione del modello non sia un predittore affidabile delle performance in questo task specifico.

L'analisi comparativa delle tre configurazioni ha rivelato il ruolo cruciale del retrieval ibrido. L'esclusione del TF-IDF ($\alpha = 1.0$), sebbene produca un apparente miglioramento del tasso di successo aggregato (dal 56% al 66%), introduce errori di dialetto fatali su Q29 che nella configurazione ibrida non si verificano. Il TF-IDF agisce quindi come *guardrail sintattico*, garantendo che le query generate siano almeno eseguibili, a scapito di un tasso di successo quantitativo inferiore ma a favore della robustezza complessiva del sistema.

La triplice analisi (Radar, Barre, Tassonomia) ha isolato il Collasso Topologico come il principale ostacolo residuo. Mentre gli errori sintattici vengono completamente eliminati nella configurazione ottimale, e quelli semantici drasticamente ridotti, la difficoltà nel preservare le relazioni del grafo persiste sistematicamente su query di aggregazione complesse come Q28. Questo suggerisce che il limite attuale non risiede nei modelli LLM, ma nella capacità del prompt di comunicare l'importanza di restituire il path completo quando la domanda richiede un grafo.

La resilienza dimostrata da Codestral-2501, in particolare sulle aggregazioni relazionali e sul dialetto AGE, suggerisce che i modelli open-source specializzati per il codice possono rappresentare un'alternativa valida e più accessibile rispetto ai modelli proprietari. Questi temi saranno approfonditi nel Capitolo 7, dove verranno discusse le implicazioni dei risultati e le possibili direzioni per futuri sviluppi.

7 Conclusioni e Sviluppi Futuri

Il presente lavoro di tesi ha esplorato il potenziale dei Large Language Models come interfaccia in linguaggio naturale per l'interrogazione di sistemi multistore basati su PostgreSQL, integrando Apache AGE per la gestione del grafo e TimescaleDB per le serie temporali. L'obiettivo principale era valutare la capacità dei modelli di tradurre richieste espresse in linguaggio naturale in query ibride Cypher+SQL, garantendone la correttezza sintattica e la coerenza semantica rispetto alle sorgenti dati eterogenee.

A tal fine, è stato progettato e sviluppato un framework automatizzato end-to-end, il cui codice sorgente è disponibile su GitHub [15]. Il framework supporta l'intero processo di interrogazione: dalla costruzione e ottimizzazione dei prompt, alla generazione delle query, fino alla loro esecuzione e valutazione. La validazione sperimentale è stata condotta su un benchmark di 10 interrogazioni inedite (Q21-Q30), rappresentative di scenari applicativi concreti nel dominio dell'agricoltura di precisione.

7.1 Considerazioni conclusive

I risultati sperimentali confermano la validità dell'architettura end-to-end proposta con una configurazione ottimale individuata in K=5, $\alpha = 0.5$, basata sul Dynamic Few-Shot Prompting combinato a un Information Retrieval ibrido, ha permesso di raggiungere prestazioni elevate mantenendo un system prompt generico. Questo approccio garantisce un'ampia applicabilità sul dominio dell'agricoltura di precisione, evitando l'overfitting sul set di test.

In questa configurazione, il modello open-weight Codestral-2501 ha registrato le prestazioni migliori, risolvendo l'80% delle interrogazioni inedite con zero eccezioni sintattiche, lo segue GPT-4o con il 70%, mentre LLaMA-3.3-70B, nonostante le dimensioni maggiori, si ferma al 50%, principalmente a causa di problemi di aderenza al formato. Questo dato è particolarmente significativo: dimostra che i modelli open-source specializzati per la generazione di codice possono rappresentare una valida alternativa ai modelli proprietari generalisti, rendendo l'adozione di interfacce interrogative intelligenti economicamente e infrastrutturalmente sostenibile.

L'analisi ha inoltre evidenziato il ruolo cruciale dell'ancoraggio lessicale

introdotto dal TF-IDF, infatti l'esclusione di questa componente ($\alpha = 1.0$) produce un apparente miglioramento del tasso di successo aggregato, ma introduce errori di dialetto fatali (Q29) che nella configurazione ibrida non si verificano. Il TF-IDF garantisce quindi che le query generate siano almeno eseguibili, privilegiando la robustezza del sistema rispetto a un mero incremento quantitativo delle query corrette.

7.2 Limitazioni

La sperimentazione ha evidenziato alcune limitazioni che delineano l'attuale perimetro di efficacia del framework. La più significativa è il collasso topologico, l'errore più frequente in tutte le configurazioni che si manifesta sistematicamente su Q28: i modelli estraggono correttamente i nodi tramite SQL ma falliscono nel ricostruire i path Cypher, producendo nodi isolati e zero archi. La persistenza di questo problema anche nella configurazione ottimale suggerisce che il limite non risiede nei singoli modelli ma nella capacità del prompt di comunicare l'importanza di preservare le relazioni del grafo. A ciò si aggiunge la dimensione ridotta del set di validazione: sebbene le 10 interrogazioni di test siano state progettate per coprire diverse tipologie di complessità, un benchmark più esteso sarebbe necessario per consolidare la significatività statistica delle metriche rilevate. Completa il quadro la specificità del dominio: il Ground Truth è attualmente calibrato sulla semantica e sulla topologia hardware dell'agricoltura di precisione, quindi l'estensione ad altri domini richiederebbe di ricostruire l'intera knowledge base degli esempi dimostrativi.

7.3 Sviluppi futuri

Le criticità evidenziate suggeriscono alcune direzioni per migliorare il sistema. La priorità è rafforzare il prompt per mitigare il collasso topologico, aggiungendo una regola che imponga al modello di restituire sempre il path quando la domanda richiede un grafo, poiché questo rimane l'ostacolo principale emerso dalla sperimentazione. Parallelamente, si può estendere il dataset di few-shot con esempi che mostrano come gestire il collasso topologico e come usare INTERSECT per il modello EAV, ampliando anche

il set di test per una valutazione più robusta, inoltre è opportuno testare modelli più recenti come DeepSeek-V3 per confrontare le performance con GPT-4o e Codestral-2501. Un layer di post-processing potrebbe correggere automaticamente errori di dialetto comuni, come l'uso di NOT EXISTS nella forma compatta, riducendo ulteriormente i crash. Infine, il feedback umano sui pattern di errore identificati potrebbe essere sfruttato per migliorare iterativamente il prompt, rendendo il sistema sempre più affidabile.

In definitiva, l'automazione dell'interrogazione su sistemi multistore tramite LLM si conferma una traiettoria di ricerca promettente, capace di coniugare l'espressività semantica dell'intelligenza artificiale con il rigore formale delle moderne architetture di archiviazione dati. I risultati ottenuti, con Codestral-2501 che raggiunge l'80% di query corrette nella configurazione ottimale, confermano che l'approccio è non solo teoricamente valido ma anche praticamente applicabile, contribuendo a ridurre il divario tra la complessità tecnica dei sistemi multistore e le esigenze degli utenti finali.

Riferimenti bibliografici

[1] Bechar, A. et al. *The IoT and AI in Agriculture: The Time Is Now A Systematic Review*. PMC / MDPI, 2025. https://pmc.ncbi.nlm.nih.gov/articles/PMC12196926/

[2] Vogt, M., Stiemer, A., & Schuldt, H. *ICARUS: Towards a Multistore Database System*. In *Proceedings of the 2017 IEEE International Conference on Big Data (BigData '17)*, pp. 24902499. IEEE, 2017. https://doi.org/10.1109/BigData.2017.8258207

[3] Codd, E. F. *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, Vol. 13, No. 6, pp. 377387, 1970. https://doi.org/10.1145/362384.362685

[4] Pasini, M. *I database NoSQL* [Dispense del corso]. FITSTIC, Università di Bologna, 2025. https://github.com/ManuelePasini/FITSTIC_NoSQL_2025/blob/main/slides/01%20-%20T1%20-%20I%20database%20NoSQL.pdf

[5] The openCypher Project. *openCypher Specification*. https://www.opencypher.org/, 2015.

[6] Wiese, L. *Polyglot Database Architectures = Polyglot Challenges*. In *Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB*, Trier, Germany, October 7-9, 2015, CEUR Workshop Proceedings, Vol. 1458, pp. 422-426. http://ceur-ws.org/Vol-1458/H05_CRC85_Wiese.pdf

[7] Lu, J., Holubová, I., & Cautis, B. *Multi-model Databases and Tightly Integrated Polystores: Current Practices, Comparisons, and Open Challenges*. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*, pp. 22892290. ACM, 2018. https://doi.org/10.1145/3269206.3274269

[8] Stonebraker, M., & Rowe, L. A. *The Design of Postgres*. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*, pp. 340355. ACM, 1986. https://doi.org/10.1145/16894.16888

[9] Apache Software Foundation. *Apache AGE Graph Database Extension for PostgreSQL*. https://github.com/apache/age/blob/master/README.md, 2022.

[10] Timescale Inc. *TimescaleDB: Time-series data at scale*. https://github.com/timescale/timescaledb, 2017.

[11] *The Few-shot Dilemma: Over-prompting Large Language Models.*. https://arxiv.org/abs/2509.13196

[12] *Prompting Strategies and In-Context Learning*. https://www.techrxiv.org/doi/full/10.36227/techrxiv.176784493.36112281/v1

[13] *Azure AI Inference SDK for Python*. https://github.com/MicrosoftDocs/azure-docs-sdk-python/blob/main/docs-ref-services/preview/ai-inference-readme.md.

[14] *Cohere Embed v3 Multilingual – Documentation*. https://cohere.com/blog/introducing-embed-v3 https://docs.cohere.com/docs/multimodal-embeddings.

[15] *Analisis-llm*. https://github.com/MattiaRocchi/Analisi-LLM/tree/main/Tesi

8 Ringraziamenti

Giunto al termine di questo percorso, vorrei rivolgere un pensiero di profonda gratitudine a tutte le persone che hanno reso possibile questo traguardo e hanno reso speciali questi tre anni.

Il primo ringraziamento va al mio relatore, professor Golfarelli Matteo e al mio correlatore, dott. Manuele Pasini, per la disponibilità e la fiducia che mi hanno dimostrato sin dal nostro primo incontro. La loro guida, i loro insegnamenti e consigli sono stati fondamentali per la realizzazione di questo elaborato.

Un ringraziamento dal profondo del cuore va ai miei genitori, con i quali ho condiviso ogni momento di questo viaggio: le gioie, le difficoltà, le tristezze e i successi. Grazie per esserci sempre stati, per avermi donato tutto ciò di cui avevo e ho bisogno e per avermi insegnato che ogni traguardo deve essere vissuto con serenità e senza rimpianti.

Ringrazio i miei compagni di corso, *"la mia famiglia acquisita"*, per la loro costante presenza e per aver reso questi anni memorabili e magici. Alcuni di loro li conosco da molto tempo, da ben prima dell'università, e il loro affetto è un pilastro su cui so di poter sempre contare; altri sono amicizie nate tra queste mura e diventate ugualmente importanti non solo durante le lezioni, ma anche nella mia vita quotidiana, grazie per esserci sempre! Ci sono state anche amicizie che, per varie ragioni, si sono perse nel tempo, ma il ricordo di quel primo anno di studi e del legame che avevamo creato rimarrà per sempre impresso nella mia mente.

Un pensiero speciale va a tutti coloro che, fanno parte della mia famiglia: quelli più stretti, che mi hanno visto crescere e mi hanno sempre spronato ad essere fiero delle mie scelte, e quegli amici di famiglia con i quali ho condiviso avventure e ricordi indelebili. Ai miei nonni, anche se alcuni possono solo guardarmi da lassù, dico GRAZIE! Grazie per l'amore che mi avete donato, per avermi cresciuto insieme ai miei magnifici genitori, per quelle telefonate fatte solo per sentire come stavo, siete stati e sarete sempre un punto di riferimento fondamentale per me. Alle mie due fantastiche zie e al mio piccolo cuginetto, grazie per la vostra presenza discreta ma costante, per i vostri consigli e per avermi sempre trasmesso calore e affetto in ogni momento della mia crescita.

Ringrazio i miei compagni di vita, quei ragazzi che hanno condiviso con me l'infanzia e che, pur essendo ormai fuori dal contesto universitario, conti-

nuano a rallegrarmi le giornate. So che, qualunque cosa accada, potrò sempre contare su di loro. Allo stesso modo, un grazie va a tutti gli amici che, anche se da poco, sono ormai entrati nella mia vita, non so bene come o perché, ma dopo aver visto anche le mie stranezze, hanno deciso di restare. Spero con tutto me stesso che questo non cambi, perché ci sono persone che desidero fortemente avere accanto anche in futuro.

Infine, un ringraziamento va al mio fratellino. Ogni volta che lo chiamo così, la gente sgrana gli occhi e mi chiede "Ma tu da quando hai un fratello?". Io vorrei rispondere "Da sempre!", e non sarebbe una bugia. Lui è parte della mia vita da prima che io ne abbia memoria. È un punto di riferimento, un luogo di salvezza, quel singolo chiodo che può reggere l'intero castello. Ti ringrazio per esistere, per esserci. Sapere che, nella situazione peggiore, saresti disposto a tutto pur di esserci per me, mi dà una forza immensa. Grazie.