UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Magistrale in
Ingegneria Informatica e dell'Automazione

Un sistema di query answering per un Semantic Data Lake

A query aswering system for a Semantic Data Lake

Relatore:
Prof. Potena Domenico

Correlatore:
Prof. Storti Emanuele

Laureanda:
Rossetti Cristina

Anno Accademico 2022/2023

Indice

Capitolo 1

Introduzione

Il mondo dei Big Data pone in campo diverse sfide, a causa delle difficoltà nella gestione di grandi quantità di dati eterogenei e con diversi formati. I dati devono essere memorizzati ed elaborati in maniera efficace, al fine di assicurare analisi di qualità e di valore per le organizzazioni. É pertanto fondamentale assicurare una buona governance dei dati, con una fase di pre-processing che garantisca dati puliti e di qualità e che identifichi le caratteristiche delle sorgenti dati, come la provenienza, il tipo di dati, e così via.
Nel tempo sono state sviluppate diverse architetture per la memorizzazione, gestione, elaborazione e analisi dei dati. Tra queste, una delle più importanti è quella dei Data Lake. Un Data Lake consente di memorizzare tutti i dati provenienti dalle diverse sorgenti disponibili in una repository, nella quale ci deve essere un'importante gestione dei metadati. La repository costituisce un insieme di sorgenti dati eterogenee, di diversa provenienza, con tipi di dato e formati differenti, la cui elaborazione può essere demandata ad una fase successiva, nel momento in cui l'utente ha bisogno di un certo insieme di dati per effettuare analisi specifiche. Un Data Lake costituisce, quindi, uno strumento potente per gestire grandi quantità di dati. L'architettura può essere estesa mediante tecnologie semantiche, diventando così un *Semantic Data Lake*, come verrà descritto più dettagliatamente nel corso di questo elaborato.
Ad ogni modo, all'interno di un Data Lake ci possono essere diverse funzionalità per consentire all'utente una maggior facilità d'uso e di elaborazione dei dati memorizzati. Un'importante funzione è la possibilità di effettuare delle query per poter ricavare, secondo le informazioni richieste, determinate sorgenti. Prima di effettuare delle interrogazioni, tuttavia, è necessario che le sorgenti siano integrate, al fine di ottenere una visione omogenea e unificata dei dati. L'integrazione comprende la risoluzione delle eterogeneità e la creazione di mapping tra le sorgenti e uno schema globale di riferimento. Tuttavia, i mapping vengono già generati in fase di caricamento di una sorgente, per cui le sorgenti dati memorizzate nel Semantic Data Lake esistente vengono considerate integrate.
A questo punto, il sistema di query answering implementato si occupa di gestire

la complessità intrinseca dell'esecuzione di query nel contesto dei Data Lake. Infatti, cercare dati in una repository composta da molte sorgenti non è semplice, considerando che, molto spesso, non si sa a priori in quali sorgenti cercare le informazioni richieste. Inoltre, potrebbe accadere che i dati richiesti siano memorizzati in più sorgenti dati, generando la necessità di effettuare molteplici join.
L'esecuzione di join multipli nel contesto di un Data Lake può non essere efficiente. Si vuole proporre, pertanto, uno strumento che stimi la dimensione finale di una potenziale integrazione tra le sorgenti dati trovate, fornendo contestualmente un ranking che ordini in maniera decrescente le combinazioni di sorgenti dati le quali, messe in join, restituiscono un maggior numero di informazioni. Ciò consente all'utente di avere una panoramica dei join migliori senza, di fatto, eseguirli. Sarà poi a discrezione dell'utente scegliere le combinazioni d'interesse e procedere con l'effettiva esecuzione.

L'obiettivo di questo elaborato è di illustrare lo sviluppo di un sistema di *query answering* per un Semantic Data Lake. L'architettura del Semantic Data Lake, proposta da Diamantini et al. [1] , utilizza le tecnologie semantiche, come ontologie e Knowledge Graph, al fine di fornire uno schema di accesso globale ai dati e di rappresentare la semantica delle possibili entità contenute nelle sorgenti dati (dimensioni, livelli, indicatori, ...) assieme alle loro relazioni. Come verrà mostrato nei prossimi capitoli, l'ontologia KPIOnto è stata implementata allo scopo di fornire un vocabolario di termini utili alla rappresentazione e concettualizzazione degli elementi presenti nelle sorgenti dati. Il Knowledge Graph, invece, sfruttando l'ontologia, rappresenta le istanze effettivamente presenti all'interno del Semantic Data Lake con un modello *graph-based*, dove i nodi costituiscono le diverse entità, mentre gli archi le possibili relazioni tra essi (sempre indicate con terminologie fornite dall'ontologia). Questo strato di conoscenza aggiunge una semantica all'architettura del Data Lake, che consente di uniformare le diverse sorgenti eterogenee memorizzate e permette di navigare tra i concetti del Knowledge Graph, nonché di effettuare ragionamenti logici.
Nell'architettura esistente è presente anche uno strato che gestisce i metadati delle sorgenti. I metadati vengono rappresentati anch'essi con un modello graph-based, dove i singoli nodi rappresentano gli elementi dello schema di una sorgente mentre gli archi specificano le proprietà che legano le entità presenti.
Tra i concetti della conoscenza del Knowledge Graph e i metadati delle sorgenti vengono memorizzati dei mapping. Se un metadato è collegato con un mapping ad un concetto del Knowledge Graph, significa che l'entità rappresentata dal metadato (es. una colonna di un dataset) ha una certa similarità di significato (è semanticamente simile) con quel concetto.
In questo elaborato si estende il Semantic Data Lake esistente con un sistema di query answering, il cui obiettivo principale è rispondere alla query di un utente restituendo le sorgenti dati contenenti le informazioni richieste. L'utente ha la possibilità di specificare indicatori e livelli dimensionali desiderati. A questo

punto, sono due le attività principali svolte dal sistema di query answering: (1) *source discovery*, ovvero il processo di ricerca delle sorgenti dati guidato dalla query dell'utente e il cui output sarà l'insieme di tutte le sorgenti in grado di restituire le informazioni richieste; (2) calcolo del *joinability index*, ovvero di un indice in grado di stimare le cardinalità finali dei join tra le sorgenti dati trovate.

Di seguito si descrive la struttura di questo elaborato. Nel capitolo 2 viene mostrato lo stato dell'arte delle principali tecnologie adottate. In particolare, nella sezione 2.1 si introducono i Data Lake insieme alle architetture principali; nella sezione 2.2 si descrivono le principali tecnologie semantiche, con particolare attenzione ai Linked Data, le ontologie e i Knowledge Graph, e le loro applicazioni ai sistemi di gestione dei dati; nella sezione 2.3 c'è un breve sommario delle implementazioni in letteratura del query answering; nella sezione 2.4 si introduce il problema della stima della joinability tra sorgenti dati.
Nel capitolo 3 si descrive in dettaglio l'architettura del Semantic Data Lake esistente su cui è basato il lavoro di questo elaborato.
Nel capitolo 4 viene mostrata l'implementazione del sistema di query answering, dalla fase di scoperta delle sorgenti dati richieste dall'utente (*source discovery*) fino al calcolo del *joinability index* per la stima del grado di similarità tra le sorgenti trovate al fine di approssimare le cardinalità dei join.
Infine, nel capitolo 5 vengono illustrati gli esperimenti eseguiti e i risultati ottenuti. In dettaglio, viene descritta la metodologia per l'esecuzione degli esperimenti, e vengono poi riportati i risultati inerenti ai tempi di esecuzione per la fase di source discovery, per il calcolo del joinability index e per il calcolo dei join (per effettuare il confronto con i risultati ottenuti dal joinability index); e i risultati inerenti alla qualità del joinability index, confrontando le cardinalità ottenute con l'indice con quelle reali ottenute dai join.

Capitolo 2

Stato dell'Arte

In questo capitolo viene mostrato lo stato dell'arte riguardante l'utilizzo dei Data Lake per la memorizzazione dei Big Data, le tecnologie semantiche e le loro applicazioni nei sistemi di gestione dati con particolare attenzione ai Knowledge Graph, tecniche di *dataset discovery* e *query answering* e infine diversi metodi per la scoperta di *joinable dataset*.

2.1 Data Lake

La continua evoluzione tecnologica ha consentito, nel corso degli anni, di poter produrre, memorizzare ed elaborare quantità di dati sempre più grandi, fino ad arrivare al concetto di *Big Data*. Con l'avvento anche del Web, i dati prodotti sono cresciuti a dismisura, generando quindi la necessità di sviluppare nuove tecnologie per la gestione di questi dati. Oltre alla capacità di memorizzare enormi quantità di dati, c'è anche il bisogno di strumenti per la pulizia, l'organizzazione e l'elaborazione degli stessi. Questi dati, infatti, non solo sono tanti ma vengono anche generati con velocità elevatissime (ad es.: flussi di dati provenienti da sensori), portando ad accumulare molti dati in breve tempo. Inoltre, i dati sono eterogenei e possono avere formati diversi tra loro. É possibile avere dati strutturati (es. tabelle relazionali), ma anche dati semi-strutturati e non strutturati i quali sono più difficili da gestire. Oltre tutto, è necessario assicurare la qualità dei dati a disposizione per far sì che questi possano portare valore alle organizzazioni.
Si rende quindi necessario lo sviluppo di architetture di memorizzazione, gestione, organizzazione ed elaborazione dei Big Data.
Negli ultimi anni, è stata proposta l'architettura dei *Data Lake* per la gestione dei dati nel contesto dei Big Data. Varie definizioni sono state proposte, ma in generale si può dire che un Data Lake è una soluzione per la memorizzazione nel loro formato nativo di dati grezzi con strutture eterogenee provenienti da diverse sorgenti (locali o esterne all'organizzazione). Un Data Lake consente l'elaborazio-

ne di questi dati da differenti prospettive e da parte di diverse tipologie di utenti specializzati (data scientist, data analysts, e così via) per l'analisi statistica, la Business Intelligence, il Machine Learning ecc. [2] Nei Data Lake è possibile memorizzare i dati così come arrivano dalle diverse sorgenti a disposizione dell'organizzazione. Non c'è una pre-elaborazione di questi dati prima della loro memorizzazione. Dato che in molti casi non è possibile sapere a priori se i dati provenienti da molteplici sorgenti saranno utili agli scopi dell'organizzazione, si presuppone che tutti i dati possono essere potenzialmente di valore per analisi future, ed è per questo che l'architettura dei Data Lake è pensata per memorizzare tutti i dati.
Un'altra soluzione utilizzata ampiamente ancora tutt'oggi per la gestione e la memorizzazione di Big Data è quella dei *Data Warehouse*, ovvero uno storage integrato e storico progettato in maniera specifica per l'analisi dei dati [3], tipicamente mediante tecnologia OLAP. I dati sono strutturati con il modello multidimensionale che si basa su tre concetti principali: *fatto*, ovvero l'informazione su cui centrare l'analisi; *misura*, ovvero la proprietà atomica di un fatto; *dimensione*, ovvero la prospettiva lungo cui effettuare l'analisi. Nonostante i Data Warehouse siano uno strumento potente per i dati strutturati, non sono tuttavia adatti per i dati semi-strutturati e non strutturati . I Data Lake, invece, sono sistemi di memorizzazione, di analisi e di gestione di grandi quantità di dati di qualsiasi formato [3].
Inoltre, mentre nei Data Warehouse si utilizza il processo di ETL (Extract, Transform, Load) dove i dati estratti vengono prima elaborati, puliti ed organizzati e solo successivamente caricati, nel contesto dei Data Lake, si sfrutta l'ELT (Extract, Load, Transform): i dati vengono estratti dalle diverse sorgenti disponibili, caricati e quindi memorizzati e solo successivamente, se necessario, vengono elaborati per la loro analisi.
Memorizzando dati di qualsiasi tipo e provenienti da qualsiasi sorgente, i Data Lake necessitano di una buona governance al fine di garantire la qualità dei dati ed evitare che si trasformi in un *Data Swamp*, ovvero una repository di dati praticamente inutilizzabile a causa di una mancata gestione ed organizzazione dei metadati. In particolare, dopo la memorizzazione, la semantica e la qualità dei dati sono sconosciute ed è possibile si perda la loro provenienza. Nel tempo è nata quindi l'esigenza di utilizzare e gestire in maniera efficace i metadati al fine di assicurare costantemente qualità e una corretta gestione dei dati [4].
I metadati possono essere classificati in tre tipologie principali [5] (Figura 2.1):

- Operazionali: rappresentano informazioni automaticamente generate durante la fase di processamento dei dati. Includono descrizioni riguardanti la posizione della sorgente, la dimensione del file, e così via.

- Business: definiti come un insieme di descrizioni che rendono i dati più comprensibili e aiutano a definire regole per il business.

- Tecnici: include informazioni sul formato dei dati e l'eventuale schema.

Diamantini et al. [5] sostengono come queste tipologie di metadati possono intersecarsi.

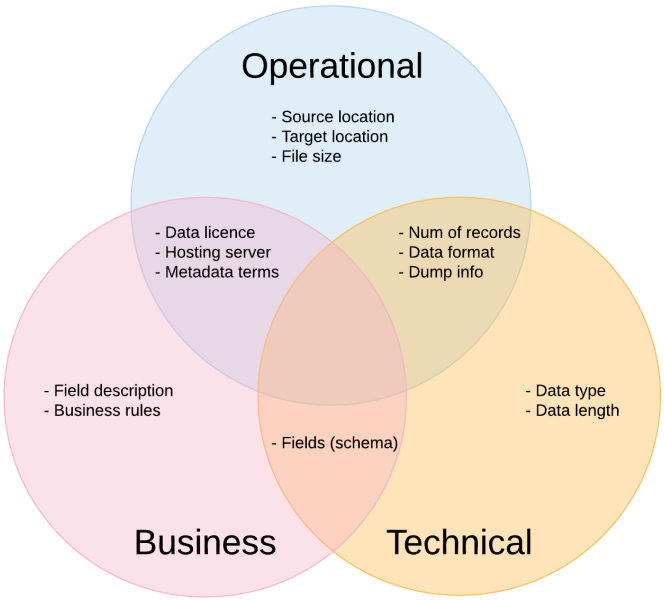

Figura 2.1: Tipologie di metadati (Diamantini et al. [5])

Nei Data Lake, i metadati possono essere modellati in diverse maniere. Uno degli approcci maggiormente adottati in letteratura è la modellazione mediante grafi [3] ed è come vengono modellati anche nel Semantic Data Lake su cui si basa questo elaborato.

In letteratura sono state proposte molte architetture per la strutturazione di un Data Lake. Una prima versione di un Data Lake consisteva nella memorizzazione dei dati grezzi nel loro formato nativo in un unico spazio (*mono-zone architecture*). [2]. Un'evoluzione consente di avere un'architettura *multi-zone*, che separa il ciclo di vita di un dataset in più fasi per il caricamento dei dati grezzi, il controllo della qualità, l'elaborazione, la scoperta dei dati e il loro utilizzo per successive analisi. La *transient loading zone* è usata tipicamente per il controllo della qualità dei dati che vengono caricati. La *raw data zone* gestisce i dati provenienti dalla zona precedente tipicamente in formato grezzo. La *trusted zone* viene usata per memorizzare i dati una volta che sono stati standardizzati e puliti. La *discovery sandbox* contiene dati provenienti dalla zona precedente ed è utilizzata dai data scientist per l'esplorazione dei dati stessi. La *consumption zone* viene acceduta dagli utenti dell'organizzazione mediante l'utilizzo di dashboard. Infine, la *governance zone* permette di gestire i metadati, la qualità dei

dati e la sicurezza [3].
Inmon [6] propone un'architettura che prevede invece l'utilizzo di *data pond*, dove ogni pond è uno specifico spazio di memorizzazione con un certo tipo di elaborazione dei dati. La prima zona, chiamata *Raw data pond*, contiene tutti i dati grezzi, che vengono poi trasformati e memorizzati, se possibile, nelle zone *analog data pond*, *application data pond*, o *textual data pond*. Nell'*analog data pond* i dati sono caratterizzati da un'alta frequenza di misurazione (sono memorizzati ad esempio dati provenienti da sensori IoT). Nell'*application data pond* i dati provengono da applicazioni software, e sono generalmente dati strutturati provenienti da DBMS relazionali. Nel *textual data pond* si gestiscono dati non strutturati testuali. Infine, l'*archivial data pond* memorizza dati che non sono usati correntemente, ma che potrebbero essere utili in futuro [3].
Hai et al. [4] propongono un'architettura *three-tier function-oriented* dove per ogni strato vengono eseguite delle funzioni. L'*ingestion tier* è responsabile nell'importazione dei dati dalle sorgenti. Le funzioni nello strato medio (*maintainance tier*) si occupano della gestione e l'organizzazione dei dati importati nello strato precedente e fungono da pre-elaborazione per successive query. Infine, l'*exploration tier* consente agli utenti di esplorare e accedere ai dati contenuti nel Data Lake.

In questo elaborato si sviluppa un approccio di query-answering su un *Semantic Data Lake*. L'architettura, proposta da Diamantini et al. [1], struttura un Data Lake utilizzando un modello graph-based per l'organizzazione dei metadati e un Knowledge Graph per la rappresentazione della conoscenza al fine di individuare gli indicatori, le loro formule matematiche e le gerarchie dimensionali.
L'utilizzo di tecnologie semantiche e modelli di dati a grafo ha consentito lo sviluppo di nuove architetture per i Data Lake. Nel paragrafo successivo 2.2 vengono descritte le principali tecnologie semantiche utilizzate nel contesto dei Data Lake.

2.2 Tecnologie Semantiche

Il World Wide Web ha cambiato completamente il modo con cui viene condivisa la conoscenza, consentendo a qualsiasi utente di pubblicare ed accedere alle informazioni in uno spazio globale. Il World Wide Web è evoluto nel tempo nel *Semantic Web*, detto anche *Web of Data*. Il Semantic Web è costituito da uno spazio globale interconnesso, che consente di integrare dati (i Linked Data) da diverse sorgenti e stabilire collegamenti tra loro, favorendo la scoperta e l'uso dei dati [7].
I Linked Data fanno uso di un meccanismo chiamato URI (*Uniform Resource Identifier*), il quale consente di identificare in maniera univoca una risorsa e l'accesso avviene mediante il protocollo HTTP.
Tim Berners-Lee [8] si riferisce ai Linked Data come un insieme di *best practi-*

ce per pubblicare e collegare dati strutturati sul Web. Queste pratiche vengono conosciute come *Linked Data Principles*, e sono le seguenti:

- Usa gli URI come nomi per le risorse
- Usa HTTP URI per consentire di dereferenziare gli URI
- Quando un utente ricerca un URI, fornisci informazioni utili usando degli standard (RDF, SPARQL)
- Includi collegamenti ad altri URI così da poter scoprire più cose

Lo standard RDF (Resource Description Framework) fornisce un modo flessibile di descrivere la conoscenza del mondo e come le entità del mondo sono connesse tra loro mediante relazioni. L'RDF fornisce un modello di dati globale *graph-based* per la rappresentazione e il collegamento dei dati. L'RDF Schema (RDFS) è un'ontologia estensione dello standard RDF utile per fornire un vocabolario per la strutturazione e rappresentazione delle entità e le loro relazioni sotto forma di grafi.
Lo standard SPARQL, invece, è un linguaggio di query usato ampiamente per interrogare i dati RDF [7].
In sostanza, il Semantic Web è visto come un'estensione del World Wide Web con informazioni comprensibili alla macchina, a differenza del Web che ha come principale target l'utente umano. Le informazioni per la macchina sono arricchite da metadati espressivi generati mediante ontologie, o almeno con un linguaggio formale con una semantica basata sulla logica che ammette il reasoning sui dati [9]. Hitzler [9] sostiene che il Semantic Web abbia attraversato tre fasi principali: (1) Ontologie, (2) Linked Data, ovvero un insieme di grafi RDF interconnessi (3) Knowledge Graph, ovvero un'evoluzione dei Linked Data soprattutto in ambito industriale.
Il termine Ontologia, nel contesto dell'informatica, definisce un insieme di primitive di rappresentazione per la modellazione della conoscenza di un certo dominio. Queste primitive possono essere classi, attributi e relazioni, e includono tipicamente anche una descrizione sulla semantica e i vincoli dei dati. Nel Semantic Web, le ontologie formali hanno il ruolo di definire logicamente e semanticamente le informazioni sulle risorse pubblicate e condivise [10]. Le ontologie, quindi, consentono di rappresentare in modo formale i concetti della conoscenza di un certo dominio assieme alle loro relazioni. Lo standard principale di riferimento è il *Web Ontology Language* (OWL) e RDFS, entrambi utili per la strutturazione dei grafi RDF.
Per fornire una definizione più dettagliata, sempre secondo Tim Berners-Lee et al. [11], i Linked Data si riferiscono ad un metodo per la pubblicazione di dati strutturati, così che possano essere interconnessi e divenire più utili mediante query semantiche, sulla base di HTTP, RDF e URI. I Linked Data, quindi, consentono la pubblicazione e la condivisione di dati sul Semantic Web in maniera

tale da rendere le informazioni comprensibili ed utilizzabili dalla macchina. Per fare questo, i Linked Data sono rappresentati mediante grafi RDF e le ontologie sono utilizzate per descrivere logicamente la semantica delle informazioni condivise. I Linked Data hanno l'obiettivo di rendere più semplice il processo di *data discovery* mediante la condivisione e l'integrazione dei dati, e consente anche di offrire meccanismi di query answering integrati. Grazie all'utilizzo di tecnologie come grafi RDF e URI, dati provenienti da sorgenti eterogenee vengono rappresentati in maniera univoca ed omogenea, così da facilitarne l'integrazione [12].
Un Knowledge Graph può essere definito come una rappresentazione strutturata di fatti, che consistono in entità, relazioni e descrizioni semantiche di un certo dominio. Quando un Knowledge Graph incorpora definizioni semantiche, può essere considerato come una base di conoscenza per interpretazioni e inferenze sui fatti. La conoscenza viene tipicamente rappresentata mediante triple RDF [13] . Ad oggi, i Knowledge Graph sono ampiamente usati in molti contesti, come viene mostrato in Figura 2.2, dove ci sono le diverse applicazioni possibili.

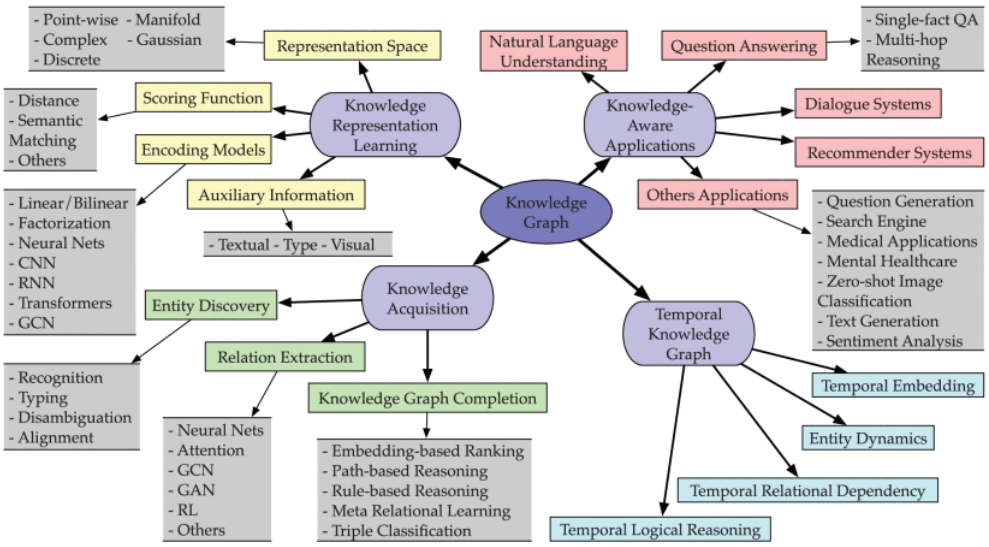

Figura 2.2: Applicazioni Knowledge Graph [13]

Come si vedrà successivamente, i Knowledge Graph sono ampiamente utilizzati nel contesto della gestione dei dati.

Il Semantic Web ha influenzato fortemente il contesto dei sistemi di gestione dei dati. Nel tempo le organizzazioni hanno iniziato ad utilizzare il modello di dati RDF nel proprio business, consentendo una migliore integrazione e scoperta semantic-based dei dati, come riportato da Hassanzadeh et al. [14]. Gli autori sostengono che l'introduzione della semantica in un sistema di gestione dei dati consente un più veloce processo di integrazione di nuove sorgenti, la generazione di mapping tra concetti, una migliore ricerca e identificazione delle sorgenti.

Ad esempio, le ontologie possono essere utilizzate per un sistema di gestione dei dati. Lenzerini [15] propone un sistema di gestione dati *ontology-based* (OBDM) che ha l'obiettivo di fornire un accesso unificato ai dati e di migliorare le attività di governance e integrazione dei dati. Il sistema OBDM è un'architettura a tre livelli che comprende l'ontologia, le sorgenti dati e i mapping tra i due. L'ontologia si propone come una rappresentazione esplicita del dominio di interesse dell'organizzazione. Sostanzialmente, lo schema globale di accesso ai dati è costituito dall'ontologia, vista come un vero e proprio modello concettuale che rappresenta la conoscenza di dominio.
Nel contesto dei Data Lake le tecnologie semantiche hanno consentito lo sviluppo dei *Semantic Data Lake*, definiti da Mami et al. [16] come un'estensione di un Data Lake con un middleware semantico che consente un accesso uniforme a sorgenti dati eterogenee. Gli autori propongono un'architettura, chiamata Squerall, che fa uso di tecniche di Semantic Web come ontologie, mapping e query SPARQL.
Nel contesto dei Data Lake, come già detto precedentemente, è fondamentale la gestione dei metadati. Anche i metadati possono essere modellati con l'ausilio delle tecnologie semantiche, in particolare con modelli *graph-based* [4].

Una tecnologia utilizzata per i Semantic Data Lake è quella relativa ai *Knowledge Graph*, che consente di aiutare l'integrazione, la gestione e l'estrazione di valori da diverse sorgenti di dati su larga scala [17]. Hogan et al. [17] sostengono che i grafi consentono di catturare differenti relazioni tra le entità di un dominio e permettono di posporre la definizione di uno schema, in modo tale che i dati possano evolvere in una maniera più flessibile. I linguaggi di interrogazione delle query supportano la navigazione tra le entità mediante le loro relazioni e le ontologie e le regole possono essere utilizzate per definire e ragionare sulle semantiche dei termini usate nel grafo.
Chessa et al. [18] propongono un Semantic Data Lake costituito da un'ontologia e un Knowledge Graph. L'ontologia utilizza delle classi per descrivere le entità del dominio e rappresenta le relazioni tra queste entità. Le sorgenti dati e l'ontologia vengono prese in input per la generazione di un Knowledge Graph utilizzando l'RDF Mapping Language (RML) per la modellazione del grafo mediante triple RDF.
L'insieme di triple RDF può costituire un grafo RDF ed essere utilizzato per la generazione di un Knowledge Graph. Una tripla RDF è costituita da soggetto, predicato e oggetto [7].
Fernandez et al. [19] propongono AURUM, un sistema per la costruzione, il mantenimento e l'interrogazione di un EKG (Enterprise Knowledge Graph) nel contesto dei Data Lake. L'EKG costruito è un iper-grafo i cui nodi rappresentano le colonne delle sorgenti dati e gli archi le relazioni tra i nodi. L'architettura consente un processo di *dataset discovery* e di interrogazione efficienti, grazie al supporto della semantica espressa mediante l'EKG.

L'architettura su cui viene implementato il sistema sviluppato in questo elaborato si basa sull'utilizzo di tecnologie semantiche graph-based sia per la costruzione dei metadati ma anche per la rappresentazione della conoscenza sui dati contenuti nelle sorgenti presenti nel Data Lake. Il Knowledge Graph incluso nell'architettura viene costruito con il supporto di un'ontologia chiamata KPIOnto, la quale fornisce un vocabolario per la descrizione semantica di dimensioni, livelli dimensionali, membri dei livelli, nonché indicatori e le formule matematiche per il loro calcolo.
Questo elaborato ha l'obiettivo di illustrare l'implementazione di un sistema di *query answering* nel contesto di un Semantic Data Lake. Di seguito vengono mostrati i principali contributi in letteratura rispetto a questa tematica.

2.3 Query Answering

L'obiettivo del *query answering* è quello di rispondere alla query di un utente posta, in questo particolare caso, su un sistema di gestione di dati. In letteratura sono state esplorate diverse tecniche di Query Answering.
Nei sistemi di gestione dati, in particolare nei Data Lake, la maggiore sfida è quella di supportare il query answering, che comprende la scoperta, l'estrazione, la pulizia e l'integrazione dei dati durante l'esecuzione della query su grandi collezioni di dati. Nella letteratura si parla di *query-driven discovery*, ovvero il processo di scoperta delle sorgenti dati guidato dalla query. In particolare, lo scopo principale è quello di trovare dataset simili alla query oppure dataset che possono essere integrati con la query [20].
Giacometti et al. [21] propongono l'estensione dei cosiddetti *recommender systems* nel contesto dei sistemi OLAP. I sistemi di raccomandazione hanno lo scopo di aiutare l'utente a navigare in grandi quantità di dati. L'idea di base degli autori è quella di calcolare la similarità tra la sequenza di query dell'utente corrente e le precedenti sequenze di query registrate dal server. L'approccio viene ripreso dalle tecniche utilizzate nella ricerca sul web e nell'e-commerce, dove si consigliano degli *items* sulla base delle ricerche precedenti di un utente.
Hai et al. [22] propongono un Data Lake chiamato *Constance* che implementa un linguaggio di query sottoinsieme di JSONiq con lo scopo di rendere più semplici le query a dati semi-strutturati come i JSON. Nel sistema ci sono delle annotazioni semantiche assimilabili a mapping global-as-view. L'utente ha anche la possibilità di esprimere la query nel linguaggio naturale.
Yuan et al. [23] propongono un framework per il query answering, chiamato *LakeAns*, per Data Lake. Uno schema globale relazionale consente l'interrogazione di sorgenti eterogenee multiple. Gli autori sostengono che il problema principale del query answering nei Data Lake sia la creazione di uno schema globale che rappresenti le associazioni tra le diverse sorgenti disponibili. Cogliere tali relazioni consente di estrarre concetti e classi con la stessa semantica o simile tra diverse

sorgenti eterogenee.
Fernandez et al. [19], con il framework AURUM discusso precedentemente, introducono un *source retrieval query language (SRQL)* con lo scopo di effettuare query sul Knowledge Graph (EKG) del loro Data Lake.

Successivamente, verrà discusso l'approccio *query-driven* adottato in questo lavoro. Si vuole anticipare che il framework di query answering implementato non ha soltanto lo scopo di scoprire le sorgenti dati che rispondono alla query dell'utente, ma anche di calcolare la *joinability* tra le sorgenti dati. Nella sezione successiva 2.4 si discuteranno gli approcci presenti nella letteratura per la scoperta di sorgenti dati che possono essere messe in join (*joinable dataset*), ovvero dataset con un certo grado di similarità che, combinati tra loro (tipicamente mediante join), forniscono il maggior numero di informazioni.

2.4 Scoperta di *joinable dataset*

Nei processi di *dataset discovery* e *query answering*, è importante valutare quali sorgenti possono essere combinate tra loro per diversi scopi. Eseguire il join tra i dataset a disposizione è un modo per unire informazioni di interesse, per arricchire i dati già in possesso, per eseguire l'integrazione dei dati. Nel contesto dei Data Lake, tuttavia, le sorgenti dati sono eterogenee, spesso hanno schemi diversi e si possono avere diverse terminologie per gli stessi concetti [24]. Inoltre, l'esecuzione di join nel contesto dei Data Lake può essere oneroso al livello computazionale e di occupazione in memoria.
Nella letteratura viene esplorata la possibilità di calcolare la similarità tra le sorgenti dati che possono essere messe in join.
Zhu et al. [25] parlano di *overlap set similarity search* nel contesto dei Data Lake, dove la misura di similarità è l'intersezione tra gli insiemi considerati. In particolare, gli autori introducono un framework chiamato JOSIE, dove gli insiemi di cui si calcola la similarità sono le colonne delle tabelle. Viene proposto un approccio top-k che fornisce scalabilità all'aumentare della dimensione dei dataset e ritorna i risultati migliori (top-k) senza la necessità che l'utente specifichi un threshold [4].
Dong et al. [24] propongono un framework chiamato PEXESO per la scoperta di *joinable dataset*, ovvero sorgenti dati con elevato grado di similarità. La joinability di una tabella è misurata dal numero di record corrispondenti nella colonna della query, che sono definiti utilizzando una funzione di distanza e un threshold. L'approccio viene utilizzato soprattutto per dati testuali.
Nel framework AURUM [19], il grafo EKG collega nodi (colonne delle tabelle) mediante relazioni che rappresentano la similarità degli insiemi cosiderati. In particolare, per la costruzione dell'EKG c'è prima una fase di *signature-building* dove un modulo profiler mantiene informazioni sulle colonne come i *content sketches*

mediante MinHash. La seconda fase costruisce le relazioni tra i nodi basandosi sui profili creati. L'etichetta di ogni arco dice quanto è forte la relazione tra i due nodi, ovvero quanto i due insiemi considerati sono simili, e questo viene calcolato mediante un algoritmo chiamato LSH Ensemble.
Un altro approccio chiamato Auctus [26] consente di supportare il processo di *data integration query*, ovvero la scoperta di sorgenti dati che possono essere concatenate o messe in join con un *query dataset*. Il framework proposto utilizza LAZO, ovvero un metodo per il calcolo del *set-overlap* basata su MinHash e LSH, al fine di creare un indice per attributi categorici.

Nell'architettura su cui viene implementato il framework di questo elaborato, la similarità tra le sorgenti dati presenti nel Semantic Data Lake è calcolata sia per la profilazione dei dataset ma anche in fase di esecuzione delle query per trovare le sorgenti il cui join restituisce maggiori informazioni. In particolare, in fase di interrogazione, viene combinato l'utilizzo del MinHash con l'LSH Ensemble, come verrà mostrato nei capitoli successivi.

Gli *sketches MinHash* sono definiti come strutture sommarie randomizzate di sottoinsiemi. Gli sketches codificano informazioni sulla cardinalità di un insieme e hanno la proprietà che sketches di insiemi simili sono simili. [27]. Il MinHash può essere usato stimare la *Jaccard Similarity* tra due insiemi in tempo lineare utilizzando uno spazio di memoria piccolo e fisso. L'indice Jaccard, tuttavia, quando viene utilizzato per l'intersezione tra insiemi, produce un bias per cui i dataset più grandi sono penalizzati. Per l'intersezione, è possibile utilizzare la misura di *set containment*, calcolata come il rapporto tra la dimensione dell'intersezione e la dimensione di uno dei due insiemi. Per il calcolo del set containment, è possibile utilizzare l'algoritmo LSH Ensemble applicato sui *MinHash data sketches* [28].
L'algorithmo LSH Ensemble è stato proposto da Zhu et al. [29] per risolvere il problema noto come *domain search problem*, che consiste nel ricercare domini dei dataset presenti nel Web simili oltre ad un certo threshold ad altri domini di altri dataset. In particolare, viene formalizzato nella seguente maniera: "*Dato un dominio di query Q e una soglia di rilevanza t*, trova tutti i domini X la cui rilevanza a Q è compresa in t**.". Infatti, nel problema posto, un dominio è rilevante se contiene tanto quanto il dominio di ricerca. Per fare ciò, gli autori propongono l'utilizzo dell'LSH Ensemble per la stima del set containment tra i domini. La ricerca dei domini con un certo valore di set containment rispetto al dominio di query viene chiamata dagli autori "ricerca di *joinable tables*". L'algoritmo LSH Ensemble viene cosiderato come una soluzione efficiente e scalabile per questo problema. Prima dell'applicazione di questo algoritmo, gli autori convertono i domini in una firma *MinHash* utilizzando un insieme di funzioni *minwise hash*. Per ogni funzione, l'hash value è ottenuto usando una funzione di hash che mappa tutti i valori di dominio in valori di hash interi e ritorna il valore di hash minimo

osservato. Data questa funzione e il valore di hash minimo del dominio X e del dominio Y, la probabilità che i due valori di hash siano uguali è la Jaccard Similarity tra X e Y. Quindi, date le firme di X e Y, è possibile ottenere una stima della Jaccard Similarity conteggiando il numero di collisioni nei corrispondenti valori di hash minimi e dividendolo per il numero totale di valori di hash in una singola firma.
Dato quindi un insieme di firme MinHash, l'algoritmo LSH divide ogni firma in b "bande". Per l'i-esima banda, è definita una funzione che restituisce la concatenazione dei valori hash minimi in quella banda. Data la firma del dominio di query, l'LSH mappa la firma in *buckets* usando le funzioni applicate alle bande. I domini le cui firme sono mappate in almeno un bucket dove è mappata la firma della query sono i possibili candidati. Dato un threshold nella Jaccard similarity, i possibili candidati sono quei domini che hanno quella soglia di probabilità di divenire candidati. Il contributo di Zhu et al. [29] è stato quello di utilizzare l'algoritmo per la stima non della Jaccard Similarity, bensì del set containment tra insiemi.
In questo elaborato, l'algoritmo LSH Ensemble sviluppato dagli autori viene utilizzato per la stima della similarità tra *joinable dataset*, così da prevedere le dimensioni finali dei possibili join tra le sorgenti dati restituite dalla query.

Capitolo 3

Architettura

In questo capitolo viene descritto il Semantic Data Lake esistente e il framework di *query answering* implementato.

3.1 Semantic Data Lake

Il Semantic Data Lake considerato è stato sviluppato da Diamantini et al. [1]. L'architettura è definita mediante una tupla $SDL =< S, G, K, m >$, dove S è l'insieme delle sorgenti dati presenti nel SDL, G è l'insieme dei metadati corrispondenti, K è un Knowledge Graph e m è una funzione di mapping che correla i metadati e i concetti nel Knowledge Graph.
In particolare, l'architettura è composta da due strati principali: il *Metadata Layer* e il *Knowledge Layer*.
In Figura 3.1 viene data una rappresentazione del Semantic Data Lake. Come verrà dettagliato successivamente, il Knowledge Layer è composto da un'ontologia *KPIOnto* grazie alla quale è possibile definire un *Knowledge Graph* per la rappresentazione semantica dei concetti presenti nelle sorgenti dati memorizzate, e da un insieme di regole logiche con cui si forniscono meccanismi di *reasoning* per il calcolo algebrico delle formule degli indicatori presenti. Il Metadata Layer è sostanzialmente un catalogo di metadati, rappresentato da grafi. I due strati (Metadata e Knowledge) sono collegati tra loro mediante dei mapping tra Knolwedge Graph e i diversi grafi dei metadati.

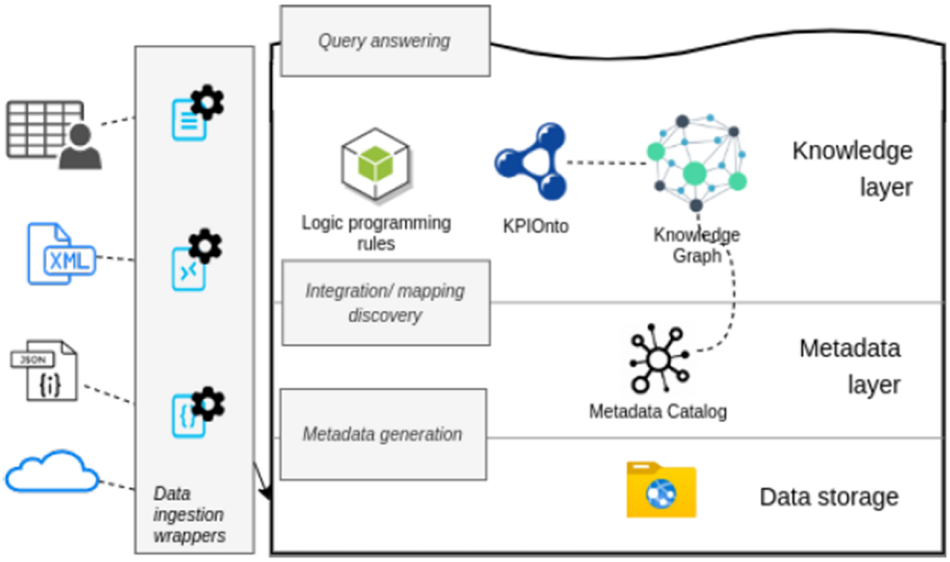

Figura 3.1: Semantic Data Lake

Metadata Layer
Come già mostrato nel capitolo precedente, la gestione dei metadati è fondamentale nel contesto dei Data Lake ed è possibile strutturare i metadati con diversi approcci. Nel Semantic Data Lake proposto [1], per ogni sorgente S_k i metadati sono rappresentati come un grafo diretto (*Metadata Graph*), costruito incrementalmente ogni volta che viene inserita una nuova sorgente all'interno del SDL.
In particolare, ogni nodo è un elemento dello schema della sorgente (ad es. tabelle o attributi in database relazionali) e gli archi rappresentano le relazioni tra i diversi nodi. Le relazioni sono definite mediante diversi vocabolari, al fine di conferire una semantica agli elementi del Metadata Graph e quindi di dare una rappresentazione uniforme ai metadati delle diverse sorgenti eterogenee. Ad esempio, per legare un nodo *Source* (sorgente) ad un nodo *domains* (dominio, ad esempio un attributo della tabella), l'arco che li collega ha l'etichetta *dl:contains*.

Di seguito si riporta un esempio di Metadata Graph per una sorgente. In Figura 3.2 si mostrano i vocabolari utilizzati per la rappresentazione dei grafi. Per migliorare la leggibilità del codice, vengono introdotti dei prefissi.

```
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix dl: <http://kdmg.dii.univpm.it/datalake/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix void: <http://rdfs.org/ns/void#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

Figura 3.2: Vocabolari utilizzati per la definizione di un Metadata Graph

In Figura 3.3 viene mostrato un esempio di Metadata Graph. La proprietà *dl: Source* definisce che l'URIRef alla sinistra è una sorgente, la proprietà *void: Dataset* indica che è un'istanza della classe *Dataset*. Per i nodi collegati al nodo sorgente che rappresentano gli elementi della sorgente si utilizza la proprietà *dl: contains*. Nell'esempio è possibile vedere come la sorgente contenga due livelli dimensionali, due attributi e un indicatore. I restanti elementi del grafo sono metadati aggiuntivi riguardanti, ad esempio, la posizione della sorgente nel file system.

```
<http://kdmg.dii.univpm.it/datasources/source1> a dl:Source,
        void:Dataset ;
    dl:contains <http://kdmg.dii.univpm.it/datasources/source1_L1_D0>,
        <http://kdmg.dii.univpm.it/datasources/source1_L1_D1>,
        <http://kdmg.dii.univpm.it/datasources/source1_attr0>,
        <http://kdmg.dii.univpm.it/datasources/source1_attr1>,
        <http://kdmg.dii.univpm.it/datasources/source1_ind1> ;
    dl:domains "20"^^xsd:int ;
    dl:items "1000"^^xsd:int ;
    dl:location "datasets/source1.csv"^^xsd:string ;
    dcterms:date "2023-07-21"^^xsd:date .
```

Figura 3.3: Metadata Graph

I nodi relativi agli elementi della sorgente (attributi, livelli, misure) hanno a loro volta degli archi che specificano i loro metadati. Nella Figura 3.4 è possibile vedere come il nodo relativo al livello *L1_D0* sia innanzitutto un'istanza della classe *Domain*. Una proprietà molto importante è *dl: mapTo* che, se presente, indica che quel nodo è mappato ad un concetto del Knowledge Graph, e alla destra della relazione è specificato l'URIRef del nodo nel Knowledge Graph.

```
<http://kdmg.dii.univpm.it/datasources/source1_L1_D0> a dl:Domain ;
    rdfs:label "L1_D0"^^xsd:string ;
    dl:hasProfileElement [ ... ],
    dl:mapTo <http://kdmg.dii.univpm.it/kg/L1_D0> ;
    dl:type "prova"^^xsd:string .
```

Figura 3.4: Metadata Graph

Lo strato dei metadati è stato successivamente esteso in un altro lavoro [30], specificando i cosiddetti *profili*, ovvero insiemi di metadati che forniscono informazioni sintetiche sui valori di uno specifico dominio di una sorgente dati. Il processo di assegnazione di questi metadati alle sorgenti viene chiamato *profilazione*. Per la profilazione dei domini, il sistema assegna, se possibile, il tipo di valore contenuto in un dominio (es. stringa).

Knowledge Layer
Lo strato di conoscenza consente di rappresentare la semantica del Data Lake. I concetti che vengono definiti sono (1) le gerarchie multidimensionali, quindi dimensioni, livelli e membri, nonché le relazioni tra questi elementi (es. *rollup* tra due livelli), (2) indicatori (cioè misure) e formule per il loro calcolo.
In particolare, il Knowledge Layer è composto dalle seguenti parti:

- *KPIOnto*: è un'ontologia [1] che fornisce la terminologia per modellare i concetti del Semantic Data Lake. L'ontologia fornisce diverse classi e proprietà per rappresentare gerarchie ed indicatori con le loro formule. La figura 3.5 mostra le principali classi e proprietà dell'ontologia KPIOnto.

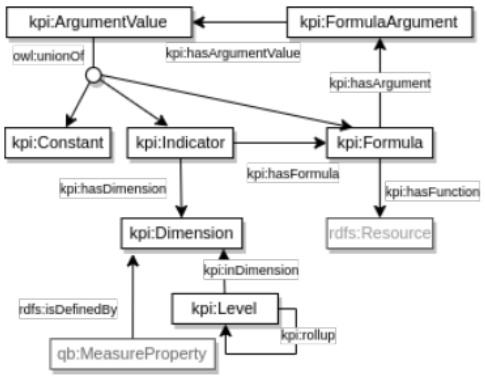

Figura 3.5: Principali classi e proprietà nell'ontologia *KPIOnto*

Ad esempio, la classe *Dimension* serve per la definizione delle dimensioni lungo cui gli indicatori sono misurati. La classe *Level* rappresenta uno speci-

[1] `http://w3id.org/kpionto`

fico livello collegato alla sua dimensione mediante la proprietà *inDimension* e può anche essere collegato ad un livello superiore mediante la proprietà di rollup, dando la possibilità di rappresentare le relazioni gerarchiche tra i livelli.
La classe *Indicator* specifica quelli che sono gli indicatori, o misure, e la proprietà *Formula* rappresenta l'espressione matematica per il calcolo dell'indicatore a cui fa riferimento.

- Knowledge Graph: i nodi rappresentano i concetti della conoscenza, mentre gli archi con le rispettive etichette definiscono le relazioni tra i concetti. L'ontologia KPIOnto viene sfruttata per la costruzione di questo grafo. I concetti sono rappresentati come grafi RDF, garantendo la possibilità di sfruttare lo standard di querying SPARQL per l'interrogazione e quindi la navigazione dei concetti all'interno del grafo. La figura 3.6 mostra una porzione di un Knowledge Graph. Si può vedere come, ad esempio, *Time* sia la dimensione temporale, mentre *Month* è il livello della dimensione tempo. Per legare questi due concetti, viene posto un arco con etichetta *kpi:InDimension* mediante la terminologia fornita dall'ontologia KPIOnto. Nel caso (b) vengono invece rappresentati gli indicatori con la possibilità di esplorare anche le formule che calcolano gli indicatori. É possibile riferirsi al caso (b) come *Formula Graph*, ovvero una vista sul Knowledge Graph che rappresenta le relazioni matematiche tra gli indicatori. Nell'esempio, si può vedere come l'indicatore *Cases*, che misura il numero totale di casi positivi al Covid, si può ottenere sommando il numero di persone attualmente positive, le morti e i ricoverati.

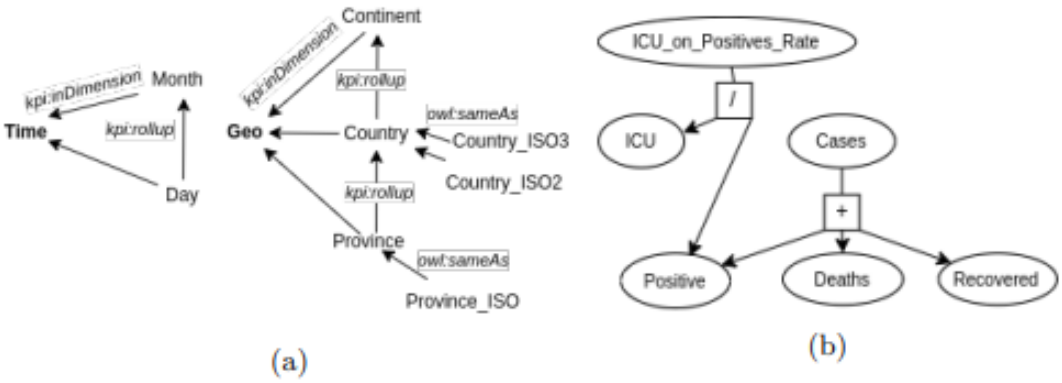

Figura 3.6: Esempio di Knowledge Graph con (a) dimensioni e livelli, (b) indicatori con le loro formule

- Regole di programmazione logica: nell'architettura è presente anche un

insieme di regole logiche, grazie alla presenza di un reasoner logico capace di eseguire manipolazioni matematiche sulle formule degli indicatori. Questo meccanismo è stato introdotto al fine di consentire, in futuro, di ricavare le formule di un indicatore al fine di riscrivere una query posta dall'utente.

Knowledge Graph e Metadata Graph sono legati tra loro mediante un meccanismo di *mapping*. Infatti, quando una nuova sorgente dati viene caricata all'interno del Semantic Data Lake, non solo vengono costruiti i metadati principali, ma si cercano anche eventuali mapping tra ogni dominio (es. attributo) della sorgente e un concetto del Knowledge Graph. Se, ad esempio, viene caricata una sorgente con un attributo *State*, viene ricercato nel Knowledge Graph un concetto semanticamente vicino al livello dimensionale considerato. Se esiste un concetto *Country* nel Knowledge Graph con elementi simili, allora viene costruito il mapping.
In particolare, questo viene fatto mediante la combinazione del MinHash con l'algoritmo Locality Sensitivity Hashing (LSH), trovando quindi concetti simili ad un certo dominio di una sorgente oltre un determinato threshold.
I mapping vengono sfruttati ampiamente per il query answering, in quanto consentono di ricercare le sorgenti dati i cui domini sono semanticamente simili agli elementi inseriti nella query dall'utente. Partendo dai concetti del Knowledge Graph, è infatti possibile ricavare i domini presenti nei metadati al fine di scoprire le sorgenti contenenti le informazioni richieste.

3.2 Sistema di query answering

I mapping definiti tra i grafi dei metadati e il Knowledge Graph sono sfruttati per supportare il *query answering* nel contesto dei Data Lake [1].
Lo scopo di questo elaborato è l'implementazione di un framework di query answering che sia in grado di rispondere alla query di un utente e quindi di restituire le sorgenti dati che contengono le informazioni richieste. Un ulteriore obiettivo è quello di effettuare un ranking delle soluzioni trovate in base alla cardinalità delle sorgenti (o combinazioni di sorgenti), al fine di individuare quelle che restituiscono una maggiore quantità di informazioni.

Viene rappresentata una query Q di un utente come una tupla $Q =< ind, \{L_1, ..., L_n\} >$, dove ind è un indicatore, mentre $\{L_1, ..., L_n\}$ è l'insieme dei livelli dimensionali specificati tali che non possono esistere più livelli appartenenti alla stessa dimensione.
L'idea originaria del lavoro di Diamantini et al. [1] era quello di sfruttare il servizio di reasoner logico al fine di prendere in input l'indicatore specificato nella query, ricercare gli indicatori componenti delle formule che calcolato l'indicatore dato ed eseguire tante riscritture della query quante sono le formule disponibili per quell'indicatore. Questo approccio è stato proposto perché può accadere che

(1) una o più sorgenti contengono l'indicatore specificato, ma si vuole avere la possibilità di trovare altre sorgenti con gli indicatori di una formula per arricchire il risultato finale, (2) non ci sono sorgenti che contengono l'indicatore specificato, quindi è necessario trovare altre sorgenti dati con gli indicatori che sono operandi di una o più formule matematiche che calcolano l'indicatore di input. Poiché non è stato possibile in questo lavoro integrare il servizio di reasoning, gli esperimenti sono stati effettuati consentendo all'utente di specificare più indicatori nella query, come verrà meglio illustrato nel capitolo successivo 4.

Considerata la query in input, lo scopo, quindi, è quello di trovare le soluzioni, cercando innanzitutto le sorgenti con lo stesso schema dimensionale. In dettaglio, data una sorgente dati S e una query Q, si dice che S ha lo stesso schema dimensionale di Q se e solo se l'insieme dei livelli dimensionali contenuti in S è esattamente pari all'insieme dei livelli dimensionali $\{L_1, ..., L_n\}$ di Q.
In implementazioni future è possibile estendere questa definizione, consentendo al sistema di restituire una sorgente S che ha schema dimensionale diverso da Q ma che diventa uguale se almeno un livello dimensionale di S è in relazione di *drill-down* o *roll-up* con un livello di Q.
Le sorgenti trovate, comunque, dovranno anche avere l'indicatore specificato nella query per poter rappresentare una soluzione corretta.
L'approccio adottato sfrutta quindi l'architettura già presente e questo consente di ridurre lo spazio di ricerca identificando solo le sorgenti dati più rilevanti al livello semantico in accordo con le necessità di scoperta. Le richieste dell'utente sono espresse con termini ontologici come le query OLAP [1].
Una volta che le soluzioni sono state identificate, può essere utile fornire all'utente una classifica delle sorgenti dati con maggiori informazioni. Nel caso in cui, per soddisfare la query dell'utente, sia necessario eseguire dei join tra le sorgenti trovate, viene proposto un approccio per stimare le cardinalità finali del join. Nel contesto dei Data Lake, infatti, eseguire join multipli tra sorgenti che possono essere anche molto grandi, può risultare molto oneroso sia al livello di computazione che di occupazione di memoria. Viene quindi proposto un *joinability index*, ovvero un indice che consente di stimare la cardinalità dei join. Questo indice viene costruito mediante l'applicazione congiunta del MinHash e dell'algoritmo LSH Ensemble, come verrà meglio dettagliato nel capitolo successivo 4.

Capitolo 4

Implementazione

In questo capitolo verrà dettagliata l'implementazione del sistema di *query answering*.
Nella sezione 4.1 sarà mostrato l'algoritmo per la scoperta delle sorgenti dati che rispondono alla query, mentre nella sezione 4.2 si illustrerà come è stato calcolato il joinability index per il ranking dei risultati.

4.1 Source discovery

La prima fase dello sviluppo è stata dedicata alla ricerca delle sorgenti dati che risondono alla query dell'utente.
Al fine di illustrare meglio il procedimento, ci si riferirà nel corso di questa sezione all'esempio in Figura 4.1, dove si ha un grafo dei metadati che si riferisce alla sorgente S0, la quale possiede due livelli dimensionali L0_D0 e L1_D1, e un indicatore ind1. D'altra parte, il Knowledge Graph possiede due dimensioni D0 e D1 con i suoi livelli e un indicatore. Si vuole sottolineare che i termini sono gli stessi in entrambi i grafi per semplificare la spiegazione, ma nella realtà i nomi di due concetti semanticamente uguali possono essere diversi. Ad ogni modo, si può vedere come i domini della sorgente S0 sono legati ai concetti del Knowledge Graph con dei mapping.

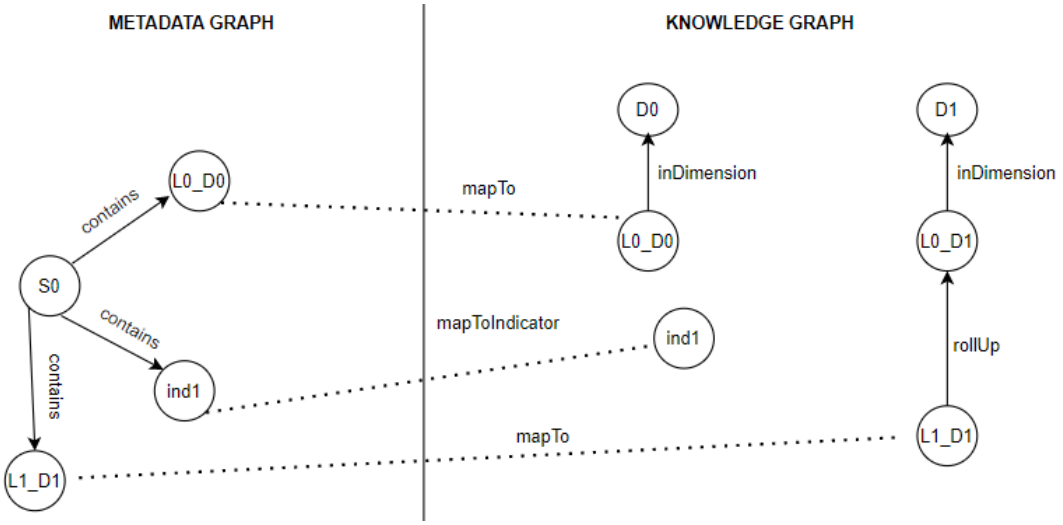

Figura 4.1: Esempio di un Metadata Graph e il Knowledge Graph con i mapping

Com'è stato già discusso nel capitolo 3, per trovare le sorgenti dati che rispondono alla query bisogna prima identificare quelle che hanno lo stesso schema dimensionale della query. Prima di fare ciò, bisogna però verificare che la query sia corretta, cioè che i livelli e gli indicatori specificati esistano nel Semantic Data Lake. Per questa verifica, viene sfruttato il Knowledge Graph, che consente di rappresentare tutta la conoscenza di dominio del Data Lake. Poiché il Knowledge Graph è di fatto strutturato mediante RDF, è possibile esplorarlo mediante linguaggio SPARQL, che viene quindi sfruttato per navigare il grafo e ricercare i concetti che fanno riferimento agli elementi specificati nella query.
L'utente, quindi, deve esprimere la query secondo i termini dell'ontologia, rendendo quindi il Knowledge Layer un vero e proprio schema globale di accesso ai dati.
Ad esempio, in Figura 4.2 viene mostrata una query SPARQL eseguita sul Knowledge Graph per trovare il livello *levelURI* specificato. Nelle variabili precedute da un punto interrogativo (es. *?x)* si memorizza il risultato della query. Quindi, in x ci sarà, se esiste, un oggetto di tipo *Level* nel Knowledge Graph con l'URIRef specificato in input.

```
result = self.graph.query(
        """SELECT ?x WHERE {?x rdf:type kpi:Level .filter(?x=<%s>)}"""%levelURI)
```

Figura 4.2

In Figura 4.3, invece, è rappresentata una query SPARQL eseguita sul Metadata Graph per ricercare i mapping con il Knowledgre Graph. Viene preso l'URIRef del livello presente nel Knowledge Graph e si trovano tutti i domini che

hanno un mapping di tipo *mapTo* a tale livello.

```
result = self.graph.query(
          """SELECT ?x  WHERE {?x dl:mapTo <%s>}""" % levelURI)
```

Figura 4.3

Il Knowledge Graph viene sfruttato anche per verificare che l'utente non inserisca nella stessa query livelli che fanno riferimento alla stessa dimensione.
Viene ora mostrato l'algoritmo 1 che, dati i livelli dimensionali della query, ricerca le sorgenti dati che hanno lo stesso schema dimensionale e quindi gli stessi livelli.

Algoritmo 1 Source Discovery

1: $levels \leftarrow [L_1, ..., L_n]$
2: $domains \leftarrow []$
3: $sources \leftarrow []$
4: **for** $level$ in $levels$ **do**
5: $\quad mapping \leftarrow getmappingFromMG(level)$
6: $\quad$ **if** $len[mapping] > 0$ **then**
7: $\quad\quad append(domains, mapping)$
8: $\quad$ **else**
9: $\quad\quad raiseNoDomain$
10: **for** $domain$ in $domains$ **do**
11: $\quad source \leftarrow getSourceFromDomain(domain)$
12: $\quad append(sources, source)$
13:
14: $sourcesSameSchema \leftarrow []$
15: $sourcesSet \leftarrow set(sources)$
16: **for** $source$ in $sourcesSet$ **do**
17: $\quad numSource \leftarrow$ numero di $source$ in $sources$
18: $\quad$ **if** $numSource == len(levels)$ & $len(getLevelsFromSource) == len(levels)$ **then**
19: $\quad\quad append(sourcesSameSchema, source)$
20: **return** $sourcesSameSchema$

Alla riga 1 viene dichiarata la variabile *levels* che fa riferimento ai livelli dimensionali presenti nella query.
Nelle righe 4-9 si trovano i domini nei grafi dei metadati che contengono i livelli specificati. In dettaglio, per ogni livello, la funzione $getmappingFromMG$ (riga 5) esegue una query SPARQL ai grafi dei metadati per ricercare il dominio che è mappato al livello considerato nel Knowledge Graph.

In Figura 4.4 è possibile vedere graficamente questo passaggio: supponendo che uno dei livelli nella query sia L0_D0, la funzione, individuato tale livello nel Knowledge Graph, cerca i domini nello strato dei metadati che hanno una relazione *mapTo* con questo livello.

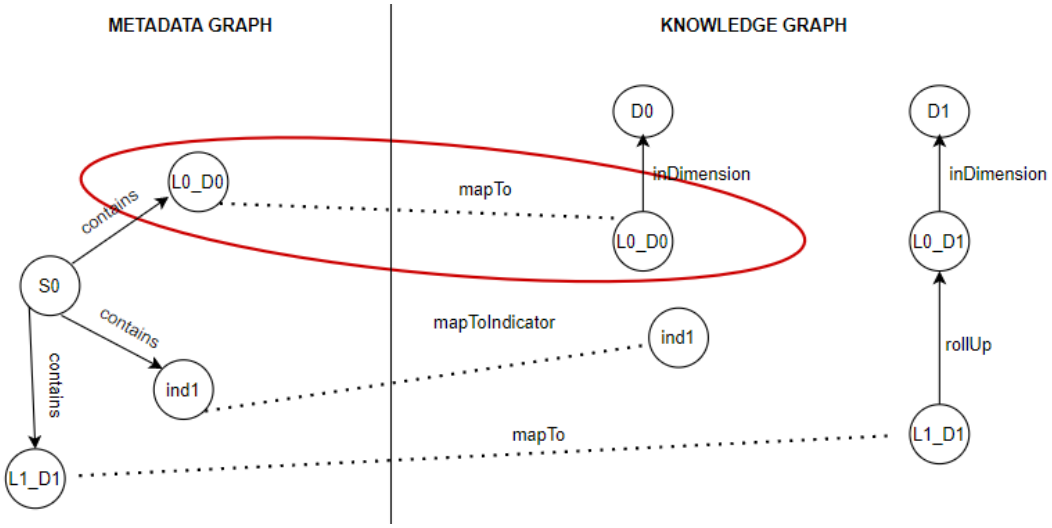

Figura 4.4: Esempio di un Metadata Graph e il Knowledge Graph con i mapping

Viene poi eseguita la funzione *getSourceFromDomain* (riga 11) che, per ogni dominio trovato, ricerca nel grafo dei metadati la sorgente corrispondente, e questa viene memorizzata. In Figura 4.5, è possibile vedere come, nell'esecuzione di questa funzione, venga sfruttata la relazione *contains* definita in uno dei vocabolari sfruttati per la costruzione dei grafi dei metadati. Vengono quindi restituite tutte le sorgenti che hanno questa relazione con un loro dominio.

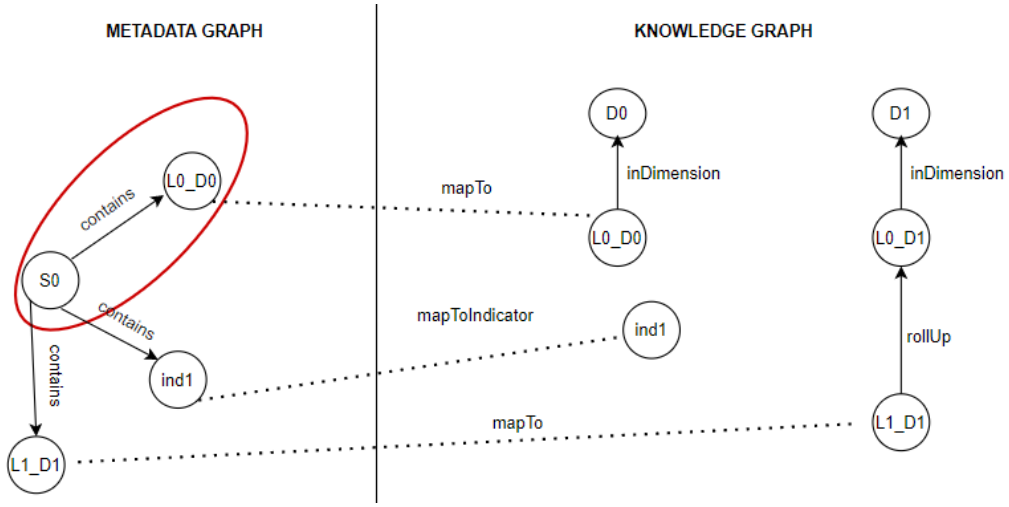

Figura 4.5: Esempio di un Metadata Graph e il Knowledge Graph con i mapping

Nelle righe 14-19 vengono cercate le sorgenti con lo stesso schema dimensionale della query. Viene creato un insieme (quindi senza duplicati) della lista *sources* e per ogni sorgente contenuta in questo insieme, si conta il numero di volte in cui compare nella lista *sources*. Se la sorgente è presente tante volte quante sono i livelli specificati nella query, allora significa che tutti i domini trovati relativi a quella sorgente sono tutti i livelli nella query. A riga 18, contemporaneamente allo schema dimensionale viene verificato che siano memorizzate le sole sorgenti che hanno esattamente i livelli contenuti nella query.

A questo punto, è necessario vedere quali sorgenti tra quelle trovate contengono gli indicatori specificati nella query. Per fare ciò, si è deciso di lavorare con una struttura dati che memorizzi per ogni indicatore specificato nella query, l'insieme delle sorgenti che lo contengono. Ad esempio, se l'insieme degli indicatori nella query è *{ind1, ind2}*, la struttura dati risultante sarà: $\{ind1 : \{S1, S2\}, ind2 : \{S3, S4\}$, se le sorgenti *S1* ed *S2* contengono l'indicatore *ind1* e *S3* ed *S4* contengono *ind2*. L'algoritmo 2 mostra la procedura per trovare le sorgenti contenenti gli indicatori nella query.
La funzione *getSourcesFromInd* a riga 2 restituisce la struttura dati appena illustrata. A riga 3 viene eseguito un for che prende ad ogni iterazione una coppia (chiave, valore), dove la chiave è uno degli indicatori della query mentre il valore è l'insieme delle sorgenti che contengono quell'indicatore. Per ogni sorgente contenuta nell'insieme, si verifica se questa è tra quelle memorizzate nell'algoritmo precedente, cioè se ha lo stesso schema dimensionale della query. Se è vero, allora la sorgente viene memorizzata in una struttura dati dizionario uguale alla struttura dati di partenza, ma con le sole sorgenti che sono in grado di rispondere alla query. Prendendo l'esempio precedente, se *S2* e *S4* non contengono gli stessi

livelli della query, allora *correctSources* sarà: $\{ind1 : \{S1\}, ind2 : \{S3\}$.

Algoritmo 2 Source Discovery

```
1: correctSources ← dict
2: sourcesIndicators ← getSourcesFromInd
3: for ind, sources in sourcesIndicators do
4:     for source in sources do
5:         if source in sourcesSameSchema then
6:             correctSources[ind] = source
```

In dettaglio, l'esecuzione della funzione a riga 2, esegue ciò che è rappresentato in Figura 4.6. Identificato l'indicatore nel Knowledge Graph, vengono cercati i domini che hanno un mapping di tipo *mapToIndicator* all'indicatore stesso. Dal dominio, sempre mediante una query SPARQL nel grafo dei metadati, viene restituita la sorgente corrispondente.

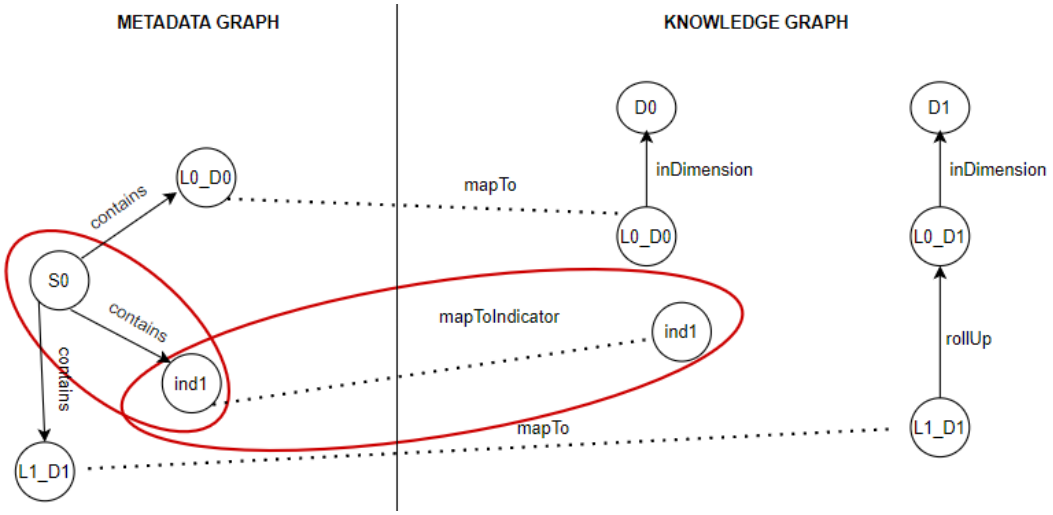

Figura 4.6: Esempio di un Metadata Graph e il Knowledge Graph con i mapping

Ora è necessario distinguere il processo di *source discovery* in due casi:

- Un solo indicatore specificato nella query: in questo caso, la struttura dati è semplicemente costituita da un indicatore con l'insieme delle sorgenti che lo contengono. Non sono necessari ulteriori step, in quanto l'algoritmo 2 restituisce già le soluzioni alla query. Il ranking viene fatto ordinando le sorgenti per la loro dimensione (cardinalità).
- Più indicatori: questo caso è più complesso, in quanto diverse sorgenti possono contenere i diversi indicatori della query. Per integrare tutta l'informazione richiesta dalla query, può essere quindi necessario eseguire dei

join tra le sorgenti. Da questo momento in poi ci si concentrerà su questo secondo caso, mostrando innanzitutto come vengono elaborate le soluzioni.

Per ottenere tutte le possibili combinazioni di sorgenti che restituiscono le informazioni richieste, viene sfruttata la struttura dati *correctSources* data in output dall'algoritmo 2. Più in dettaglio, viene eseguito il prodotto cartesiano tra tutti gli insiemi di sorgenti.
Il seguente esempio rende più chiara la procedura:

$$correctSources = \{ind1 : \{S1, S2, S3\}, ind2 : \{S4, S5, S6\}\}$$

$$\{S1, S4\}, \{S1, S5\}, \{S1, S6\}, ... \leftarrow prodottoCartesiano(\{S1, S2, S3\}, \{S4, S5, S6\})$$

Alla sinistra del secondo passaggio vengono trovate le combinazioni di sorgenti risultato della query. Ogni combinazione è costituita da n sorgenti e ogni sorgente S_i contiene l'indicatore i-esimo. Per ottenere la soluzione della query, è necessario eseguire il join tra le sorgenti di ogni combinazione.
Come già discusso, si vuole evitare l'esecuzione del join nel contesto di un Data Lake, e si è introdotto quindi un *joinability index*, la cui implementazione si discute nella prossima sezione (4.2).

4.2 Joinability Index

Il calcolo del *joinability index* (JI) viene effettuato mediante l'esecuzione dell'algoritmo Locality Sensitive Hashing (LSH) Ensemble, che consente di stimare il *set containment* tra due insiemi sopra un certo threshold impostato. Gli insiemi da prendere in considerazione per il calcolo della similarità sono gli attributi di join delle sorgenti. I domini delle sorgenti, quindi, vengono confrontati per determinare quanti valori sono condivisi tra le sorgenti mediante set containment. Prima dell'applicazione dell'algoritmo, viene applicata una funzione di hashing ai domini delle sorgenti tramite MinHash. Questo viene fatto in quanto un confronto tra i valori reali dei domini può portare a tempi di esecuzione elevati, nonostante l'ottimizzazione dell'algoritmo. Pertanto, il confronto della similarità delle sorgenti viene fatto sui valori di hash ottenuti con il MinHash.
Sorge, tuttavia, un problema: l'LSH Ensemble, per il calcolo della similarità tra insiemi, si basa su un singolo attributo di join alla volta, mentre nell'approccio attuale si ha bisogno di considerare tutti gli attributi di join delle sorgenti, ovvero i livelli dimensionali della query. A tal proposito, è stato quindi adottata una strategia che combini i domini che rappresentano i livelli dimensionali così da mapparli in un singolo dominio prima di applicare la funzione di hashing [1]. In particolare, è stata applicata una concatenazione di stringhe dei domini delle sorgenti, così da ottenere un unico dominio con valori concatenati. Se, ad esempio, si hanno due domini nella sorgenti *Country* e *Year* e si hanno valori

come *Italy* e *2020*, allora la concatenazione mapperà i due domini in un unico attributo *Country_ Year*, dove i valori saranno del tipo *Italy_ 2020*. Sui valori così ottenuti viene applicato il MinHash, i cui valori di hash sono memorizzati in maniera permanente in fase di caricamento della sorgente nel Semantic Data Lake.
Il calcolo del joinability index avviene durante l'esecuzione della query, una volta che sono state ottenute le combinazioni di sorgenti dalla fase precedente (sezione 4.1). Durante l'esecuzione dell'LSH Ensemble vengono ripresi i valori di hash calcolati in fase di caricamento delle sorgenti considerate.

L'algoritmo sottostante (3) mostra il procedimento per il calcolo del joinability index.

Algoritmo 3 Calcolo *joinability index*

```
1:  function ComputeJoinability(< S_1, ..., S_m >)
2:      first = 0.0
3:      last = 1.0
4:      delta = 0.049
5:      Search  S* ∈ {S_1, ..., S_m} s.t. |S*| = min_{i=1,...,m}|S_i|
6:      while  first <= (last − delta)  do
7:          q = (first + last)/2
8:          Λ = LSH_Ensemble(S*, {S_1, ..., S_m}\S*, q)
9:          if |Λ| < (m − 1) then last = q
10:         else
11:             first = q
12:     return q
```

La funzione *ComputeJoinability* viene richiamata per ogni combinazione di sorgenti trovata, e questa viene data in input alla funzione stessa. La ricerca del joinability index viene effettuata in maniera dicotomica. Infatti, nelle righe 2-4 si inizializzano i parametri per questa ricerca.
Alla riga 5 viene restituita la sorgente S^* che ha la cardinalità più piccola delle sorgenti all'interno della combinazione.
Alla riga 7 viene inizializzato il valore di threshold q sommando l'estremo inferiore e l'estremo superiore della ricerca dicotomica e dividendo per 2.
Alla riga 8 viene richiamato l'LSH Ensemble nella seguente maniera: (1) Prima viene indicizzato l'algoritmo con tutti i valori di hash delle sorgenti della combinazione, tranne S^*, (2) Viene eseguita una query all'algoritmo passando come parametro la sorgente con cardinalità più piccola S^*. L'algoritmo viene impostato con threshold pari a q. In questa maniera, si ricercano tutte le sorgenti che hanno valore di *set containment* con S^* maggiore del threshold q.
Se il numero di sorgenti restituite è minore del numero di sorgenti nella combina-

zione eccetto S^* (riga 8), significa che non tutte le sorgenti hanno un valore di *set containment* con S^* maggiore di q, quindi viene diminuito il threshold partendo con un estremo superiore più basso nella ricerca dicotomica. Altrimenti, si alza il valore dell'estremo inferiore, e quindi il threshold aumenta.
L'algoritmo continua finché non si trova un valore di threshold oltre il quale tutte le sorgenti hanno quel valore di *set containment* con S^*. Il valore finale del threshold q corrisponderà al *joinability index*.
Per ottenere la stima della cardinalità finale del join tra le sorgenti nella combinazione, il valore di *joinability index* viene moltiplicato per la cardinalità della sorgente più piccola.

Una volta ottenute le cardinalità stimate, viene restituito un ranking delle combinazioni in ordine decrescente sulla base della cardinalità ottenuta.
Al fine di valutare la qualità e l'efficienza del *joinability index*, in fase di esperimenti, sono stati eseguiti in maniera congiunta anche i relativi join delle sorgenti, così da poter confrontare cardinalità stimata e reale nonché i tempi di esecuzione.
Per quanto riguarda la valutazione della qualità, si è deciso di calcolare il Mean Percentage Error (MAPE) tra la cardinalità stimata dal *joinability index* e la cardinalità reale ottenuta mediante join, così come mostrato di seguito:

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \frac{|card(join)_i - card(stima)_i|}{card(join)_i}$$

Negli esperimenti, il MAPE verrà calcolato su più combinazioni, in modo da ottenere una media dell'errore.

Capitolo 5

Esperimenti e Risultati

Gli esperimenti, condotti su una macchina Linux [1], avevano l'obiettivo di:

- Misurare il tempo di ricerca delle sorgenti nel *Semantic Data Lake* (fase di *Source Discovery*) dalla ricerca dei livelli e degli indicatori della query nel Knowledge Graph fino alla restituzione delle combinazioni di sorgenti che soddisfano la query
- Misurare l'efficienza del joinability index, calcolando sia il tempo di esecuzione dell'algoritmo per il joinability index che il tempo di esecuzione del join, così da poter confrontare i due processi
- Valutare la qualità del joinability index, confrontando la cardinalità stimata dal joinability index e la cardinalità reale ottenuta mediante join tramite MAPE

Per gli esperimenti sono stati innanzitutto generati dei dataset da 100'000 e 1'000'000 di righe con le seguenti caratteristiche (Tabella 5.1):

Numero colonne	20
Numero Livelli Dimensionali	4
Numero Indicatori	4
Percentuale Rumore	Variabile dal 10% al 50%

Tabella 5.1: Caratteristiche Dataset Esperimenti

É stato prima generato un Knowledge Graph con 4 dimensioni e 5 livelli, da cui sono stati presi i membri per la generazione dei datasets. Nella generazione di un Knowledge Graph vengono anche specificati il numero di membri per ogni

[1] Ubuntu Linux 22.04.2 LTS, CPU: Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz (8 core), RAM: 64GB

livello e il fattore di crescita. Se vengono specificati n livelli, significa che per una data dimensione ci sarà una gerarchia dimensionale data da n livelli legati da relazioni di roll-up. Il fattore di crescita indica quanto cresce il numero di membri per ogni livello della gerarchia di una dimensione. Se, come in questo caso, sono specificati 10 membri e fattore di crescita 1, significa che al livello più in alto nella gerarchia ci saranno 10 membri, al secondo 100, al terzo 1'000, al quarto 10'000 e infine al quinto 100'000.
Sono stati eseguiti gli esperimenti separatamente prima per i dataset da 100'000 righe, poi per quelli da 1'000'000 di righe. Per entrambi i casi, gli esperimenti sono stati divisi in 5 test diversi, ognuno per una percentuale di rumore dei dataset. Dalla Tabella 5.1 si può vedere come la percentuale di rumore all'interno dei dataset vari dal 10% al 50%: ciò significa che si avranno 5 gruppi di dataset diversi con una percentuale di righe totali rumorose, ovvero variate con valori casuali rispetto ai valori originali presi dal Knowledge Graph. In particolare, quindi, si avranno dataset con il 10%, 20%, 30%, 40% e 50% di righe rumorose.
Per ogni test vengono generati 11 dataset differenti, di cui 1 senza rumore mentre le altre 10 copie del dataset senza rumore con la stessa percentuale di rumore prestabilita. Ad esempio, sul test dei dataset da 100'000 righe con il 10% di rumore, (1) viene generato dal Knowledge Graph un dataset, (2) vengono create 10 copie da questo dataset, (3) per ogni dataset copia, sono variate in maniera random 10'000 righe (il 10%).
Tutti i dataset all'interno dello stesso test possiedono lo stesso schema dimensionale (quindi gli stessi livelli dimensionali), ma indicatori differenti (vengono aggiunti successivamente alla generazione dei membri dei livelli). Nello specifico, il primo dataset senza rumore ha determinati indicatori, mentre gli altri 10 dataset copia hanno gli stessi indicatori ma diversi dal primo dataset. Ciò viene fatto per facilitare gli esperimenti, come viene spiegato di seguito con un esempio.

In questo esempio, si hanno per semplicità di spiegazione 3 dataset con 10 righe, di cui 1 senza rumore mentre gli altri 2 con rumore pari al 10%. Si avranno quindi i 2 dataset con 1 riga variata rispetto al dataset originale. In Figura 5.1 ci sono i 3 dataset: S1 è la sorgente senza rumore che ha due livelli e un indicatore, S2 ed S3 sono i dataset copia con due indicatori uguali tra loro e una riga rumorosa.

S1		
L1	L2	ind1
L1_1	L2_1	20
L1_2	L2_2	15
L1_3	L2_3	12
L1_4	L2_4	10
L1_5	L2_5	21
L1_6	L2_6	5
L1_7	L2_7	12
L1_8	L2_8	22
L1_9	L2_9	20
L1_10	L2_10	30

S2			
L1	L2	ind2	ind3
L1_1	L2_1	11	10
L1_2	L2_2	1	2
L1_3	L2_3	2	12
L1_4	L2_4	5	20
L1_5	L2_5	20	4
100	2	25	2
L1_7	L2_7	10	50
L1_8	L2_8	1	45
L1_9	L2_9	6	12
L1_10	L2_10	9	10

S3			
L1	L2	ind2	ind3
L1_1	L2_1	11	10
L1_2	L2_2	1	2
L1_3	L2_3	2	12
L1_4	L2_4	5	20
L1_5	L2_5	20	4
4	12	25	10
L1_7	L2_7	10	50
L1_8	L2_8	1	45
L1_9	L2_9	6	12
L1_10	L2_10	9	10

Figura 5.1: Esempio per gli esperimenti

Se l'utente inserisce la query $< (ind1, ind2), (L1, L2) >$, i dataset così generati permetteranno al sistema di query answering di trovare le seguenti combinazioni: $\{S_1, S_2\}, \{S_1, S_3\}$. Se invece si specifica la query $< (ind1, ind2, ind3), (L1, L2) >$, si troveranno le seguenti combinazioni: $\{S_1, S_2, S_3\}, \{S_1, S_3, S_2\}, \{S_1, S_2\}, \{S_1, S_3\}$. Per facilitare gli esperimenti, le combinazioni dove ci sono le stesse sorgenti sono state eliminate, ovvero viene preso solo uno dei duplicati (ad esempio si prende solo $\{S_1, S_2, S_3\}$).
Inoltre, sempre per semplicità di analisi, quando vengono inseriti n indicatori nella query, saranno restituite le combinazioni con esattamente n sorgenti. Prendendo ad esempio il caso precedente in cui si hanno i tre indicatori nella query, un risultato completo comprenderebbe anche le combinazioni da due sorgenti. Tuttavia, poiché negli esperimenti queste vengono già considerate quando si inseriscono due indicatori nella query, in fase di test queste combinazioni vengono filtrate. É bene notare che nel framework corretto questo filtraggio non avviene, perché tutte le combinazioni risultanti dal prodotto cartesiano restituiscono le informazioni richieste dall'utente e vanno quindi considerate.

Negli esperimenti si costruiscono i dataset con 4 indicatori, così da poter arrivare a combinazioni con 5 sorgenti ciascuna. Quindi, per ogni test con una certa percentuale di rumore, si effettuano 4 query con numero di indicatori diversi, da 2 a 5.
Dato che si hanno 11 dataset di cui 10 copie con rumore e visto l'esempio precedente, per ogni test si otterranno i risultati illustrati nella Tabella 5.2.

Numero indicatori	**Numero combinazioni ottenute**
2	10
3	45
4	120
5	210

Tabella 5.2: Combinazioni ottenute per ogni test

Al numero di indicatori corrisponde il numero di sorgenti all'interno di ogni combinazione.

Di seguito vengono riportati i risultati ottenuti nella fase di *Source Discovery* (sezione 5.1) e successivamente nella fase di calcolo del *joinability index* (sezione 5.2).

5.1 Tempo di source discovery

Il tempo medio totale per la ricerca di sorgenti nel Semantic Data Lake è pari a 0.206847 secondi. Dalla Tabella 5.3 viene mostrato il tempo medio di ricerca delle sorgenti per numero di indicatori nella query, ovvero per numero di sorgenti all'interno di ogni combinazione. Il tempo medio cresce di poco all'aumentare del numero di indicatori e ciò mostra l'efficienza dell'approccio adottato.

Numero indicatori	**Tempo medio ricerca sorgenti** (s)
2	0.190686
3	0.196649
4	0.204579
5	0.228317

Tabella 5.3: Tempo medio ricerca sorgenti per numero di indicatori nella query o sorgenti in ogni combinazione

Il tempo massimo registrato per la ricerca delle sorgenti è pari a 0.347277 secondi. La deviazione standard risultante è molto piccola (è pari circa a 0.02), dimostrando che non ci sono grandi oscillazioni nei tempi.

Si è voluto poi analizzare se ci fosse un andamento crescente dei tempi all'aumentare delle dimensioni del Metadata Graph. Si è ipotizzato che, eseguendo diverse interrogazioni nei grafi dei metadati, i tempi sarebbero aumentati con una dimensione totale più grande. Dal grafico a barre in Figura 5.2 è possibile

vedere che non è così. Sull'asse delle ascisse ci sono delle dimensioni riassuntive di quelle totali registrate, espresse in MB. Non si notano andamenti crescenti significativi da 60MB a 300MB. Tuttavia, in futuri esperimenti, si potrebbero testare dimensioni più grandi nell'ordine dei GigaByte.

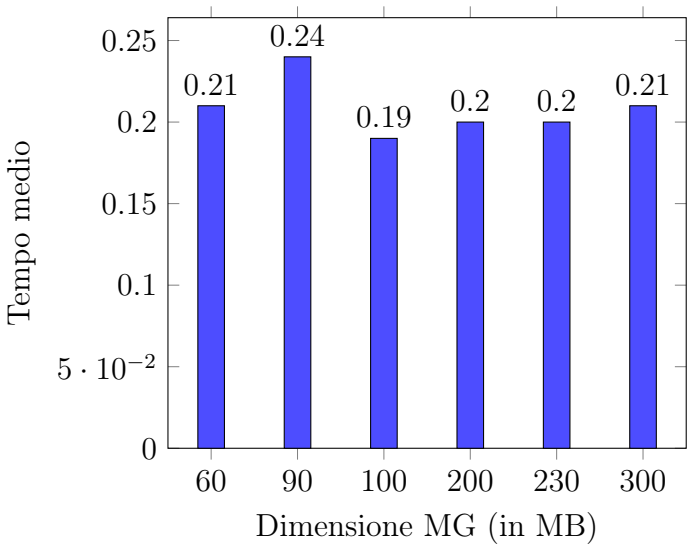

Figura 5.2: Tempo medio ricerca sorgenti per dimensione MG

Si può quindi concludere che i tempi totali di ricerca delle sorgenti che rispondono alla query in input siano soddisfacenti.

5.2 Calcolo Joinability Index

Gli esperimenti che riguardano il calcolo del *joinability index* sono stati eseguiti (1) valutando il suo tempo di esecuzione, confrontandolo con i tempi dei join (sezione 5.2.1), (2) valutando la qualità dell'indice mediante MAPE (sezione 5.2.2). Come già discusso precedentemente, sono stati eseguiti due esperimenti principali, uno con dataset da 100'000 righe e il secondo con dataset da 1'000'000 di righe. Per ognuno, sono stati fatti 5 test distinti per percentuale di rumore nelle sorgenti dati. Per ciascun test sono state eseguite 4 query per generare un numero diverso di sorgenti nelle combinazioni.
Durante gli esperimenti i risultati sono stati memorizzati in un unico file specificando per ogni combinazione il tempo di esecuzione del join tra le sorgenti, il tempo di calcolo del joinability index, la cardinalità stimata, la cardinalità vera, il MAPE.
Il join viene effettuato mediante funzione *merge* della libreria *Pandas*, memoriz-

zando a run-time i risultati parziali sul disco. Questo è stato necessario soprattutto quando c'era bisogno di eseguire il join tra 4 o 5 datasets, poiché la RAM non era sufficiente alla memorizzazione dei risultati parziali.

5.2.1 Tempi di esecuzione

A partire dal file contenente tutti i risultati ottenuti dagli esperimenti, è stato ricavato il tempo medio di calcolo del joinability index e risulta essere pari a 2.415634 secondi. Rispetto a un totale di 3850 combinazioni ottenute da tutti i diversi test, circa 60 contenevano degli outliers dei tempi di calcolo e sono stati esclusi da questa valutazione.
Il tempo medio di calcolo dei join è invece pari a 18.50 secondi con un minimo pari a 1.53 secondi per le sorgenti più piccole e un massimo valore pari a 49.1 secondi per i dataset da 1'000'000 di righe.
La deviazione standard dei tempi di calcolo dell'indice è pari a 0.088 secondi mentre quella dei tempi di esecuzione dei join è 16.34 secondi. Da questo risulta che l'indice rimanga piuttosto costante, mentre il tempo di esecuzione di un join varia.
Dalla Tabella 5.4 è possibile vedere i tempi medi di esecuzione del joinability index e dei join raggruppati per numero di righe. Si nota subito come il tempo di esecuzione del joinability index rimanga piuttosto costante all'aumentare delle dimensioni dei dataset, mentre il tempo di esecuzione dei join aumenta.

Numero righe	**Tempo medio JI** (s)	**Tempo medio join** (s)
100'000	2.4	3.1
1'000'000	2.42	33.9

Tabella 5.4: Tempi medi di calcolo del joinability index e dei join per numero di righe nei dataset

Dato che, per quanto riguarda le considerazioni sui tempi, si nota come un notevole vantaggio in efficienza si ha al crescere della dimensione dei dataset, da qui fino alla fine di questa sezione (5.2.1) si riportano i risultati solo per i dataset da 1'000'000 di righe.
Dal grafico in Figura 5.3 è possibile distinguere i tempi medi di esecuzione dei join e di calcolo del joinability index raggruppando i dati per numero di sorgenti nelle combinazioni.
L'esecuzione del join cresce in complessità all'aumentare del numero di sorgenti da combinare insieme, anche per la necessità di memorizzare sul disco i risultati

intermedi. Il tempo di esecuzione del joinability index rimane costante.

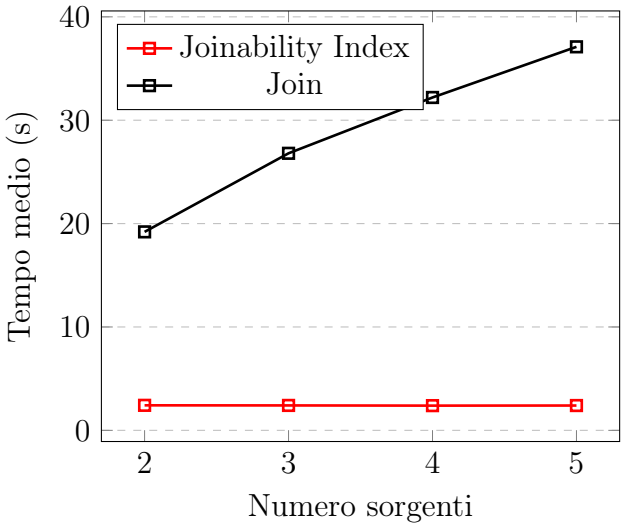

Figura 5.3: Tempi medi di calcolo per numero di sorgenti

In Figura 5.4 è possibile visualizzare la differenza nei tempi medi di esecuzione raggruppando i dati per percentuale di rumore. Sul joinability index non si notano differenze, mentre sul join si hanno sempre tempi di esecuzione maggiori ma che decrescono all'aumentare della percentuale di rumore. Il massimo tempo registrato per l'esecuzione del join (49.1 secondi) è infatti relativo al join tra 5 sorgenti di cui 4 con il 10% di rumore.

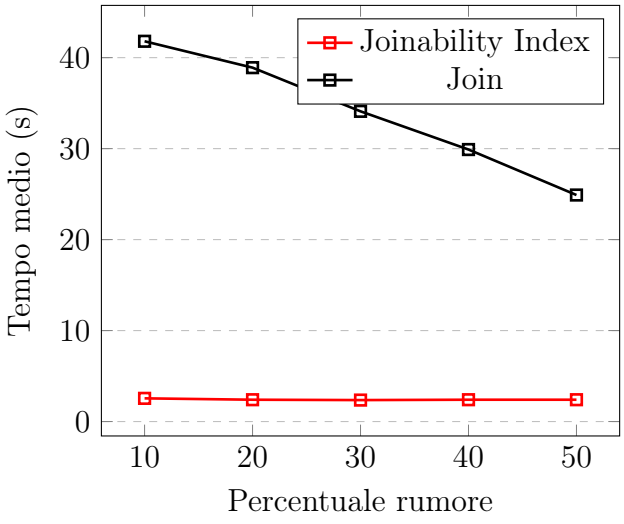

Figura 5.4: Tempi medi di calcolo per numero di sorgenti

Dai risultati, emerge quindi una notevole differenza di efficienza tra l'esecuzione di join e il calcolo del joinability index. Si prevede che con ulteriori esperimenti con dataset più grandi, la differenza aumenti maggiormente. I tempi di calcolo del joinability index non aumentano all'aumentare della dimensione dei dataset in quanto il calcolo viene fatto su valori di hash precalcolati dove il numero di permutazioni viene pre-impostato ed è fisso per tutte le sorgenti dati che vengono caricate nel Data Lake. I vettori di hash hanno, pertanto, dimensione fissa a prescindere dalla dimensione del dataset. L'esecuzione del join necessita invece la considerazione delle sorgenti dati nella loro interezza. La complessità intrinseca dell'operazione di join assieme alla necessità di memorizzare i risultati parziali sul disco comportano dei tempi di calcolo piuttosto lunghi. É necessario far notare che nel contesto dei Data Lake, le sorgenti possono essere molto più grandi di quelle considerate negli esperimenti. Anche il numero di combinazioni possibili può aumentare considerevolmente. Il tempo totale per eseguire join tra tutte le sorgenti che possono rispondere alla query può essere pertanto molto grande, comportando una grande attesa da parte dell'utente che ha fatto la richiesta. L'utilizzo, quindi, del joinability index come stima delle cardinalità finali dei join può essere molto utile in questi casi.
Sarà successivamente l'utente, una volta ottenute tutte le possibili combinazioni con un ranking di quelle con maggiori informazioni (grazie alla stima data dal joinability index), a scegliere le combinazioni più opportune e ad eseguire eventualmente dei join per le proprie necessità.

5.2.2 Qualità del Joinability Index

Una volta dimostrata l'efficienza di questo approccio, è necessario mostrare la qualità del joinability index mediante il calcolo del MAPE, considerando cardinalità reali ottenute mediante join e cardinalità stimate dall'indice.
Il Joinability Index tende a fornire una sovrastima della similarità reale tra le sorgenti dati e funziona al meglio quando il confronto è eseguito su due insiemi. Infatti, come si vedrà successivamente, se le combinazioni di sorgenti considerate sono costituite da due sorgenti, la cardinalità stimata e quella reale sono piuttosto simili. Quando, invece, aumenta il numero di sorgenti da confrontare insieme, il joinability index non lavora come il join, il quale elabora di volta in volta i risultati parziali per restituire solo gli elementi in comune tra tutte le sorgenti. Difatti, l'indice elabora le sorgenti a coppie, non considerando la similarità complessiva tra le sorgenti dati e fornendo quindi una stima la cui precisione decresce all'aumentare degli insiemi da considerare.
Inoltre, dai risultati emerge anche che la qualità dell'indice è più alta per i dataset poco rumorosi e che quindi hanno un maggior numero di elementi in comune.
Dal grafico in Figura 5.5 vengono mostrati i risultati ottenuti dal MAPE ordinati per percentuale di rumore e numero di sorgenti in una combinazione. Poiché tra

le sorgenti da 100'000 righe e quelle da 1'000'000 di righe non si notano miglioramenti significativi (i valori del MAPE sono piuttosto simili a parità di rumore e numero di sorgenti in una combinazione), i risultati sono stati raggruppati in un unico grafico.
Quando si hanno 2 sorgenti, si hanno i migliori risultati in termini di qualità. Al crescere del numero di sorgenti nelle combinazioni, i risultati peggiorano. Si nota anche che con il 10% di rumore si hanno i risultati migliori.

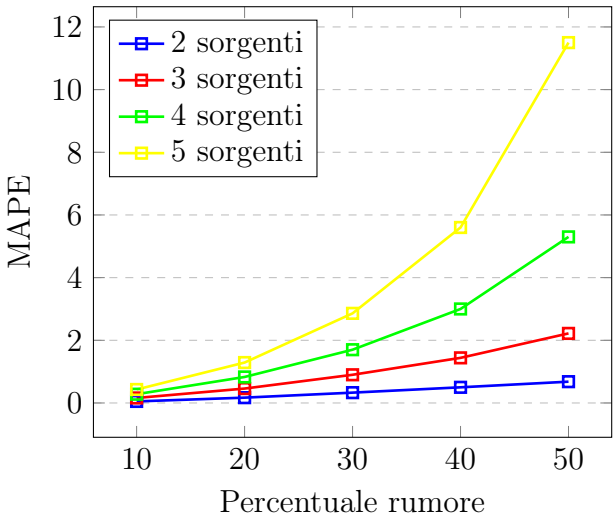

Figura 5.5: MAPE

Dai risultati, si può vedere ad esempio che per una combinazione da due sorgenti e il 10% di rumore, la cardinalità vera del join è pari a 90'000 righe, mentre quella stimata è di 93'750. Con il 50% di rumore, si ha in una combinazione una cardinalità vera pari a 50'000 mentre una cardinalità stimata pari a 78'125: in questo caso il MAPE è più alto e la qualità del joinability index è quindi peggiore.

Nonostante la qualità del joinability index peggiori all'aumentare della diversità tra i dataset e del numero di sorgenti da combinare, si sostiene comunque che sia uno strumento potente per stimare la cardinalità finale dei join tra le sorgenti che rispondono alla query perché è in grado di distinguere tra un insieme di sorgenti che hanno molti elementi in comune e un altro insieme di sorgenti che hanno invece un numero ridotto di elementi in comune. Infatti, si auspica di utilizzare questo indice al fine di mostrare all'utente quali sono le combinazioni che possono potenzialmente restituire il maggior numero di informazioni. Se, ad esempio, l'utente sta cercando il numero di positivi al Covid per Stato ma è interessato solo a quelli europei e la query restituisce due combinazioni di sorgenti (S1, S2), (S3, S4), dove S3 è una sorgente in cui gli Stati sono tutti quelli degli USA, mentre in S4, S1 ed S2 ci sono Stati europei, è chiaro che un eventuale join tra S3 e S4 non porterebbe ad alcun risultato utile in quanto S3 non contiene le

informazioni che l'utente desidera. Può anche accadere che alcune delle sorgenti presenti nei risultati abbia poche righe rispetto ad altre sorgenti: un join tra una sorgente da 10'000'000 di righe e un'altra da 1'000 righe porta chiaramente meno informazioni di un join tra due sorgenti con 10'000'000 righe ciascuna.

Il joinability index, in un contesto di Data Lake, evita join dispendiosi ed inutili, individuando, seppur con una sovrastima, i potenziali join con più dati di valore. Con ulteriori esperimenti, si vuole dimostrare che il joinability index è in grado di restituire un ranking molto simile rispetto a quello generato con le cardinalità reali mediante join. A tal proposito, si utilizza un indice chiamato *Kendall Tau* [2], ovvero una misura per la corrispondenza tra due rankings. Valori di *Tau* vicini a 1 indicano una stretta vicinanza, mentre valori vicini a -1 indicano che i rankings confrontati sono molto diversi.
La formula per il calcolo di questo indice è la seguente:

$$Tau = (P - Q)/sqrt((P + Q + T) * (P + Q + U))$$

Dove P è il numero di coppie concordanti, Q è il numero di coppie discordanti, T è il numero di legami che ci sono solo in x (primo ranking), e U è il numero di legami solo in y (secondo ranking). Se un legame occorre per la stessa coppia sia in x che in y, allora non è aggiunto né in T né in U.
Quando, per ogni combinazione di sorgenti, viene calcolato sia il joinability index che il join, vengono generati due rankings separati. In particolare, vengono memorizzate in due dizionari separati le informazioni inerenti alla cardinalità stimata e a quella reale. Quindi, data la combinazione di sorgenti (S_1, S_2), nel ranking del joinability index si scriverà $(S_1, S_2) = card\_stimata$, mentre nel ranking del join si scriverà $(S_1, S_2) = card\_reale$. Questo procedimento viene eseguito per tutte le combinazioni di sorgenti trovate. I due rankings vengono infine ordinati in ordine decrescente in base alle cardinalità. Le combinazioni più in alto nel ranking indicano quelle sorgenti che, integrate, possono potenzialmente fornire all'utente il maggior numero di informazioni. Se il joinability index restituisce, quindi, certe combinazioni come quelle più ricche di dati, ci si aspetta che, eseguendo il join, si ha una cardinalità finale piuttosto alta. Al fine quindi di confrontare i due rankings e verificare che il joinability index sia capace di fornire questa classificazione, è stato eseguito un piccolo esperimento calcolando l'indice *Kendall Tau* tra i due rankings.

In dettaglio, sono state considerate 11 sorgenti, di cui la prima da 1'000'000 di righe mentre le restanti 10 sono copie della prima sorgente con determinate righe eliminate, con il seguente numero di righe: 900'000, 800'000, 700'000, 600'000, 500'000, 300'000, 200'000, 100'000, 50'000, 10'000. Come nei precedenti

[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kendalltau.html

esperimenti, al fine di ottenere un certo numero di sorgenti nelle combinazioni e di standardizzare in tutti i test l'esecuzione dei join, nel primo datasets c'è una misura, mentre gli altri 10 datasets hanno lo stesso insieme di misure ma diverso dalla misura del primo dataset.
Dai risultati emerge che il MAPE, come già osservato nei precedenti esperimenti, aumenta all'aumentare del numero di sorgenti in una combinazione e, nonostante in certi casi sia piuttosto grande, i risultati ottenuti calcolando il *Kendall Tau* sono molto soddisfacenti.

Al termine di questo esperimento, vengono ottenuti in totale 4 coppie di rankings, ognuna inerente ad un test dove ci sono n sorgenti all'interno delle combinazioni, con n che varia da 2 a 5. La tabella 5.5 mostra tali risultati. É possibile osservare che, nonostante gli errori di stima non indifferenti, il joinability index sia in grado di restituire una classificazione delle combinazioni di sorgenti molto vicina al ranking reale ottenuto con le cardinalità finali dei join.

Numero sorgenti	**Tau**
2	0.9999999
3	0.9999999
4	0.9999998
5	0.9999998

Tabella 5.5: *Tau* ottenuto per i diversi rankings

Guardando direttamente ai dati, si riportano i risultati ottenuti con combinazioni di due sorgenti (Tabella 5.6). Il joinability index ordina le combinazioni di sorgenti nello stesso ordine del ranking dei join, nonostante le stime peggiorino al diminuire delle dimensioni delle sorgenti (es.: la cardinalità del join quando S2 è di 10'000 righe è esattamente 10'000, ma il joinability index restituisce una cardinalità stimata pari a circa 312).

Cardinalità S1	**Cardinalità S2**	**Cardinalità join**	**Cardinalità stimata (ji)**
1'000'0000	900'000	900'000	843'750
1'000'0000	800'000	800'000	750'000
1'000'0000	700'000	700'000	656'250
1'000'0000	600'000	600'000	562'500
1'000'0000	500'000	500'000	390'625
1'000'0000	300'000	300'000	121'875
1'000'0000	200'000	200'000	56'250
1'000'0000	100'000	100'000	21'875
1'000'0000	50'000	50'000	10'937.5
1'000'0000	10'000	10'000	312.5

Tabella 5.6: Risultati ottenuti per le combinazioni con due sorgenti

Questi risultati mostrano la bontà del joinability index nel contesto del *query answering* in un Data Lake, in quanto capace di distinguere le sorgenti che, integrate, possono portare più o meno informazioni all'utente. Anche se le stime hanno degli errori più o meno grandi, l'utente è comunque in grado di scegliere quelle combinazioni con maggiori informazioni e può scartare senza problemi quelle sorgenti con un numero ridotto di dati.

Capitolo 6

Conclusioni e sviluppi futuri

L'obiettivo di questo elaborato era la realizzazione di un sistema di query answering che estendesse l'architettura del Semantic Data Lake esistente. All'utente è stata data la possibilità di ricercare le sorgenti dati specificando indicatori e livelli dimensionali, e il sistema risulta in grado di restituire tutte le sorgenti con le informazioni richieste e di stimare la cardinalità finale di eventuali join.
Le tecnologie semantiche adottate hanno supportato una ricerca efficiente ed efficace delle sorgenti dati, consentendo di navigare nella base di conoscenza del Data Lake, al fine di trovare l'insieme completo dei dataset capaci, singolarmente o integrandoli tra loro, di soddisfare la richiesta di un utente. Il Knowledge Graph presente nell'architettura si pone come uno schema globale di accesso alle informazioni che uniforma i concetti presenti nelle sorgenti dati memorizzate, in particolare dimensioni, livelli dimensionali, membri dei livelli, nonché indicatori con le formule per il loro calcolo. I mapping generati in fase di caricamento di una sorgente tra i suoi metadati e i concetti del Knowledge Graph, generano informazioni fondamentali per una ricerca semantica delle sorgenti dati. Infatti, dato un livello dimensionale, il sistema restituisce tutte quelle sorgenti dati in cui c'è un dominio che corrisponde semanticamente al livello presente nel Knowledge Graph. Questo meccanismo è fondamentale nel mondo dei Big Data, dove le sorgenti a disposizione di un sistema di gestione dati sono innumerevoli, con schemi e formati diversi. L'eterogeneità viene risolta grazie allo strato di conoscenza semantica, il quale uniforma tutta la conoscenza a disposizione.
Una volta implementata la fase di *Source Discovery*, lo sviluppo del joinability index ha consentito una stima efficiente delle cardinalità dei join tra le diverse sorgenti trovate. Anziché eseguire molteplici join costosi, l'indice implementato permette all'utente di avere una panoramica sulle possibili combinazioni di sorgenti che possono portare maggiori informazioni.
Dai risultati è emerso come il sistema implementato garantisca un elevato grado di efficienza dal punto di vista dei tempi di esecuzione. É interessante come il tempo di calcolo del joinability index rimanga più o meno costante all'aumentare delle dimensioni delle sorgenti dati, mentre il tempo di esecuzione del join aumen-

ta sempre di più sia all'aumentare della cardinalità dei dataset che all'aumentare delle sorgenti da combinare insieme.
Se è vero che la qualità del joinability index non sia ottimale in tutti i casi, è stato possibile vedere come sia comunque in grado di generare dei buoni rankings anche se la stima della cardinalità finale delle sorgenti integrate non è, in alcune situazioni, molto vicina a quella reale. Nonostante ci sia un certo errore, è comunque soddisfacente avere dei ranking corretti, o che comunque distinguano le integrazioni che restituiscono poche informazioni da quelle che invece ne restituiscono un numero nettamente maggiore.

Il sistema può essere comunque migliorato sotto vari aspetti. Innanzitutto può essere opportuno avere, in alcuni casi, una stima della cardinalità finale dei join più precisa. Pertanto, negli sviluppi futuri si possono proporre delle modifiche all'algoritmo esistente.
Inoltre, è prevista in futuro l'implementazione del servizio di reasoning che può consentire un meccanismo di *query rewriting*. In dettaglio, specificando un singolo indicatore nella query, il reasoner logico permette di navigare nel Knowledge Graph sulle possibili formule matematiche di quell'indicatore e riscrivere tante query quante sono le possibili soluzioni. Questo consente di restituire un grande numero di combinazioni di sorgenti, garantendo all'utente una risposta completa alla query.
É possibile estendere il sistema di query answering anche sfruttando ulteriori metadati, ampliando la profilatura delle sorgenti dati con l'aggiunta di nuove informazioni.

Oltre ai possibili sviluppi futuri, si ritiene che il sistema così implementato abbia soddisfatto le specifiche definite in fase di progetto, garantendo all'utente un Semantic Data Lake fornito di un meccanismo di query answering efficiente.

Bibliografia

[1] C. Diamantini, D. Potena, and E. Storti. A knowledge-based approach to support analytic query answering in semantic data lakes. In S. Chiusano et al., editor, *Advances in Databases and Information Systems*, pages 179–192, Cham, 2022. Springer International Publishing.

[2] Franck Ravat and Yan Zhao. Data lakes: Trends and perspectives. *International Conference on Database and Expert Systems Applications (DEXA 2019)*, 2019.

[3] Pegdwendé Sawadogo and Jérôme Darmont. On data lake architectures and metadata management. *Journal of Intelligent Information Systems*, 56:97–120, 2021.

[4] Rihan Hai, Christoph Quix, Christos Koutras, and Matthias Jarke. Data lake concept and systems: a survey. *ArXiv*, abs/2106.09592, 2021.

[5] Claudia Diamantini, Paolo Lo Giudice, Lorenzo Musarella, Domenico Potena, Emanuele Storti, and Domenico Ursino. A new metadata model to uniformly handle heterogeneous data lake sources. In András Benczúr, Bernhard Thalheim, Tomáš Horváth, Silvia Chiusano, Tania Cerquitelli, Csaba Sidló, and Peter Z. Revesz, editors, *New Trends in Databases and Information Systems*, pages 165–177, Cham, 2018. Springer International Publishing.

[6] Bill Inmon. *Data Lake architecture: Designing the Data Lake and avoiding the garbage dump*. Technics Publications, 2016.

[7] Tom Heath and Bizer Christian. *Linked Data: Evolving the Web into a Global Data Space*. Morgan Claypool, 2011.

[8] Tim Berners-Lee. Linked data. In *Design Issues*. 2006.

[9] Pascal Hitzler. A review of the semantic web field. *Commun. ACM*, 64(2):76–83, jan 2021.

[10] Fabio Ciotti and Francesca Tomasi. Formal ontologies, linked data, and tei semantics. *Journal of the Text Encoding Initiative*, (9), 2016.

[11] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI global, 2011.

[12] Michalis Mountantonakis and Yannis Tzitzikas. Large-scale semantic integration of linked data: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–40, 2019.

[13] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE transactions on neural networks and learning systems*, 33(2):494–514, 2021.

[14] Oktie Hassanzadeh, Anastasios Kementsietsidis, and Yannis Velegrakis. Data management issues on the semantic web. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1204–1206, 2012.

[15] Maurizio Lenzerini. Managing data through the lens of an ontology. *AI Magazine*, 39(2):65–74, Jul. 2018.

[16] Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, Sören Auer, and Jens Lehmann. Uniform access to multiform data lakes using semantic technologies. In *Proceedings of the 21st International Conference on Information Integration and Web-Based Applications Services*, iiWAS2019, page 313–322, New York, NY, USA, 2020. Association for Computing Machinery.

[17] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37, jul 2021.

[18] Alessandro Chessa, Gianni Fenu, Enrico Motta, Diego Reforgiato Recupero, Francesco Osborne, Angelo Salatino, and Luca Secchi. Enriching data lakes with knowledge graphs. In *1st International Workshop on Knowledge Graph Generation From Text and the 1st International Workshop on Modular Knowledge, TEXT2KG 2022 and MK 2022*, 2022.

[19] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012, 2018.

[20] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989, aug 2019.

[21] Arnaud Giacometti, Patrick Marcel, Elsa Negre, and Arnaud Soulet. Query recommendations for olap discovery driven analysis. In *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, pages 81–88, 2009.

[22] Rihan Hai, Sandra Geisler, and Christoph Quix. Constance: An intelligent data lake system. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2097–2100, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Qin Yuan, Ye Yuan, Zhenyu Wen, He Wang, and Shiyuan Tang. An effective framework for enhancing query answering in a heterogeneous data lake. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '23, page 770–780, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 456–467, 2021.

[25] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. pages 847–864, 06 2019.

[26] Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. Auctus: A dataset search engine for data discovery and augmentation. *Proc. VLDB Endow.*, 14(12):2791–2794, jul 2021.

[27] Edith Cohen. Min-hash sketches., 2016.

[28] Datasketch. `http://ekzhu.com/datasketch.html`.

[29] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. Lsh ensemble: Internet-scale domain search. *arXiv preprint arXiv:1603.07410*, 2016.

[30] Claudia Diamantini, Alessandro Mele, Domenico Potena, and Emanuele Storti. Assessment of data quality through multi-granularity data profiling. In Alberto Abelló, Panos Vassiliadis, Oscar Romero, and Robert Wrembel, editors, *Advances in Databases and Information Systems*, pages 195–209, Cham, 2023. Springer Nature Switzerland.

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi