UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Magistrale in
Ingegneria Informatica e dell'Automazione

Profilazione di sorgenti dati in Data Lake basata su knowledge graph

Knowledge graph-based profiling of data sources in Data Lake

Relatore:
Prof. Potena Domenico

Correlatore:
Prof. Storti Emanuele

Laureando:
Mele Alessandro

Anno Accademico 2022/2023

Indice

1 Introduzione **5**

2 Tecnologie per la memorizzazione dei Big Data **7**
2.1 Data Warehouse 7
2.1.1 Modello multidimensionale dei dati 7
2.1.2 Architettura di un Data Warehouse 9
2.1.3 Limiti dei Data Warehouse 11
2.2 Data Lake 13
2.2.1 Architetture di un Data Lake 13
2.2.2 Gestione dei metadati 15
2.2.3 Rappresentazione dei metadati nei Data Lake 17
2.2.4 Tecnologie semantiche nei Data Lake 18

3 Definizione delle specifiche **21**
3.1 Definizione dei profili 24
3.1.1 Profilo relativo ai domini associati a livelli del knowledge graph 28
3.1.2 Profilo relativo ai domini con valori interi 32
3.1.3 Profilo relativo ai domini con valori reali 35
3.1.4 Profilo relativo ai domini con valori categorici 37
3.1.5 Profilo relativo ai domini con valori date 39
3.1.6 Profilo relativo ai domini con valori stringhe 41
3.2 Calcolo della similarità tra sorgenti 43

4 Implementazione **45**
4.1 Workflow Per il calcolo del profilo 45
4.2 Calcolo dei profili 48
4.3 Calcolo della similarità 54

5 Valutazione delle performance **57**
5.1 Valutazione del calcolo dei profili 57
5.2 Valutazione del calcolo della similarità 60

6 Conclusioni e sviluppi futuri **63**
6.1 Ringraziamenti 64

Bibliografia **65**

Capitolo 1

Introduzione

Nell'era dell'informazione, la quantità di dati generati e raccolti è cresciuta in maniera esponenziale. Questo fenomeno, noto come *Big Data*, rappresenta un'opportunità straordinaria per le organizzazioni, al fine di acquisire conoscenza e orientare le proprie strategie basandosi su fatti.

Tuttavia, la gestione, l'elaborazione e l'analisi dei *Big Data* presenta sfide significative; le tradizionali tecnologie basate sull'utilizzo dei *Data Warehouse* presentano diverse limitazioni, tra cui:

- *rigidità* del modello, con conseguenti problemi di gestione di dati non strutturati;
- *complessità* nella costruzione dello schema stesso, il quale richiede un notevole sforzo di progettazione dovuto alla difficoltà di integrazione delle diverse sorgenti.

I *Data Lake* nascono in risposta alle limitazioni dei *Data Warehouse*: in un *Data Lake* i dati vengono raccolti e memorizzati nel loro formato originale, consentendo alle organizzazioni di acquisire dati provenienti da diverse fonti senza necessità di adattamento ad uno schema predefinito; ciò agevola la fase di progettazione del sistema, ma introduce una serie di problematiche in termini di:

- *Data integration*, il cui obiettivo è fornire una visione unificata di diverse fonti eterogenee correlate;
- *Data discovery*, poiché per fare analisi spesso il primo problema è scegliere le sorgenti, e quindi analizzare il contenuto di quelle potenzialmente utili;

Il *Data Lake*, senza un'adeguata *governance*, rischia di diventare un *Data Swamp* (Khine et al.[1]), ossia un *repository* contenente dati difficilmente accessibili, manipolabili e, inevitabilmente, analizzabili. Dibowski et al.[2] affermano che questo rischio è significativo in tutti i moderni *Data Lake* privi di un *middleware* che descrive semanticamente i dati e introduce una nuova classe di *Data Lake* noti come *Semantic Data Lake*.

Tale elaborato si propone come un'estensione del *Semantic Data Lake* di Diamantini et al.[3], con l'obiettivo di realizzare un sistema di gestione dei metadati per il *Data Lake* semantico dotato di un *knowledge graph.*

Nel capitolo 2 verrà fornita una panoramica sui *Data Warehouse*, il primo sistema di memorizzazione dati nel contesto *Big Data*, per poi illustrare vantaggi, svantaggi e le motivazioni che hanno portato all'introduzione dei *Data Lake.* Con riferimento a questi ultimi, verranno illustrate le architetture, il problema della gestione dei metadati al fine di evitare il fenomeno del *Data Swamp* e una panoramica sulle *tecnologie semantiche* per fornire una descrizione semantica del contenuto di una sorgente.

Nel capitolo 3 verranno definite le proprietà per descrivere sorgenti e profili, discutendo inoltre del modello di rappresentazione delle stesse in un *metadata graph*; infine, si definirà una prima applicazione dei profili per calcolare la similarità tra due sorgenti.

Nel capitolo 4 verranno illustrate le modalità di calcolo dei profili ed il *workflow* della fase di importazione di una sorgente nel *Data Lake*; si discuteranno, inoltre, alcune problematiche legate alle tecnologie utilizzate per la realizzazione degli stessi.

Si concluderà, nel capitolo 5, valutando le prestazioni del sistema sia relativamente al calcolo dei profili che delil calcolo della similarità.

Infine, nel capitolo 6, si illustrerà un riepilogo del lavoro svolto, per terminare con una casistica di possibili sviluppi futuri al fine di ottimizzare e ampliare le funzionalità del *framework.*

Capitolo 2

Tecnologie per la memorizzazione dei Big Data

2.1 Data Warehouse

Inmon et al.[4] definiscono un *Data Warehouse* come *"una raccolta di dati subject-oriented, non volatile, integrata e variabile nel tempo, a supporto delle decisioni del management"*. Formalmente, un *Data Warehouse* è un grande archivio di dati:

- utilizzato per il supporto decisionale;
- integrato, dove i dati provengono da diversi sistemi OLTP (*On-line Transaction Processing*);
- che contiene dati storici, usualmente aggregati per effettuare stime e valutazioni;
- *offline*, ovvero i dati vengono aggiunti periodicamente estraendoli dal sistema OLTP;
- mantenuto separatamente dal sistema OLTP.

Essendo uno strumento fondamentale per il supporto alle decisioni, un *Data Warehouse* può essere utilizzato per implementare la tecnologia OLAP (*Online Analytical Processing*), la quale consente di analizzare e visualizzare i dati basandosi su una rappresentazione multidimensionale degli stessi.

2.1.1 Modello multidimensionale dei dati

Nei sistemi OLAP si adotta come modello logico il *modello multidimensionale dei dati*, il quale si basa su tre concetti principali:

- fatto, ovvero il concetto sul quale centrare l'analisi;
- misura: proprietà atomica di un fatto; tipicamente è un valore numerico, come ad esempio il conteggio delle occorrenze;
- dimensione: prospettiva lungo la quale effettuare l'analisi; vengono espresse come delle gerarchie, le quali possono essere presentare delle diramazioni, come mostrato in figura 2.1.

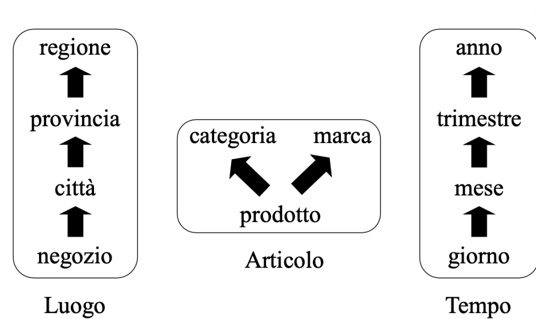

Figura 2.1: Esempio di gerarchie dimensionali.

Di seguito, in figura 2.2, si illustra un esempio di rappresentazione multidimensionale dei dati avente come *fatto* le vendite.

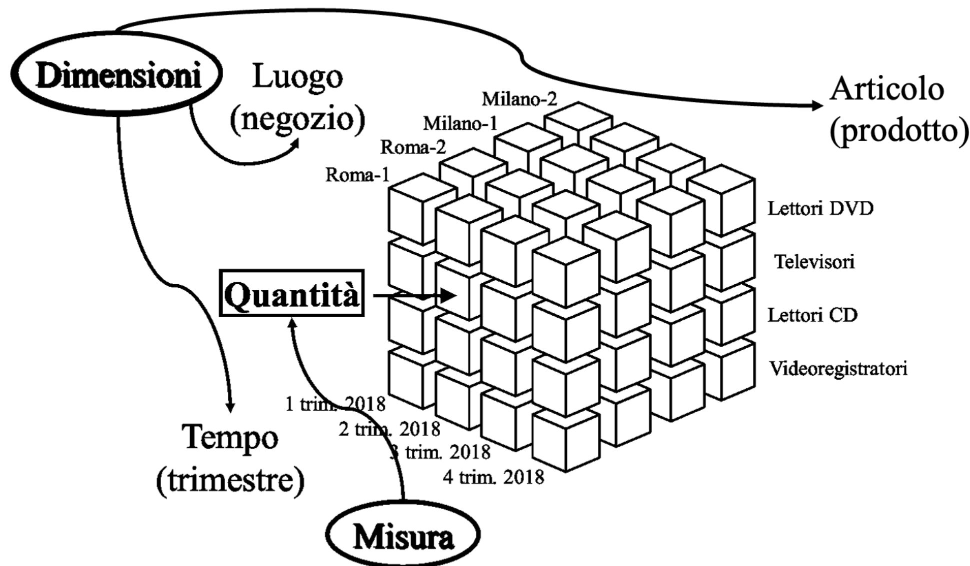

Figura 2.2: Esempio di modello multidimensionale dei dati con *fatto* le vendite.

Il *modello multidimensionale*, come illustrato in figura 2.2, viene espresso come un cubo, il quale rappresenta l'insieme dei dati sul quale è possibile applicare gli operatori OLAP:

- *Slice & dice*: selezione e proiezione, permette di ridurre la dimensionalità del cubo fissando un valore per una delle dimensioni;

- *Roll up*: aggrega i dati eliminando un livello di dettaglio da una gerarchia;

- *Drill Down*: disaggrega i dati introducendo un ulteriore livello di dettaglio in una gerarchia;

- *Pivot*: ruota il cubo in per riorganizzare le celle secondo una prospettiva diversa, ovvero portando in primo piano una differente combinazione di dimensioni.

Il *modello multidimensionale* si presenta quindi come uno strumento fondamentale per l'organizzazione dei dati in un *Data Warehouse* e per la successiva analisi e visualizzazione nei sistemi OLAP.

2.1.2 Architettura di un Data Warehouse

In letteratura si evidenziano principalmente tre tipologie di architetture per i *Data Warehouse*:

- *Single-tier architecture*, si ha un unico *repository* di dati ed un'unica componente per l'elaborazione sia analitica che transazionale; motivi per i quali non è frequentemente utilizzata nella pratica;

- *Two-tier architecture*, separa fisicamente le fonti disponibili e il *Data Warehouse* attraverso una *Staging area*, garantendo che i dati caricati nel *Data Warehouse* siano puliti ed in un formato adeguato;

- *Three-tier architecture*, è l'architettura più diffusa (Han et al.[5]), consiste in tre livelli:

  - nel livello inferiore i dati vengono puliti, trasformati e caricati tramite strumenti esterni;
  - il livello intermedio è un server OLAP che astrae il database agendo da mediatore tra l'utente finale ed il database stesso.
  - il livello superiore è costituito dagli strumenti utilizzati per connettersi e interagire con il *Data Warehouse.*

Di seguito, in figura 2.3, si illustra l'architettura di un *Data Warehouse*:

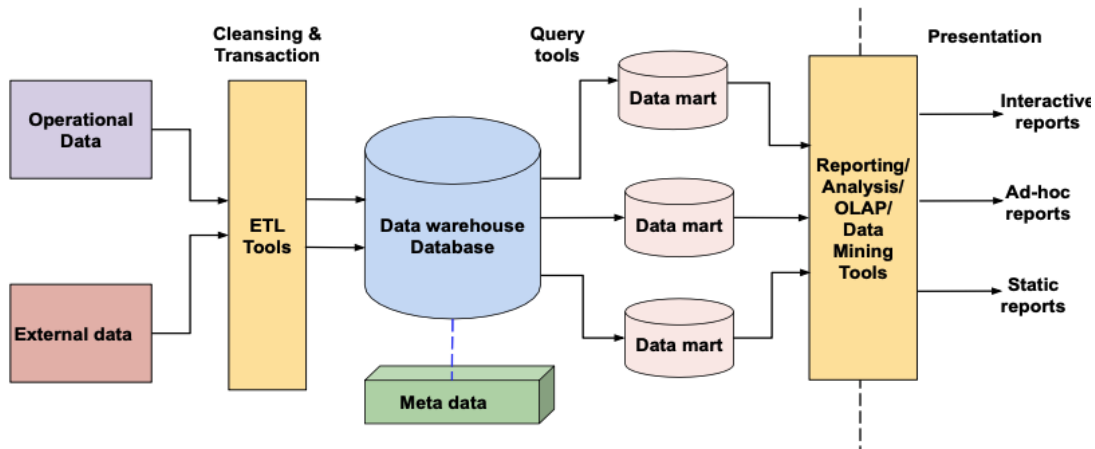

Figura 2.3: Architettura di un *Data Warehouse*(Nambiar et al.[6]).

Tra le componenti dell'architettura di un *Data Warehouse* si evidenziano:

- *Data Warehouse database*, è il cuore del *Data Warehouse*, viene implementato solitamente utilizzando la tecnologia RDBMS, la quale, non essendo ottimizzata per i *Data Warehouse*, presenta diverse limitazioni. Un *Data Warehouse* viene costruito a partire dall'unione dei suoi *Data marts* e tipicamente la progettazione avviene con una strategia *bottom-up*, progettando quindi un *Data mart* per volta, ed integrandoli fino ad ottenere il *Data Warehouse* completo. Questa strategia è più efficace rispetto all'approccio *top-down*, tipicamente non fattibile nel caso dei *Data Warehouse*.

- *Extract, transform, and load (ETL) tools*, comprende tutte le procedure di estrazione, trasformazione e caricamento dei dati necessarie per trasformare i dati in un formato unico:

  – Estrazione: si estraggono i dati di interesse dalle sorgenti;

  – Pulizia: è la fase che mira ad incrementare la qualità dei dati; tipici errori ed inconsistenze sono valori duplicati, errati e mancanti; tra le principali funzionalità vi sono la *correzione*, che utilizza dei dizionari appositi per correggere errori di scrittura, e l'*omogeneizzazione*, che utilizza sinonimi e regole proprie del dominio applicativo per stabilire le corrette corrispondenze tra valori;

  – Trasformazione: è la fase che converte i dati dal formato operazionale sorgente a quello del *Data Warehouse*; la corrispondenza con il livello sorgente è in genere complicata dalla presenza di più fonti distinte eterogenee, che richiede, durante la fase di progettazione, una complessa fase di integrazione;

  - Caricamento: l'ultimo passo è il caricamento dei dati nel *Data Warehouse*, che può avvenire facendo *refresh*, se i dati vengono riscritti integralmente sostituendo quelli precedenti, o *update*, se vengono aggiunti i soli cambiamenti senza alterare i dati esistenti.

- Metadati: sono i *"dati che descrivono dati"*(Hari et al.[7]) che aiutano a costruire, mantenere e gestire l'intero *Data Warehouse*. Svolgono un ruolo importante nella fase di ETL, poiché definiscono la fonte, l'utilizzo, le caratteristiche del *Data Warehouse* e le modalità di aggiornamento ed elaborazione dei dati dello stesso. È lo strumento più difficile da scegliere, in quanto non vi è nessuna standardizzazione.

- Strumenti OLAP: gli strumenti di elaborazione analitica online sfruttano i concetti di un database multidimensionale e aiutano ad analizzare i dati utilizzando viste multidimensionali (Codd et al.[8]). Possono essere implementati utilizzando tre diverse tecnologie:
  - ROLAP (*Relational OLAP*): i dati sono memorizzati in DBMS relazionali. Prevede l'utilizzo di un modello logico relazionale per la rappresentazione di un *Data mart*, tipicamente per mezzo di uno schema *a stella* oppure *snowflake*.
  - MOLAP (*Multidimensional OLAP*): i dati sono memorizzati in *array multidimensionali* implementati con strutture dati proprietarie. MOLAP permette di ottenere un risultato immediato, poiché tutti i dati sono già pre-aggregati nel cubo (Xu et al.[9]); tuttavia ci sono alcuni svantaggi, come la mancanza di uno standard, con conseguente difficoltà di portabilità, sparsità dei dati e lo spreco di memoria, poiché viene allocato spazio in memoria per contenere tutte le celle e mediamente solo il 20% dei dati viene acceduto;
  - HOLAP (*Hybrid OLAP*): si usa ROLAP per gestire i dati sul disco, mentre MOLAP quando sono caricati in memoria centrale.

2.1.3 Limiti dei Data Warehouse

I *Data Warehouse* sono stati lo standard di fatto per la gestione di grandi quantità di dati per molto tempo; ciò nonostante presentano alcune limitazioni:

- si richiede che i dati siano *aggregabili*: è il caso tipico di dati provenienti da sistemi OLTP, e quindi necessariamente strutturati.

- è indispensabile la progettazione dello schema, e quindi conoscere in anticipo le sorgenti disponibili;

- l'aggiunta di una nuova sorgente comporta la ri-progettazione dell'intero *Data Warehouse*, a causa della costruzione di uno schema integrato.

- bisogna decidere in anticipo quali porzioni dei dati conservare e quali no, ciò dipende dai requisiti e obiettivi aziendali; le altre andranno perse e difficilmente possono essere recuperate.

Questi limiti sono decisamente stringenti se si considerano le attuali esigenze del mondo *Big Data*, influenzate dallo sviluppo *agile* e dal Web, dove si hanno moltissime sorgenti, eterogenee, che producono dati non strutturati (Miloslavskaya et al.[10]) e che possono essere tutte potenzialmente utili.

L'idea, piuttosto recente, è dunque di non selezionare in anticipo le sorgenti e collezionare tutti i dati di interesse senza alcuna pre-elaborazione, selezionando quelli utili solo nel caso in cui servano per delle specifiche analisi: ciò si concretizza nel concetto di *Data Lake*.

2.2 Data Lake

Un *Data Lake* è definito come un *"sistema scalabile*(Sawadogo et al.[11]) *di archiviazione e analisi di dati di qualsiasi tipologia, conservati nel loro formato nativo e utilizzati da specialisti (statistici, data scientist o analisti) per l'estrazione di conoscenza"*(Madera et al.[12]). Un *Data Lake*, per essere definito tale, dovrebbe inoltre:

- utilizzare un catalogo di metadati per garantire la qualità dei dati;
- adottare politiche e strumenti di *governance* dei dati;
- essere accessibile a diverse classi di utenti;
- integrare qualsiasi tipologia di dati;
- avere un'organizzazione logica e fisica;
- essere scalabile in termini di archiviazione ed elaborazione.

2.2.1 Architetture di un Data Lake

In letteratura si identificano due principali architetture nel mondo *Data Lake*, *Pond architecture* e *Zone architectures*; di seguito, si illustra una panoramica di entrambe le tecnologie.

Nella *Pond architecture* (Inmon et al.[13]), un *Data Lake* è definito come un insieme di componenti (*ponds*), ognuna delle quali processa una specifica tipologia di dati. Di seguito, in figura 2.4, si illustra l'architettura *Pond*.

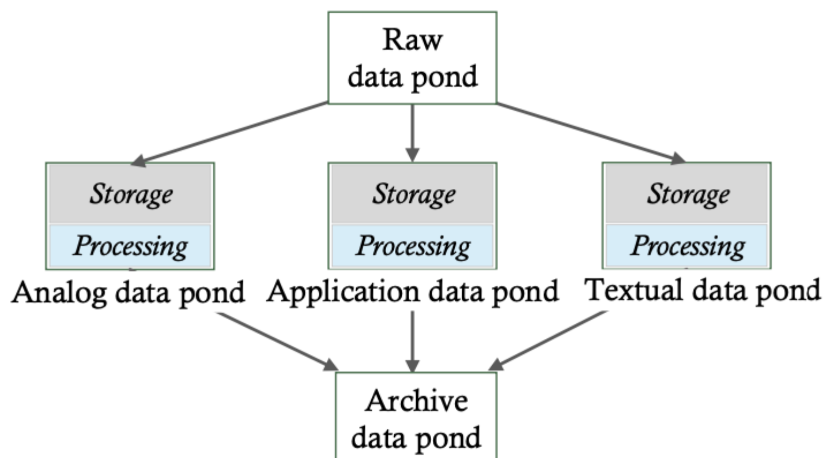

Figura 2.4: Architettura *Data pond*.

Più precisamente, si identificano cinque componenti:

- *Raw data pond*, contiene i dati grezzi appena importati; è una zona di transito, poiché i dati verranno poi trasferiti in altre componenti; a differenza degli altri, non vi è associato alcun sistema di gestione dei metadati.

- *Analog data pond*, contiene i dati caratterizzati da un'alta frequenza di misurazioni; tipicamente dati semi-strutturati provenienti dal mondo *IoT* (*Internet of Things*);

- *Application data pond*, contiene i dati provenienti da applicazioni software e quindi generalmente strutturati. Tali dati vengono integrati e trasformati per l'analisi; Inmon et al.[13] classifica l'*Application data pond* come un vero e proprio *Data Warehouse*;

- *Textual data pond*, gestisce i dati testuali non strutturati; è supportato dall'utilizzo di diverse tecniche di *Natural Language Processing* per facilitare l'analisi, come ad esempio la disambiguazione del significato del testo;

- *Archivial data pond*, si occupa di memorizzare dati provenienti dai diversi *ponds* che non vengono utilizzati attivamente, ma che potrebbero rivelarsi utili nel futuro.

Le *Pond architecture* hanno lo svantaggio che, una volta trasferiti i dati dal *Raw data pond* ad un altro bacino, non sono più disponibili nel loro formato naturale. D'altra parte, le *Zone architectures* assegnano i dati a una zona in base al loro grado di raffinatezza (Giebler et al.[14]). Di seguito, in figura 2.5, si illustra la *Zone architecture* di Zaloni (Laplante et al.[15]).

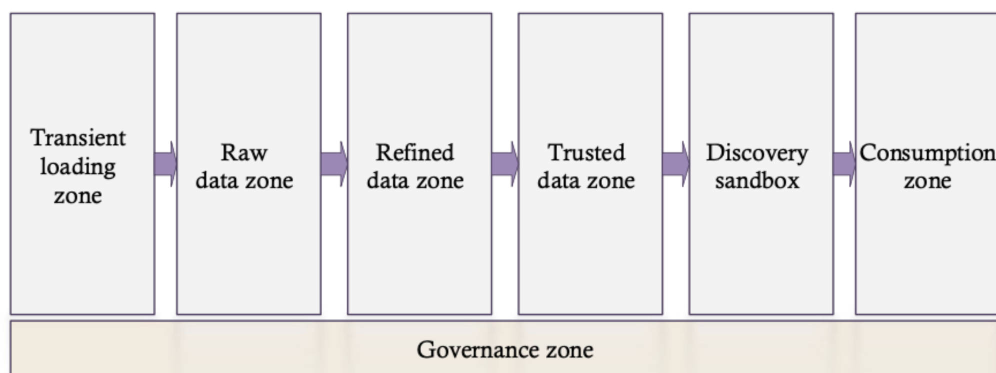

Figura 2.5: *Zone architecture* di Zaloni.

Nel dettaglio, la *Zone architecture* di Zaloni è composta da sei livelli:

- *Transient loading zone*, gestisce i dati in fase di caricamento eseguendo controlli sulla qualità degli stessi;

- *Raw data zone*, gestisce i dati in formato semi-grezzo provenienti dalla zona precedente;

- *Trusted zone*, contiene i dati trasferiti una volta standardizzati e puliti;

- *Sandbox zone*, contiene i dati prelevati dalla *Trusted zone* per effettuare *Data discovery*;

- *Consumption zone*, permette agli utenti aziendali di analizzare i dati attraverso strumenti di dashboard *user-friendly*;

- *Governance zone*, consente di gestire e monitorare i metadati, il catalogo dei dati e la sicurezza.

Nella *Zone architecture* di Zaloni si ha lo svantaggio che i dati fluiscono attraverso tutte le zone, con conseguente possibilità di copie multiple degli stessi dati.

2.2.2 Gestione dei metadati

Sebbene i metadati fossero presenti anche nell'architettura di un *Data Warehouse*, analizzando la definizione di *Data Lake* essi acquisiscono molta più importanza. Appare evidente come i metadati, ed una loro corretta gestione, rappresentino la chiave per far fronte alle sfide a cui sono posti. La mancanza di una corretta gestione dei metadati può trasformare il *Data Lake* in un *Data Swamp* (Khine et al.[1]), ovvero un *repository* contenente dati difficilmente accessibili, manipolabili e, inevitabilmente, analizzabili.

In letteratura si definiscono due tipologie di metadati, *funzionali* e *strutturali*; in figura 2.6, si propone una classificazione (Oram et al.[16]) dei metadati *funzionali* in tre categorie.

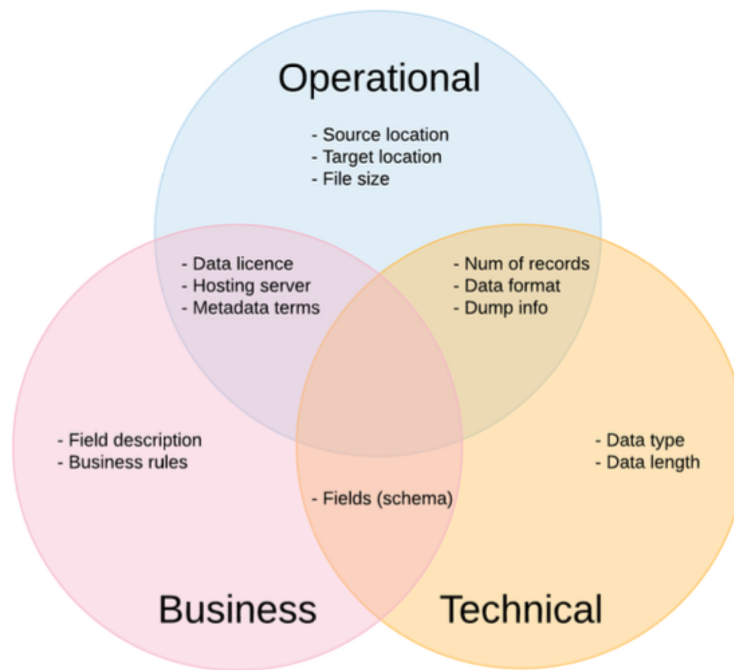

Figura 2.6: Classificazione a tre livelli dei metadati[17].

In particolare, si hanno:

- metadati tecnici: includono la la struttura dello schema, la tipologia ed il formato dei dati;
- metadati operazionali: sono generati automaticamente in fase di importazione di una sorgente nel *Data Lake*, ad esempio la dimensione ed il numero dei record;
- metadati di business: includono i nomi e le descrizioni assegnati ai campi delle sorgenti; vengono utilizzati per comprendere la semantica dei dati.

Per i metadati strutturali, Sawadogo et al.[11] propongono una classificazione rispetto al concetto di *oggetto*, che può essere interpretato come una generalizzazione di una sorgente dati, come una tabella relazionale, un foglio di calcolo, un documento XML o un file immagine. Nella classificazione vengono proposte tre tipologie di metadati:

- *intra-object metadata*, composti da:
  - *properties*, proprietà generali di un oggetto, come nome, dimensione e posizione nel *file-system*;
  - *Previsualization and summary metadata*, forniscono una panoramica del contenuto o della struttura di un oggetto; esempi sono uno schema E-R per una tabella relazionale o vettori di parole per i dati testuali;

  - *Version and representation metadata*, sono generati a partire da un oggetto già esistente quando viene modificato; ad esempio, una raffinazione è il filtraggio di un documento testuale eliminando le *stopwords*;
  - *Semantic metadata*, sono annotazioni che descrivono il significato dei dati in un oggetto; possono essere generati utilizzando le ontologie.

- *inter-object metadata*, si suddividono in tre categorie:
  - *Object groupings*, ogni oggetto può essere associato a una o più collezioni; Tali associazioni possono essere dedotti automaticamente da alcuni *intra-object metadata*, come formato dei dati o proprietario;
  - *Similarity links*, esprimono in maniera quantitativa la somiglianza tra oggetti; sono ottenibili con misure di somiglianza, ad esempio, Maccioni et al.[18] definiscono le misure di affinità e di *joinability* per esprimere la somiglianza tra oggetti;
  - *Parenthood links*, hanno lo scopo di preservare l'ordine di discendenza dei dati, ad esempio quando un oggetto viene creato a partire dalla combinazione di altri; vengono generati automaticamente quando si uniscono dati provenienti da diversi oggetti.

- *Global metadata*, non sono associati a un oggetto specifico, ma riguardano l'intero *Data Lake*; essi sono:
  - *Semantic resources*, sono basi di conoscenza come ontologie, tassonomie e tesauri; ad esempio, un'ontologia può essere utilizzata per sostituire automaticamente i termini di una interrogazione con altri equivalent;
  - *Indexes*, migliorano il processo di ricerca dei dati basato su termini o modelli;
  - *Logs*, mantengono uno storico delle interazioni dei diversi utenti con il *Data Lake.*

Si osserva che la classificazione di Sawadogo et al.[11] include la maggior parte delle caratteristiche della classificazione di Oram et al.[16], in quanto i metadati aziendali sono paragonabili a quelli semantici, i metadati operativi possono essere considerati come log, i metadati tecnici sono equivalenti a quelli di pre-visualizzazione, i metadati strutturali possono essere considerati come un'estensione della classificazione di quelli funzionali.

2.2.3 Rappresentazione dei metadati nei Data Lake

La maggior parte dei modelli di rappresentazione dei metadati nei *Data Lake* sono basati sui grafi, un formalismo matematico che permette di descrivere e

rappresentare una relazione binaria su una collezione finita e discreta di elementi; formalmente $G = (V, E)$ è una coppia di insiemi finiti, dove V è l'insieme dei nodi, mentre E è l'insieme degli archi.

Sawadogo et al.[19] definiscono tre principali modelli di rappresentazione dei metadati basati su grafi, in relazione alle proprietà che descrivono:

- *Data provenance-centered graph models*: i metadati vengono rappresentati come un grafo aciclico diretto dove i nodi rappresentano utenti, ruoli o oggetti, mentre gli archi descrivono le interazioni tra le entità, ad esempio il tipo di operazione eseguita, come lettura, creazione e modifica; questa tipologia di informazioni garantisce la tracciabilità e la ripetibilità dei processi nel *Data Lake*, pertanto possono essere utilizzati per comprendere, spiegare e riparare le incoerenze presenti nei dati (Beheshti et al.[20]);
- *Similarity-centered graph models*: i metadati vengono rappresentati come un grafo indiretto, in cui i nodi sono oggetti e gli archi esprimono la somiglianza, la quale può essere specificata attraverso un'etichetta:
  - *pesata*: identifica la forza della somiglianza, ad esempio l'affinità e la *joinability*(Maccioni et al.[18]), la quale indica la *facilità* con cui due sorgenti possono essere unite o combinate in base a un criterio di associazione comune;
  - *non pesata*: rileva se due oggetti sono connessi (Farrugia et al.[21]). Un esempio sono i sistemi di raccomandazione, i quali mostrano automaticamente i dati correlati a quelli che l'utente visualizza (Maccioni et al.[18]);
- *Composition-centered graph models*: i metadati vengono rappresentati come un grafo aciclico diretto, in cui i nodi rappresentano oggetti o attributi (ad esempio colonne), mentre gli archi da un nodo A a un nodo B esprimono il vincolo $B \subseteq A$(Oram et al.[16]). Ad esempio, Diamantini et al.[22] utilizzano una misura, espressa come stringa, per rilevare i collegamenti tra oggetti eterogenei confrontando i rispettivi tag.

2.2.4 Tecnologie semantiche nei Data Lake

Fin dall'inizio degli anni '80, la comunità *IA* ha cercato di promuovere il concetto di *"fornire una conoscenza generale e formalizzata del mondo ai sistemi e agli agenti intelligenti"* (Sheth et al.[23]), senza però ottenere buoni riscontri; gli eventi chiave furono nel 2001, quando Tim Berners-Lee, direttore del *World Wide Web Consortium* (W3C), propose la sua visione del *Semantic Web* e, nel 2006, definì i principi base per collegare tra loro gli insiemi di dati sul Web:

- URI (Uniform Resource Identifier), per identificare e referenziare le risorse.

- HTTP (Hypertext Transfer Protocol), per accedere alle risorse identificate tramite URI.
- RDF (Resource Description Framework), per descrivere le relazioni tra le risorse.

La comunità scientifica si rese conto che alcuni di questi principi potevano trovare riscontro nel mondo *Data Lake*: aggiungendo uno strato semantico che descrive il contenuto dei dati, è possibile migliorare la scoperta, l'integrazione e l'interpretazione dei dati. Mami et al.[24] definisce un *Semantic Data Lake* come *"un'estensione dei Data Lake che fornisce un middleware semantico per migliorare la comprensione dei dati e l'estrazione di conoscenza aggiungendo il contesto semantico ai dati all'interno del repository stesso"*.

Le tecnologie semantiche utilizzate nei *Data Lake* si basano principalmente su:

- ontologie: modelli di dati che definiscono le relazioni tra i concetti all'interno di un dominio specifico; possono essere utilizzate per organizzare e strutturare i dati in modo che siano coerenti e facilmente accessibili. Un esempio è OWL (*Web Ontology Language*), un linguaggio di rappresentazione che consente di definire concetti, relazioni e proprietà all'interno di un dominio specifico;
- *knowledge graph*: basati su ontologie, consentono di rappresentare e navigare attraverso le relazioni i dati;
- RDF (*Resource Description Framework*): è uno standard del *World Wide Web Consortium* (W3C) per rappresentare i dati come una terna costituita da un soggetto, un predicato e un oggetto. RDF consente di modellare le relazioni tra i dati e di creare ontologie per organizzare e strutturare le informazioni all'interno del *Data Lake*;
- SPARQL (*SPARQL Protocol and RDF Query Language*): è un linguaggio di interrogazione che consente di estrarre informazioni dai dati RDF. È ampiamente utilizzato per interrogare e recuperare dati all'interno di un *Data Lake* basato su tecnologie semantiche;
- *Natural Language Processing* (NLP): Le tecnologie di NLP sono utilizzate per estrarre informazioni semantiche dai testi non strutturati presenti nel *Data Lake*. Questo può includere l'elaborazione di linguaggio naturale, l'estrazione di entità, l'analisi dei sentimenti e altre tecniche per comprendere il significato dei testi;
- Database a grafo: sono ampiamente utilizzati per la gestione e l'analisi dei dati semantici all'interno dei *Data Lake*. Questi database consentono

di rappresentare i dati come nodi e relazioni, facilitando la navigazione e l'analisi dei dati basata sulle relazioni semantiche;

In letteratura si evidenziano diversi casi di utilizzo di tecnologie semantiche applicate al mondo *Data Warehouse* e *Data Lake*: Romero et al.[25] definiscono un approccio *user-centered* basato sull'ontologia OWL EU-Car Rental[26] per supportare la progettazione del *Data Warehouse*; Dibowski et al.[2] estendono il loro *Data Lake* aggiungendo un'ontologia e un *knowledge graph* per fornire una descrizione semantica significativa delle risorse; Diamantini et al.[3] definiscono un *Semantic Data Lake* per l'analisi di indicatori aggregati in scenari con cambiamento dinamico delle esigenze informative e coesistenza di un numero elevato e crescente di sorgenti.

Capitolo 3

Definizione delle specifiche

Questo elaborato si propone come un'estensione del *Semantic Data Lake* di Diamantini et al.[3], perciò di seguito si illustra una breve descrizione del *framework* evidenziandone gli aspetti rilevanti per la comprensione del proseguo.

Il *Semantic Data Lake framework* è composto da:

- $S = \{S1, ..., Sn\}$ insieme delle sorgenti, le quali possono essere strutturate o semi-strutturate;
- $G = \{G1, ..., Gn\}$ il relativo insieme dei cataloghi di metadati;
- K il relativo *knowledge graph*;
- $m \subseteq G \times K$ la funzione che associa i metadati ai concetti di conoscenza;

Di seguito, in figura 3.1, si illustra uno schema del *framework*:

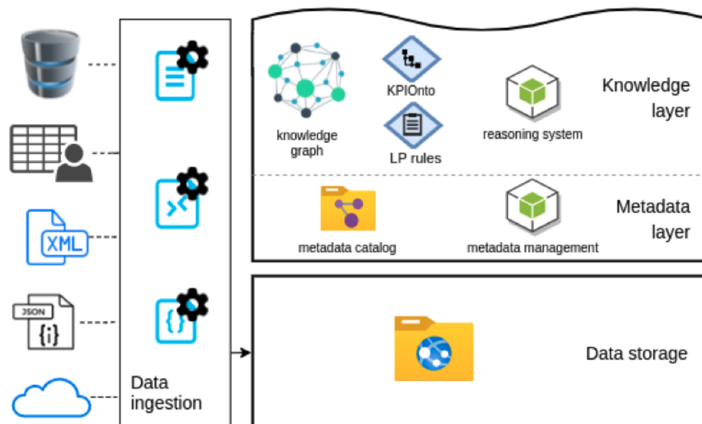

Figura 3.1: Schema del *Semantic Data Lake framework*.

In particolare, si osservano:

- *Metadata layer*: con riferimento alla classificazione in figura 2.6, il *framework* è focalizzato sulla rappresentazione dei metadati tecnici, utilizzando come modello un grafo. Data una sorgente dati S_k:

  – si indica con M_k l'insieme dei suoi metadati, che appartengono ad un catalogo dei metadati;

  – i metadati vengono rappresentati come un grafo diretto $G_k = \langle N_k, A_k, \Omega_k \rangle$ dove N_k sono i nodi, A_k gli archi e $\Omega_k : A_k \rightarrow \Lambda_k$ una funzione che associa ad ogni arco $a \in A_k$ un'etichetta $l \in \Lambda_k$.

  Il grafo viene costruito in maniera incrementale da un sistema di gestione dei metadati, a partire dalla definizione di un nodo $n \in N_k$ per ogni $o \in M_k$. Un arco $(n_x, n_y) \in A_k$ rappresenta la relazione strutturale esistente tra gli oggetti o_x, o_y , ad esempio una relazione tra una tabella e una colonna.

- *Knowledge layer*, composto da:

  – KPIOnto, un'ontologia definita con l'obiettivo di modellare indicatori in termini di descrizione, unità di misura e formula per il loro calcolo; fornisce inoltre classi e relazioni per rappresentare gerarchie multidimensionali.

  – un *knowledge graph* che fornisce una rappresentazione della conoscenza del dominio in termini di definizioni di indicatori, gerarchie dimensionali e membri delle dimensioni; i concetti sono rappresentati come *Linked data* in accordo all'ontologia KPIOnto.

  – Regole di programmazione logica applicate da un *reasoner* per eseguire manipolazioni di formule, risoluzione di equazioni, deduzione di tutte le formule per un dato indicatore e valutazione di tutte le dipendenze di un dato indicatore.

Per ogni sorgente, i nodi dei grafi dei metadati sono semanticamente allineati con i concetti del *knowledge graph* attraverso la definizione di associazioni $\langle c, n \rangle$ dove c denota un concetto ontologico nel *knowledge graph* e n un nodo dei metadati che rappresenta un indicatore o una dimensione nella sorgente. Questo può essere eseguito manualmente o con un approccio semi-automatico basato sulla corrispondenza delle stringhe e sulla somiglianza semantica con i concetti presenti in *knowledge bases* esterne. Di seguito, in figura 3.2, si illustra un esempio di modellazione e allineamento di tre sorgenti dati.

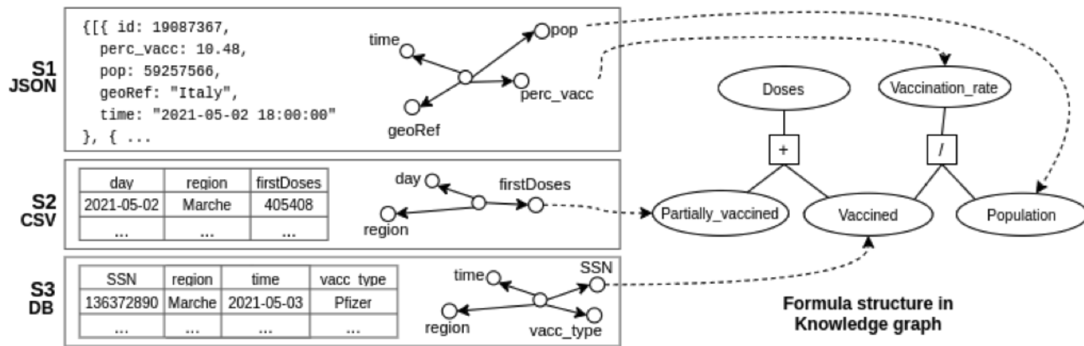

Figura 3.2: Esempio di modellazione e allineamento di sorgenti dati.

3.1 Definizione dei profili

Si definisce *profilo* l'insieme dei metadati che forniscono informazioni sintetiche sui valori di uno specifico dominio di una sorgente dati. Ogni elemento di sintesi è definito come una *proprietà* del profilo stesso.

Per esprimere le proprietà sono stati utilizzati i seguenti vocabolari:

- *rdfs*, definito da W3C, fornisce un vocabolario per la modellazione dei dati RDF ed è un'estensione del vocabolario base di RDF. Vengono utilizzate diverse proprietà, come `type` per esprimere che una risorsa x è un'istanza di una classe X e `Literal` per esprimere che un nodo è una foglia, ovvero contiene dati semplici e non può avere archi uscenti. Il *namespace* viene espresso con il prefisso `rdfs` e indicato con il seguente *URIRef*.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
```

Listato 3.1: URIRef del namespace `rdfs`

- *xsd*: definito da W3C, viene utilizzato per specificare il tipo di dato associato ad un dato semplice, espresso come `Literal`. Il *namespace* viene espresso con il prefisso `xsd` e indicato con il seguente *URIRef*.

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
```

Listato 3.2: URIRef del namespace `xsd`.

- *dcterms*, definito da Dublin Core, è un dizionario standard per descrivere generiche risorse sul web; nel seguente elaborato viene utilizzata la proprietà `date` per associare ad una sorgente la data di caricamento all'interno del *Data Lake*. Il *namespace* viene espresso con il prefisso `dc` e indicato con il seguente *URIRef*.

```
@prefix dcterms <http://purl.org/dc/terms/>.
```

Listato 3.3: URIRef del namespace `dcterms`.

- *void*: definito da W3C, fornisce un vocabolario per esprimere metadati nelle sorgenti dati RDF. Viene utilizzata la proprietà `Dataset` per esprimere che una sorgente dati x è un'istanza della classe *Dataset*. Il *namespace* viene espresso con il prefisso `void` e indicato con il seguente *URIRef*.

```
@prefix void: <http://rdfs.org/ns/void#>.
```

Listato 3.4: URIRef del namespace `void`.

- *KPIOnto*, definito precedentemente, è un dizionario che fornisce classi e relazioni per rappresentare gerarchie multidimensionali nel *knowledge graph*. Il *namespace* viene espresso con il prefisso `kpi` e indicato con il seguente *URIRef*.

```
@prefix kpi: <http://w3id.org/kpionto/> .
```

Listato 3.5: URIRef del namespace `KPIOnto`.

- *dl*, vocabolario interno definito appositamente per la realizzazione del framework, ad esempio per esprimere che una sorgente dati x è un'istanza della classe *Source*. Le proprietà vengono espresse dal prefisso `dl` e indicate con il seguente *URIRef*.

```
@prefix dl: <http://kdmg.dii.univpm.it/datalake/>.
```

Listato 3.6: URIRef del namespace `dl`.

Di seguito si illustra, nella tabella 3.1, le proprietà comuni a tutte le sorgenti dati:

Item	**Domain**	**Property**	**Range**
Posizione nel *file system*	dl:Source	*dl:location*	rdf:Literal
Data di caricamento	dl:Source	*dcterms:date*	rdf:Literal
Numero di domini	dl:Source	*dl:domains*	rdf:Literal
Numero di righe	dl:Source	*dl:items*	rdf:Literal
Tipo	dl:Source dl:Source	*rdf:type* *rdf:type*	dl:Source void:Dataset
Dominio sorgente	dl:Source	*dl:contains*	dl:Domain

Tabella 3.1: Elenco delle proprietà che descrivono una sorgente.

Di seguito, si illustra un esempio sia testuale (nel listato 3.7) che visuale (in figura 3.3) delle proprietà associate ad una sorgente.

```
<http://kdmg.dii.univpm.it/test/file_0> a dl:Source,
        void:Dataset ;
    dl:contains <http://kdmg.dii.univpm.it/test/file_0_D0>,
        <http://kdmg.dii.univpm.it/test/file_0_D1>,
        <http://kdmg.dii.univpm.it/test/file_0_D2>,
        <http://kdmg.dii.univpm.it/test/file_0_D3>,
        <http://kdmg.dii.univpm.it/test/file_0_D4> ;
    dl:domains "5"^^xsd:int ;
    dl:items "7"^^xsd:int ;
    dl:location "./datasets/file.csv"^^xsd:string ;
    dcterms:date "2023-06-26"^^xsd:date .
```

Listato 3.7: Proprietà associate ad una sorgente.

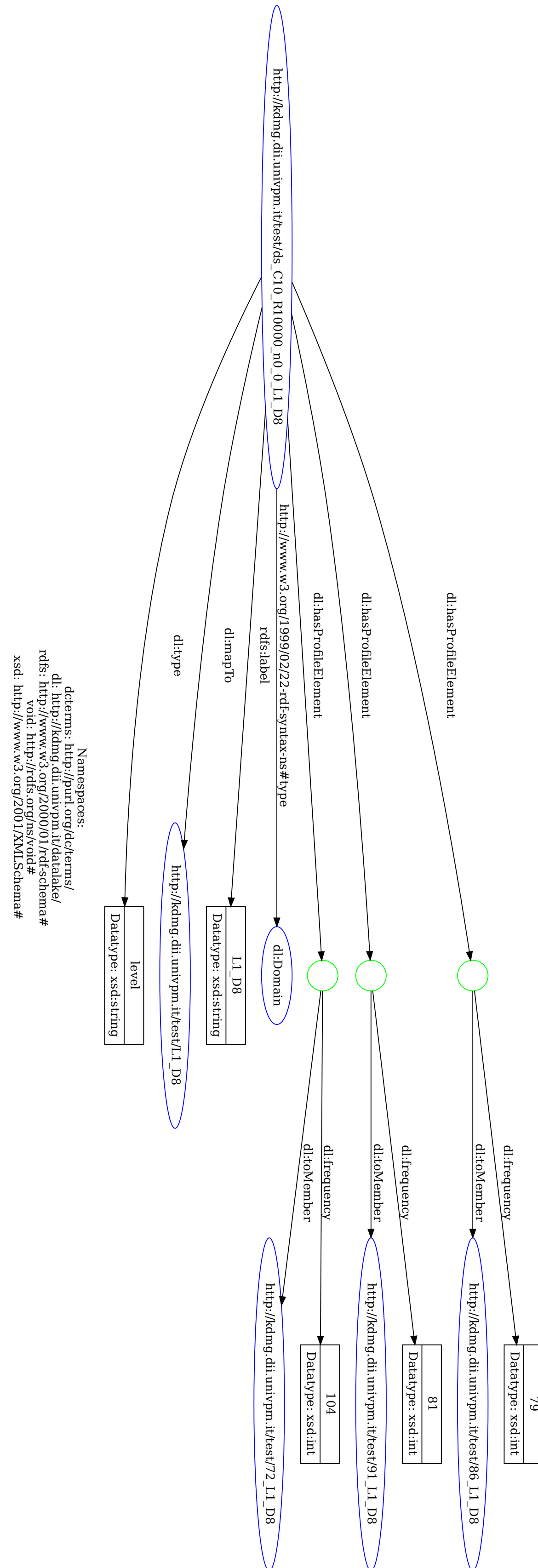

Figura 3.3: Rappresentazione delle proprietà di una sorgente.

Per la profilazione dei domini, si definisce una funzione $\Omega(a) = b$ dove a è una colonna (dominio) di una sorgente S_k, mentre b è un valore categorico tale che $b \in X$ e $X = \{"level", "integer", "real", "categorical", "date", "string", ""\}$. Si osserva che "" viene associato ad un dominio se e solo se l'algoritmo non è in grado di rilevare la tipologia dei valori contenuti nello stesso; in tal caso, il profilo non viene né calcolato né aggiunto nel *grafo dei metadati*.

Di seguito si illustra, nella tabella 3.2, le proprietà comuni a tutte i domini:

Item	**Domain**	**Property**	**Range**
Nome dominio	dl:Domain	*rdfs:label*	rdf:Literal
Tipologia profilo	dl:Domain	*rdf:type*	rdf:Literal
Profilo	dl:Domain	*dl:hasProfileElement*	dl:ProfileElement

Tabella 3.2: Elenco delle proprietà comuni a tutti i domini.

3.1.1 Profilo relativo ai domini associati a livelli del knowledge graph

Il calcolo del profilo relativo ai domini associati ai livelli del *knowledge graph* è parte integrante del *Semantic Data Lake framework*; di seguito, in tabella 3.3, si illustra, per completezza, l'elenco degli elementi che compongono il profilo relativo ai domini associati ai livelli del *knowledge graph*.

Profile item	**Domain**	**Property**	**Range**
Livello associato al knowledge graph	dl:Domain	*dl:mapTo*	dl:ProfileElement
Membro del knowledge graph che occorre nella sorgente	dl:ProfileElement	*dl:toMember*	kpi:level
Numero di occorrenze del precedente	dl:ProfileElement	*dl:frequency*	rdf:Literal
Numero di occorrenze degli elementi del dominio mancanti nel livello del knowledge graph	dl:ProfileElement	*dl:other*	rdf:Literal

Tabella 3.3: Elenco delle proprietà del profilo di categoria *level*.

Di seguito, si illustra un esempio sia testuale (nel listato 3.8) che visuale (in figura 3.4) di profilo associato a un dominio classificato *level*.

```
<http://kdmg.dii.univpm.it/test/ds_C10_R10000_n0_0_L1_D8> a dl:
   Domain;
   rdfs:label "L1_D8"^^xsd:string ;
   dl:hasProfileElement [ dl:frequency "79"^^xsd:int ;
          dl:toMember <http://kdmg.dii.univpm.it/test/86_L1_D8>
             ],
       [ dl:frequency "81"^^xsd:int ;
          dl:toMember <http://kdmg.dii.univpm.it/test/91_L1_D8>
             ],
       [ dl:frequency "104"^^xsd:int ;
          dl:toMember <http://kdmg.dii.univpm.it/test/72_L1_D8> ]
              ;
   dl:mapTo <http://kdmg.dii.univpm.it/test/L1_D8> ;
   dl:type "level"^^xsd:string .
```

Listato 3.8: Esempio di rappresentazione delle proprietà del profilo di categoria *level*.

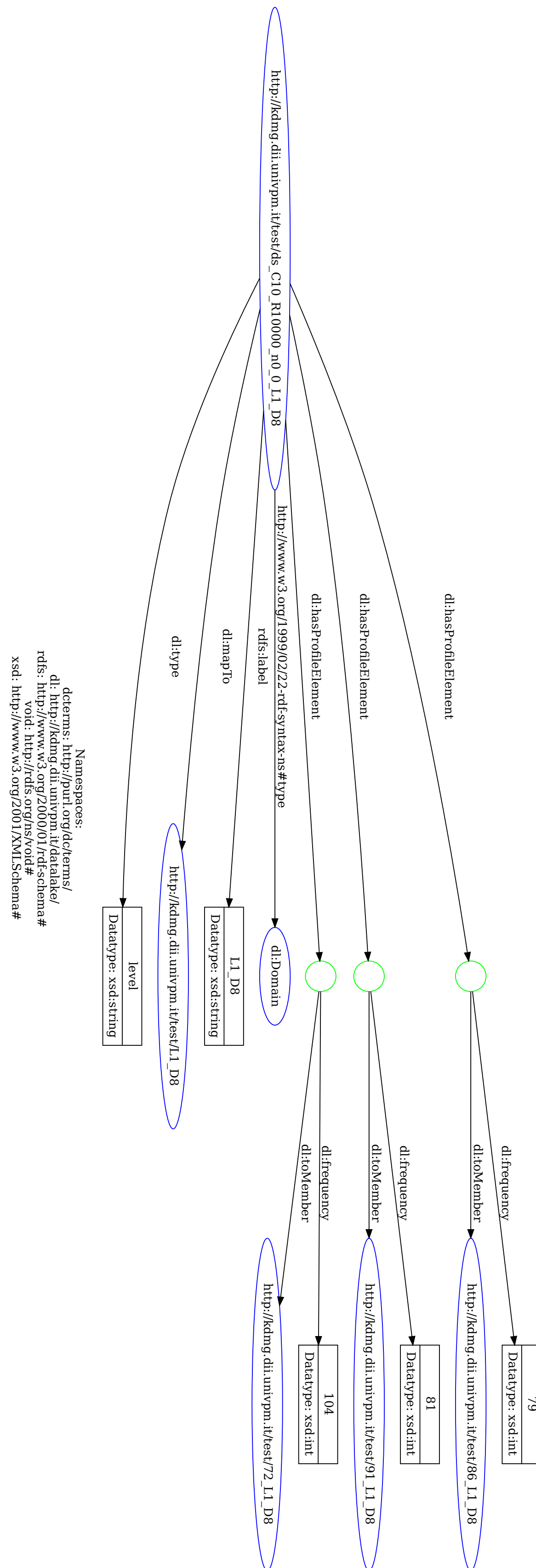

Figura 3.4: Esempio di rappresentazione delle proprietà del profilo di categoria *level*.

3.1.2 Profilo relativo ai domini con valori interi

Di seguito, in tabella 3.4, si illustra l'elenco degli elementi che compongono il profilo relativo ai domini con valori interi.

Profile item	**Domain**	**Property**	**Range**
Somma	dl:ProfileElement	*dl:sum*	rdf:Literal
Valore massimo	dl:ProfileElement	*dl:max*	rdf:Literal
Valore minimo	dl:ProfileElement	*dl:min*	rdf:Literal
Valor medio	dl:ProfileElement	*dl:mean*	rdf:Literal
Mediana	dl:ProfileElement	*dl:median*	rdf:Literal
Conteggio valori distinti	dl:ProfileElement	*dl:distinct*	rdf:Literal
Conteggio valori nulli	dl:ProfileElement	*dl:nr_null*	rdf:Literal
Distribuzione valori	dl:ProfileElement dl:DistributionElement dl:DistributionElement dl:DistributionElement	*dl:distribution* *dl:start_range* *dl:end_range* *dl:count*	dl:DistributionElement rdf:Literal rdf:Literal rdf:Literal

Tabella 3.4: Elenco delle proprietà del profilo di categoria *integer*.

Di seguito, si illustra un esempio sia testuale (nel listato 3.9) che visuale (in figura 3.5) di profilo associato a un dominio classificato *integer*.

```
<http://kdmg.dii.univpm.it/test/file_0_D0> a dl:Domain ;
   rdfs:label "D0"^^xsd:string ;
   dl:hasProfileElement [ dl:max "5446"^^xsd:int ],
       [ dl:sum "18623"^^xsd:int ],
       [ dl:min "581"^^xsd:int ],
       [ dl:median "3005.5"^^xsd:float ],
       [ dl:mean "3103.833"^^xsd:float ],
       [ dl:distribution [ dl:count "0"^^xsd:int ;
                  dl:end_range "4229.75"^^xsd:float ;
                  dl:start_range "3013.5"^^xsd:float ],
              [ dl:count "3"^^xsd:int ;
                  dl:end_range "1797.25"^^xsd:float ;
                  dl:start_range "581.0"^^xsd:float ],
              [ dl:count "0"^^xsd:int ;
                  dl:end_range "3013.5"^^xsd:float ;
                  dl:start_range "1797.25"^^xsd:float ],
              [ dl:count "3"^^xsd:int ;
                  dl:end_range "5446.0"^^xsd:float ;
                  dl:start_range "4229.75"^^xsd:float ] ],
       [ dl:nr_null "1"^^xsd:int ],
       [ dl:distinct "6"^^xsd:int ] ;
   dl:type "integer"^^xsd:string .
```

Listato 3.9: Esempio di rappresentazione delle proprietà del profilo di categoria *integer*.

Figura 3.5: Esempio di rappresentazione delle proprietà del profilo di categoria *integer*.

3.1.3 Profilo relativo ai domini con valori reali

Di seguito, in tabella 3.5, si illustra l'elenco degli elementi che compongono il profilo relativo ai domini con valori reali; la differenza rispetto al caso precedente è l'assenza della componente relativa alla distribuzione dei valori.

Profile item	**Domain**	**Property**	**Range**
Somma	dl:ProfileElement	*dl:sum*	rdf:Literal
Valore massimo	dl:ProfileElement	*dl:max*	rdf:Literal
Valore minimo	dl:ProfileElement	*dl:min*	rdf:Literal
Valor medio	dl:ProfileElement	*dl:mean*	rdf:Literal
Mediana	dl:ProfileElemente	*dl:median*	rdf:Literal
Conteggio valori distinti	dl:ProfileElement	*dl:distinct*	rdf:Literal
Conteggio valori nulli	dl:ProfileElement	*dl:nr_null*	rdf:Literal

Tabella 3.5: Elenco delle proprietà del profilo di categoria *real*.

Di seguito, si illustra un esempio sia testuale (nel listato 3.10) che visuale (in figura 3.6) di profilo associato a un dominio classificato *real*.

```
<http://kdmg.dii.univpm.it/test/file_0_D1> a dl:Domain ;
    rdfs:label "D1"^^xsd:string ;
    dl:hasProfileElement [ dl:max "17.123"^^xsd:float ],
        [ dl:sum "32.725"^^xsd:float ],
        [ dl:min "2.233"^^xsd:float ],
        [ dl:mean "6.545"^^xsd:float ],
        [ dl:nr_null "2"^^xsd:int ],
        [ dl:distinct "4"^^xsd:int ],
        [ dl:median "4.123"^^xsd:float ] ;
    dl:type "real"^^xsd:string .
```

Listato 3.10: Esempio di rappresentazione delle proprietà del profilo di categoria *real*.

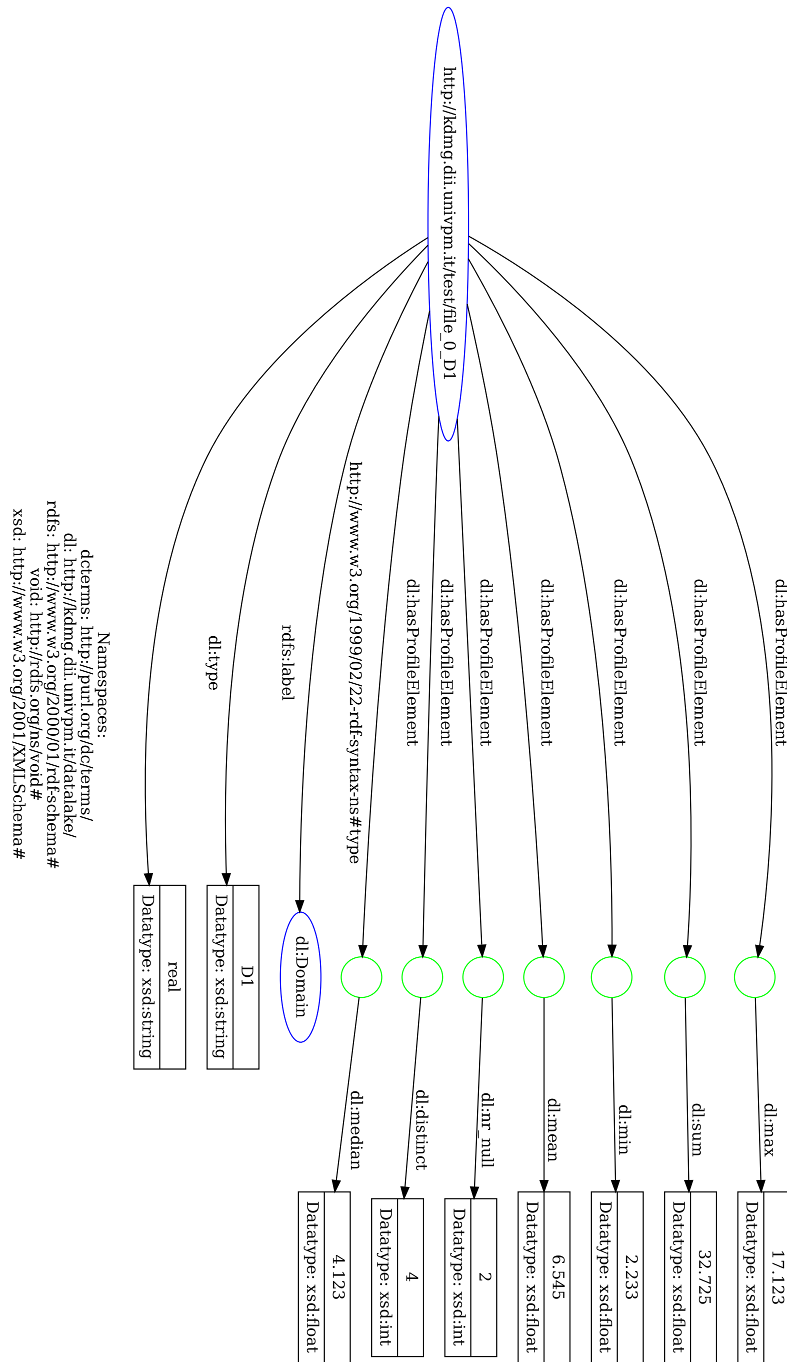

Figura 3.6: Esempio di profilo relativo ad un dominio classificato *real*.

3.1.4 Profilo relativo ai domini con valori categorici

Di seguito, in tabella 3.6, si illustra l'elenco degli elementi che compongono il profilo relativo ai domini con valori categorici.

Profile item	**Domain**	**Property**	**Range**
Conteggio delle occorrenze per ogni categoria	dl:ProfileElement	*dl:category*	dl:CategoryElement
	dl:CategoryElement	*dl:name*	rdf:Literal
	dl:CategoryElement	*dl:count*	rdf:Literal

Tabella 3.6: Elenco delle proprietà del profilo di categoria *categorical*.

Si osserva che:

- il numero di occorrenze tale per cui si classifica un dominio categorico sarà un parametro definito in fase di esecuzione dell'algoritmo;
- il numero di valori nulli è anch'essa una categoria, perciò per poter definire N categorie, in fase di esecuzione dell'algoritmo bisognerà impostare come limite $N + 1$.

Di seguito, si illustra un esempio sia testuale (nel listato 3.11) che visuale (in figura 3.7) di profilo associato a un dominio classificato *categorical*.

```
<http://kdmg.dii.univpm.it/test/file_0_D2> a dl:Domain ;
    rdfs:label "D2"^^xsd:string ;
    dl:hasProfileElement [ dl:category [ dl:name "Human"^^xsd:
        string ;
                dl:value "3"^^xsd:int ],
              [ dl:name "Machine"^^xsd:string ;
                dl:value "3"^^xsd:int ],
              [ dl:name "nan"^^xsd:string ;
                dl:value "1"^^xsd:int ] ] ;
    dl:type "categorical"^^xsd:string .
```

Listato 3.11: Esempio di rappresentazione delle proprietà del profilo di categoria *categorical*.

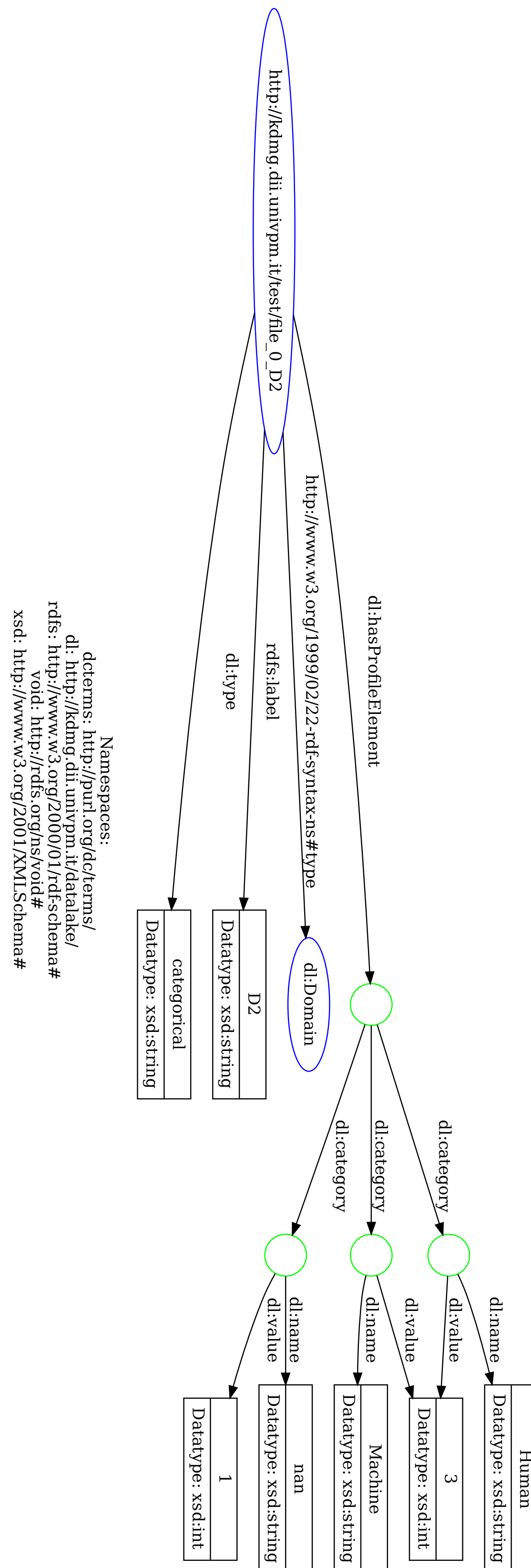

Figura 3.7: Esempio di profilo relativo ad un dominio classificato *categorical*.

3.1.5 Profilo relativo ai domini con valori date

Di seguito, in tabella 3.7, si illustra l'elenco degli elementi che compongono il profilo relativo ai domini con valori date.

Profile item	Domain	Property	Range
Data più recente	dl:ProfileElement	*dl:max*	rdf:Literal
Data meno recente	dl:ProfileElement	*dl:min*	rdf:Literal
Conteggio valori nulli	dl:ProfileElement	*dl:nr_null*	rdf:Literal
Conteggio valori distinti	dl:ProfileElement	*dl:distinct*	rdf:Literal
Distribuzione degli anni	dl:ProfileElement dl:YearElement dl:YearElement	*dl:years* *dl:year* *dl:count*	dl:YearElement rdf:Literal rdf:Literal

Tabella 3.7: Elenco delle proprietà del profilo di categoria *date*.

Di seguito, si illustra un esempio sia testuale (nel listato 3.12) che visuale (in figura 3.8) di profilo associato a un dominio classificato *date*.

```
<http://kdmg.dii.univpm.it/test/file_0_D3> a dl:Domain ;
    rdfs:label "D3"^^xsd:string ;
    dl:hasProfileElement [ dl:min "1998-02-02T00:00:00"^^xsd:
        dateTime ],
        [ dl:max "1998-02-07T00:00:00"^^xsd:dateTime ],
        [ dl:years [ dl:count "6"^^xsd:int ;
                    dl:year "1998"^^xsd:year ] ],
        [ dl:nr_null "1"^^xsd:int ] ;
    dl:type "date"^^xsd:string .
```

Listato 3.12: Esempio di rappresentazione delle proprietà del profilo di categoria *date*.

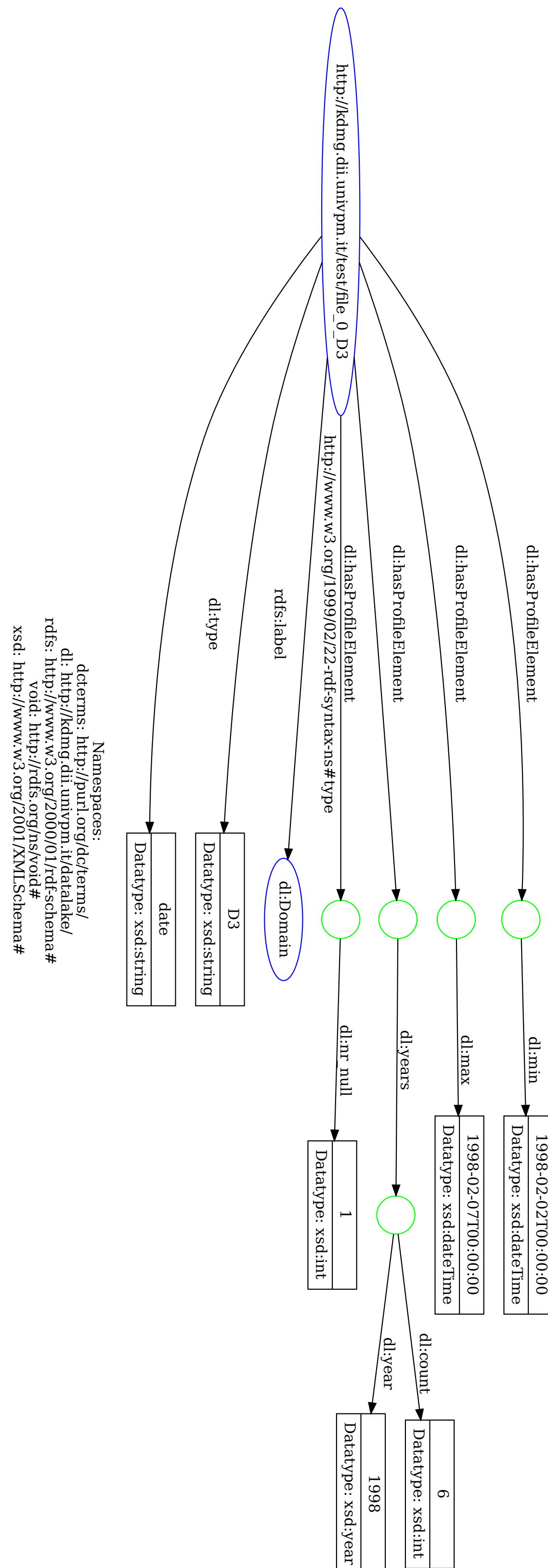

Figura 3.8: Esempio di profilo relativo ad un dominio classificato *date*.

3.1.6 Profilo relativo ai domini con valori stringhe

Di seguito, in tabella 3.6, si illustra l'elenco degli elementi che compongono il profilo relativo ai domini con valori stringhe.

Profile item	**Domain**	**Property**	**Range**
Conteggio valori nulli	dl:ProfileElement	*dl:nr_null*	rdf:Literal
	dl:ProfileElement	*dl:mfw*	dl:WordElement
n parole più frequenti	dl:WordElement	*dl:word*	rdf:Literal
	dl:WordElement	*dl:count*	rdf:Literal

Tabella 3.8: Elenco delle proprietà del profilo di categoria *string*.

Si osserva che il numero di parole più frequenti da estrarre sarà un parametro definito in fase di esecuzione dell'algoritmo; nel caso in cui venisse specificato un numero di parole maggiore rispetto alla dimensione della lista delle parole stesse, l'algoritmo restituirà l'elenco di tutte le parole. Di seguito, si illustra un esempio sia testuale (nel listato 3.13) che visuale (in figura 3.9) di profilo associato a un dominio classificato *string*.

```
<http://kdmg.dii.univpm.it/test/file_0_D4> a dl:Domain ;
    rdfs:label "D4"^^xsd:string ;
    dl:hasProfileElement [ dl:nr_null "1"^^xsd:int ],
        [ dl:mfw [ dl:count "3"^^xsd:int ;
                    dl:word "would"^^xsd:string ],
                [ dl:count "2"^^xsd:int ;
                    dl:word "world"^^xsd:string ],
                [ dl:count "2"^^xsd:int ;
                    dl:word "know"^^xsd:string ],
                [ dl:count "2"^^xsd:int ;
                    dl:word "free"^^xsd:string ],
                [ dl:count "2"^^xsd:int ;
                    dl:word "human"^^xsd:string ] ],
        [ dl:words "40"^^xsd:int ] ;
    dl:type "string"^^xsd:string .
```

Listato 3.13: Esempio di rappresentazione delle proprietà del profilo di categoria *string*

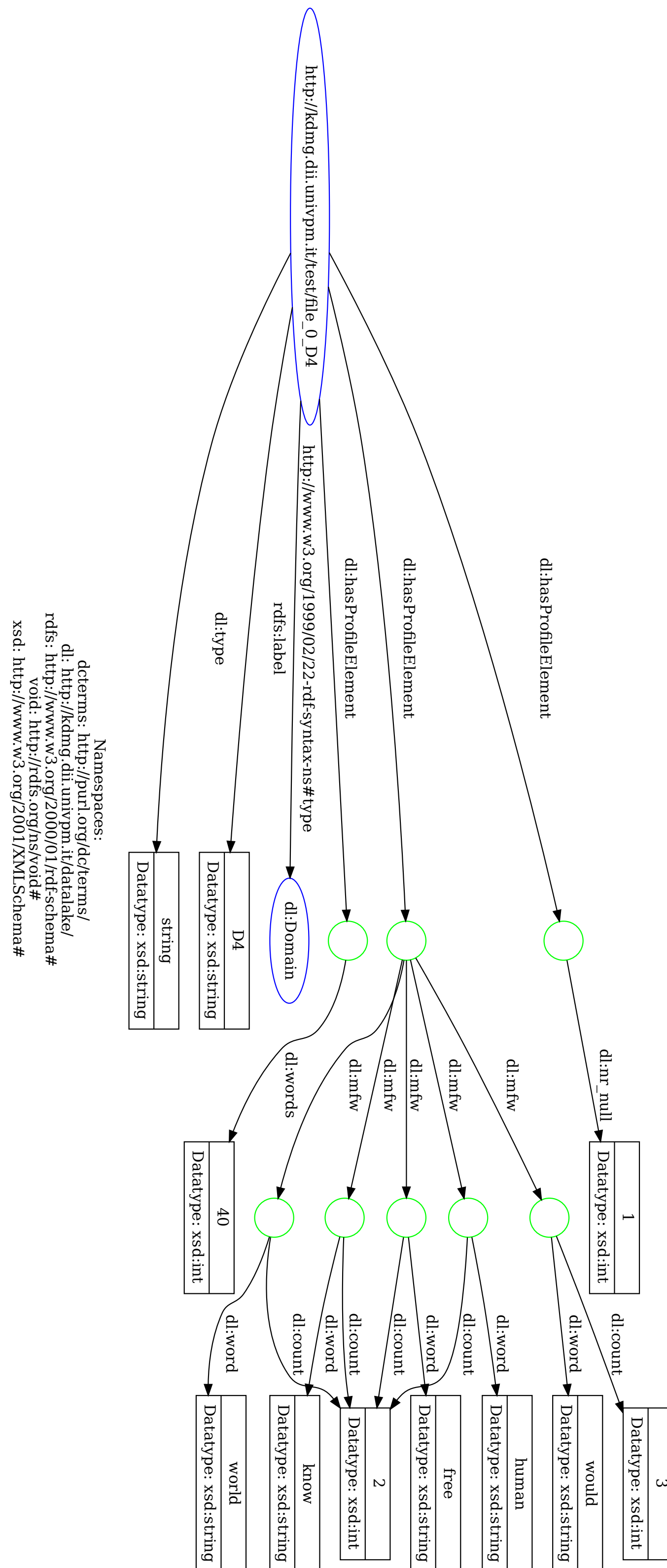

Figura 3.9: Esempio di profilo relativo ad un dominio classificato *string*.

3.2 Calcolo della similarità tra sorgenti

Successivamente alla definizione dei profili, una prima applicazione è, date due sorgenti, calcolare la similarità dei domini associati agli stessi livelli del *knowledge graph*. Una possibile strategia consiste nell'utilizzo della *cosine similarity*, una tecnica ampiamente utilizzata nel *Machine learning* quando si elaborano dati di grandi dimensioni o sparsi, come ad esempio per le somiglianze tra testi e *clustering* di documenti. Il requisito per poter calcolare la *cosine similarity* è l'*embedding* dei dati in input, ossia una trasformazione dal loro formato originale in forma vettoriale, dove i concetti simili sono posizionati vicini l'uno all'altro nello spazio.

La *cosine similarity* è una metrica di similarità utilizzata per confrontare la similarità tra due vettori misurando l'angolo nello spazio geometrico; definiti **t** ed **e** due vettori di dimensione n, allora la *cosine similarity* è definita come:

$$\mathbf{cos}(\mathbf{t},\mathbf{e}) = \frac{\mathbf{te}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{t_i e_i}}{\sqrt{\sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{t_i})^2}\sqrt{\sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{e_i})^2}} \tag{3.1}$$

Il numeratore rappresenta il prodotto scalare tra i vettori **t** ed **e**, mentre il denominatore è il prodotto delle norme di **t** ed **e**, le quali corrispondono alla lunghezza o modulo dei vettori nello spazio. Il risultato della *cosine similarity* tra due vettori è un valore sempre compreso tra -1 e 1:

- se il risultato vale 1, allora i due vettori sono completamente simili;
- se il risultato vale -1, allora i due vettori sono completamente dissimili, ovvero puntano in direzioni opposte;
- se il risultato vale 0, allora i due vettori sono ortogonali e non hanno nessuna somiglianza tra di loro.

Di seguito, in figura 3.10, si illustra un esempio di rappresentazione di due testi nello spazio geometrico, rispettivamente *"Hello, world"* e *"Hi, world!"*.

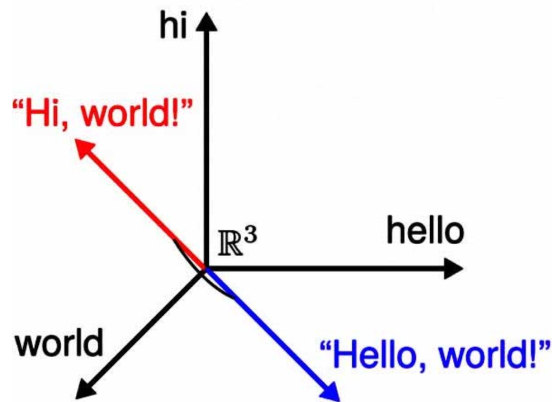

Figura 3.10: Esempio di rappresentazione di due testi nello spazio geometrico, fonte: engati.

La *cosine similarity*:

- è una tecnica euristica;
- non particolarmente costosa dal punto di vista computazionale;

I domini associati agli stessi livelli del *knowledge graph* possono essere confrontati utilizzando la notazione posizionale fornita dallo stesso, inserendo il valore zero in corrispondenza degli elementi del vettore che compaiono nel *knowledge graph* ma non nel dominio della sorgente. Di seguito, in figura 3.11 si illustra un esempio di *embedding* dei vettori per il calcolo della *cosine similarity*.

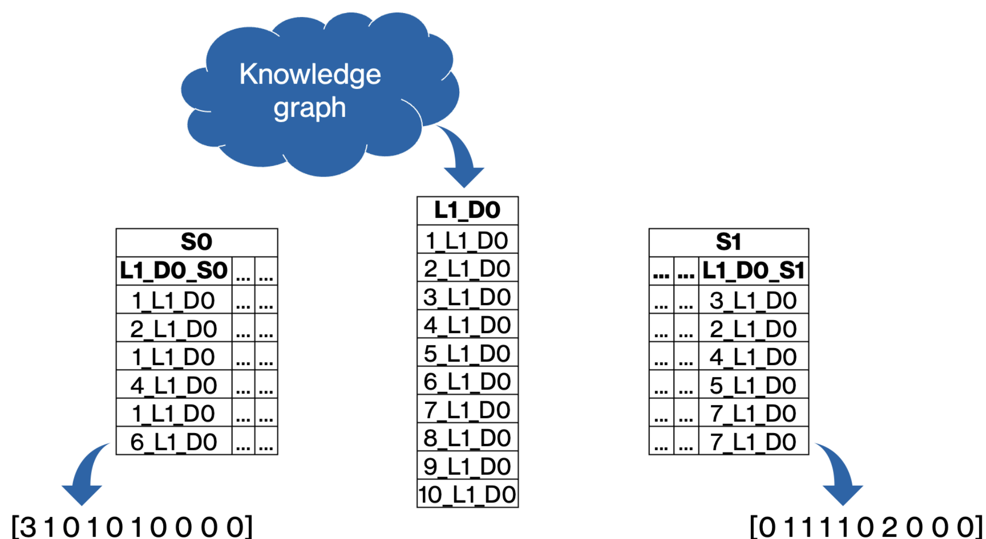

Figura 3.11: Esempio di *embedding* di vettori.

Capitolo 4

Implementazione

4.1 Workflow Per il calcolo del profilo

La procedura `Mount` prende in input il percorso della sorgente che si vuole aggiungere al *Data Lake*, il *metadata graph*, il *knowledge graph* ed il *flag string processing* che abilita o disabilita il processamento dei domini di tipo *string*. In particolare, la procedura si compone dei seguenti passi:

- si estraggono il nome, il numero di righe e la lista delle colonne dal file sorgente;
- Si manipola il nome della sorgente per risolvere eventuali conflitti di sorgenti con lo stesso nome nel *metadata graph*: in particolare, si aggiunge al nome della sorgente il suffisso `_n` dove `n` viene valorizzato in maniera incrementale;
- si calcolano le proprietà della sorgente e si aggiungono al *metadata graph*;
- si itera sulla lista delle colonne estratta precedentemente:
  - Si importa la colonna della sorgente come *Serie pandas*;
  - estraendo i valori contenuti nella colonna del *DataFrame*;
  - si verifica se la colonna è associabile a qualche livello del *knowledge graph*, inserendo i potenziali candidati in una lista;
  - se la lista non è vuota:
    - si estraggono i membri del livello candidato del *knowledge graph*;
    - si calcola il profilo *level* per la colonna in questione;
  - se la lista è vuota:
    - si calcola il profilo opportuno per la colonna in questione;
  - si aggiunge al *metadata graph* il profilo della colonna appena calcolato;

Nell'algoritmo 1 si illustra, mediante pseudo-codice, l'implementazione della procedura `Mount`.

Algoritmo 1 Importazione di una sorgente dati nel *Data Lake*

```
1: function MOUNT(source_path, m_graph, k_graph, string_proc)
2:     source_name, columns, n_rows ← GET_SOURCE_INFO(source_path)
3:     m_graph.ADD_SOURCE(source_name, n_rows, columns)
4:     lsh_ensemble ← INITIALIZE_LSH(k_graph)
5:     for col in columns do
6:         df ← READ_COLUMN(source_name, col)
7:         values ← EXTRACT_VALUES(df)
8:         list_mappings ← MAP_SOURCE_DOMAIN(values, lsh_ensemble)
9:         if list_mappings then
10:            level ← list_mappings[0]
11:            MAP(m_graph, source_name, col, level)
12:            members ← GET_MEMBERS_FROM_LEVEL(k_graph, level)
13:            col_profile, col_type ← CALCULATE_LEV_PROFILE(values, members)
14:        else
15:            col_profile, col_type ← CALCULATE_COLUMN_PROFILE(df, string_proc)
16:        ADD_PROFILE(m_graph, source_name, col, col_type, col_profile)
17:    m_graph.SERIALIZE()
```

L'estensione del *framework* riguarda la funzione `Calculate_column_profile`, la quale ha come parametri in input:

- una colonna (*Serie*) del *DataFrame*;
- il *flag* che abilita o disabilita il processamento delle colonne di tipo *string*.

In particolare:

- si verifica se il tipo di dato (*dtype*) associato alla colonna equivale a *int*:
  - in caso affermativo, si calcola il profilo *integer* e l'algoritmo termina;
- altrimenti, si verifica se equivale al tipo di dato *real*:
  - in caso affermativo, si calcola il profilo *real* e l'algoritmo termina;
- altrimenti:
  - si verifica se la colonna è compatibile con il tipo *categorical*:

    * in caso affermativo, si associa alla colonna il profilo *categorical* e l'algoritmo termina;
  - altrimenti, si verifica se la colonna è compatibile con il tipo *date*:
    * in caso affermativo, si associa alla colonna il profilo *date* e l'algoritmo termina;
  - altrimenti, se il *flag* per il processamento delle stringhe è abilitato:
    * si associa alla colonna un profilo *String* e l'algoritmo termina;
  - altrimenti, l'algoritmo termina senza associare alla colonna in questione un profilo.

4.2 Calcolo dei profili

La componente relativa al calcolo dei profili è stata realizzata utilizzando le librerie:

- *pandas*, per il calcolo degli elementi dei profili e l'importazione delle sorgenti nelle strutture dati *DataFrame* e *Serie*;

- *collections*, per contare, tramite l'oggetto *Counter*, le occorrenze degli elementi in una sequenza, come una lista o una stringa;

- *rdflib*, per l'accesso e la manipolazione al *knowledge graph* ed al *metadata graph*;

- *numpy*, per il calcolo della distribuzione dei valori interi e l'ottimizzazione della memoria, ove possibile, utilizzando la struttura dati *Array* anziché il tipo primitivo *List* fornito dal linguaggio stesso.

La categoria di profilo da associare ad un dominio viene scelta tenendo conto di alcune limitazioni delle librerie utilizzate, poiché *pandas*, in fase di importazione di una sorgente, associa ad ogni colonna del *DataFrame* una proprietà (`dtype`) che identifica il *super tipo* tra tutti i valori presenti nella stessa. I `dtype` messi a disposizione dalla libreria, quando si importa un file esterno, sono *integer, float* e *object*; ciò potrebbe essere un problema, poiché, supponendo di avere una colonna composta da tutti valori interi ad eccezione di uno nullo (`NaN`, il cui `dtype` è *float*), il meccanismo assegnerebbe a tale colonna un tipo *float*; con conseguente calcolo del profilo *real*. Un'altra osservazione è che, basandosi sul `dtype`, non è possibile distinguere tra domini *date* e *string*; per cui verranno definite delle funzioni personalizzate per soddisfare queste esigenze.

Nell'algoritmo 2 si illustra, mediante pseudo-codice, l'implementazione della funzione `Calculate_column_profile`.

Algoritmo 2 Funzione per il calcolo del profilo

```
1: function CALCULATE_COLUMN_PROFILE(col, string_processing, perc_limit)
2:     if col.dtype = int then
3:         column_info, column_type ← INTEGER_COLUMN(col)
4:         return column_info, column_type
5:     else if col.dtype = float then
6:         column_info, column_type ← REAL_COLUMN(col)
7:         return column_info, column_type
8:     else
9:         try
10:            column_info ← CATEGORICAL_COLUMN(col)
11:            column_type ← "categorical"
12:            return column_info, column_type
13:        except CategoriesReachedException as e:
14:            PRINT(e)
15:        try
16:            column_info ← DATE_COLUMN(col)
17:            column_type ← "date"
18:            return column_info, column_type
19:        except TresholdReachedException as e:
20:            PRINT(e)
21:        if string_processing then
22:            column_info ← STRING_COLUMN(column)
23:                column_type ← "string"
24:            return column_info, column_type
25:    return {},""
```

Per il calcolo del profilo *integer* si utilizzano le funzioni messe a disposizione dalla libreria *pandas*, le quali garantiscono, tramite una serie di ottimizzazioni, complessità computazionali dell'ordine $nlog(n)$. L'unica eccezione si ha per il calcolo della distribuzione, in quanto *pandas* permette la visualizzazione grafica ma senza un agevole accesso ai dati della stessa; dunque si è optato per l'utilizzo della funzione `histogram` di *numpy*, il cui risultato viene manipolato trasformandolo in un dizionario con *key* il range della distribuzione e *value* il numero di occorrenze all'interno del *bin*. Nell'algoritmo 3 si illustra, mediante pseudo-codice, l'implementazione del profilo *integer*.

Algoritmo 3 Profilo per colonne *integer*

```
1:  function INTEGER_COLUMN(column)
2:      column_type ← "integer"
3:      profile ← {}
4:      profile["sum"] ← column.sum()
5:      profile["max"] ← column.max()
6:      profile["min"] ← column.min()
7:      profile["mean"] ← round(column.mean(), 3)
8:      profile["median"] ← round(column.median(), 3)
9:      profile["distinct"] ← column.nunique()
10:     profile["nr_null"] ← column.isna().sum()
11:     count, limits ← histogram(column, bins="auto")
12:     profile["distribution"] ← MANIPULATE_DISTRIBUTION(count, limits)
13:     return profile, column_type
```

Il calcolo del profilo *real* è analogo al caso *integer*, ad eccezione dell'assenza della componente relativa alla distribuzione;

Il problema dovuto al `dtype` precedentemente illustrato viene risolto contando le occorrenze degli elementi nulli nella colonna:

- se vale zero, si procede con il calcolo del profilo *real*;

- altrimenti, si eliminano i valori nulli e si verifica se la colonna contiene tutti valori interi:

  - in caso affermativo, il profilo viene arricchito con la componente relativa alla distribuzione, ottenendo un profilo equivalente al caso *integer*;

  - altrimenti, non aggiungendo la componente della distribuzione, si ottiene il profilo *real*.

Nell'algoritmo 4 si illustra, mediante pseudo-codice, l'implementazione del profilo *real*.

Algoritmo 4 Profilo per colonne *real*

```
1: function REAL_COLUMN(column)
2:     profile ← {}
3:     profile["sum"] ← column.sum()
4:     profile["max"] ← column.max()
5:     profile["min"] ← column.min()
6:     profile["mean"] ← round(column.mean(), 3)
7:     profile["median"] ← round(column.median(), 3)
8:     profile["distinct"] ← column.nunique()
9:     profile["nr_null"] ← column.isna().sum()
10:    if profile["nr_null"] == 0 then
11:        column_type ← "real"
12:    else
13:        column.dropna(inplace=True)
14:        column_type ← CHECK_IF_INTEGER(count, limits)
15:    if column_type == "integer" then
16:        count, limits ← histogram(column, bins="auto")
17:        profile["distribution"] ← MANIPULATE_DISTRIBUTION(count, limits)
18:    return profile, column_type
```

Il calcolo del profilo *categorical* avviene tramite la funzione predefinita di *pandas* `value_counts()`, la quale permette di creare un dizionario dove le chiavi sono i nomi delle categorie ed i valori il numero di occorrenze. Una volta creato il dizionario, si verifica se il numero di chiavi (potenziali categorie) è inferiore alla soglia prestabilita; in caso affermativo alla colonna viene associato un profilo *categorical*, altrimenti l'algoritmo termina. Si osserva che il valore `NaN` viene espresso anch'esso come una categoria; perciò per identificare N categorie, il parametro *max_categories* dovrebbe valere $N + 1$. Nell'algoritmo 5 si illustra, mediante pseudo-codice, l'implementazione del profilo *categorical*.

Algoritmo 5 Profilo per colonne *categorical*

```
1: function CATEGORICAL_COLUMN(column, max_categories)
2:     counter ← column.value_counts(dropna=False)
3:     if len(counter) < max_categories then
4:         return counter
5:     else
6:         raise CategoriesReachedException("Categories limit reached.")
```

Come precedentemente illustrato, in fase di importazione di una sorgente, *pandas* associa alle colonne che non contengono valori numerici il tipo *Object*, non distinguendo tra valori *date* e *string*; si rende dunque necessaria una soglia

tale per cui, oltre un certo numero di tentativi falliti di estrazione della data, si interrompe il calcolo e inizia il processamento del profilo *string*, se abilitato. Il processamento inizia con il conteggio dei valori nulli e la ridefinizione della variabile *column*, iterando su tutti gli elementi della colonna e convertendo in oggetti `dateTime` i valori compatibili e sostituendo con `NaN` tutti gli elementi la cui conversione fallisce. Dunque, se il numero di conversioni fallite supera la soglia, allora il processamento si interrompe; altrimenti, si procede con il calcolo della distribuzione degli anni e l'estrazione della data più recente e meno recente. Nell'algoritmo 6 si illustra, mediante pseudo-codice, l'implementazione del profilo *date*.

Algoritmo 6 Profilo per colonne *Date*

```
1: function DATE_COLUMN(column, perc_limit)
2:     limit_exception ← int(perc_limit × len(column))
3:     nr_null ← column.isna().sum()
4:     column ← to_datetime()
5:     if (column.isna().sum() − nr_null) > limit_exception then
6:         raise TresholdReachedException("Date limit reached.")
7:     profile[""nr_null""] ← nr_null
8:     profile["max"] ← column.max()
9:     profile["min"] ← column.min()
10:    profile["years"] ← EXTRACT_YEAR(column)
11:    return profile
```

Il processamento delle colonne *string* avviene, ad eccezione del conteggio delle occorrenze dei valori nulli, definendo una *Lambda function* dove, per ogni elemento di tipo testo nella colonna, si aggiorna il `counter` applicando una *pipeline* all'elemento testuale, la quale:

- separa il testo in parole usando come delimitatore lo spazio;
- filtra le parole rimuovendo le *stopwords* fornite dal modulo *nltk*;
- filtra le parole rimuovendo tutti i caratteri che non sono alfanumerici;

Successivamente, si aggiungono come elementi del profilo il numero di parole filtrate e le n parole più frequenti, specificando l'occorrenza per ciascuna di esse. Nell'algoritmo 7 si illustra, mediante pseudo-codice, l'implementazione del profilo *string*.

Algoritmo 7 Profilo per colonne *string*

```
1: function STRING_COLUMN(column, max_items)
2:     profile ← {}
3:     profile["nr_null"] ← column.isna().sum()
4:     counter ← FILTER_TEXT(column)
5:     profile["words"] ← len(counter.items())
6:     if profile["words"] > max_items then
7:         profile["mfw"] ← EXTRACT_N_WORDS(column, max_items)
8:     else
9:         profile["mfw"] ← counter
10:    return profile
```

4.3 Calcolo della similarità

La componente relativa al calcolo della similarità è stata realizzata utilizzando le librerie:

- *scipy*: per il calcolo della *cosine similarity*;
- *rdflib*: per la manipolazione dei dati estratti dal grafo dei metadati;
- *numpy*: per ottimizzare l'occupazione di memoria, ove possibile, utilizzando la struttura dati *array* anziché il tipo primitivo *List* fornito dal linguaggio stesso.

La procedura `Estimate_similarity` prende in input una stringa che contiene, separate da una virgola, i nomi delle sorgenti di cui si vuole calcolare la similarità, il *metadata graph* ed il *knowledge graph*:

- si verifica se la stringa un input contiene due sorgenti:
  - in caso affermativo, si prosegue;
  - altrimenti, viene generata un'eccezione;
- si verifica se le due sorgenti hanno un profilo associato nel *metadata graph*:
  - in caso affermativo, si prosegue;
  - altrimenti, viene generata un'eccezione;
- si verifica se le due sorgenti hanno dei domini associati ai livelli del *knowledge graph*:
  - in caso affermativo, si prosegue;
  - altrimenti, viene generata un'eccezione;
- si itera sulle coppie $(d_S_0, l_S_0_kg)$ della prima sorgente, dove d_S_0 è il dominio della sorgente, mentre $l_S_0_kg$ è il livello del *knowledge graph* a cui è associato d_S_0:
  - si itera sulle coppie $(d_S_1, l_S_1_kg)$ della seconda sorgente, dove d_S_1 è il dominio della sorgente, mentre $l_S_1_kg$ è il livello del *knowledge graph* a cui è associato d_S_1:
    * se $l_S_0_kg$ e $l_S_1_kg$ fanno riferimento allo stesso livello del *knowledge graph*:
      · si estraggono i membri del livello dal *knowledge graph*;
      · si estraggono i membri dei domini d_S_0, d_S_1 dal *metadata graph*;

· si fa *embedding* dei membri di d_S_0, d_S_1 estratti precedentemente;

· si calcola la *cosine similarity* dei due *embedding* calcolati precedentemente.

Nell'algoritmo 8 si illustra, mediante pseudo-codice, l'implementazione della procedura per il calcolo della similarità.

Algoritmo 8 Algoritmo per il calcolo della similarità

```
1: function ESTIMATE_SIMILARITY(sources, m_graph, k_graph)
2:     if CHECK_SOURCES_IN_KGRAPH(source_0, source_1) then
3:         mapped_domains ← EXTRACT_MAPPED_DOMAINS(sources)
4:         if mapped_domains_s0 and mapped_domains_s1 then
5:             for (domain_s0, kg_level_s0) in mapped_domains_s0 do
6:                 for (domain_s1, kg_level_s1) in mapped_domains_s1 do
7:                     if kg_level_s0 = kg_level_s1 then
8:                         kg_members ←
                           GET_MEMBERS_FROM_LEVEL(kg_level_s0)
9:                         level_members ← GET_LEVEL_PROFILE(domain_s0,
                           domain_s1)
10:                        embedded_profiles ←
                           GET_EMBEDDING(level_members_s0,
                           level_members_s1)
11:                        similarity ← CALCULA-
                           TE_SIMILARITY(embedded_profiles)
```

La funzione `Get_Embedding`:

- prende in input:
  - un dizionario, contenente come chiave gli elementi del dominio di una sorgente e come valore il numero di occorrenze;
  - i membri del livello del *knowledge graph* a cui il dominio è associato.
- restituisce in output l'*embedding* del dominio passato come parametro in input.

L'algoritmo opera nel seguente modo:

- si inizializza una variabile *embedding* come un *numpy array* vuoto;
- si ordina la lista dei membri del livello del *knowledge graph*;
- si itera sugli elementi della lista ordinata precedentemente:

- se l'elemento è presente nelle chiavi del dizionario che definiscono gli elementi del dominio:
  * si aggiunge al vettore il numero di occorrenze (valore del dizionario) dell'elemento alla variabile *embedding*.
  * altrimenti, si aggiunge al vettore *embedding* il valore zero.

Nell'algoritmo 9 si illustra, mediante pseudo-codice, l'implementazione della procedura di *embedding* per un dominio di una sorgente.

Algoritmo 9 Calcolo *embedding* per dominio di una sorgente

```
1: function GET_EMBEDDING(level_members, kg_members)
2:     embedding ← Array([])
3:     for member in SORTED(kg_members) do
4:         if member in KEYS(level_members) then
5:             embedding ← APPEND(embedding, level_members[member])
6:         else
7:             embedding ← APPEND(embedding, 0)
8:     return embedding
```

La funzione `Calculate_similarity` prende in input due *embedding* e sottrae, al valore 1, il risultato della funzione `spatial.distance.cosine` con parametri gli *embedding*. Nell'algoritmo 10 si illustra, mediante pseudo-codice, l'implementazione della procedura di calcolo della *cosine similarity* tra due *embedding*.

Algoritmo 10 Calcolo della similarità tra due *embedding*

```
1: function CALCULATE_SIMILARITY(embedding1, embedding2)
2:     similarity ← 1 − SPATIAL.DISTANCE.COSINE(embedding1, embedding2)
3:     return similarity
```

Capitolo 5

Valutazione delle performance

Le prestazioni dell'algoritmo realizzato sono state valutate in due fasi, concentrandosi dapprima sulla componente relativa al calcolo del profilo, per poi concludere con la componente relativa al calcolo della similarità tra due sorgenti.

I test sono stati entrambi ripetuti cinque volte su un calcolatore MacBook Pro ARM M1 Pro con CPU a 8 core e memoria unificata da 16 GB.

5.1 Valutazione del calcolo dei profili

Per poter valutare le prestazioni del sistema relativamente al calcolo del profilo di una sorgente, sono state generate delle sorgenti dati in formato *csv* composte da un numero variabile di colonne (10, 20, 30, 40, 50) e righe (10^4, 10^5, 10^6): il numero di domini associati a una particolare categoria di valori è un parametro definito in fase di generazione delle sorgenti, in accordo con la tabella 5.1:

Tipologia di domini	**Percentuale**
Level	20
Integer	30
Real	10
Date	10
Categorical	10
String	20

Tabella 5.1: Percentuale della tipologia di domini per ogni sorgente.

Si avranno dunque, ipotizzando una sorgente dati formata da 10 colonne: 2 domini *Level* e *String*, 3 *Integer* e 1 *Real*, *Date* e *Categorical*.

Le prestazioni sono state valutate calcolando, per cinque iterazioni, i profili delle sorgenti dati ordinate alfabeticamente in ordine ascendente per nome delle stesse.

Di seguito, in figura 5.1, si illustra il tempo medio di esecuzione del calcolo del profilo al variare sia della cardinalità della sorgente, espressa nell'asse x che del numero di domini, espresso come colore nel grafico stesso.

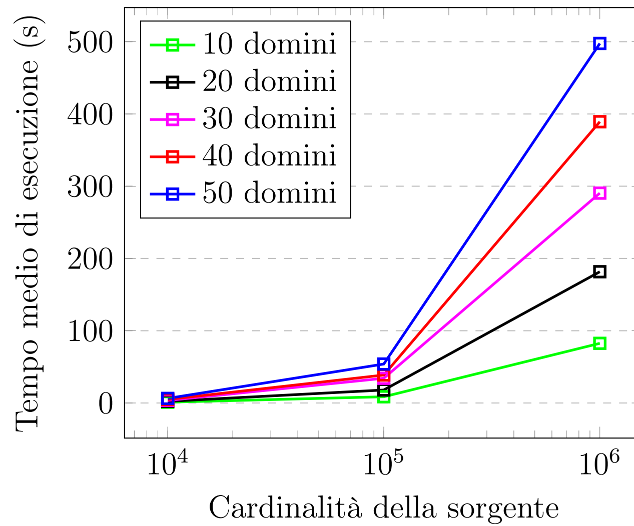

Figura 5.1: Tempo medio calcolo del profilo di una sorgente al variare della cardinalità della stessa.

Come osservabile in figura 5.1, il calcolo del profilo di una sorgente ha una complessità computazionale lineare con la cardinalità della sorgente stessa.

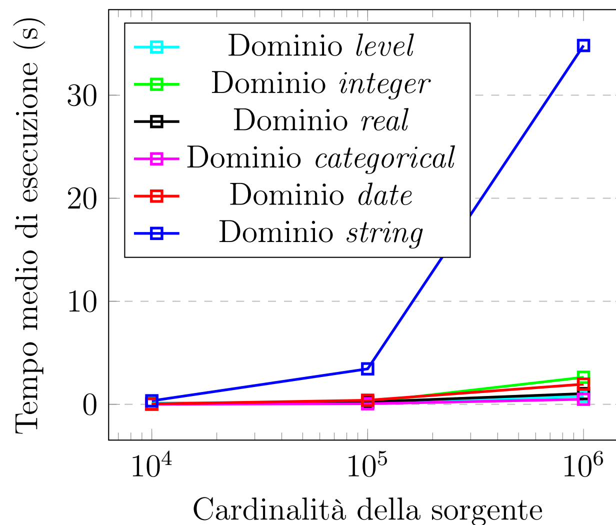

Figura 5.2: Tempo medio calcolo del profilo per singola categoria al variare della cardinalità della sorgente.

Come osservabile in figura 5.2, il calcolo del profilo relativo ad uno specifico dominio ha una complessità computazionale lineare con la cardinalità della sorgente stessa. Si osserva inoltre che il dominio *string*, nonostante la linearità, presenta un ordine di grandezza superiore rispetto agli altri; ciò accade poiché la *pipeline* di processamento, come osservato nella sezione precedente, è piuttosto onerosa.

5.2 Valutazione del calcolo della similarità

Sono stati generati:

- un *knowledge graph* composto da un'unica dimensione con cinque livelli, dove il primo avrà 10^1 membri, il secondo 10^2 membri, fino ad arrivare al quinto composto da 10^5 membri.

- sorgenti dati composte da 10^5, 10^6, 10^7 righe e cinque colonne, ognuna delle quali è associata ad uno specifico livello della dimensione del *knowledge graph*; gli elementi vengono estratti in maniera casuale.

Di seguito, nella tabella 5.2 si illustra un esempio di sorgente di test per il calcolo della similarità:

Dominio	**Livello associato**	**Numero potenziali membri per livello**
D0	L0_D0	10^1
D1	L1_D0	10^2
D2	L2_D0	10^3
D3	L3_D0	10^4
D4	L4_D0	10^5

Tabella 5.2: Esempio di dataset di test per il calcolo della similarità.

In particolare, il test è stato eseguito:

- generando il *knowledge graph*;

- importando la sorgente dati e calcolando il profilo relativo ai domini associati ai livelli del *knowledge graph*;

- calcolando la *cosine similarity* confrontando il profilo precedentemente calcolato con un altro analogo.

Di seguito, in figura 5.3, si illustra il tempo medio di calcolo della similarità tra domini associati agli stessi livelli del *knowledge graph*:

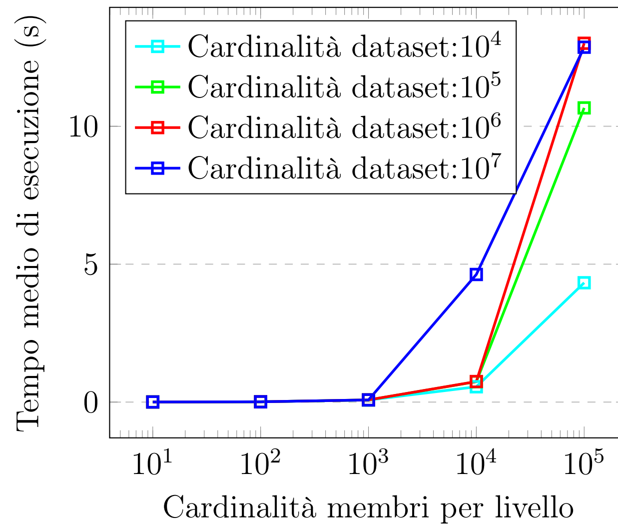

Figura 5.3: Tempo medio calcolo della similarità tra domini associati agli stessi livelli del *knowledge graph*.

Come osservabile in figura 5.3, si ottiene un tempo medio di esecuzione pressoché costante nell'intervallo $[10^1, 10^3]$ membri per livello; nell'intervallo $(10^3, 10^5]$ si nota un comportamento lineare rispetto alla cardinalità dei membri del livello, dove il coefficiente angolare è funzione della cardinalità della sorgente stessa.

Capitolo 6

Conclusioni e sviluppi futuri

Il seguente elaborato ha illustrato la definizione e lo sviluppo di un sistema di gestione dei metadati in ambito *Data Lake* dotato di uno strato semantico che arricchisce e descrive semanticamente il contenuto delle sorgenti. Si è descritta dapprima l'architettura del *Semantic Data Lake framework* di Diamantini et al.[3], di cui questo elaborato si propone come estensione, per proseguire illustrando le proprietà associate ad una sorgente ed il profilo relativo ad un dominio associato ad un livello nel *knowledge graph*. Successivamente, si è arricchito il sistema di gestione dei metadati già esistente definendo, oltre al profilo per i domini associati ai livelli del *knowledge graph*, dei profili di categoria *integer*, *real*, *categorical*, *date* e *string*; per ognuno di essi sono state illustrate le proprietà definite, la metodologia di calcolo, con le relative problematiche dovute agli strumenti utilizzati, e la loro modalità di rappresentazione nel *metadata graph*. Si è quindi ottenuto, complessivamente, un sistema di gestione dei metadati che, data in input una sorgente:

- aggiunge le proprietà della sorgente stessa nel *metadata graph*;
- calcola, per ogni dominio, il profilo più opportuno in base alla tipologia dei valori contenuti all'interno dello stesso e lo aggiunge al *metadata graph*;

Sono state poi valutate le prestazioni del calcolo del profilo e si è osservato che presenta una complessità lineare con la cardinalità della sorgente. Un ulteriore risultato ottenuto è la definizione di una componente per calcolare la similarità tra due sorgenti, tramite l'utilizzo del profilo, confrontando le proprietà dei domini associati agli stessi livelli del *knowledge graph*.

Possibili sviluppi futuri includono il collaudo del *framework* verificando l'efficienza e l'efficacia su sorgenti di dati reali. Inoltre, il profilo potrebbe essere utilizzato per fornire una misura complessiva di somiglianza tra le sorgenti, anziché un indicatore valido solo per coppie di colonne associate agli stessi livelli del *knowledge graph*. Ciò può essere utile per supportare diversi *task*, tra cui *dataset discovery*, *data exploration* e *query discovery*.

6.1 Ringraziamenti

Oggi, al termine di questo percorso universitario, desidero ringraziare tutte le persone che hanno contribuito a far sì che tutto ciò potesse avverarsi.

Ringrazio la mia famiglia: nonostante non sia una persona loquace, sapere di avere qualcuno che ti sostiene, nonostante il poco tempo passato assieme, è una sicurezza in più rispetto alla variabilità di tutto il resto.

Ringrazio Sofia: sei stata l'unica persona a cui ho mostrato tutte le mie fragilità, sei riuscita a comprenderle e agire di conseguenza senza che dicessi mai una parola; con la tua dolcezza, e soprattutto pazienza, sei riuscita a rendere meno ostico questo percorso.

Ringrazio Davide: l'unica ancora in questi cinque anni di percorso universitario; sempre disponibile nel momento del bisogno e, spesso e volentieri, mettendo al primo posto gli altri piuttosto che te stesso.

Ringrazio i compagni di corso ed i coinquilini: grazie per aver addolcito questi due anni, alternando sessioni di studio con momenti di svago e divertimento: li porterò sempre con me.

Ringrazio i miei amici delle scuole superiori e del percorso di Laurea Triennale: nonostante non siano state molte le occasioni per rivederci, quelle poche sono bastate per avere la certezza che non è mai cambiato nulla.

Infine, ringrazio i Professori Domenico Potena ed Emanuele Storti, per avermi concesso la possibilità di svolgere il seguente elaborato sotto la loro guida e supervisione, fornendomi costante supporto e disponibilità per ogni evenienza.

Bibliografia

[1] Pwint Phyu Khine and Zhao Shun Wang. Data lake: a new ideology in big data era. In *ITM web of conferences*, volume 17, page 03025. EDP Sciences, 2018.

[2] Henrik Dibowski, Stefan Schmid, Yulia Svetashova, Cory Henson, and Tuan Tran. Using semantic technologies to manage a data lake: Data catalog, provenance and access control. In *SSWS@ ISWC*, pages 65–80. Athen, 2020.

[3] Claudia Diamantini, Domenico Potena, and Emanuele Storti. A semantic data lake model for analytic query-driven discovery. In *The 23rd International Conference on Information Integration and Web Intelligence*, pages 183–186, 2021.

[4] William H Inmon. *Building the data warehouse*. John wiley & sons, 2005.

[5] Sang-Woo Han. Three-tier architecture for sentinel applications and tools: separating presentation from functionality. Master's thesis, Citeseer, 1997.

[6] Athira Nambiar and Divyansh Mundra. An overview of data warehouse and data lake in modern enterprise data management. *Big Data and Cognitive Computing*, 6(4):132, 2022.

[7] Hari Mailvaganam. Introduction to olap-slice, dice and drill. *Data Warehousing Review*, 2007.

[8] Edgar F Codd, Sharon B Codd, and Clynch T Salley. Providing olap (on-line analytical processing) to user-analysts. *An IT Mandate. White Paper. Arbor Software Corporation*, 4, 1993.

[9] Jian Xu, Yong Qiang Luo, and Xin Xin Zhou. Solution for data growth problem of molap. In *Applied Mechanics and Materials*, volume 321, pages 2551–2556. Trans Tech Publ, 2013.

[10] Natalia Miloslavskaya and Alexander Tolstoy. Big data, fast data and data lake concepts. *Procedia Computer Science*, 88:300–305, 2016.

[11] Pegdwendé Sawadogo, Tokio Kibata, and Jérôme Darmont. Metadata management for textual documents in data lakes. *arXiv preprint arXiv:1905.04037*, 2019.

[12] Cedrine Madera and Anne Laurent. The next information architecture evolution: the data lake wave. In *Proceedings of the 8th international conference on management of digital ecosystems*, pages 174–180, 2016.

[13] Bill Inmon. *Data Lake Architecture: Designing the Data Lake and avoiding the garbage dump*. Technics publications, 2016.

[14] Corinna Giebler, Christoph Gröger, Eva Hoos, Holger Schwarz, and Bernhard Mitschang. Leveraging the data lake: Current state and challenges. In *Big Data Analytics and Knowledge Discovery: 21st International Conference, DaWaK 2019, Linz, Austria, August 26–29, 2019, Proceedings 21*, pages 179–188. Springer, 2019.

[15] Alice LaPlante. *Architecting data lakes*. O'Reilly Media, 2016.

[16] Andrew Oram. *Managing the Data Lake: Moving to Big Data Analysis*. O'Reilly Media, 2015.

[17] Claudia Diamantini, Paolo Lo Giudice, Domenico Potena, Emanuele Storti, and Domenico Ursino. An approach to extracting topic-guided views from the sources of a data lake. *Information Systems Frontiers*, 23:243–262, 2021.

[18] Antonio Maccioni and Riccardo Torlone. Kayak: a framework for just-in-time data preparation in a data lake. In *Advanced Information Systems Engineering: 30th International Conference, CAiSE 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings 30*, pages 474–489. Springer, 2018.

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

[20] Amin Beheshti, Boualem Benatallah, Reza Nouri, Van Munin Chhieng, HuangTao Xiong, and Xu Zhao. Coredb: a data lake service. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 2451–2454, 2017.

[21] Ashley Farrugia, Rob Claxton, and Simon Thompson. Towards social network analytics for understanding and managing enterprise data lakes. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1213–1220. IEEE, 2016.

[22] 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 *New Trends in Databases and Information Systems: ADBIS 2018 Short Papers and Workshops, AI* QA, BIGPMED, CSACDB, M2U, BigDataMAPS, ISTREND, DC, Budapest, Hungary, September, 2-5, 2018, Proceedings 22*, pages 165–177. Springer, 2018.

[23] Amit Sheth. Panel: Data semantics: what, where and how? In *Database Applications Semantics: Proceedings of the IFIP WG 2.6 Working Conference on Database Applications Semantics (DS-6) Stone Mountain, Atlanta, Georgia USA, May 30–June 2, 1995*, pages 601–610. Springer, 1997.

[24] 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*, pages 313–322, 2019.

[25] Oscar Romero and Alberto Abelló. A framework for multidimensional design of data warehouses from ontologies. *Data & Knowledge Engineering*, 69(11):1138–1157, 2010.

[26] Leonor Frias, Anna Queralt Calafat, and Antoni Olivé Ramon. Eu-rent car rentals specification. 2003.

[27] Renée J Miller. Open data integration. *Proceedings of the VLDB Endowment*, 11(12):2130–2139, 2018.

[28] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*, pages 847–864, 2019.

[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] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1190–1201. IEEE, 2019.

[31] Franck Ravat and Yan Zhao. Data lakes: Trends and perspectives. In *Database and Expert Systems Applications: 30th International Conference,*

DEXA 2019, Linz, Austria, August 26–29, 2019, Proceedings, Part I 30, pages 304–313. Springer, 2019.

[32] Barry A. Devlin and Paul T. Murphy. An architecture for a business and information system. *IBM systems Journal*, 27(1):60–80, 1988.

[33] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment*, 12(12):1986–1989, 2019.

Elenco delle figure

2.1 Esempio di gerarchie dimensionali. 8
2.2 Esempio di modello multidimensionale dei dati con *fatto* le vendite. 8
2.3 Architettura di un *Data Warehouse*(Nambiar et al.[6]). 10
2.4 Architettura *Data pond*. 13
2.5 *Zone architecture* di Zaloni. 14
2.6 Classificazione a tre livelli dei metadati[17]. 16

3.1 Schema del *Semantic Data Lake framework*. 21
3.2 Esempio di modellazione e allineamento di sorgenti dati. 23
3.3 Rappresentazione delle proprietà di una sorgente. 27
3.4 Esempio di rappresentazione delle proprietà del profilo di categoria *level*. 31
3.5 Esempio di rappresentazione delle proprietà del profilo di categoria *integer*. 34
3.6 Esempio di profilo relativo ad un dominio classificato *real*. 36
3.7 Esempio di profilo relativo ad un dominio classificato *categorical*. . 38
3.8 Esempio di profilo relativo ad un dominio classificato *date*. 40
3.9 Esempio di profilo relativo ad un dominio classificato *string*. . . . 42
3.10 Esempio di rappresentazione di due testi nello spazio geometrico, fonte: engati. 44
3.11 Esempio di *embedding* di vettori. 44

5.1 Tempo medio calcolo del profilo di una sorgente al variare della cardinalità della stessa. 58
5.2 Tempo medio calcolo del profilo per singola categoria al variare della cardinalità della sorgente. 58
5.3 Tempo medio calcolo della similarità tra domini associati agli stessi livelli del *knowledge graph*. 61

Elenco delle tabelle

3.1 Elenco delle proprietà che descrivono una sorgente. 25
3.2 Elenco delle proprietà comuni a tutti i domini. 28
3.3 Elenco delle proprietà del profilo di categoria *level*. 29
3.4 Elenco delle proprietà del profilo di categoria *integer*. 32
3.5 Elenco delle proprietà del profilo di categoria *real*. 35
3.6 Elenco delle proprietà del profilo di categoria *categorical*. 37
3.7 Elenco delle proprietà del profilo di categoria *date*. 39
3.8 Elenco delle proprietà del profilo di categoria *string*. 41

5.1 Percentuale della tipologia di domini per ogni sorgente. 57
5.2 Esempio di dataset di test per il calcolo della similarità. 60

Elenco dei listati

3.1 URIRef del namespace `rdfs` 24
3.2 URIRef del namespace `xsd`. 24
3.3 URIRef del namespace `dcterms`. 24
3.4 URIRef del namespace `void`. 24
3.5 URIRef del namespace `KPIOnto`. 25
3.6 URIRef del namespace `dl`. 25
3.7 Proprietà associate ad una sorgente. 26
3.8 Esempio di rappresentazione delle proprietà del profilo di categoria *level*. 30
3.9 Esempio di rappresentazione delle proprietà del profilo di categoria *integer*. 33
3.10 Esempio di rappresentazione delle proprietà del profilo di categoria *real*. 35
3.11 Esempio di rappresentazione delle proprietà del profilo di categoria *categorical*. 37
3.12 Esempio di rappresentazione delle proprietà del profilo di categoria *date*. 39
3.13 Esempio di rappresentazione delle proprietà del profilo di categoria *string* 41

Elenco degli algoritmi

1 Importazione di una sorgente dati nel *Data Lake* 46
2 Funzione per il calcolo del profilo 49
3 Profilo per colonne *integer* 50
4 Profilo per colonne *real* 51
5 Profilo per colonne *categorical* 51
6 Profilo per colonne *Date* 52
7 Profilo per colonne *string* 53
8 Algoritmo per il calcolo della similarità 55
9 Calcolo *embedding* per dominio di una sorgente 56
10 Calcolo della similarità tra due *embedding* 56