UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Tecnologie Di Reasoning Per Big Data E Knowledge Graphs

Laureando

Marco Faretra

Matricola 460573

Relatore

Prof. Paolo Atzeni

Correlatore

Ing. Luigi Bellomarini

Anno Accademico 2016/2017

Questa è la dedica

Ringraziamenti

Grazie a tutti

Introduzione

Nell'ultimo decennio, grandi, medie e piccole aziende si trovano sempre più di fronte alla sfida di dover produrre, trattare ed utilizzare una quantità sempre maggiore di dati, ciò è dovuto principalmente alla diffusione di strumenti tecnologici, all'aumento della capacità di calcolo ed allo sviluppo di internet, dei sensori e delle reti di comunicazione. Tale crescita porta ad una proliferazione delle sorgenti dei dati disponibili, con un conseguente incremento delle mole di dati da gestire e potenzialmente utilizzare per fini operativi e decisionali. Questo fenomeno, spesso conosciuto come Big Data, ha introdotto diversi problemi: i tempi di risposta sono aumentati drasticamente; lo spazio e l'hardware cominciano a non bastare più.

L'importanza della conoscenza è stata evidente fin dagli anni '70 e il suo utilizzo si è presto diffuso in molti ambiti industriali e tecnici, tra cui il supporto alle decisioni e il data warehousing.

Al giorno d'oggi, l'avvento dei Big Data, ha reso ancora più essenziale elaborare e trattare tale conoscenza, con il fine ultimo di produrne una nuova.

In effetti, il termine attuale "economia della conoscenza" indica proprio l'insieme delle attività umane volte a trarre valore, tangibile e intangibile, da tale conoscenza. La conoscenza, e quindi l'informazione che la costituisce, è sempre più vista come forza motrice nelle attività di business. Ci sono molte aziende informatiche che hanno come business plan la conoscenza, ad esempio aziende che vendono dati estratti dal web, grosse aziende e banche che vendono e utilizzano i dati per sofisticate inferenze relative ai propri clienti, con vari fini: profilazione, mantenimento della clientela, sviluppo di sistemi di recommendation, contrasto alle frodi, vendita dei dati a terze parti e molto altro.

L'esplosione è avvenuta negli ultimi anni, in passato erano presenti diverse difficoltà, sia tecniche che teoriche, i linguaggi di programmazione erano molto complessi e gli ingegneri erano in numero limitato, l'hardware, i database, erano inadeguati, rappresentavano un collo di bottiglia per le elaborazioni troppo grandi.
Con il passare degli anni, l'avvento tecnologico ha subito una crescita esponenziale, l'hardware si è evoluto, le tecnologie dei database sono migliorate notevolmente, i linguaggi di programmazione sono diventati molto più semplici, sono nati nuovi paradigmi di programmazione, inoltre è possibile utilizzare framework architetturali che fungono da middleware e permettono quindi la riduzione di scrittura di codice da parte di uno sviluppatore. Ovviamente questa semplificazione ha portato ad un enorme vantaggio nell'elaborazione di conoscenza e reasoning su grandi quantità di dati.
Il termine Knowledge Graph è stato originariamente riferito solo a quello di Google, ovvero "una base di conoscenze utilizzata da Google per migliorare i risultati di ricerca del suo motore, con informazioni di ricerca semantica raccolte da una grande varietà di fonti". Nel frattempo, altri colossi come Facebook, Amazon, ecc., hanno costruito i proprio Knowledge Graph, e molte altre aziende vorrebbero mantenere un knowledge graph privato aziendale che contiene molte quantità di dati in forma di fatti. Tale Knowledge Graph aziendale dovrebbe contenere conoscenze di business rilevanti, ad esempio su clienti, prodotti, prezzi, ecc. Questo dovrebbe essere gestito da un KGMS, cioè un sistema di gestione delle basi di conoscenza e che svolge altri task basati su grandi quantità di dati fornendo strumenti per il data analytics e il machine learning. La parola '*graph*' in questo contesto viene spesso fraintesa: molti credono che avere un Graph Database System e alimentare tale database con i dati, sia sufficiente per ottenere un Knowledge Graph aziendale, altri pensano che siano limitati alla memorizzazione e all'analisi dei dati, ma non dovrebbero essere limitati a questo [BGPS17].
Come definito in [BGPS17] definiamo i requisiti per avere un KGMS completo. Esso deve svolgere compiti complessi di reasoning, ed allo stesso tempo ottenere performance efficienti e scalabili sui Big Data con una complessità computazionale accettabile. Inoltre necessita di interfacce con i database aziendali, il web e il machine learning. Il core di un KGMS deve fornire un linguaggio per rappresentare la conoscenza e il resoning.

Introdurremo un sistema che soddisfa tutti i requisiti del KGMS sopra citati, ed utilizza un linguaggio che fa parte della famiglia del Datalog.
Datalog [ACPT06] è un linguaggio di interrogazione per basi di dati che ha riscosso un notevole interesse dalla metà degli anni ottanta, è basato su regole di deduzione. Rappresenta un linguaggio sottoinsieme del linguaggio Prolog relativo ai database relazionali, anche Datalog è basato su regole di deduzione ma non permette l'utilizzo né di simboli di funzione né di un modello di valutazione.
Ogni regola è composta da una testa (chiamata anche head o conseguente) e da un corpo (chiamato anche body o antecedente) a loro volta formati da uno o più predicati atomici. Se tutti gli atomi del corpo sono verificati, ne consegue che anche il predicato atomico della testa lo sia. L'espressività di Datalog è dovuta alla possibilità di scrivere regole ricorsive, cioè in grado di richiamare il medesimo atomo della testa anche nel corpo della regola.
Uno dei migliori linguaggi per il reasoning basato sulla conoscenza è Datalog. Durante gli anni è stato studiato in dettaglio, nel data exchange e data integration [FGNS16].
Il problema di Datalog è che non permette una forte potenza espressiva, è quindi stata progettata una famiglia di linguaggi, chiamata Datalog$^{\pm}$ che aggiunge maggiore potenza espressiva al linguaggio nativo, mantenendo costante efficienza e scalabilità [BGPS17].
La famiglia Datalog$^{\pm}$ estende Datalog con quantificatori esistenziali nelle teste delle regole, ed allo stesso tempo limita la sua sintassi in modo da ottenere decidibilità e scalabilità dei dati [CGK13, CGP12, CGL^{+}10].

Il sistema che presentiamo ed utilizziamo in questa tesi, Vadalog Reasoner, è il contributo dell'università di Oxford al progetto VADA [VAD], in collaborazione con le università di Edimburgo e Manchester. La mia attività è stata svolta presso il Laboratorio basi di dati dell'università degli studi Roma Tre, lavorando da remoto con il mio correlatore Luigi Bellomarini.
Il Vadalog Reasoner è un KGMS, che offre un motore centrale di reasoning principale ed un linguaggio, il Vadalog, per la gestione e l'utilizzo.
Vadalog appartiene alla famiglia Datalog$^{\pm}$ sopra descritta ed è in grado di soddisfare tutti i requisiti di un KGMS completo sopra definiti.

Il core logico del Vadalog Reasoner è in grado di processare tale linguaggio, è in grado di eseguire task di reasoning ontologici e risulta computazionalmente efficiente, tale da soddisfare i requisiti citati, esso ha inoltre accesso ad un repository di regole.
Il problema principale nell'utilizzo del Datalog$^{\pm}$ è che l'introduzione della quantificazione esistenziale nelle teste porta subito all'indecidibilità. Sono quindi nati una serie di frammenti (cioè delle restrizioni) di Datalog$^{\pm}$ che sono decidibili.
Tali frammenti hanno diversa complessità computazionale (data complexity), esponenziale nel caso più generale.
Uno dei punti di forza di Datalog è la bassa complessità come descritto in precedenza, Vadalog cattura ed estende Datalog senza aumentare tale complessità, questo avviene grazie alle proprietà ereditate dal frammento Warded Datalog$^{\pm}$, il linguaggio su cui si basa Vadalog.
In termini intuitivi, il Warded Datalog$^{\pm}$ (così come altri frammenti di Datalog$^{\pm}$) contiene la propagazione dei valori nulli nell'ambito della ricorsione. Il linguaggio riesce ad imporre tali vincoli grazie a semplici restrizioni sintattiche, che individuano quali variabili possono contenere valori nulli che si propagano nella testa (variabili *dangerous*), e impongono che tali variabili debbano sempre comparire esattamente in un atomo del corpo (detto *ward*) che interagisce con gli altri atomi del corpo solo mediante variabili che non possono assumere valori nulli (*harmless*) [BGPS17].
Il Vadalog Reasoner fornisce anche degli strumenti che permettono il data analytics, l'iniezione di codice procedurale, l'integrazione con diverse tipologie di input (ad esempio database relazionali, file csv, database non relazionali, ecc.).

Il mio contributo in questa tesi è stato l'implementazione di feature core del Vadalog Reasoner per l'esecuzione di programmi Vadalog.
Inizialmente il progetto presentava un ambito di intervento molto vasto.
Non erano presenti diversi tipi di dati, anche primitivi, per permettere all'utente di effettuare maggiori statistiche. Tale requisito è fondamentale se si vogliono integrare tipi di dati nel linguaggio Vadalog.
Non erano presenti delle ottimizzazioni per permettere un guadagno sull'esecuzione di programmi Vadalog. Ciò rappresentava un limite volendo guadagnare in efficienza e

permettere quindi delle performance migliori. Ad esempio non veniva ottimizzata efficientemente la ricorsione.
Erano stati effettuati soltanto dei benchmark iniziali per testare le perfomance, ma nulla di concreto per effettuare anche confronti con i competitor esistenti, e quindi capire se il sistema fosse competitivo nel settore.
Le sorgenti a disposizione dell'utente finale erano in numero limitato. Anche questo problema rappresentava un collo di bottiglia per il nostro sistema, in quanto l'utente aveva poche sorgenti per incrociare i dati ed effettuarne analisi.
Non era possibile integrare codice sorgente da altri linguaggi, né definire funzioni all'interno del linguaggio Vadalog. Ciò era un limite, poiché non permetteva di fare computazioni laboriose o di utilizzare del codice già scritto in precedenza (magari dedicato ad un calcolo ben definito).
Era presente un'interfaccia web molto scarna e con poche funzionalità. L'utente aveva accesso soltanto a poche funzionalità rispetto a tutte le funzionalità fornite dal sistema.

Di seguito una breve lista delle funzionalità di cui mi sono occupato, che verranno illustrate in maniera più approfondita nei prossimi capitoli:

- Implementazione di nuovi tipi di dato, semplici e strutturati, per risolvere il problema dei pochi tipi di dato disponibili nel sistema.

- Tecniche di ottimizzazione in Datalog per migliorare le prestazioni. In particolare mi sono occupato delle ottimizzazioni di: *Push Selections e Projections Down*, ovvero la trasformazione di una query in un'altra equivalente, ma anticipando le selezioni e le proiezioni alle regole più vicine a quelle di input (ad esempio l'ideale sarebbe inserire le selezioni e le proiezioni alle regole di input), questo risulta efficiente poiché coinvolge un numero minore di tuple nell'operazione di join [ACPT06]; supporto per ricorsione sinistra e destra, trasformazione dove le ricorsioni destre (nel corpo di una regola l'atomo che ricorre si trova alla destra di tutti gli altri atomi) che risultano più problematiche anche a livello di computazione temporale, vengono opportunamente trasformate in ricorsioni sinistre (nel corpo di una regola l'atomo che ricorre si trova alla sinistra di tutti gli altri

atomi) che risultano più efficienti; ottimizzazione multi-join, ovvero quando si è in presenza di regole che hanno al loro interno join condivisi tra tre o più atomi, essi vengono splittati in un numero di regole proporzionale al numero di atomi, in modo di avere al più regole con un join tra due atomi.

- Creazione di benchmark per effettuare test sulle performance per effettuare analisi esaustive sulle performance del sistema ed effettuare infine confronti con altri sistemi esistenti. Mi sono occupato inoltre dello sviluppo di *iWarded*, un piccolo tool che genera programmi Warded Datalog$^{\pm}$ per effettuare benchmark. Tale sistema prende in input diversi parametri che hanno lo scopo di descrivere le particolarità del programma Vadalog, e ne restituisce quest'ultimo.

- Supporto di nuove sorgenti, file CSV e database relazionali e non relazionali. Questo permette maggiore interazione per l'utente, è così possibile combinare i dati provenienti da storage eterogenei ed effettuare statistiche e tutte le operazioni offerte dal sistema su di essi.

- Supporto alle user-defined functions. È possibile definire delle funzioni all'interno di Vadalog, al quale è possibile passare dei parametri, ed è possibile esprimere la correlazione tra esse e funzioni esterne che possono essere implementate in diversi linguaggi. Essenziali per l'utente se vuole integrare funzioni all'interno del programma Vadalog, ad esempio funzioni già scritte in passato che calcolano una determinata statistica.

- Miglioramento notevole dell'interfaccia web (Vadalog console), con l'integrazione di elementi grafici che permettono l'interazione con tutti i servizi offerti dal sistema, integrazione di un editor e tante altre funzionalità.

La tesi è organizzata come segue:

Il capitolo 1 descrive il Vadalog Engine, in particolare le proprietà di un KGMS ed in modo approfondito l'architettura del nostro sistema.

Il capitolo 2 è incentrato sul linguaggio Vadalog, in particolare è presente un approfondimento più ampio sul core logico della famiglia di linguaggi Datalog$^{\pm}$ e di Vadalog, della complessità e delle estensioni di quest'ultimo.

Il capitolo 3 è dedicato alla descrizione nel dettaglio delle funzionalità riguardanti il mio contributo al progetto.

Il capitolo 4 descrive le prove sperimentali effettuate sul sistema Vadalog.

Il capitolo 5 tratta del paragone effettuato tra il sistema Vadalog ed i competitor già presenti sul mercato.

Il capitolo 6 trae le conclusioni del lavoro.

Indice

Capitolo 1

Il linguaggio Vadalog

Vadalog è un linguaggio che raggiunge un attento equilibrio tra espressività e complessità, e può essere utilizzato come reasoning core di un KGMS.

1.1 Il core del linguaggio

Il core logico di Vadalog fa parte della famiglia Warded Datalog±. Che a sua volta è un'estensione della famiglia dei linguaggi Datalog±, nei seguenti sottocapitoli descriveremo in dettaglio tali famiglie di linguaggi.

1.1.1 Il core logico dei linguaggi Datalog±

L'obiettivo principale dei linguaggi Datalog± è quello di estendere il noto linguaggio Datalog con funzionalità di modellazione utili come quantificatori esistenziali nelle teste delle regole (il + nel simbolo ±), e contemporaneamente limitare la sintassi della regola, in modo tale che sia garantita la decidibilità e la tracciabilità dei dati del reasoning (il - nel simbolo ±).
Il core dei linguaggi Datalog± è costituito da regole note come regole esistenziali, che generalizzano le regole Datalog con quantificatori esistenziali nelle teste delle regole, un esempio di regola esistenziale:

$$Person(x) \rightarrow \exists y \; HasFather(x, y), Person(y)$$

che esprime che ogni persona, ha un padre che anch'esso a sua volta è una persona.
In generale, una regola esistenziale è una frase di primo ordine:

$$\forall \bar{x} \forall \bar{y} (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \psi(\bar{x}, \bar{z})$$

dove ϕ ()il corpo) e ψ (la testa) sono congiunzioni di atomi con costanti e variabili.
La semantica di un insieme di regole esistenziali Σ sopra un database D, chiamata Σ(D), è definita mediante una procedura. Durante questa procedura vengono aggiunti nuovi atomi a D (coinvolgendo anche valori nulli per soddisfare le variabili esistenziali), finché il risultato finale Σ(D) non soddisfa tutte le regole esistenziali, solitamente Σ(D) è infinito.
Vediamo un esempio di tale procedura:
Consideriamo un Database D = Person(Bob), e la regola esistenziale:

$$Person(x) \rightarrow \exists y \; HasFather(x, y), Person(y)$$

L'atomo del database D innesca la regola esistenziale e vengono aggiunti i seguenti atomi:

$$HasFather(Bob, \nu_1) \; e \; Person(\nu_1)$$

ν_1 è un labeled null che rappresenta un valore sconosciuto.
Il nuovo atomo Person(ν_1) innesca nuovamente la regola esistenziale, e vengono aggiunti altri atomi a D:

$$HasFather(\nu_1, \nu_2) \; e \; Person(\nu_2)$$

Dove ν_2 è un nuovo nullo. Il risultato è l'istanza:

$$\{Person(Bob), HasFather(Bob, \nu_1)\} \cup \bigcup_{i>0} \{Person(\nu_i), HasFather(\nu_i, \nu_{i+1})\}$$

ν_1, ν_2, ..., ν_i, ν_{i+1} sono labeled nulls.

Data una coppia Q = (Σ, Ans), dove Σ è un insieme di variabili esistenziali e Ans un predicato n-ario, la valutazione di Q su un database D, indicata Q(D), è definita come set di tuple sopra sopra l'insieme C_d di valori costanti che occorrono nel database D.

Il compito principale del reasoning è l'inferenza di tuple: dato un database D, una coppia Q = (Σ, Ans), ed una tupla di costanti $\bar{t}$, bisogna decidere se $\bar{t}$ appartiene a Q. Questo problema è molto difficile, infatti è indecidibile, anche quando Q è fisso e solo D è dato come input [CGK13].
Ciò ha portato all'attività di identificare restrizioni sulle regole esistenziali che rendono tale problema decidibile. Ciascuna di queste restrizioni genera un nuovo linguaggio Datalog±.

1.1.2 Il core logico di Vadalog

Il core logico di Vadalog si basa sulla nozione di *wardedness*, che genera Warded Datalog± [GP15].
In altri termini Vadalog è ottenuta estendo Warded Datalog± con ulteriori funzionalità pratiche che sono descritte nella sezione successive.
Nei linguaggi Warded Datalog±, ci sono tre tipologie di variabili che sono contraddistinte:

1. Harmless: Variabili standard presenti nel corpo di una regola, che hanno un valore noto.

2. Harmful: Rappresentano dei labeled null presenti nel corpo di una regola, e che sono a loro volta un esiste nella testa di un'altra regola.

3. Dangerous: Sono le variabili che rappresentano gli esiste nelle teste delle regole.

La wardedness applica una restrizione su come vengono utilizzate le variabili "*dangerous*" (pericolose) di un insieme di regole esistenziali. Intuitivamente una variabile dangerous è una variabile del corpo che può essere unificata con un valore nullo quando viene applicato l'algoritmo di ricerca e viene propagato anche alla testa della regola.
Per esempio dato l'insieme Σ di regole esistenziali

$$P(x) \rightarrow \exists z\ R(x, z)\ e\ R(x, y) \rightarrow P(y)$$

La variabile y nel corpo della seconda regola è dangerous. Ad esempio dato un database D = {P(a)}, l'algoritmo applica la prima regola e genera R(a, ν), dove ν è un nullo che

funge da testimone della variabile esistenziale z, e poi la seconda regola verrà applicata con la variabile y che è unificata con ν che viene propagata all'atomo ottenuto P(ν). L'obiettivo della wardedness è quello di verificare il modo in cui i valori nulli vengono propagati ponendo le seguenti condizioni:

1. Tutte le variabili dangerous devono coesistere in un singolo corpo di un atomo A, chiamato il ward, e
2. Il ward può condividere solo variabili *harmless* con il resto del corpo.

Warded Datalog± è composto da insiemi (finiti) di regole esistenziali warded, esso rappresenta una raffinatezza di *Weakly-Frontier-Guarded Datalog*±, che è una famiglia di linguaggi definita allo stesso modo ma senza la condizione 2 sopra citata [BLMS11]. Weakly-Frontier-Guarded Datalog± è intrattabile nella complessità dei dati, infatti è EXPTIME-completo.

Warded Datalog± gode di diverse proprietà che lo rendono robusto, verso linguaggi più pratici:

- L'inferenza di tuple è trattabile; infatti essa è PTIME-completa quando l'insieme di regole è fisso.
- Cattura Datalog, senza incrementare la complessità. Infatti un insieme Σ di regole Datalog è implicitamente Warded poiché non ci sono variabili dangerous per definizione.
- Generalizza linguaggi di ontologia principali come OWL 2 QL (linguaggio ontologico per la semantica del web [W3Ca])
- È adatto per la ricerca di grafi RDF (Resource Description Framework, strumento base proposto da W3C per la codifica, scambio e riutilizzo di metadati [W3Cb]). In realtà aggiungendo una negazione stratificata, si ottiene un linguaggio chiamato TriQ-Lite1.0 [GP15], che può esprimere ogni query SPARQL (linguaggio di query per gli RDF [W3Cc]) nell'ambito del regime di OWL 2 QL.

Capitolo 2

Vadalog Engine

In questo capitolo parleremo un maniera più approfondita del Vadalog Engine.
Il capitolo è organizzato come segue, nella sezione 2.1 verranno definiti in maniera più approfondita i requisiti che si devono soddisfare per avere un KGMS efficiente e performante, in particolare descriveremo tre categorie principali su cui sono basati tali requisiti. Una prima categoria (sezione 2.1.1) che descrive le proprietà che l'engine deve mettere a disposizione del linguaggio, un'altra categoria (sezione 2.1.2) che descrive l'accesso e gestione dei Big Data con le relative proprietà che l'engine deve supportare, ed infine una categoria (sezione 2.1.3) che descrive le necessità di un supporto per codice procedurale da terze parti.
Infine nella sezione 2.2 verrà descritta l'architettura del Vadalog Reasoner, in maniera approfondita tutto il processo di reasoning (sezione 2.2.1), nel quale verrà spiegata la strategia di terminazione e la gestione della ricorsione. Inoltre descriveremo come il Vadalog Reasoner gestisce i stream e tecniche per la gestione e ottimizzazione della cache (sezione 2.2.2), e le modalità con cui è possibile interfacciarsi con esso (sezione 2.2.3). Questo capitolo prende spunto su diverse nozioni da [BGPS17]

2.1 Proprietà di un KGMS

Un KGMS efficiente deve soddisfare diversi requisiti, che vengono suddivisi in tre categorie principali.

2.1.1 Linguaggio e sistema per il reasoning

Ogni KGMS necessita di avere un formalismo logico (linguaggio) per esprimere fatti e regole ed un engine per il reasoning che utilizza tale linguaggio che dovrebbe fornire le seguenti caratteristiche:

- Sintassi semplice e modulare: Deve essere semplice aggiungere e cancellare fatti e nuove regole. I fatti devono coincidere con le tuple sul database.
- Alta potenza espressiva: Datalog [CGT12, HGL11] è un buon punto di riferimento per il potere espressivo delle regole. Con una negazione molto lieve cattura PTIME [DEGV01].
- Calcolo numerico e aggregazioni: Il linguaggio deve essere arricchito con funzioni di aggregazione per la gestione di valori numerici.
- Probabilistic Reasoning: Il linguaggio dovrebbe essere adatto ad incorporare metodi di reasoning probabilistico e il sistema dovrebbe propagare probabilità o valori di certezza durante il processo di reasoning.
- Bassa complessità: Il reasoning dovrebbe essere tracciabile nella complessità dei dati. Quando possibile il sistema dovrebbe essere in grado di riconoscere e trarre vantaggio da set di regole che possono essere elaborate in classi di complessità a basso livello di spazio.
- Rule repository, Rule management and ontology editor: È necessario fornire una libreria per l'archiviazione di regole e definizioni ricorrenti, ed un'interfaccia utente per la gestione delle regole.
- Orchestrazione dinamica: Per applicazioni più grandi, deve essere presente un nodo master per l'orchestrazione di flussi di dati complessi.

2.1.2 Accesso e gestione dei Big Data

- Accesso ai Big Data: Il sistema deve essere in grado di fornire un accesso efficace alle sorgenti e ai sistemi Big Data. L'integrazione di tali tecniche dovrebbe essere possibile se il volume dei dati lo rende necessario [SYI+16].

- Accesso a Database e Data Warehouse: Dovrebbe essere concesso un accesso a database relazionali, graph databases, data warehouse, RDF stores ed i maggiori NoSQL stores. I dati nei vari storage dovrebbero essere direttamente utilizzabili come fatti per il reasoning.

- Ontology-based Data Access (OBDA): OBDA [CDGL$^+$11] consente ad un sistema di compilare una query che è stata formulata in testa ad un'ontologia direttamente all'interno del database.

- Supporto multi-query: Laddove possibile e appropriato, i risultati parziali di query ripetute dovrebbero essere valutati una volta [RSSB00] e ottimizzati a questo proposito, per ottenere un guadagno temporale.

- Pulizia dei dati, Scambio e Integrazione: L'integrazione, la modifica e la pulizia dei dati dovrebbero essere supportati direttamente (attraverso il linguaggio).

- Estrazione di dati web, Interazione e IOT: Un KGMS dovrebbe essere in grado di interagire con il web mediante estrazione dei dati web rilevanti (prezzi pubblicizzati dai concorrenti) e integrandoli in database locali e scambiare dati con moduli e server web disponibili (API).

2.1.3 Inserimento di codice procedurale e di terze parti

- Codice procedurale: Il sistema dovrebbe disporre di metodi di incapsulamento per l'incorporazione di codice procedurale scritti in vari linguaggi di programmazione e offrire un'interfaccia logica ad esso.

- Pacchetti di terze parti per il machine learning, text mining, NLP, Data Analytics e Data Visualization: Il sistema dovrebbe essere dotato di accesso diretto a potenti package esistenti per il machine learning, text mining, data analytics e data visualization. Esistono diversi software di terze parti per questi scopi, un KGMS dovrebbe essere in grado di utilizzare una moltitudine di tali pacchetti tramite opportune interfacce logiche.

2.2 Architettura

In Vadalog, il knowledge graph è organizzato come un repository, una collezione di regole Vadalog, a sua volta confezionate in librerie.
Le regole e le librerie possono essere amministrate tramite un'interfaccia utente dedicata che ne permette la creazione, la modifica e la cancellazione.
Le fonti esterne sono supportate e gestite attraverso dei trasduttori, ovvero adattatori intelligenti che consentono un'interazione con le fonti durante il processo di reasoning.
Come indicato in Figura 2.1, che rappresenta la nostra architettura di riferimento, la componente centrale di un KGMS è il suo motore di reasoning principale, che ha accesso ad un repository di regole. Sono stati raggruppati vari moduli che forniscono funzionalità pertinenti di accesso ai dati ed analisi. Come possiamo vedere ad esempio la possibilità di accedere a RDBMS utilizzando il linguaggio standard SQL, nonché l'accesso ai NoSQL Stores attraverso delle API, ecc.

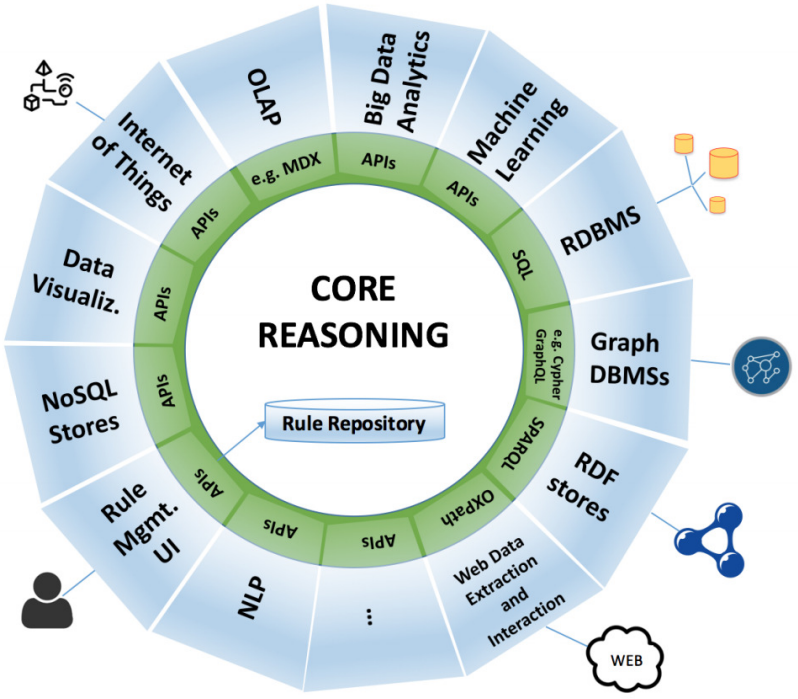

Figura 2.1: Schema dell'architettura dell'Engine di Vadalog [BGPS17].

2.2.1 Reasoning

Il linguaggio Vadalog estende la famiglia di linguaggi Warded Datalog$^{\pm}$, ovvero tutte le caratteristiche offerte dal linguaggio base, con la possibilità di utilizzare gli esiste nella testa delle regole.
In particolare tale famiglia garantisce la *wardedness*, ovvero limita l'interazione tra gli esiste (labeled null), garantendo che due esiste possano essere messi in join fra di loro, soltanto se tale elemento di join non viene proiettato in testa (Harmful Join), altrimenti potremmo trovarci nel non determinismo (NP), mentre utilizzando questa strategia viene garantita la P (polinomiale).

In questa sezione viene mostrato il processo di reasoning, ovvero come il Vadalog Reasoner sfrutta le proprietà chiave di Warded Datalog±.
Il Vadalog Reasoner fa un uso piuttosto alto di ricorsione, quindi è probabile che ci si ritrovi in un ciclo infinito all'interno del grafo di esecuzione (grafo contenente tutti i nodi per la ricerca della soluzione, partendo dai nodi di output ai nodi di input), si utilizza quindi una strategia di *Termination Control*.
Tale strategia, rileva ridondanza, ovvero la ripetizione di un nodo più volte, il prima possibile combinando il tempo di compilazione e tecniche a runtime.
A tempo di compilazione, grazie alla wardedness, che limita l'interazione tra i labeled null, l'engine riscrive il programma in modo che i joins con specifici valori di labeled null non si verificheranno mai (Harmful Join elimination).
A runtime, il Vadalog Reasoner adotta una tecnica di potatura ottimale di ridondanza e rami che non terminano mai, questa tecnica è strutturata in due parti *detection* e *pruning*.
Nella fase di detection, ogni volta che una regola genera un fatto che è simile ad uno dei precedenti, viene memorizzata la sequenza delle regole applicate, ovvero la provenienza.
Nella fase di pruning, ogni volta che un fatto presenta la stessa provenienza di un altro e sono simili, il fatto non viene generato.
Tale tecnica sfrutta a pieno le simmetrie strutturali all'interno del grafo, per scopi di terminazione i fatti sono considerati equivalenti, se hanno la stessa provenienza e sono originati da fatti simili tra loro.

2.2.2 Architettura stream-based e gestione della cache

Il Vadalog Reasoner, per essere un KGMS efficace e competitivo, utilizza un'architettura in-memory, nel quale viene utilizzata la cache per velocizzare determinate operazioni, ed utilizza le tecniche descritte in precedenza, che garantiscono risoluzione e ridondanza.
Da un insieme di regole Vadalog, viene generato un *query plan*, che rappresenta un grafo dove è presente un nodo per ogni regola definita e un arco ogni volta che la testa di una regola appare nel corpo di un'altra.
Alcuni nodi speciali sono contrassegnati come input o output, quando corrispondono a set di dati esterni (input), o ad atomi per il reasoning (output), rispettivamente. Il query plan è ottimizzato con diverse variazioni su tecniche standard, ad esempio *push selections down* e *push projections down* il più possibile vicino alla sorgente dati.
Infine il query plan viene trasformato in un piano di accesso, dove i nodi di una generica regola vengono sostituiti dalle implementazioni appropriate per i corrispondenti operatori a basso livello (ad esempio selezione, proiezione, join, aggregazione, ecc...).
Per ogni operatore sono disponibili un insieme di possibili implementazioni e vengono attivati in base a criteri di ottimizzazione comuni.
In Figura 2.2 viene rappresentato un esempio di query plan di un semplice programma Vadalog, nel quale vengono dati due input e sono definite due regole che contengono semplici proiezioni.

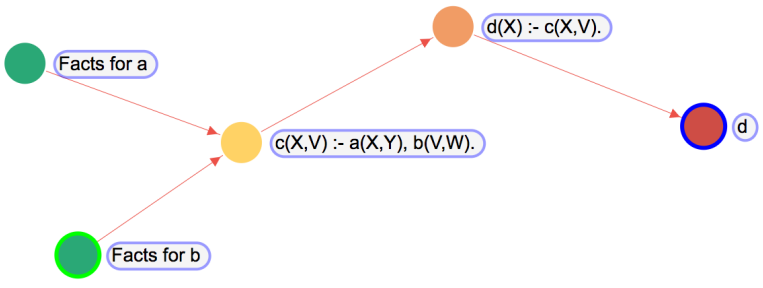

Figura 2.2: Esempio di query plan.

Il Vadalog Reasoner utilizza un approccio stream-based (o approccio a pipeline). Tale approccio [Wik] presume di avere i dati da elaborare organizzati in gruppi (stream) e che questi possano essere elaborati applicandone una serie di operazioni, spesso tali operazioni vengono elaborate tramite l'utilizzo di strutture a pipeline, e per ridurre i tempi vengono spesso utilizzate delle cache.
Nel nostro sistema i fatti sono attivamente richiesti dai nodi di output ai loro predecessori e così via, fino ad arrivare ai nodi di input, che ricavano i fatti dalla sorgente dati. L'approccio stream è essenziale per limitare il consumo di memoria, in modo che risulta efficace per grandi volumi di dati.
La nostra impostazione è resa più impegnativa dalla presenza di regole di interazione multipla e dalla presenza di ricorsioni.
Gestiamo tali problemi utilizzando delle tecniche sui buffer, i fatti di ciascun nodo vengono messi in cache.
La cache locale funziona particolarmente bene in combinazione con l'approccio basato sui stream, dato che i fatti richiesti da un successore possono essere immediatamente riutilizzati da tutti gli altri successori, senza avanzare ulteriori richieste. Inoltre questa combinazione realizza una forma di ottimizzazione multi-query, dove ogni regola sfrutta i fatti prodotti dagli altri ogni volta che risulta applicabile.
Il problema principale è che in questo caso, se abbiamo a che fare con molti dati, la memoria rappresenta un limite, per limitarne l'occupazione, le cache locali vengono pulite con un *Eager Eviction Strategy* che rileva quando un fatto è stato consumato da tutti i possibili richiedenti e quindi viene cancellato della memoria.
I casi di cache overflow vengono gestiti ricorrendo alle scritture su disco (ad esempio utilizzando strategie di scrittura differenti come LRU, LFU, ecc.).
Le cache locali sono anche componenti funzionali fondamentali nell'architettura, poiché implementano in modo trasparente ricorsione e strategia di terminazione.
Quindi, il meccanismo di stream è completamente agnostico sulle condizioni di terminazione, il suo compito è produrre dati per i nodi di output finché gli input forniscono fatti, è responsabilità delle cache locali rilevare la periodicità, controllare la terminazione e interrompere il calcolo ogni volta che ricorre un pattern noto.
Possiamo vedere un esempio nella Figura 2.3, dove si nota all'interno del rettangolo

bordato l'intero processo di stream, il nodo di output che richiede attivamente i fatti ai nodi intermedi (in questo caso un blocco contiene N nodi intermedi), e così via, fino ad arrivare ai nodi di input che chiedono i fatti alle sorgenti esterne (ad esempio Database, Datawarehouses, csv, ecc...).
Nella parte superiore possiamo notare l'interazione diretta tra il core e la cache, in questo caso si tratta di un'interazione periodica al fine di verificare ricorsioni cicliche per la strategia di terminazione.
Ed infine è presente l'interazione tra il core ed il disco fisso, in questo caso essa avviene soltanto quando ci troviamo nel caso di cache overflow, in modo da fare il flush di tutti i dati della cache su disco fisso.

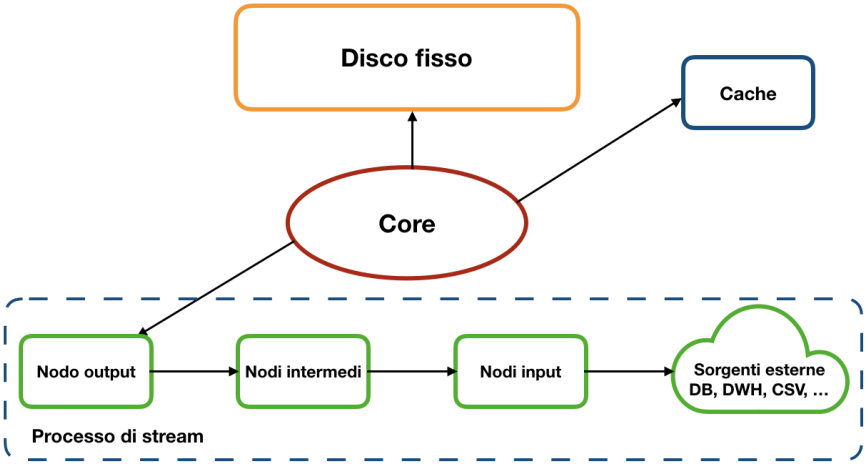

Figura 2.3: Funzionamento dell'architettura stream-based.

Per i join il Vadalog Reasoner adotta un estensione del Nested Loop Join, adatto per l'approccio stream-based ed efficiente in combinazione con le cache locali.
Tuttavia per garantire buone prestazioni, le cache locali sono migliorate dall'indicizzazione dinamica (a runtime) in memoria. In particolare, le cache associate ai join possono essere indicizzate mediante indici di hash creati a runtime, in modo da attivare un'implementazione ancora più efficiente di hash join.

2.2.3 Interfacce

L'utente ha due possibilità per interfacciarsi con l'engine Vadalog, o attraverso delle API Rest, o attraverso Java, integrando il progetto.
Le principali operazioni che si possono effettuare sull'engine sono:

- Evaluate: Permette di eseguire codice Vadalog.
- EvaluateAndStore: Permette di eseguire codice Vadalog, con il salvataggio dell'output all'interno di un database definito dall'utente.
- getExecutionPlan: Che ritorna il query plan del programma dato in input.
- transformProgram: Restituisce il codice Vadalog dopo aver effettuato le ottimizzazioni (riscritture) definite dall'engine.

Come descritto in Figura 2.4 sono presenti due controller, uno per la gestione delle chiamate rest (VadaRestController), ed uno per la gestione delle chiamate Java (LibraryController), che si interfacciano con un unico controller che gestisce le chiamate pervenute da entrambi.

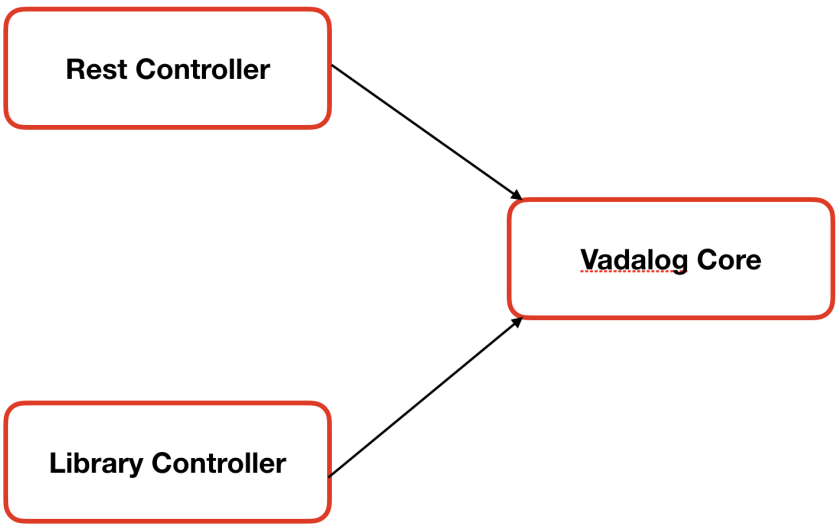

Figura 2.4: Sistemi per interfacciarsi con Vadalog Engine.

Capitolo 3

Algoritmi

Capitolo 4

Prove sperimentali

Capitolo 5

Related Work

Conclusioni e sviluppi futuri

La tesi è finita

Bibliografia

[ACPT06] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Basi di dati: modelli e linguaggi di interrogazione (seconda edizione)*. McGraw-Hill, 2006.

[BGPS17] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. Swift logic for big data and knowledge graphs. 2017.

[BLMS11] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9-10):1620–1654, 2011.

[CDGL+11] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.

[CGK13] Andrea Calı, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res*, 48:115–174, 2013.

[CGL+10] Andrea Cali, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on*, pages 228–242. IEEE, 2010.

[CGP12] Andrea Cali, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128, 2012.

[CGT12] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Science & Business Media, 2012.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001.

[FGNS16] Tim Furche, Georg Gottlob, Bernd Neumayr, and Emanuel Sallinger. Data wrangling for big data: Towards a lingua franca for data wrangling. 2016.

[GP15] Georg Gottlob and Andreas Pieris. Beyond sparql under owl 2 ql entailment regime: Rules to the rescue. In *IJCAI*, pages 2999–3007, 2015.

[HGL11] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM, 2011.

[RSSB00] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[SYI+16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149. ACM, 2016.

[VAD] VADA. http://vada.org.uk/.

[W3Ca] W3C. https://www.w3.org/TR/owl2-profiles/.

[W3Cb] W3C. https://www.w3.org/TR/rdf-schema/.

[W3Cc] W3C. https://www.w3.org/TR/rdf-sparql-query/.

[Wik] Wikipedia. `https://it.wikipedia.org/wiki/Stream_processing`.