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, piccole, medie e grandi 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, 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; la memoria centrale è insufficiente a contenere i dati necessari alle elaborazioni; lo spazio di storage e l'hardware cominciano a non essere più sufficienti.
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, cogliendo la sfida di sfruttarla in attività complesse, spesso producendone di nuova (*reasoning*).
In effetti, il termine attuale "economia della conoscenza" [Wika] 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 di consulenza informatica e di business la cui azione è centrata sulla conoscenza: ad esempio start-up che vendono dati estratti dal web; banche che vendono e utilizzano i dati per sofisticate inferenze relative ai propri clienti, con vari fini tra cui: profilazione, mantenimento della clientela,

sviluppo di sistemi di recommendation, contrasto alle frodi, vendita dei dati a terze parti e molto altro.

La forte crescita di interesse verso la conoscenza 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 e rappresentavano un collo di bottiglia per grandi elaborazioni.

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 dello 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 mostrano di voler mantenere un knowledge graph privato aziendale che contiene grandi quantità di dati. Tale Knowledge Graph aziendale dovrebbe contenere conoscenze di business rilevanti, ad esempio su clienti, prodotti, prezzi, ecc. Questo dovrebbe essere gestito da un Knowledge Graph Management System (KGMS), cioè un sistema di gestione delle basi di conoscenza in grado di svolgere task 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 (Database a grafo) e alimentare tale database con i dati, sia sufficiente per ottenere un Knowledge Graph aziendale, altri pensano erroneamente che i knowledge graphs siano limitati alla memorizzazione e all'analisi dei dati di grafi [BGPS17].

Come definito in [BGPS17] poniamo ora attenzione ai requisiti di 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 ac-

cettabile. Inoltre necessita di interfacce con i database aziendali, il web e librerie per il machine learning. Il core di un KGMS deve fornire un linguaggio per rappresentare la conoscenza e permettere il reasoning.

Questa tesi descrive il contributo allo sviluppo di un sistema che soddisfa tutti i requisiti del KGMS, ed utilizza un linguaggio che fa parte della famiglia del Datalog.
Datalog è un linguaggio di interrogazione per basi di dati che ha riscosso un notevole interesse dalla metà degli anni ottanta. È un linguaggio affine al Prolog, utilizzato nell'ambito dei database relazionali. In un certo senso è un sottoinsieme di Prolog poiché, nella sua versione base, Datalog è basato su regole di deduzione che non permettono l'utilizzo di simboli di funzione e non adotta di un modello di valutazione [ACPT06] .
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 (o semplicemente *atomi*). 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.
Datalog si è affermato come uno dei migliori linguaggi per il reasoning basato sulla conoscenza. Durante gli anni è stato studiato in dettaglio, nel data exchange e nella data integration [FGNS16].
Tuttavia Datalog ha un limitato potere espressivo, a causa dell'assenza di quantificazione esistenziale. È quindi stata progettata una famiglia di linguaggi, chiamata Datalog$^{\pm}$, che aggiunge maggiore potenza espressiva al linguaggio nativo [BGPS17].
In particolare 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 del Dipartimento di Ingegneria dell'Università 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.
Il problema principale nell'utilizzo del Datalog$^{\pm}$ è che l'introduzione della quantificazione esistenziale nelle teste rende il linguaggio indecidibile. Sono quindi nati una serie di frammenti (cioè delle restrizioni) di Datalog$^{\pm}$ che hanno l'obiettivo minimo di garantire la decidibilità. Tali frammenti hanno diversa complessità computazionale, esponenziale nel caso più generale.
Vadalog cattura ed estende Datalog senza aumentare tale complessità, questo avviene grazie alle proprietà ereditate dal frammento di Datalog$^{\pm}$ (Warded Datalog$^{\pm}$ in particolare) su cui si basa. 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 la gestione di 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 confronta il sistema Vadalog con i competitor già presenti sul mercato.

Il capitolo 6 trae le conclusioni del lavoro.

Indice

Capitolo 1

Il linguaggio Vadalog

In questo capitolo parleremo in maniera più approfondita del linguaggio Vadalog e di tutte le sue particolarità.
Vadalog è un linguaggio che raggiunge un attento equilibrio tra espressività e complessità, e può essere utilizzato come reasoning core di un KGMS. Fa parte della famiglia Warded Datalog$^{\pm}$ un frammento della famiglia Datalog$^{\pm}$ descritta in precedenza.
Il capitolo è organizzato come segue, nella sezione 1.1 verrà descritto in maniera approfondita il core del linguaggio, che a sua volta è diviso in due sottosezione, la sezione 1.1.1 che descrive il core logico dei linguaggi Datalog$^{\pm}$, tutte le proprietà e i vincoli che esso induce, arricchito con degli esempi, la sezione 1.1.2 che descrive il core logico di Vadalog,: le proprietà della famiglia Warded Datalog$^{\pm}$, la sezione 1.1.3 descrive la complessità e le caratteristiche del linguaggio Vadalog, infine nella sezione 1.1.4 vengono descritte le estensioni adottate dal linguaggio Vadalog che permettono funzionalità aggiuntive al linguaggio base.
Nella sezione 1.2 è possibile trovare dei programmi Vadalog di esempio per permettere una maggiore comprensione del linguaggio. Questo capitolo prende spunto da diversi articoli pubblicati dal gruppo VADA, sia per alcune nozioni teoriche che per esempi pratici descritti [BGPS17, GP15].

1.1 Il core del linguaggio

Il linguaggio Vadalog, fa parte della famiglia di linguaggi Warded Datalog$^{\pm}$, che a sua volta rappresenta un frammento della famiglia di linguaggi Datalog±, ovvero una restrizione della famiglia Datalog$^{\pm}$, in particolare garantendone la decidibilità.
In questa sezione andremo a focalizzarci proprio su tali famiglie di linguaggi e le relative complessità, sono presenti tre sezioni, una che descrive il core logico dei linguaggi appartenenti alla famiglia di linguaggi Datalog$^{\pm}$, una che descrive il core logico di Vadalog, e come soddisfa la decidibilità, infine è presente una sezione per descrivere la complessità del core logico Vadalog.

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 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 della famiglia Datalog±.

1.1.2 Il core logico di Vadalog

Vadalog è un linguaggio logico e rappresenta un'estensione di Datalog, quindi un programma Vadalog è composto da un insieme di regole.

In ogni regola è possibile trovare una testa ed un corpo, nel quale ogni testa è composta da un insieme di atomi, ed ogni corpo da un insieme di atomi e un insieme di condizioni. Gli atomi del corpo possono rappresentare: degli atomi di input (collegamento diretto ad unità di storage), presenti in teste di altre regole o nella regola stessa (ricorsione). Ogni atomo può essere composto da un numero indefinito di argomenti, che possono essere variabili o costanti.
Le condizioni sono di varia natura, possono rappresentare l'assegnazione di una variabile, una condizione booleana che si vuole verificare e tante altre possibilità implementate dal linguaggio Vadalog.
Un esempio di regola contenente atomi composti da variabili e costanti:

$$a(X) : -b(X, "Hi")$$

Poiché Vadalog appartiene alla famiglia di linguaggi Warded Datalog$^{\pm}$, contiene quantificatori esistenziali nelle teste generando valori nulli.
Nel linguaggio Vadalog, ci sono tre tipologie di variabili che sono contraddistinte:

1. Harmless: Variabili standard presenti nel corpo di una regola, che non possono assumere valori nulli.

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

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

Vadalog si basa sulla nozione di *wardedness*, proprietà ereditata dalla famiglia di linguaggi Warded Datalog$^{\pm}$ [GP15].
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. 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).

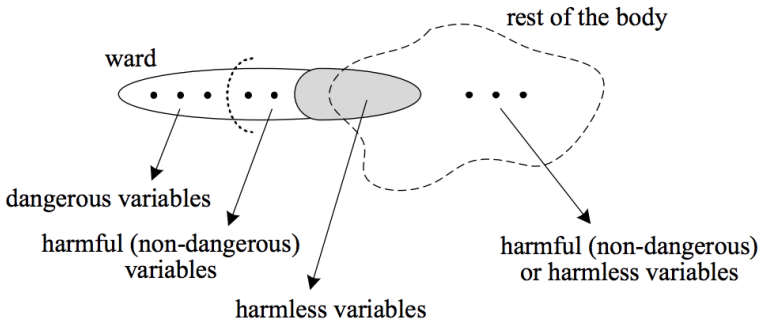

Figura 1.1: Corpo di una regola warded [GP15].

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.

Una regola è warded se vengono rispettate entrambe le condizioni appena citate, in Figura 1.1 possiamo vederne un esempio grafico.

1.1.3 Complessità

In questa sezione viene descritta la complessità di Warded Datalog$^{\pm}$, anche facendo riferimenti a possibili frammenti di esso.
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.

Anche se la complessità dei dati è accettabile per applicazioni convenzionali, può risultare inadatta per applicazioni Big Data.
Ciò solleva una questione se ci siano frammenti di Warded Datalog$^{\pm}$ che garantiscano una minore complessità dei dati, ma mantengano allo stesso tempo le proprietà descritte in precedenza. Tale frammento dovrebbe essere più debole di Datalog completo, poiché quest'ultimo è PTIME-completo nella complessità dei dati.
Tale frammento Warded Datalog$^{\pm}$, definito "*StronglyWarded*", dovrebbe avere complessità NLOGSPACE, e si potrebbe ottenere limitando il modo in cui viene utilizzata la ricorsione. Prima di dare la definizione di StronglyWarded definiamo la nozione di grafo dei predicati.
Il grafo dei predicati di Σ, chiamato PG(Σ), è un grafo diretto (V,E), in cui l'insieme

dei nodi V è composto da tutti i predicati presenti in Σ, ed abbiamo un arco da un predicato P ad un predicato R se esiste $\sigma \in \Sigma$ tale che P è presente nel corpo di σ ed R è presente nella testa di σ. Si consideri un insieme di nodi S $\subseteq$ V e un nodo R $\in$ V, diciamo che R è Σ-raggiungibile da S se esiste almeno un nodo P $\in$ S che può raggiungere R attraverso un percorso in PG(Σ).
Possiamo ora introdurre la definizione di *strong wardedness*. Un insieme di regole esistenziali Σ viene chiamato strongly-warded se Σ è warded e per ogni $\sigma \in \Sigma$ nella forma

$$\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} P_1(\bar{x}, \bar{y}), ..., P_n(\bar{x}, \bar{y})$$

esiste al massimo un atomo di $\phi(\bar{x}, \bar{y})$ il cui predicato è Σ-raggiungibile da $\{P_1, ..., P_n\}$. Strongly-Warded Datalog$^{\pm}$ è composto da insiemi finiti di regole esistenziali che sono strongly-warded.
Intuitivamente, in un insieme di regole esistenziali strongly-warded, ogni regola σ non è ricorsiva. Si può quindi dimostrare che il nostro principale compito inferenza di tuple nel processo di reasoning in Strongly-Warded Datalog$^{\pm}$ è NLOGSPACE nella complessità dei dati. Inoltre questo linguaggio raffinato rimane abbastanza potente per catturare OWL 2 QL e esteso da una lieve negazione può esprimere ogni query SPARQL.
Come detto in precedenza, la complessità dei dati NLOGSPACE esclude immediatamente il datalog completo. Tuttavia Strongly-Warded Datalog+- include alcuni importanti e ben studiati frammenti di Datalog:

- Datalog non ricorsivo, dove il grafo dei predicati è quindi aciclico.
- Linear Datalog, in cui ogni regola può avere al massimo un predicato nel suo corpo.

Per ottenere il core logico di Vadalog, i linguaggi appena discussi, ovvero Warded, Strongly-Warded e Linear Datalog$^{\pm}$, sono arricchite di utili funzionalità senza pagarne un prezzo in complessità.
Infatti si considerano i vincoli negativi della forma $\forall \bar{x}(\phi(\bar{x}) \rightarrow \bot)$, dove ϕ è una congiunzione di atomi e $\bot$ denota la costante false. Vengono presi in considerazione anche i vincoli di uguaglianza (come condizioni) della forma $\forall \bar{x}(\phi(\bar{x}) \rightarrow x_i = x_j)$, dove ϕ è una congiunzione di atomi e x_i e x_j sono variabili di $\bar{x}$, che non interagiscono con le

regole esistenziali. Questa classe di vincoli di uguaglianza è conosciuta come non conflittuale [CGP12], ovvero non avrò mai dei conflitti con variabili esistenziali.
Si noti che se consideriamo vincoli arbitrari di uguaglianza, senza restrizioni, allora il nostro compito di reasoning diventa indecidibile [CV85].

1.1.4 Estensioni di Vadalog

Al fine di essere efficace per applicazioni nel mondo reale, abbiamo esteso il core logico di Vadalog descritto in precedenza con un insieme di funzionalità aggiuntive di utilità pratica. Risultano essenziali per permettere un'analisi più dettagliata sui dati. Nel seguito della sezione elencheremo tutte le estensioni che sono state applicate in Vadalog.

Tipi di dato: È possibile definire costanti e variabili. Il linguaggio supporta i tipi di dato più comuni: Integer, Float, String, Boolean, Date, ma esiste anche il supporto per tipi di dato compositi, come i Set.

Espressioni: Le variabili e le costanti possono essere combinate in espressioni, definite ricorsivamente come variabili, costanti o combinazioni di esse, per le quali sosteniamo molteplici operazioni per diversi tipi di dato: somma, moltiplicazione, divisione, per Integer e Float; contenimento, aggiunta, cancellazione degli elementi per i Set; operazioni su stringhe (contains, starts-with, ends-with, index-of, substring, ecc.), operazioni sui booleani (and, or, not, ecc.).
Le espressioni possono essere usate nei corpi di regola come lato sinistro (LHS) di una condizione, cioè il confronto (<,>,=,<=,>=,<>) di una variabile del corpo con l'espressione stessa o come LHS di un'assegnazione, cioè la definizione di un valore calcolato specificatamente, spesso utilizzata come variabile di testa quantificata esistenzialmente.

Funzioni di Skolem: I valori dei nulli possono essere calcolati con funzioni. Si presume che siano deterministici (restituendo un nullo univoco) e avere range disgiunti. Alle funzioni di Skolem è possibile applicare delle implementazioni procedurali, che possono essere definite in diversi linguaggi, definendo il calcolo che effettuerà la funzione.

Aggregazioni monotoniche: Vadalog supporta l'aggregazione (min, max, somma, prodotto, conteggio), mediante un'estensione alla nozione di aggregazioni monotoniche [SYZ15], che consente di adottare l'aggregazione anche in presenza di ricorsioni conservando monotonicità. Le recenti applicazioni di Vadalog in casi di utilizzi industriali impegnativi hanno dimostrato che tali aggregazioni sono molto efficienti in molte impostazioni Big Data del mondo reale.

Annotazioni: Gli atomi nelle regole sono decorati con annotazioni, in modo che un passo nel processo di reasoning innesca il componente esterno. Le fonti di dati e gli obiettivi possono essere dichiarati adottando annotazioni di input/output. Le annotazioni sono fatti particolari che aumentano le serie di regole esistenti con comportamenti specifici. Ad esempio una semplice annotazione che può essere di input/output, può prendere/persistere i dati da un sistema esterno, come un database relazionale. I fatti possono essere quindi derivati, per esempio da un database relazione o un graph db, che vengono acceduti con le due annotazioni di esempio, rispettivamente:

$$@bind(\text{"}Own\text{"},\text{"}rdbms\text{"},\text{"}companies.ownerships\text{"}).$$

$$@qbind(\text{"}Own\text{"},\text{"}graphDB\text{"},$$
$$\text{"}MATCH(a)-[o:Owns]->(b)$$
$$RETURN a,b,o.weight\text{"}).$$

Dove la prima annotazione effettua una proiezione di tutta la tabella "ownerships", mentre nella seconda viene effettuata una vera e propria query al graphDB in Cypher (linguaggio per interrogazione di graphDB).
Un approccio simile viene utilizzato anche per colmare le piattaforme di machine learning esterne e di estrazione dati nel sistema. Facciamo un esempio, utilizzando il framework di estrazione dati OXPath [FGG+13], un'estensione di XPath (linguaggio che permette di individuare nodi all'interno di un documento XML) che interagisce con le applicazioni web per estrarre le informazioni ottenute durante la navigazione web.
Supponiamo che le nostre informazioni sulla proprietà di un'azienda siano parziali, mentre le altre informazioni possono essere reperite dal web. In particolare, supponiamo

che un registro aziendale agisca come motore di ricerca web, prendendo come input un nome di un'azienda e restituendo come pagine separate le società di proprietà. Queste informazioni possono essere ottenute come segue:

$$
\begin{aligned}
&@qbind(\text{"}Own\text{"},\text{"}oxpath\text{"},\\
&\text{"}doc('http://company_register.com/ownerships')\\
&/descendant :: field()[1]/\{\$1\}\\
&/following :: a[.\#\ =' Search']/\{click/\}\\
&/(//a[.\#\ =' Next']/\{click/\})*\\
&//div[@class =' c'] :< comp >\\
&[./span[1] :< name = string(.) >]\\
&[./span[3] :< percent = string(.) >]\text{"}).
\end{aligned}
$$

Interazioni interessanti si possono osservare in scenari più sofisticati, in cui il processo di reasoning e l'elaborazione dei componenti esterni sono più intrecciati.
Inoltre è possibile definire delle annotazioni di mapping, che hanno lo scopo di definire diverse informazioni sui dati di output, ad esempio i tipi, il nome della colonna, la posizione, ecc., ma possono essere anche evitati, poiché il sistema è in grado di inferire il tipo di dato.

Reasoning probabilistico: Vadalog offre il supporto per i casi di base in cui è possibile garantire il calcolo scalabile. I fatti sono considerati probabilisticamente indipendenti ed una forma minimalistica di inferenza probabilistica è offerta come risposta alla query. I fatti possono essere adornati con misure di probabilità.
Quindi se l'insieme delle regole esistenziali rispetta specifiche proprietà sintattiche che garantisco il calcolo probabilistico, i fatti risultanti della query sono arricchiti dalla loro probabilità marginale.
Nell'esempio utilizziamo il probabilistic reasoning per tenere conto delle proprietà incerte (ad esempio a causa di fonti inaffidabili), stabilendo i fatti con la loro probabilità,

per trarre conclusioni sui rapporti di controllo aziendali:

$$0.8 :: Own(\text{"}ACME\text{"}, \text{"}COIN\text{"}, 0.7)$$
$$0.3 :: Own(\text{"}COIN\text{"}, \text{"}SAVERS\text{"}, 0.3)$$
$$0.4 :: Own(\text{"}ACME\text{"}, \text{"}GYM\text{"}, 0.55)$$
$$0.6 :: Own(\text{"}GYM\text{"}, \text{"}SAVERS\text{"}, 0.4).$$

Post-processing Annotations: Poiché spesso sono necessari dopo che il risultato è stato prodotto, Vadalog supporta molti di essi mediante annotazioni (in questo caso annotazioni diverse rispetto a quelle generali, poiché vengono richiamate soltanto alla fine del processo di reasoning): ordinamento dei valori risultanti, ad esempio @orderby("Control", 1) ordina i fatti di control in base al primo campo; rimozione dei duplicati (@distinct), che in condizioni specifiche possono risultare indesiderati; aggregazioni non monotoniche sul risultato finale, senza le limitazioni indotte dalla ricorsione.

Nel complesso, il linguaggio consente di superare il reasoning basato sulla logica e il machine learning in tre modi. In primo luogo, il linguaggio supporta l'inferenza probabilistica in casi di base come visto in precedenza.
In secondo luogo, le estensioni del linguaggio di base forniscono tutte le funzionalità necessarie per estrarre e incorporare algoritmi avanzati di inferenza in modo che possano essere eseguiti direttamente dal sistema Vadalog e quindi sfruttare le proprie strategie di ottimizzazione.
Infine, per le applicazioni di machine learning più sofisticate, le estensioni consentono una semplice interazione con librerie e sistemi specializzati.

1.2 Esempi di programmi Vadalog e loro utilizzo

In questa sezione verranno presentati diversi programmi Vadalog di esempio, utilizzati per effettuare calcoli particolari ed utilizzare alcune delle estensioni descritte sopra, e spiegati nel dettaglio.
Partiremo da esempi molto semplici, dove interagiscono pochi elementi (join tra due elementi), fino ad arrivare ad esempi più complessi che utilizzano ricorsione, espressioni,

funzioni di skolem, integrazione di database, post processing annotations e così via.
Un primo esempio di un programma Vadalog molto semplice:

```
a(1,0).
b(1,2).
c(X,Y,Z) :- a(X,Y), b(X,Z).
@output("c").
```

In questo esempio viene effettuato il join tra gli atomi 'a' e 'b', sulla prima variabile e vengono proiettate tutte le variabili. In questo caso, i fatti di input vengono aggiunti manualmente a(1,0) e b(1,2), quindi il risultato sarà c(1,0,2).
I join possono coinvolgere anche più atomi tra di loro, e possono essere abbinati anche alla ricorsione, nel seguente esempio ne vediamo un'applicazione:

```
a(1.0,"string").
b(1.0,2).
c(1.0,4,2017-09-28 12:29:00).
c(X,Y,W) :- a(X,Y), b(X,Z), c(X,V,W).
@post("unique", "c").
@output("c").
```

In questo esempio, viene effettuato il join tra tre atomi, di cui uno ricorsivo, vengono dati i fatti di input (anche un input per 'c'), e viene eseguita anche un'operazione di post-processing 'unique' che ha l'obiettivo di rimuovere i duplicati.
I fatti di input in questo caso sono vari: double, interi, stringhe e date. Come detto in precedenza, oltre ad essere dichiarati esplicitamente come fatti possono rappresentare tabelle su un database, relazionale e non, o anche le righe di un file csv, ecc.
Attraverso questa integrazione è possibile effettuare delle statistiche incrociando dati provenienti da storage distinti, combinando dati, e producendo dati di output che possono rappresentare nuova conoscenza, alla base del processo di reasoning.
Infatti è anche possibile direzionare gli output verso una determinata fonte, oltre la stampa a schermo. È possibile salvare un output ad esempio in un database, in un file csv e quant'altro. Questo risulta molto importante poiché uno degli obiettivi del processo reasoning è produrre nuova conoscenza, analizzando ed incrociando dati già acquisiti.

Nel prossimo esempio vedremo un'interazione che può avvenire tra diverse sorgenti dati all'interno di un programma Vadalog con persistenza del risultato:

```
@input("a").
//a composto da due colonne
@bind("a", "postgres", "vadaschema", "Table").
@input("b").
//b composto da due colonne
@bind("b", "csv", "absolute/path/to/folder", "filename.csv").
c(X,Y,Z) :- a(X,Y), b(X,Z).
@output("c").
@bind("c", "csv", "absolute/path/to/folder", "result.csv").
```

In questo esempio i fatti di input vengono presi da un database postgres (con schema "vadaschema" e tabella "Table") contenente due colonne e da un file csv (che è nella directory "absolute/path/to/folder" chiamato "filename.csv") contenente anch'esso due elementi per riga. Viene poi effettuato il join tra i due input in base al primo elemento, ed il risultato viene salvato in un nuovo file csv, che viene appositamente creato nella directory "absolute/path/to/folder".
C'è anche la possibilità di effettuare delle preselezioni dal database, con un'annotazione chiamata *qbind*, che permette di effettuare una vera e propria query in SQL, di seguito vediamo un esempio sulla base del precedente:

```
@input("a").
//a composto da due colonne
@qbind("a", "postgres", "vadaschema",
"select name1, name2
from Table
where name1 = 'teststring'").
@input("b").
//b composto da due colonne
@bind("b", "csv", "absolute/path/to/folder", "filename.csv").
c(X,Y,Z) :- a(X,Y), b(X,Z).
@output("c").
@bind("c", "csv", "absolute/path/to/folder", "result.csv").
```

È anche possibile inserire delle espressioni all'interno dei programmi Vadalog, esse possono essere utilizzate, come condizioni o assegnazioni di una variabile esistenziale. Per entrare più nel dettaglio vediamo qualche esempio:

```
a(1,5).
b(2,7).
c(X,V) :- a(X,Y), b(V,W), Y >= 5, W >= 5.
@output("c").
```

In questo esempio sono presenti due condizioni Y >= 5 e W >= 5, quindi la proiezione in 'c' viene fatta soltanto se entrambe sono verificate, poiché Y e W, con i fatti specificati valgono 5 e 7, le condizioni sono verificate e quindi l'output sarà c(1,2).

```
a(1,5).
b(2,7).
c(X,V,J) :- a(X,Y), b(V,W), J = X+V.
@output("c").
```

In quest'altro esempio invece le espressioni vengono utilizzate per assegnare un valore ad una variabile esistenziale, in questo caso J=X+V, quindi l'output sarà c(1,2,3). È possibile integrare skolem function (che vengono considerate come espressioni), al quale è possibile applicare delle funzioni esterne con cui possono essere definite, o altrimenti generano un nullo. Vediamone alcuni esempi:

```
b(1,2).
a(X,Y,Z) :- b(X,Y),Z=#f(X,Y).
c(K) :- b(X,Y),K=#f(X,Y).
d(K) :- b(X,Y),K=#g(X,Y).
@output("a").
@output("c").
@output("d").
```

```
b(1,2).
c(3,2).
a(Y,Z) :- b(X1,Z),c(X2,Z),Y=#f(X1,X2).
@output("a").
@implement("#f","java","com.package1.package2.myClass","staticmethodName").
```

Nel primo esempio non vengono definite le implementazioni delle funzioni, quindi ritorneranno un valore nullo, #f(X,Y) rappresenta una funzione in questo esempio. La particolarità delle funzioni è che se in due regole la stessa funzione prende come input gli

stessi valori, allora quest'ultima produrrà lo stesso nullo. In questo esempio il risultato prodotto sarà a(1,2,z_1), c(z_1) e d(z_2), dove z_1 e z_2 rappresentano i nulli. Nel secondo esempio invece, la funzione 'f' viene implementata attraverso una procedura esterna (in questo caso un metodo Java) del quale non entriamo nel dettaglio implementativo, ma il risultato della funzione Java sarà lo stesso risultato di 'f'.

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.

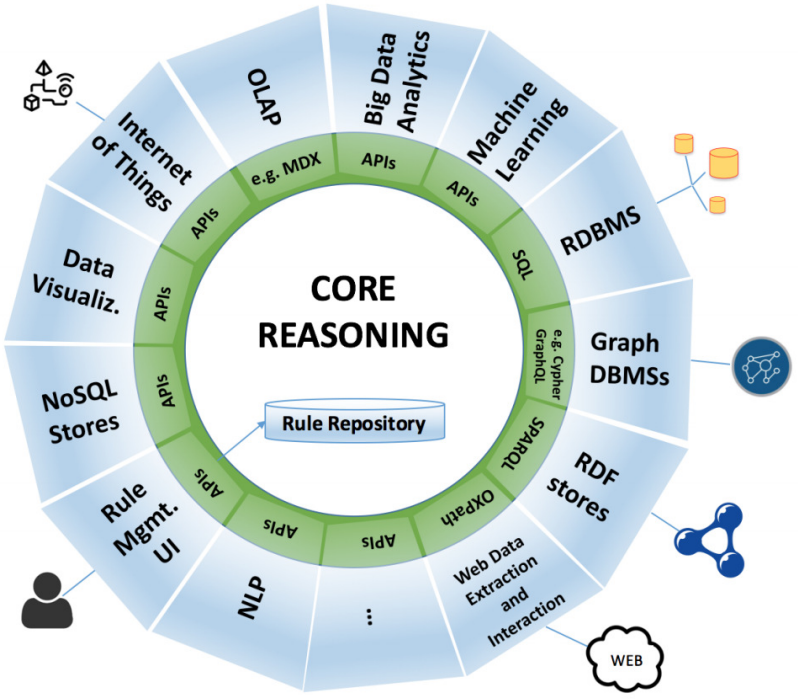

Figura 2.1: Schema dell'architettura dell'Engine di Vadalog [BGPS17].

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.

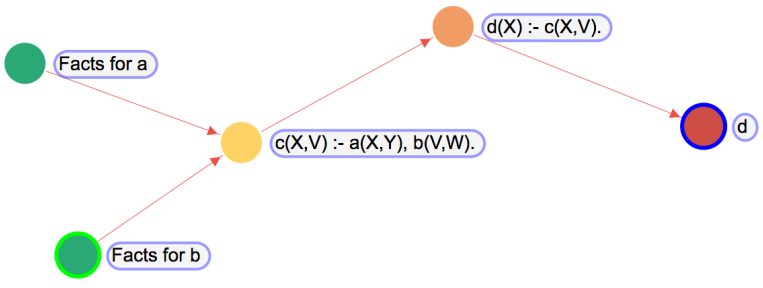

Figura 2.2: Esempio di query plan.

Il Vadalog Reasoner utilizza un approccio stream-based (o approccio a pipeline). Tale approccio [Wikb] 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.

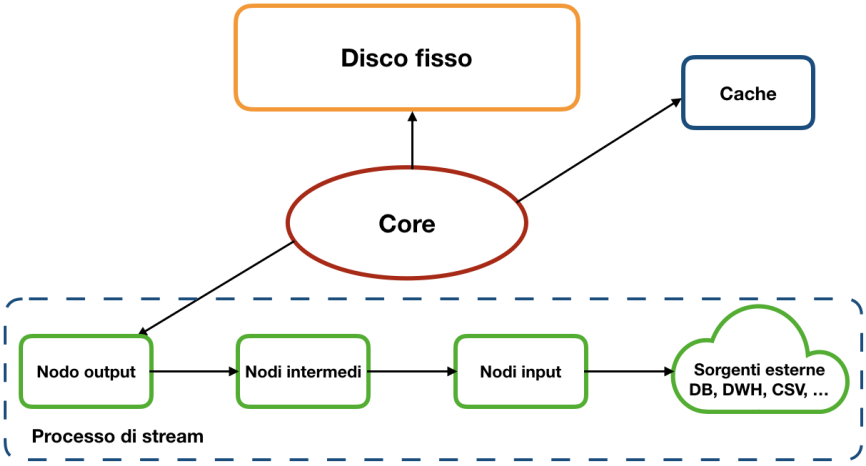

Figura 2.3: Funzionamento dell'architettura stream-based.

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.

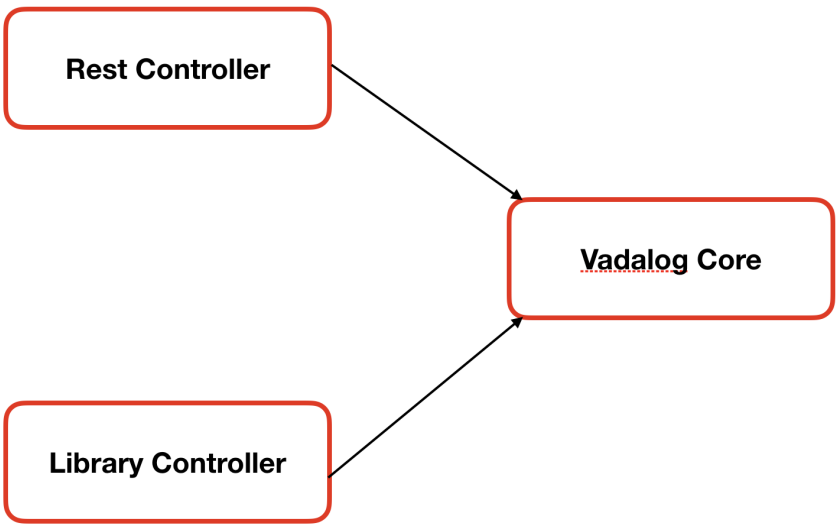

Figura 2.4: Sistemi per interfacciarsi con Vadalog Engine.

Capitolo 3

Supporto per nuove features

In questo capitolo verranno descritte nel dettaglio le implementazioni di feature core del Vadalog Reasoner, di cui mi sono occupato durante il mio periodo di Tesi, per l'esecuzione di programmi Vadalog.
Il capitolo è organizzato come segue, nella sezione 3.1 verranno descritte nel dettaglio le tecniche di ottimizzazione e le loro implementazioni, in particolare le ottimizzazioni di push selections e projections down (sezione 3.1.1), la gestione di teste e join multipli (sezione 3.1.2) e l'individuazione e inversione delle ricorsioni destre (sezione 3.1.3).

3.1 Tecniche di ottimizzazione

Inizialmente il Vadalog Reasoner, effettuava ben poche ottimizzazioni, ad esempio non veniva ottimizzata efficientemente la ricorsione, non venivano applicate delle ottimizzazioni note come *Push Selections e Projections Down*. Non erano presenti ottimizzazioni che permettevano un guadagno sull'esecuzione di programmi, ciò portava ad un limite del Vadalog Reasoner anche in ambito Big Data.
Le tecniche di ottimizzazione sono basate sulla riscrittura, ovvero quando viene lanciato un programma esso viene riscritto applicando le ottimizzazioni ed infine dato in pasto al reasoner.

3.1.1 Push selections e projections down

In questa sezione verranno descritte come sono state gestite le operazioni di push selections e projections down all'interno del Vadalog Reasoner.
Queste operazioni rappresentano delle ottimizzazioni note nel settore basi di dati da diversi anni, il loro obiettivo è la trasformazione di una query in un'altra equivalente (stesso risultato), ma anticipando selezioni e proiezioni, in modo da avere dei benefici sui costi temporali e computazionali.
Nel nostro caso, poiché ci troviamo di fronte ad un linguaggio della famiglia Datalog$^{\pm}$, tali operazioni creano molte problematiche in più rispetto a linguaggi base di interrogazione per database, come SQL.
Questo perché Datalog ha molti più casi da gestire rispetto al linguaggio SQL, ad esempio, vanno gestite le ricorsioni, le variabili esistenziali, regole che possono avere in testa l'atomo A, sul corpo l'atomo B ed un'altra regola che può avere in testa l'atomo B e sul corpo l'atomo A (un ciclo) e quant'altro.

Nella fase di push selections down, l'obiettivo è quello di anticipare le selezioni il prima possibile, ovvero cercare di anticipare le selezioni il più vicino possibile alle regole che coinvolgono nodi di input. Come possiamo vedere nell'esempio (un caso base):

```
b(1).
b(2).
a(X) :- b(X).
d(X) :- a(X), X>10.
@output("d").
```

Dopo la trasformazione diventa:

```
b(1).
b(2).
a(X) :- b(X), X>10.
d(X) :- a(X).
@output("d").
```

Dove la proiezione con la variabile X è stata anticipata alla regola più vicina al nodo di input.

Spesso è necessario che una variabile va anticipata in N regole che coinvolgono (selezionano) tutte la stessa variabile che viene selezionata, come possiamo vedere nell'esempio:

```
b(1).
c(30,33).
a1(X) :- b(X).
a2(X,Y) :- c(X,Y).
d(X) :- a1(X), a2(X,Y), X>1.
@output("d").
```

Dopo la trasformazione diventa:

```
b(1).
c(30,33).
a1(X) :- b(X), X>1.
a2(X,Y) :- c(X,Y), X>1.
d(X) :- a1(X), a2(X,Y).
@output("d").
```

In questo caso la variabile X viene utilizzata in due regole, quindi la selezione viene anticipata in entrambe le regole che coinvolgono tale variabile.
In entrambe le fasi di push selections e projections down, si tiene conto della posizione della variabile nell'atomo anziché del nome della variabile, questo perché in altre regole i nomi delle variabili possono essere in ordine inverso, possiamo vedere un esempio pratico:

```
b1(10,25).
c1(25,10).
b(Y,X) :- b1(X,Y).
c(X,Y) :- c1(X,Y).
a(X) :- b(X,Y),c(Y,X),X>20.
@output("a").
```

Viene trasformato in:

Come possiamo vedere la variabile su cui viene effettuata la selezione 'X', che corrisponde alla prima posizione dell'atomo b ed alla seconda posizione dell'atomo c, quando viene anticipata viene opportunamente cambiata di nome in base alla determinata regola.

```
b1(10,25).
c1(25,10).
b(Y,X) :- b1(X,Y), Y>20.
c(X,Y) :- c1(X,Y), Y>20.
a(X) :- b(X,Y),c(Y,X).
@output("a").
```

La fase di push projections down si occupa di anticipare le selezioni il più vicino possibile ai nodi di input ed inoltre di rimuovere le proiezioni inutilizzate, ad esempio se in una regola proietto due variabili, in cui una non è coinvolta con l'output o con un'interazione per produrre l'output. Un esempio di rimozione di variabile inutilizzata è il seguente:

```
a(2).
b(1,1).
q(X,Y) :- b(X,Y).
c(X) :- a(X), q(X,Y).
d(X) :- c(X).
@output("d").
```

Che viene trasformato in:

```
a(2).
b(1,1).
q(X,Y) :- b(X,Y).
c(X) :- a(X), q(X).
d(X) :- c(X).
@output("d").
```

Come possiamo vedere nella regola c(X) :- a(X), q(X,Y) viene eliminata la variabile 'Y' poiché inutilizzata.
Un esempio di proiezione anticipata è il seguente:

```
b(1).
c(2,5).
a1(X) :- b(X).
a2(X,Y) :- c(X,Y).
d(X) :- a1(X), a2(X,5).
@output("d").
```

Che viene trasformato in:

```
b(1).
c(2,5).
a1(X) :- b(X).
a2(X) :- c(X,5).
d(X) :- a1(X), a2(X).
@output("d").
```

Nel quale viene anticipata la proiezione dell'atomo 'c', e viene rimossa la variabile 'Y' dall'atomo 'a2', poiché inutilizzata.

L'implementazione delle procedure di push selections e projections down è stata effettuata come una riscrittura del programma, ovvero prima di mandare il codice in pasto al reasoner, esso viene opportunamente trasformato seguendo le tipologie di ottimizzazioni sopra descritte.
Nella fase di push selections down, si visita il grafo d'esecuzione partendo dai nodi di output, durante la quale vengono controllate tutte le variabili delle condizioni della regola corrente, e si fa un match con le variabili degli atomi nel corpo della regola, a questo punto ci sono due possibili strade da percorrere:

- La variabile occorre in N atomi nel corpo e tutti gli atomi sono di input. In questo caso la condizione viene lasciata nella regola corrente, poiché non è possibile anticipare la selezione, si continua poi la visita del grafo per vedere se ci sono altre selezioni che è possibile anticipare.

- La variabile occorre in atomi che sono di input e in atomi che non sono di input. In questo caso, viene lasciata la condizione alla regola corrente, ma si tiene conto degli altri atomi (che non sono di input) coinvolti, proseguendo la visita del grafo, quando arriveremo alle regole che hanno tali atomi in testa, vengono nuovamente verificati gli atomi nel corpo, nel caso in cui siano tutti di input allora viene aggiunta la selezione anche a queste regole, altrimenti si prosegue finché non arriviamo alla regola contenente soltanto atomi di input.

La fase di push projections down è molto simile, ma anziché controllare le condizioni, in ogni regola si verifica la presenza di costanti, in caso positivo si procede con lo stesso criterio del push selections down. Inoltre si verifica anche la presenza di variabili inutili,

ad esempio nella testa ho le variabili [X,Y] e nel corpo [X,Y,J], in questa caso, J viene rimossa dall'atomo nel corpo della regola e conseguentemente dalla testa della regola contenente l'atomo coinvolto.

3.1.2 Gestione di teste multiple e join multipli

In questa sezione verranno descritte le ottimizzazioni in presenza di teste e join multipli. Vadalog è un linguaggio che permette regole standard da poter utilizzare nella forma head :- body, dove head è rappresentato da un singolo atomo e body da una lista di atomi e una lista di condizioni.
Tuttavia, vogliamo gestire diverse funzionalità che permettono un'espressività maggiore, una di queste è la possibilità di definire una regola con più teste, ad esempio se ho diversi atomi che sono composti dallo stesso corpo anziché far scrivere al programmatore un numero di regole pari al numero di atomi, viene effettuata una riscrittura che permette di scrivere una regola con più teste. Un esempio è il seguente:

h1(X,Y),h2(Y,Z),h3(Z,W) :- a(X,Y,Z),b(Z,M).

Dove gli atomi h1, h2, h3 condividono lo stesso corpo.
In questo caso viene riscritto, prima di essere dato in pasto al reasoner nel seguente programma:

h_tmp(X,Y,Z,M) :- a(X,Y,Z),b(Z,M).
h1(X,Y) :- h_tmp(X,Y,Z,M).
h2(Y,Z) :- h_tmp(X,Y,Z,M).
h3(Z,W) :- h_tmp(X,Y,Z,M).

Viene quindi definita una regola standard che contiene il corpo condiviso con le N teste, e aggiunta una regola lineare (uno ed un solo atomo nel corpo) per ogni testa.

Spesso sono presenti regole, che nel corpo hanno più di 2 atomi che sono vincolati tramite join, l'uno con l'altro (ad esempio join tra tre elementi o join incrociati tra tre atomi).
Quando sono presenti join tra tre o più atomi, l'aumento del tempo di computazione è

proporzionale alla crescita del numero di atomi interessati. Ad esempio data una regola di questo tipo:

c(Z,X,M) :- a(X,Y),b(Y,K),c(K,M).

In questo esempio gli atomi 'a', 'b' e 'c' sono coinvolti in un join incrociato, ovvero l'atomo 'a' è in join con 'b' e quest'ultimo con 'c'.
Nella riscrittura la regola generalizzata composta da N atomi, di cui M (>=3) sono in join tra loro, viene splittata in M regole in cui vengono definiti join tra due elementi soltanto, e gli atomi non coinvolti nel join, vengono inseriti nella regola finale.
Quindi nel nostro esempio il programma dopo la riscrittura diventa:

v_atom1(X,K) :- a(X,Y),b(Y,K).
c(Z,X,M) :- v_atom1(X,K),c(K,M).

Il join viene quindi splittato in due regole, dando il medesimo risultato di output.

3.1.3 Individuazione e inversione delle ricorsioni destre

Un altro collo di bottiglia per il tempo di computazione e l'efficienza è la ricorsione, in particolare le ricorsioni destre (atomo che ricorre si trova alla fine del corpo della regola) risultano meno efficienti delle ricorsioni sinistre (atomo che ricorre si trova all'inizio del corpo della regola).
Tale perdita di efficienza si ha sopratutto in presenza di join. Ricordiamo che Vadalog utilizza un algoritmo di join simile al nested loop (ogni ennupla della prima relazione viene paragonata ad ogni ennupla della seconda relazione).
Supponiamo che ci sia una regola Vadalog del tipo head :- A,B, dove A è un'atomo non ricorsivo e B ricorsivo, quindi una ricorsione destra, e che A abbia N ennuple, e B abbia M ennuple.
Quando andiamo ad effettuare il nested loop join, ogni ennupla di A effettua una chiamata ricorsiva su B (per paragonarla) per tutte le ennuple di B, vengono quindi fatte NxM chiamate ricorsive circa.
Nel caso in cui la regola abbia una ricorsione sinistra (quindi dopo la riscrittura), ovvero head :- B,A, in questo caso è B che viene paragonata ad A, quindi vengono prima fatte

le chiamate ricorsive e poi il risultato delle chiamate paragonato alle ennuple di B, quindi vengono fatte M chiamate ricorsive circa. Quindi la riscrittura porta un guadagno notevole sulla base del numero di chiamate ricorsive.
Vediamo un'esempio pratico di come avviene la riscrittura:

a(Y) :- b(X,Y), a(X).

Viene trasformato in:

a(Y) :- a(X), b(X,Y).

L'algoritmo di riscrittura ha lo scopo di portare l'atomo ricorsivo come primo atomo del corpo, il resto della regola rimane invariato, come il risultato finale.

3.2 Supporto a nuovi tipi di dato, sorgenti e funzionalità

3.3 Benchmark

3.4 Vadalog console

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.

[CV85] Ashok K Chandra and Moshe Y Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.

[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.

[FGG+13] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Sellers. Oxpath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal*, 22(1):47–72, 2013.

[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.

[SYZ15] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 867–878. IEEE, 2015.

[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/.

[Wika] Wikipedia. `https://en.wikipedia.org/wiki/Knowledge_economy`.

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