UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea Magistrale 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

Alla mia famiglia

Ringraziamenti

Ringrazio innanzitutto il Professor Paolo Atzeni, Relatore e l'Ing. Luigi Bellomarini, Correlatore, per i suggerimenti e il supporto ricevuto durante questo periodo.

In questi cinque anni di università, ho conosciuto persone con le quali ho condiviso molto che ringrazio con tutto il cuore.

In particolare ringrazio il gruppo studio che abbiamo formato qui all'università: Federico, Gabriele, Valerio, Alessandro, Carlo, Antonio, Dalila, Emanuele.

Indice

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" [33] 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 [5].

Come definito in [5] 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 accettabile.

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 [2] .
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 logica. Durante gli anni è stato studiato in dettaglio, nel *data exchange* e nella *data integration* [17]. 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 Datalog [5].
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 [7, 9, 8].

Il sistema che presentiamo ed utilizziamo in questa tesi, Vadalog Reasoner, è il contributo dell'Università di Oxford al progetto VADA [28], 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. 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. 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*) [5].
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 operazioni aritmetiche e semplici aggregazioni.
- Non erano presenti ottimizzazioni particolari tali da permettere un guadagno sull'esecuzione di programmi Vadalog.

- Erano stati valutati 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.
- I tipi di sorgente dati a disposizione dell'utente finale erano in numero limitato. Ciò rappresentava un limite alle analisi possibili con il nostro sistema.
- Non era possibile integrare codice sorgente da altri linguaggi, né definire *user-defined functions* all'interno del linguaggio Vadalog. Ciò era un limite, poiché non permetteva di fare computazioni particolarmente difficili da specificare in un approccio dichiarativo e non era possibile 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 feature rispetto a tutte quelle fornite dal back-end.

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 ampliare la gamma di tipi di dato disponibili nel sistema.
- Tecniche di ottimizzazione in Datalog per migliorare le prestazioni. In particolare mi sono occupato delle ottimizzazioni di: 1. *push selections e push projections down*, ovvero la trasformazione di una query in un'altra equivalente, ma anticipando le selezioni e le proiezioni alle regole più "vicine ai dati". Ciò aumenta l'efficienza poiché coinvolge un numero minore di tuple nell'operazione di join. 2. Supporto per *ricorsione sinistra e destra*, in particolare trasformando le ricorsioni destre che risultano più onerose computazionalmente, vengono opportunamente trasformate in ricorsioni sinistre che risultano più efficienti. 3. 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 riallocati in un numero di regole pari al numero di atomi, in modo da ricondursi a efficienti join tra due atomi.
- Creazione di benchmark per effettuare test sulle performance del sistema e confrontarlo con sistemi esistenti. Mi sono occupato inoltre dello sviluppo di *iWarded*,

un piccolo tool che genera benchmark Warded Datalog$^{\pm}$. Tale sistema produce programmi Vadalog casuali, con caratteristiche configurabili attraverso un insieme di parametri.

- Supporto di nuove sorgenti: file CSV e database relazionali e non relazionali. Questo permette di combinare i dati provenienti da sistemi di storage eterogenei ed avvalersi di linguaggi e funzionalità di questi ultimi.

- Supporto alle user-defined functions. È possibile definire delle user-defined functions nell'ambito di programmi Vadalog. Tali funzioni possono essere parametriche e l'implementazione si può specificare in un linguaggio esterno a piacere, ad esempio Python o Java. Ciò risulta essenziale per l'utente per integrare all'interno di un programma Vadalog, funzionalità precedentemente sviluppate.

- Miglioramento notevole dell'interfaccia web (Vadalog console), con l'integrazione di elementi grafici che permettono l'interazione con tutti i servizi offerti dal sistema. L'interfaccia è stata anche estesa con un IDE (Integrated Development Environment) per lo sviluppo di codice Vadalog.

La tesi è organizzata come segue.

Il capitolo 1 è incentrato sul linguaggio Vadalog, in particolare è presente un approfondimento sul core logico della famiglia di linguaggi Datalog$^{\pm}$ e di Vadalog, sulla complessità computazionale e su varie estensioni del linguaggio.
Il capitolo 2 descrive il Vadalog Reasoner, in particolare le proprietà di un KGMS ed in modo approfondito l'architettura del nostro sistema.
Il capitolo 3 è dedicato alla descrizione nel dettaglio delle funzionalità riguardanti il mio contributo al progetto.
Il capitolo 4 descrive le prove sperimentali che ho effettuato 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.

Capitolo 1

Il linguaggio Vadalog

In questo capitolo parleremo in maniera più approfondita del linguaggio Vadalog e di tutte le sue particolarità. Questo capitolo descrive il linguaggio in linea generale, il mio contributo a riguardo è approfondito nei capitoli successivi.
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$^{\pm}$.
Il capitolo è organizzato come segue. Nella sezione 1.1 verrà descritto in maniera approfondita il core del linguaggio, 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. Nella sezione 1.2 è possibile trovare dei programmi Vadalog di esempio per 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 [5, 19].

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$^{\pm}$, 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 re-

lative complessità, la sezione 1.1.1 che descrive il core logico dei linguaggi Datalog$^{\pm}$, tutte le proprietà e i vincoli che esso induce, la sezione 1.1.2 che descrive il core logico di Vadalog, le proprietà della famiglia Warded Datalog$^{\pm}$, la sezione 1.1.3 che descrive la complessità e le caratteristiche del linguaggio Vadalog, infine la sezione 1.1.4, dove vengono descritte le estensioni adottate dal linguaggio Vadalog che introducono funzionalità aggiuntive al linguaggio base.

1.1.1 Il core logico dei linguaggi Datalog$^{\pm}$

L'obiettivo principale dei linguaggi Datalog$^{\pm}$ 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$^{\pm}$ 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 e che anche questo a sua volta è una persona.
In generale, una regola esistenziale è una formula del 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 dell'applicazione di un insieme di regole esistenziali Σ su un database D, in simboli Σ(D), è definita mediante una procedura di *chase* [15]. 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 coinvolte.
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 fissato e solo D è dato come input [7].
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 essere: atomi di input (che traggono i loro fatti direttamente dalla persistenza), atomi già presenti in teste di altre regole o nella regola stessa

(cioè la presenza di ricorsione).
Ogni atomo può essere composto da un numero indefinito di argomenti, che possono essere variabili o costanti.
Riportiamo un esempio di regola contenente atomi composti da variabili e costanti:

$$a(X) : -b(X, \text{“}Hi\text{”})$$

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

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 prodotti da un quantificatore esistenziale nella testa di un'altra regola.

3. *Dangerous*: Sono le variabili i cui valori, potenzialmente nulli, si propagano nella testa delle regola.

Vadalog si basa sulla nozione di *wardedness*, proprietà ereditata dalla famiglia di linguaggi Warded Datalog$^{\pm}$ [19].
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)}, il chase 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).

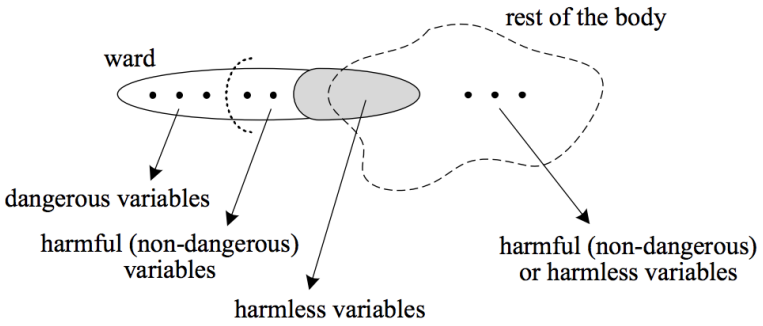

Figura 1.1: Corpo di una regola warded [19].

L'obiettivo della wardedness è quello di porre dei vincoli sulla modalità in cui i valori nulli vengono propagati come segue:

1. Tutte le variabili dangerous devono coesistere in un singolo corpo di un atomo A, chiamato il ward;

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 una rappresentazione grafica.

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$^{\pm}$ è composto da insiemi (finiti) di regole esistenziali warded, esso rappresenta un raffinamento di *Weakly-Frontier-Guarded Datalog*$^{\pm}$, che è una famiglia di linguaggi definita allo stesso modo ma senza la condizione 2 citata nella sezione 1.1.2 [4]. Weakly-Frontier-Guarded Datalog± è intrattabile nella complessità dei dati, infatti è EXPTIME-completo.

Warded Datalog± gode di diverse proprietà che lo rendono robusto ed utilizzabile in pratica:

- 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 ontologici principali come OWL 2 QL (linguaggio ontologico per la semantica del web [30])
- È adatto per la ricerca di grafi RDF (Resource Description Framework, strumento base proposto da W3C per la codifica, scambio e riutilizzo di metadati [31]). In realtà aggiungendo la negazione stratificata, si ottiene un linguaggio chiamato TriQ-Lite1.0 [19], che può esprimere ogni query SPARQL (linguaggio di query per gli RDF [32]) nell'ambito del regime di OWL 2 QL.

Anche se la data complexity di Vadalog è accettabile per applicazioni convenzionali, può risultare inadatta per applicazioni Big Data.
Ciò solleva il problema relativo all'esistenza di frammenti di Warded Datalog$^{\pm}$ che garantiscano una minore complessità dei dati, ma mantengano allo stesso tempo le proprietà descritte in precedenza.
Tale frammento di 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 diamo 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, l'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 con 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$^{\pm}$ 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 arricchiti 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 [9], ovvero senza conflitti con variabili esistenziali.
Si noti che se consideriamo vincoli arbitrari di uguaglianza, senza restrizioni, allora il nostro compito di reasoning diventa indecidibile [12].

1.1.4 Estensioni di Vadalog

Al fine di essere efficace per applicazioni nel mondo reale, il core logico di Vadalog descritto in precedenza è stato esteso con un insieme di funzionalità aggiuntive di utilità pratica, che risultano essenziali per permettere un'analisi più dettagliata sui dati. Nel seguito della sezione elenchiamo 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 supportiamo 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 monotone: Vadalog supporta l'aggregazione (min, max, somma, prodotto, conteggio), mediante un'estensione alla nozione di aggregazioni monotone [26], che consente di adottare l'aggregazione anche in presenza di ricorsioni. Le recenti applicazioni di Vadalog in casi di utilizzi industriali impegnativi hanno dimostrato che tali aggregazioni sono molto efficienti in vari scenari 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 due annotazioni mostrate negli esempi 1 e 2, rispettivamente:

Esempio 1.

```
@bind("Own","rdbms","companies.ownerships").
```

Esempio 2.

```
@qbind("Own", "graphDB",
"MATCH (a) - [o:Owns] -> (b)
RETURN a,b,o.weight").
```

Dove la prima annotazione effettua una proiezione di tutta la tabella "ownerships", mentre nella seconda viene effettuata una vera e propria query a Neo4J, uno specifico database a grafo, in Cypher (linguaggio per interrogazione di Neo4J).
Un approccio simile viene utilizzato anche per supportare le piattaforme di machine learning esterne e di estrazione dati nel sistema. Facciamo un esempio, utilizzando il framework di estrazione dati OXPath [16], 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. Nell'esempio 3 possiamo vederne un'esempio di come possano essere accedute queste informazioni:

Esempio 3.

```
@qbind("Own", "oxpath",
"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(.)>]").
```

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 delle colonne di destinazione, 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("ACME", "COIN", 0.7)
0.3 :: Own("COIN", "SAVERS", 0.3)
0.4 :: Own("ACME", "GYM", 0.55)
0.6 :: Own("GYM", "SAVERS", 0.4)
```

Post-processing Annotations: Spesso è necessario dopo il processo di reasoning effettuare delle semplici operazioni utili per l'utente finale.
In Vadalog queste operazioni sono delle annotazioni, che rientrano nella categoria delle Post-processing annotations. La particolarità di queste annotazioni è che non aumentano la complessità computazionale del sistema, poiché effettuate dopo il processo di reasoning. Degli esempi di post-processing annotations sono orderby, rimozione dei duplicati, e tanti altri.

Nel complesso, il linguaggio supporta 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.
Nell'esempio 4 possiamo vedere un programma Vadalog molto semplice.

Esempio 4.

$$\mathrm{a}(1,0).$$
$$\mathrm{b}(1,2).$$
$$\mathrm{c}(X,Y,Z) \ :- \ \mathrm{a}(X,Y), \ \mathrm{b}(X,Z).$$

```
@output("c").
```

Nell'esempio 4 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:

Esempio 5.

```
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").
```

Nell'esempio 5, 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 di reasoning è produrre nuova conoscenza, analizzando ed incrociando dati già acquisiti.
Nell'esempio 6 vedremo un'interazione che può avvenire tra diverse sorgenti dati all'interno di un programma Vadalog con persistenza del risultato:

Esempio 6.

```
@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").
```

Dove 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, ne vediamo un utilizzo nell'esempio 7:

Esempio 7.

```
@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:

Esempio 8.

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

Nell'esempio 8 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).

Esempio 9.

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

Nell'esempio 9 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. Vediamo qualche applicazione negli esempi 10 e 11:

Esempio 10.

```
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").
```

Esempio 11.

```
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.myClass","staticmethodName").
```

Nell'esempio 10 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. Nell'esempio 11 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 Reasoner

In questo capitolo parleremo in maniera più approfondita del Vadalog Reasoner. Descriveremo principalmente l'architettura del Vadalog Reasoner, il mio contributo in questo capitolo riguarda la gestione della ricorsione, integrazione di DBMS, comunicazione con il Vadalog Reasoner attraverso interfacce.
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 il Vadalog Reasoner 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 il Vadalog Reasoner 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 [5]

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 [11, 20] è un buon punto di riferimento per il potere espressivo delle regole. Con una negazione molto lieve cattura PTIME [14].
- **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 [25].

- **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 [10] 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 [24] 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.

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±.

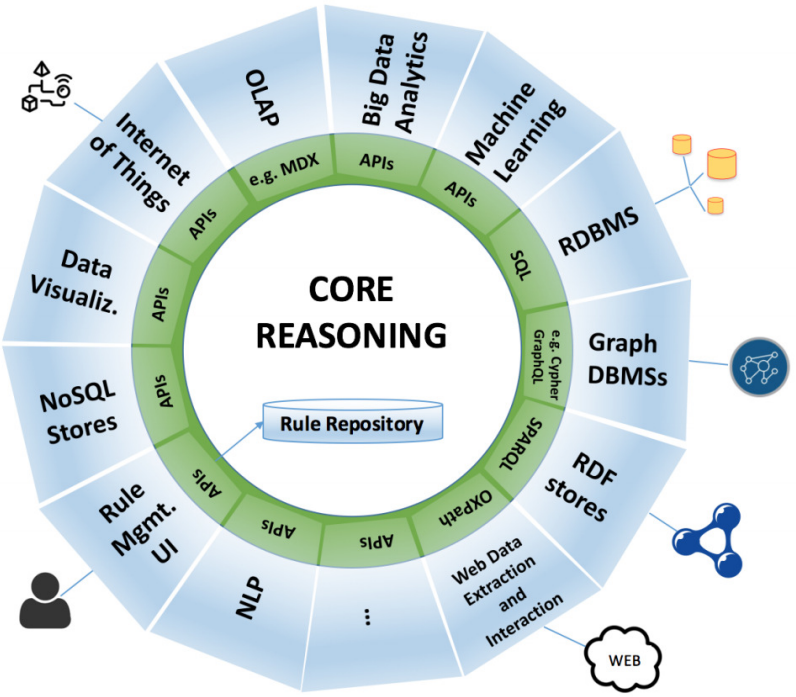

Figura 2.1: Schema dell'architettura del Vadalog Reasoner [5].

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, il Vadalog Reasoner 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, strutturata in due fasi *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.

Il Vadalog Reasoner utilizza un approccio stream-based (o approccio a pipeline). Tale approccio [34] 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.

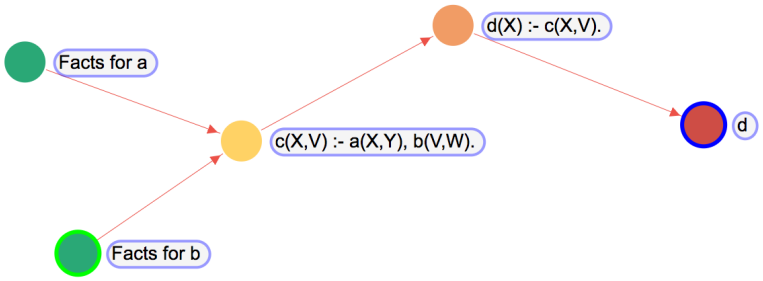

Figura 2.2: Esempio di query plan mostrata dalla Vadalog Console.

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.

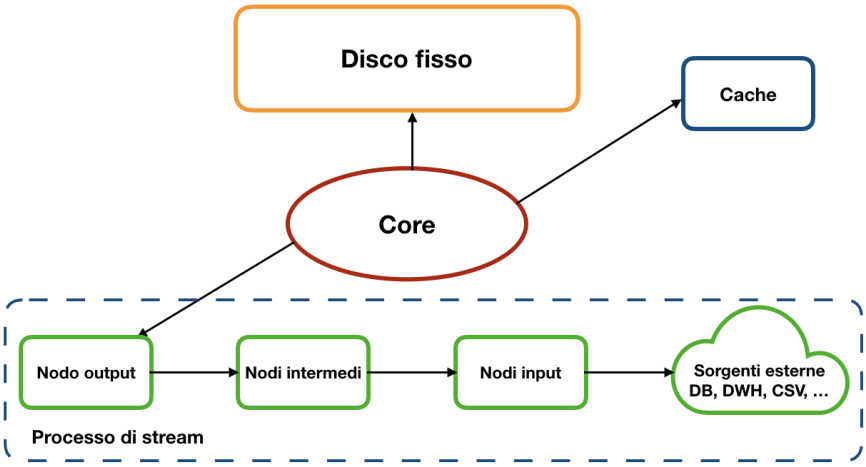

Figura 2.3: Funzionamento dell'architettura stream-based.

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 il Vadalog Reasoner: attraverso delle API Rest, o attraverso Java, integrando il progetto.
Le principali operazioni che si possono effettuare 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).

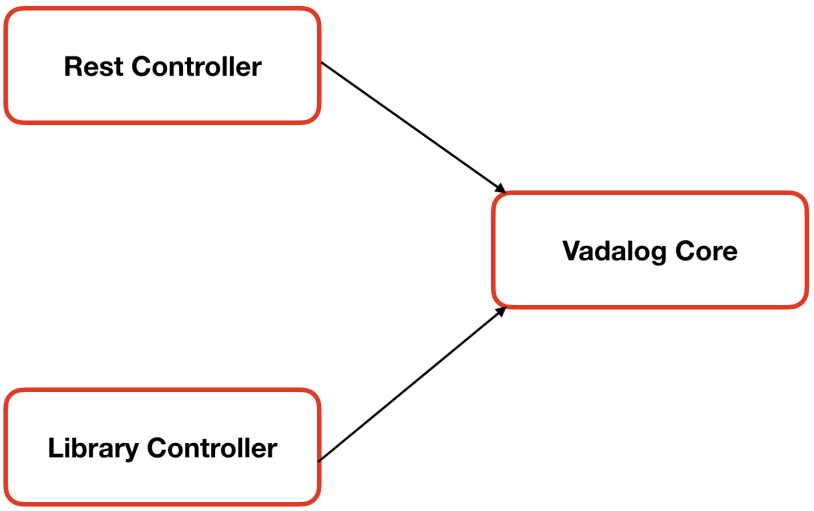

Figura 2.4: Sistemi per interfacciarsi con il Vadalog Reasoner.

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 (Li-

braryController), che si interfacciano con un unico controller che gestisce le chiamate pervenute da entrambi.

In particolare io mi sono occupato dell'implementazione delle operazioni di *getExecutionPlan* e *transformProgram*. Nella prima operazione vengono restituiti i valori da inserire nei nodi e i collegamenti tra nodi, che vengono poi utilizzati per la visualizzazione. Nella seconda operazione, molto semplicemente vengono applicate le riscritture applicate al codice Vadalog ed infine restituito.

Capitolo 3

Supporto per nuove features

Nei capitoli precedenti abbiamo parlato in linea generale del funzionamento del Vadalog Reasoner e del linguaggio Vadalog, 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, entreremo nel dettaglio dell'implementazione di ottimizzazioni note come *Push Selections Down* e *Push Projections Down*, ottimizzazioni in presenza di regole composti da più atomi nella testa ed in presenza di join multipli nel corpo di una regola ed infine dei dettagli sulla ricorsione e la loro ottimizzazione.
Nella sezione 3.2 verranno descritte le modalità di esecuzione delle annotazioni post-processing, l'architettura di gestione e l'implementazione delle operazioni dei tipi di dato nel Vadalog Reasoner, la gestione e l'implementazione delle nuove sorgenti ed infine l'implementazione e l'utilizzo delle funzioni e delle query parametriche in Vadalog.
Infine nella sezione 3.3 troveremo una descrizione sull'implementazione dell'editor grafico e della visualizzazione del grafo d'accesso.

3.1 Tecniche di ottimizzazione

Inizialmente il Vadalog Reasoner, effettuava ben poche ottimizzazioni e non 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 *Vadalog Reasoner*. In questa sezione descriveremo 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 ed inversione delle ricorsioni destre (sezione 3.1.3).

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 12 (un caso base):

Esempio 12.

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

Dopo la trasformazione diventa:

Esempio 13.

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

Nell'esempio 13 la selezione è stata anticipata alla regola più vicina al nodo 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 14:

Esempio 14.

```
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:

Esempio 15.

```
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").
```

Nell'esempio 15 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, ne vediamo un'applicazione negli esempi 16 e 17:

Esempio 16.

```
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:

Esempio 17.

```
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").
```

Come possiamo vedere nell'esempio 17 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 regola nella quale viene applicata.

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. Possiamo vederne un'applicazione nell'esempio 18 e 19:

Esempio 18.

```
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:

Esempio 19.

```
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 nell'esempio 19, nella regola c(X) :- a(X), q(X,Y) viene eliminata la variabile 'Y' poiché inutilizzata.
Mentre l'applicazione della proiezione anticipata, possiamo vederla nell'esempio 20 e 21:

Esempio 20.

```
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:

Esempio 21.

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

Nell'esempio 21 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 e Y e nel corpo X,Y e 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, possiamo vederne un'applicazione nell'esempio 22.

Esempio 22.

$$\mathrm{h1}(X,Y)\,,\mathrm{h2}(Y,Z)\,,\mathrm{h3}(Z,W) \;:-\; \mathrm{a}(X,Y,Z)\,,\mathrm{b}(Z,M)\,.$$

Dove gli atomi h1, h2, h3 condividono lo stesso corpo.
In questo caso viene riscritto, prima di essere dato in pasto al Vadalog Reasoner come nell'esempio 23:

Esempio 23.

$$\begin{aligned}
&\mathrm{h_tmp}(X,Y,Z,M) \;:-\; \mathrm{a}(X,Y,Z)\,,\mathrm{b}(Z,M)\,.\\
&\mathrm{h1}(X,Y) \;:-\; \mathrm{h_tmp}(X,Y,Z,M)\,.\\
&\mathrm{h2}(Y,Z) \;:-\; \mathrm{h_tmp}(X,Y,Z,M)\,.\\
&\mathrm{h3}(Z,W) \;:-\; \mathrm{h_tmp}(X,Y,Z,M)\,.
\end{aligned}$$

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.

Inoltre è anche possibile definire delle regole "speciali", che hanno una sintassi diversa da quelle standard. Tali regole coinvolgono la testa della regola, permettendo di utilizzare un'uguaglianza come testa.
Tali regole vengono tradotte in regole esistenziali e implicitamente trattate come regole di output. Vediamo la trasformazione negli esempi 24 e 25:

Esempio 24.

$$Y{=}Z \ :- \ \mathrm{a}(X,Y)\,,\mathrm{a}(X,Z)\,.$$

Esempio 25.

$$\mathrm{egd1}(X,Y,Z,J) \ :- \ \mathrm{a}(X,Y)\,, \ \mathrm{a}(X,Z)\,, J{=}(Y == Z)\,.$$
$$@\mathrm{output}("\mathrm{egd1}")\,.$$

Viene quindi creato un apposito atomo di testa che chiameremo egd_N ed una variabile esistenziale $\{J_1,\ J_2,\ ...,\ J_n\}$. L'atomo in testa proietta tutte le variabili del corpo (compresa la variabile esistenziale) ed aggiunge come condizione l'assegnazione della variabile esistenziale, che sarà quella espressa in testa ed infine viene aggiunta la regola di output per ogni atomo egd_N. Anche questa funzionalità è stata implementata utilizzando una riscrittura prima della procedura di reasoning.

Spesso sono presenti regole, che nel corpo hanno più di 2 atomi che sono vincolati tramite l'operazione di 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. Vediamo un'applicazione pratica nell'esempio 26:

Esempio 26.

$$\mathrm{c}(Z,X,M) \ :- \ \mathrm{a}(X,Y)\,,\mathrm{b}(Y,K)\,,\mathrm{c}(K,M)\,.$$

Nell'esempio 26 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.
Nell'esempio 27 è possibile vedere il programma dopo il processo di riscrittura:

Esempio 27.

$$\mathrm{v\_atom1}(X,K) \ :- \ \mathrm{a}(X,Y), \mathrm{b}(Y,K).$$
$$\mathrm{c}(Z,X,M) \ :- \ \mathrm{v\_atom1}(X,K), \mathrm{c}(K,M).$$

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 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 un atomo 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'applicazione pratica di come avviene la riscrittura negli esempi 28 e 29:

Esempio 28.

$$\mathrm{a}(Y) \ :- \ \mathrm{b}(X, Y), \ \mathrm{a}(X).$$

Viene trasformato in:

Esempio 29.

$$\mathrm{a}(Y) \ :- \ \mathrm{a}(X), \ \mathrm{b}(X, Y).$$

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

Nel linguaggio Vadalog vengono supportati: diversi tipi di dato, semplici e strutturati; diverse sorgenti dati da cui prendere gli input, ad esempio da un database; diverse funzionalità varie che garantiscono l'usabilità all'utente finale: ad esempio la possibilità di fare operazioni standard dopo il calcolo del risultato (orderby, distinct, min, max, argmin, argmax, ecc.), possibilità di definire funzioni e query parametriche.
In particolare mi sono occupato dell'implementazione delle operazioni di argmin e argmax (descritte nella sezione 3.2.1), dell'architettura utilizzata per la gestione dei tipi di dato e le loro operazioni con relativa implementazione (sezione 3.2.2) ed infine la gestione e l'utilizzo di funzioni e query parametriche in Vadalog (sezione 3.2.3).

3.2.1 Post-processing annotations

Le post-processing annotations sono delle annotazioni speciali in Vadalog che permettono di effettuare dei calcoli dopo tutta la procedura di reasoning. In queste annotazioni è possibile specificare molte operazioni che si possono effettuare:

- *Order by*: ha l'obiettivo di ordinare l'output per un determinato atomo, specificandone il campo (variabile) sul quale ordinare.
- *Min*: calcola il valore minimo per una o più posizioni in un atomo e raggruppa per le altre. Ad esempio se viene definito il minimo sul terzo campo ed ho come

risultato: a(1,"a",1), a(1,"a", 5) ed a(1,"b",3), il risultato finale sarà a(1,"a",1) e a(1,"b",3).

- *Max*: ha lo stesso comportamento di Min, ma prendendo il massimo.
- *Argmin*: ha la stessa funzionalità di Min, soltanto che è possibile definire le posizioni su cui raggruppare anziché prenderle tutte di default.
- *Argmax*: stesso comportamento di Argmin, ma con il massimo.
- *Unique*: ha la funzionalità di rimuovere i duplicati dall'output di un atomo definito.
- *Certain*: si occupa di rimuovere, dall'output di un atomo definito, i fatti che contengono nulli. Utile quando si devono effettuare salvataggi su database.

Io mi sono occupato dell'implementazione delle operazioni di Argmin ed Argmax.
Negli esempi 30 e 31 vediamo l'applicazione di Argmax e il risultato ottenuto.

Esempio 30.

```
f(1,3,"a",3).
f(4,3,"a",5).
f(2,6,"b",7).
f(2,6,"b",8).
f(3,6,"b",9).
g(X,Y,Z,K) :- f(X,Y,Z,K).
@output("g").
@post("g","argmax(4,<2,3>)").
```

Esempio 31.

```
g(4,3,"a",5).
g(3,6,"b",9).
```

Nell'implementazione vengono prima effettuati i raggruppamenti per le posizioni indicate nell'annotazione, e viene poi effettuata la selezione del massimo (o minimo, a seconda dell'annotazione).

3.2.2 Tipi di dato e sorgenti dati

Vadalog supporta molti tipi di dato semplici e strutturati, di conseguenza supporta le operazioni che possono essere effettuate su questi tipi di dato. Sono presenti diversi tipi di dato in Vadalog: *integer*, *double*, *string*, *boolean*, *date*, *set* e *Marked Null*.
Nel Vadalog Reasoner la gestione dei tipi di dato e le eventuali operazioni viene effettuata utilizzando il pattern architetturale *Visitor*, che permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa [29]. È presente una classe che si occupa di eseguire le operazioni che è chiamata *Visitor* e diverse classi chiamate *ConcreteElement* che estendono la classe *Element*. Il Visitor implementa un metodo "visit" per ogni classe concreta (riconoscerà quale invocare basandosi sulla classe del chiamante) e le varie classi concrete implementano un metodo "accept" che invoca "visit" passando se stesso come parametro per permettere di accedere al proprio stato ed effettuare le operazioni offerte dall'oggetto chiamante.
Nel Vadalog Reasoner il *Visitor* è chiamato *ExpressionVisitor*, e le *ConcreteElement*, rappresentano tutte le espressioni che possono essere applicate ai tipi di dato. Nella Figura 3.1 possiamo vedere una porzione del diagramma delle classi riguardante l'applicazione del pattern architetturale.

Il client invoca i metodi di accept e conseguentemente l'ExpressionVisitor che esegue le operazioni a seconda del chiamante.
Io mi sono occupato dell'integrazione completa (tipi e operazioni base) dei tipi boolean e date con l'integrazione delle operazioni base per le stringhe, dove ogni operazione rappresentava un metodo nell'Expression Visitor.
Per il tipo semplice *boolean* ho definito le operazioni di AND, OR, NOT, ==, <>, >, <, >= e <= utilizzando i confronti forniti da Java. Mentre per il tipo strutturato *date*, come prima operazione mi sono occupato del cast dal tipo utilizzato in Vadalog (YYYY-MM-DD HH24:MI:SS) al tipo Java (GregorianCalendar) ed in seguito delle operazioni di confronto ==, <>, >, <, >= e <=. Infine mi sono occupato dell'implementazione degli operatori per le stringhe in Vadalog: substring, contains, strartsWith, endsWith, concat ed indexOf, utilizzando anche qui le primitive Java.

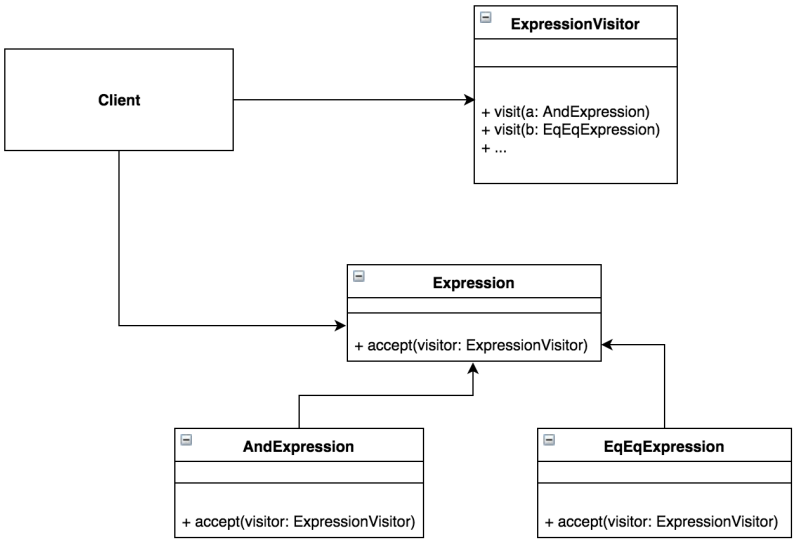

Figura 3.1: Diagramma delle classi del pattern Visitor.

Vadalog inoltre supporta diverse sorgenti dati come input, al momento sono presenti le seguenti sorgenti:

- *File CSV locali*: lettura e scrittura di dati in file CSV salvati su disco fisso.
- *File CSV remoti*: download e lettura di dati in file CSV in internet, con l'utilizzo del protocollo HTTP.
- *Postgres*: lettura e scrittura di dati su database Postgres.
- *OXPath*: lettura di dati provenienti da pagine html sul web.

Nel Vadalog Reasoner per l'implementazione e la gestione delle sorgenti dati si è fatto uso di polimorfismo, è presente una classe *RecordManager*, che viene estesa da classi concrete che rappresentano le possibili sorgenti.
In Figura 3.2 possiamo vedere un diagramma delle classi che rappresenta la gestione del record manager e le varie classi concrete nel Vadalog Reasoner.
Io mi sono occupato dell'integrazione di file CSV sia remoti che locali. Poiché le implementazioni sono molto simili, l'unica differenza sostanziale è la lettura del file, una avviene in locale ed una bisognava effettuare il download del file, ho quindi deciso di

introdurre un'ulteriore forma di polimorfismo creando una classe astratta *CSVRecord-Manager* che implementa tutte le operazioni in comune, estesa da due sottoclassi la cui unica funzione è leggere il file.

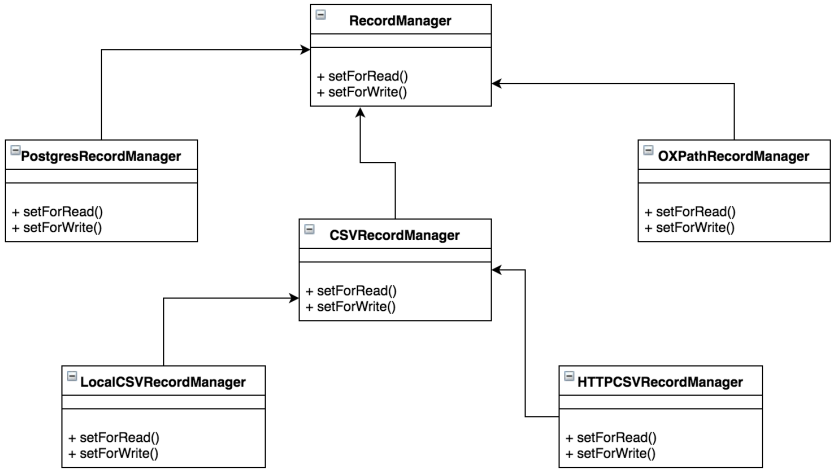

Figura 3.2: Diagramma delle classi che rappresenta la gerarchia del record manager.

Per l'implementazione di lettura vengono lette le righe del file CSV, successivamente caricate come fatti di input per l'atomo associato. La particolarità è che viene effettuata un'inferenza dei tipi di dato del file CSV attraverso una cascata di *Try/Catch* che tentano di fare il cast del valore.
Per l'implementazione di scrittura viene semplicemente creato un file CSV nel path definito dall'utente e salvato l'output al suo interno.

3.2.3 Funzioni e query parametriche

Vadalog supporta le funzioni (chiamate *funzioni di skolem*), che possono essere implementate con procedure esterne (*user-defined functions*) scritte in altri linguaggi (Java o Python), se non vengono implementate allora producono un nullo. Io mi sono occupato sia dell'integrazione delle funzioni che dell'utilizzo di procedure esterne.
Negli esempi 32 e 3.2.3 possiamo vedere due utilizzi di funzioni, nell'esempio 32 non è presente nessuna implementazione, viceversa nell'esempio 3.2.3.

Esempio 32.

```
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").
```

Esempio 33.

```
b(1,2).
c(3,2).
a(Y,Z) :- b(X1,Z), c(X2,Z), Y= #f(X1,X2).
@output("a").
@implement("#f","python","com.module1","functionName").
```

Se le funzioni non vengono implementate allora esse producono un nullo, la particolarità è che tale nullo dovrà essere uguale nel caso in cui la stessa funzione prende gli stessi input, ad esempio se nella sostituzione ho la funzione #f(1,2) e ritorna un valore nullo Z_1, allora se in un successivo passaggio viene invocata nuovamente la stessa funzione, dovrà restituire lo stesso risultato.
Per implementare questa particolarità ho fatto uso di una cache creata all'inizio del processo di reasoning e distrutta al termine, la quale conteneva i risultati parziali di ogni funzione.
Le funzioni vengono gestite attraverso il Visitor, che ha diversi compiti:

- verifica se la funzione è già stata invocata, in caso positivo restituisce il valore nullo associato alla funzione, altrimenti
- verifica se la funzione non ha implementazioni associate, in questo caso genera un nuovo nullo e lo aggiunge in cache, altrimenti

- se è stato implementato con una procedura Java, allora importa e con la riflessione invoca il metodo esterno con i parametri passati alla funzione Vadalog, altrimenti
- se è stato implementato con una procedura Python, attraverso un tool di Java interpreta ed esegue il codice Python con i parametri passati alla funzione Vadalog.
- Infine salva il risultato nella pila di output, ed aggiunge il risultato della funzione in cache.

Questa procedura viene eseguita ogni volta che viene utilizzata una funzione.

Mi sono anche occupato dell'implementazione di query parametriche in Vadalog, come già detto è possibile avere diverse sorgenti, in particolare su database relazionali è possibile fare anche delle query per prenderne i risultati con l'annotazione di qbind, come mostrato nell'esempio 34.

Esempio 34.

```
test("marco","luigi").
test("fabio", "antonio").
control(X,Y) :- own(X,Y,W), W>0.5.
control(X,Z) :- control(X,Y), own(Y,Z,W).
@output("control").
@input("own").
@bind("own","postgres","vada","ownerships").
@qbind("own","postgres","vada",
"select name1, name2, weight
from ownerships2
where name1='Marco'
").
```

Le query parametriche permettono di utilizzare dei parametri nelle selezioni, come possiamo vedere nell'esempio 35.

Esempio 35.

```
test("marco","luigi").
test("fabio", "antonio").
control(X,Y) :- own(X,Y,W), W>0.5.
control(X,Z) :- test(X,Y), own(Y,Z,W).
@output("control").
@input("own").
@bind("own","postgres","vada","ownerships").
@qbind("own","postgres","vada",
"select name1, name2, weight
from ownerships2
where name1=${1}
").
```

Nelle query parametriche viene specificato un valore del tipo $1, che sta a rappresentare i valori in prima posizione dell'atomo corrente. Solitamente il parametro che viene espresso è messo in join con un altro atomo, come avviene nell'esempio 35, che viene messo in join con "control". In questo caso i valori di $1 sono "luigi" ed "antonio", vengono quindi effettuate due query una con name1 uguale a "luigi" ed una "antonio". Spesso invece i valori da sostituire possono essere presi direttamente dai fatti, come possiamo vedere nell'esempio 36.

Esempio 36.

```
own("marco",1,2).
own("luigi",2,3).
control(X,Y) :- own(X,Y,W), W>0.5.
@output("control").
@input("own").
@bind("own","postgres","vada","ownerships").
@qbind("own","postgres","vada",
"select name1, name2, weight
from ownerships2
```

```
where name1=${1}
").
```

Nell'esempio 36 vengono effettuate due query al database postgres, una con name1 uguale a "marco" ed una "luigi".
Per applicare le query parametriche è necessario però garantire la proprietà di *safety*, ovvero che l'atomo p deve avere il parametro che è coinvolto nella qbind in una posizione che non sia dangerous, cioè non deve avere nulli, anche negli altri atomi del corpo.
L'implementazione delle query parametriche è svolta a runtime, in particolare durante il processo di reasoning. L'atomo coinvolto nel qbind viene sempre spostato come ultimo atomo del corpo della regola. Per l'esecuzione di una query parametrica si svolgono le seguenti operazioni:

- Si scorre il grafo di esecuzione eseguendo prima il sottoalbero sinistro, dove vengono visitati tutti gli atomi e raccolti i fatti provenienti da essi e salvati in una cache.
- Successivamente viene eseguito il sottoalbero destro, seguendo la stessa procedura.
- Vengono poi fatte le query sostituendo i parametri con i fatti salvati in cache.
- Infine viene eseguita l'operazione in cui è coinvolto l'atomo che esegue la query parametrica.

3.3 Vadalog console

È presente una console di Vadalog che permette all'utente di eseguire programmi, e tante altre operazioni offerte dal Resoner.
Questa console è sviluppata come un client web, e permette di interagire con il sistema Vadalog attraverso delle chiamate API Rest.
In seguito descriveremo brevemente l'implementazione attraverso librerie di un editor grafico e la visualizzazione del grafo di accesso.

3.3.1 Editor

Mi sono occupato dell'integrazione di un editor grafico per la scrittura di codice Vadalog. Ho utilizzato un text editor chiamato *CodeMirror*, che rappresenta un progetto open-source, è implementato in JavaScript, integrabile nella propria pagina html, supporta tutti i browser e permette molte funzionalità da poter personalizzare, a partire dalla colorazione del codice all'autocompletamento [13].

3.3.2 Visualizzazione del grafo di accesso e dei dati

Il Vadalog Reasoner offre un'operazione che restituisce il grafo d'accesso del reasoning. Io mi sono occupato della visualizzazione di tale grafo, per farlo ho utilizzato una libreria JavaScript chiamata *SigmaJS* (integrabile nella propria pagina html). Questa libreria prende in input un file JSON, nel quale vengono specificati nodi ed archi, ogni nodo è composto da un identificativo, un'etichetta, il valore della x e della y e la grandezza, ogni arco è invece composto da un identificativo, la sorgente (identificativo del nodo) e la destinazione (identificativo del nodo) [27]. È possibile utilizzare anche funzionalità aggiuntive per nodi ed archi da aggiungere al file JSON, ad esempio il colore, sia per i nodi che per gli archi.
Inoltre ho implementato una funzionalità che permette di visualizzare i dati sia sotto forma di fatti (ad esempio a(1,0), ecc.) che sotto forma di tabella (simile all'output di un database relazionale).

Capitolo 4

Prove sperimentali

In questo capitolo parleremo in maniera approfondita delle prove sperimentali che sono state effettuate sul Vadalog Resoner da Luigi Bellomarini ed Emanuel Sallinger presso il Laboratorio dell'Università di Oxford.
Il capitolo è organizzato come segue. Nella sezione 4.1 descriveremo il tool, che ho implementato durante il mio periodo di tesi, *iWarded*, che permette la generazione di benchmark Vadalog, vedendone le funzionalità e l'implementazione.
Infine nella sezione 4.2 descriveremo diversi scenari applicativi utilizzando benchmark generati con tool (ad esempio iWarded), generandone anche i dati, per testare situazioni ben precise, ad esempio la presenza di ricorsione, di propagazione dei nulli, di join tra variabili harmful, ecc.

4.1 L'implementazione di iWarded

In questa sezione descriveremo il tool *iWarded*, che permette la generazione di benchmark con determinate caratteristiche che vengono espresse attraverso l'input. Inoltre gli input vengono presi di default da file csv generato e popolato dallo stesso iWarded. iWarded è stato implementato separatamente rispetto al core logico del Vadalog Reasoner e prende in input i seguenti parametri:

- Numero di regole che devono essere presenti nel programma finale.
- Numero medio di atomi nel corpo delle regole.

- Numero medio di variabili in una regola.
- Probabilità di avere una costante.
- Numero medio di join in una regola.
- Numero di tuple nei file di input.
- Path di destinazione per il salvataggio dei file CSV e del programma.

Attraverso questi parametri presi come input, vengono svolti dei passaggi per la creazione del programma Vadalog.
Si parte dal numero di regole, queste bisogna suddividerle in regole di input, intermedie e output, definendo un numero preciso. In particolare il numero di regole di input è definito nel range [1, n.regole totali/2], quindi ho almeno 1 regola di input ed al massimo la metà delle regole totali di input; stesso ragionamento per le regole di output definite nel range [1, n.regole totali/3]; ed infine il numero di regole intermedie definite come la sottrazione tra le regole totali con quelle di input ed output.
Successivamente si passa alla creazione delle regole, per quelle di input è presente un modulo che si occupa:

- Creazione e popolazione del file CSV nel path definito.
- Creazione delle regole di input (annotazioni di input e di bind).
- Salvataggio in cache delle regole create, che solitamente hanno un nome convenzionale che va da in_1 ad in_n.

Poi vengono create le regole intermedie con l'esecuzione dei seguenti passaggi:

- Creazione del corpo della regola utilizzando gli atomi di input o eventuali atomi intermedi già creati, utilizzando il numero medio di atomi e variabili nel corpo e facendo join sulla base del numero medio di join.
- Creazione dell'atomo di testa, utilizzando un nome convenzionale che va da intermediate_1 ad intermediate_n.

- Assegnazione delle variabili in testa casuali e aggiungendo delle variabili esistenziali con una probabilità del 20%.
- Infine vengono salvate in cache per essere utilizzate come corpo di altre regole intermedie o di output.

Vengono poi create le regole di output:

- Viene creato il corpo utilizzando atomi sia di input che intermedi.
- Viene creata la testa proiettando variabili casuali dal corpo della regola, evitando variabili dangerous che potrebbero portare ad un programma non warded.

Infine l'intero modello contenente le regole create viene salvato su disco.
Nell'esempio 37 possiamo vedere un programma generato da iWarded:

Esempio 37.

```
@output("ou_1").
@input("in_1").
@bind("in_1","csv","/Users/utente/Desktop","in_1_csv.csv").
@input("in_2").
@bind("in_2","csv","/Users/utente/Desktop","in_2_csv.csv").
intermediate_1(HARMLESS_1) :- in_1(HARMLESS_1,HARMLESS_2).
intermediate_2(HARMLESS_3) :- in_2(2,HARMLESS_3).
intermediate_3(DANGEROUS_1) :- in_1(HARMLESS_1,HARMLESS_2).
intermediate_4(HARMLESS_1) :- in_1(HARMLESS_1,HARMLESS_2).
ou_1(HARMLESS_4) :- intermediate_1(HARMLESS_4).
```

Possiamo vedere la presenza di costanti e variabili dangerous, ma in questo caso non sono presenti join.

4.2 Scenari utilizzati

In questa sezione descriveremo l'applicazione del Vadalog Reasoner in diversi scenari.
Per effettuare i test il Reasoner è stato utilizzato come una libreria, come storage sono stati utilizzati dei file CSV.
La configurazione hardware per l'esecuzione dei test è la seguente:

- Sistema operativo: Linux.

- Processore: Intel Xeon v3 con 8 cores, frequenza a 2.4GHz.
- RAM: 16GB.

Nella sezione 4.2.1 descriveremo il comportamento del Vadalog Reasoner utilizzando degli scenari generati da *iWarded*.
Nella sezione 4.2.2 utilizzeremo degli scenari generati da un altro tool simile, chiamato *iBench*, descrivendo il comportamento del Vadalog Reasoner rispetto ai suoi competitor.

4.2.1 iWarded

Per questo scenario è stato utilizzato *iWarded* descritto nella sezione 4.1, che permette di generare programmi Vadalog (e quindi Warded Datalog$^{\pm}$) con regole lineari e non lineari, con presenza di harmful o harmless join, ricorsione, ecc.

Set of rules	L rules	⋈ rules	L recursive	⋈ recursive	∃ rules	hrml ⋈ hrmf	hrml ⋈ hrml with ward	hrml ⋈ hrml w/o ward	hrmf ⋈ hrmf
synthA	90	10	27	3	20	5	4	1	0
synthB	10	90	3	27	20	45	40	5	0
synthC	30	70	9	20	40	25	20	5	20
synthD	30	70	9	20	22	10	9	1	50
synthE	30	70	15	40	20	35	29	1	5
synthF	30	70	25	20	50	35	29	1	5
synthG	30	70	9	21	30	0	10	60	0
synthH	30	70	9	21	30	0	60	10	0

Figura 4.1: Dettagli dello scenario generato con iWarded.

In Figura 4.1 sono rappresentati i dettagli dello scenario, in particolare per ogni set di regole (programma) è configurato con le seguenti caratteristiche:

- L rules: è il numero di regole lineari presenti nel programma (regole che hanno un solo atomo nel corpo).
- ⋈ rules: è il numero totale di regole con un join del programma.
- L recursive: è il numero di regole lineari ricorsive (sottoinsieme di L).

- $\bowtie$ recursive: è il numero di regole join ricorsive (sottoinsieme di $\bowtie$ rules).
- $\exists$ rules: è il numero di regole (tra lineari e $\bowtie$ rules) che hanno esattamente una variabile classificata esistenzialmente.
- hrml $\bowtie$ hrmf: è il numero di join tra variabili harmless e variabili harmful nell'intero programma.
- hrml $\bowtie$ hrml with ward: è il numero di join tra variabili harmless-harmless in regole in cui c'è una propagazione di una dangerous nella testa.
- hrml $\bowtie$ hrml w/o ward: è il numero di join harmless-harmless in cui non c'è la propagazione di alcuna dangerous nella testa.
- hrmf $\bowtie$ hrmf: è il numero di join harmful-harmful.

Sono stati creati 8 programmi contenenti 100 regole.
SynthA ha una netta superiorità di regole lineari, il 20% delle regole totali ha un quantificatore esistenziale, il 30% delle regole lineari e non lineari ha una ricorsione (sia L rules che $\bowtie$ rules), ed i join sono equamente distribuiti tra harmless-harmful e harmless-harmless, quindi non sono presenti join tra valori nulli.
SynthB ha le stesse caratteristiche di SynthA, soltanto con una netta superiorità di regole non lineari (regole di join). SynthA e SynthB insieme servono a confrontare l'impatto della presenza di regole lineari.
SynthC ha tra il 30% e il 70% di regole lineari e non lineari, ed ha una buona presenza di join harmful-harmful sui quali ci focalizziamo. *SynthD* ha lo stesso comportamento di SynthC ma ha prevalenza di join harmful-harmful.
SynthC e SynthD servono a valutare l'impatto della presenza degli join harmful-harmful.
SynthE ha tutto nella media ma con una forte ricorsione sulle regole con join. *SynthF* si comporta allo stesso modo di SynthE, ma ha una forte ricorsione sulle regole lineari anziché sulle regole con join.
SynthE e SynthF servono a verificare l'impatto della ricorsione.
SynthG ha praticamente le stesse caratteristiche di SynthC e SynthD, ma con prevalenza di join harmless-harmless, di conseguenza ha quasi le caratteristiche di un programma Datalog. *SynthH* è molto simile, ma enfatizza i join sui ward.

In Figura 4.2 sono riportati i risultati dell'esecuzione dei vari programmi. Il Vadalog Reasoner ha le migliori prestazioni sugli scenari SynthB e SynthH, con un tempo di esecuzione inferiore ai 20 secondi.
In SynthB sono presenti circa 45 join harmless-harmful ma non condizionano le prestazioni del sistema. Lo scenario SynthC ha un numero medio di regole (tra lineari e non lineari) ed ha un numero bilanciato delle tipologie di join, invece SynthD ha un numero di join harmful-harmful maggiore ed impiega soltanto 7 secondi in più di SynthC, questo poiché interviene l'ottimizzazione degli harmful-harmful join trasformandoli in join harmless-harmless dove possibile (linearizzazione). Lo scenario SynthG ha 60 join harmless-harmless senza wards, è molto simile a SynthD, ma ha un tempo di reasoning maggiore dovuto alla presenza di variabili esistenziali.

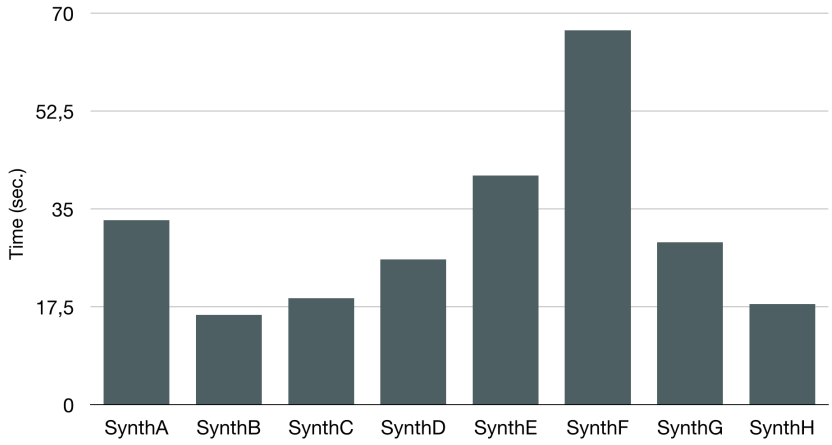

Figura 4.2: Tempo di reasoning per lo scenario iWarded.

Infine gli scenari SynthE e SynthF mostrano che l'impatto della ricorsione è significante, con tempi di 40 e 65 secondi rispettivamente. La ricorsione lineare ha un impatto molto forte, dovuto dalla presenza di nulli in combinazione con gli harmful join, non sfruttando a pieno i benefici dati dalla wardedness. Ciò è confermato dallo scenario SynthA, poiché ha una buona presenza di regole lineari e di ricorsione, ma non ha la presenza di join tra valori nulli, impiegando soltanto 32 secondi per l'esecuzione.

4.2.2 iBench

iBench [1] è un tool per generare degli scenari di data integration. Consideriamo due famosi scenari chiamati *STB-128* e *ONT-256*, trasformando i programmi generati da iBench in programmi Vadalog.
Questi scenari sono molto importanti per i nostri scopi, poiché hanno molte regole esistenziali, i nulli sono propagati nelle teste e coinvolti in molti join e le regole hanno una presenza molto alta di ricorsione.
STB-128 è un programma che ha le seguenti caratteristiche:

- Composto da un insieme di 250 regole warded, di cui il 25% sono esistenziali.
- Sono presenti 15 casi di harmful join.
- 30 casi di propagazione dei nulli.
- Ci sono 112 atomi distinti.
- La sorgente contiene 1000 fatti per ogni atomo.
- Il risultato atteso è di 800 mila fatti, di cui il 20% nulli.

In questo scenario abbiamo eseguito 16 query differenti molto complesse, coinvolte ognuna in circa 5 join.
ONT-256 è un programma che ha le seguenti caratteristiche:

- Composto da un insieme di 789 regole warded, di cui il 35% sono esistenziali.
- Sono presenti 295 casi di harmful join.
- Più di 300 casi di propagazione dei nulli.
- Ci sono 220 atomi distinti.
- La sorgente contiene 1000 fatti per ogni atomo.
- Il risultato atteso è di circa 2 milioni di fatti, di cui il 50% nulli.

In questo scenario le regole sono molto più complesse rispetto a STB-128 e contengono join multipli e ricorsione. Abbiamo eseguito 11 query differenti, coinvolte ognuna in circa 5 join.

L'obiettivo di questo scenario è testare i tempi di esecuzione del Vadalog Reasoner con altri sistemi di reasoning già esistenti, in particolare abbiamo considerato i sistemi: RDFox [23], Llunatic [18], DLV [22], Graal [3] e PDQ [6].

In Figura 4.3 vengono messi a paragone i tempi di reasoning del Vadalog Reasoner e tutti gli altri sistemi menzionati utilizzando la stessa configurazione hardware/software, utilizzando nomi generici poiché questi dati devono ancora essere pubblicati.

Come possiamo vedere dal grafico il Vadalog Reasoner ha prestazioni migliori rispetto a tutti gli altri sistemi, in entrambi gli scenari, con un tempo di 6.59 secondi per l'esecuzione di STB-128 e 51.579 secondi per ONT-256.

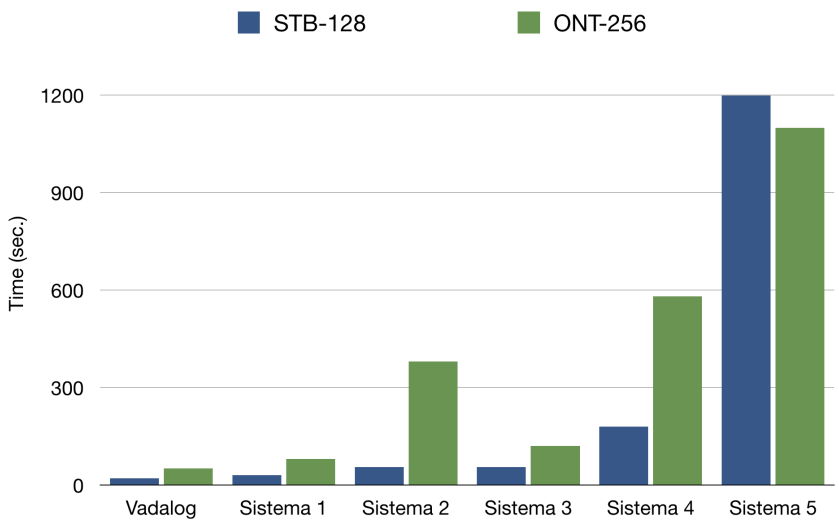

Figura 4.3: Tempo di reasoning per gli scenari STB-128 e ONT-256 su tutte le piattaforme.

Capitolo 5

Related work

In questa sezione andremo ad analizzare i sistemi simili al Vadalog Reasoner, analizzandoli e paragonandoli al nostro sistema.

5.1 Analisi e confronti

Sono presenti molti sistemi esistenti simili al Vadalog Reasoner. I principali sono Graal [3], Llunatic [18], PDQ [6] e DLV^E [21] che condividono molto ma non tutto il potere espressivo del nostro sistema.
La differenza principale di tutti questi sistemi è che nessuno da un grosso peso alla scalabilità, e quindi alla gestione dei Big Data, al contrario il Vadalog Reasoner ne fa un vero e proprio punto di forza. Inoltre il Vadalog Reasoner è l'unico tra tutti questi sistemi ad adottare un linguaggio della famiglia Warded Datalog$^{\pm}$

il sistema Graal è un toolkit implementato in Java, considera basi di conoscenza composte da dati e ontologie espresse da regole esistenziali. È basato su Datalog^{+}, quindi supporta le regole esistenziali, ma non garantisce decidibilità e tracciabilità dei dati al contrario del Vadalog Reasoner.

Llunatic offre un linguaggio da poter utilizzare dall'utente finale ed un'interfaccia grafica (applicazione desktop), è focalizzato principalmente sul mapping e il cleaning dei

dati, che lo rendono molto limitato rispetto al Vadalog Reasoner.

PDQ è un sistema che è focalizzato principalmente sul reasoning, ha dei wrapper che permettono soltanto l'integrazione di dati web e provenienti da DBMS relazionali, inoltre l'algoritmo di ottimizzazione e ricerca ha un costo computazionale molto alto, come possiamo anche vedere nei test di cui abbiamo parlato nella sezione 4.2.2, in Figura 4.3 è il sistema con le prestazioni peggiori.

Infine DLV^E è il sistema che più si avvicina al Vadalog Reasoner, infatti anch'esso utilizza un linguaggio Datalog$^{\pm}$, dove attraverso delle semplificazioni fornisce la decidibilità P-completa. La differenza sostanziale sta nel fatto che il Vadalog Reasoner è esteso da molte altre funzionalità che ne permettono l'applicazione in diversi scenari.

Molti di questi sistemi hanno una buona implementazione, tuttavia non posseggono un supporto specifico per le funzionalità richieste in importanti scenari aziendali e tendono a concentrarsi sull'approccio logico, anziché sull'architettura.

Conclusioni e sviluppi futuri

L'obiettivo di questa tesi è stato l'implementazione di feature core del Vadalog Reasoner per l'esecuzione di programmi Vadalog.

Durante quest'attività di tesi ho acquisito molte competenze, che troviamo di seguito:

- Sviluppo del core di un Reasoner.
- Gestione di un'architettura stream.
- Conoscenza avanzata del linguaggio Datalog e le sue estensioni.
- Ottimizzazioni delle query.
- Creazione di benchmark efficienti per testare determinate caratteristiche.
- Tecniche di gestione della cache.

Inizialmente era presente una versione base, che permetteva il reasoning senza applicarne ottimizzazioni, poche sorgenti, pochi tipi di dato, poche funzionalità aggiuntive ed erano stati effettuati pochi benchmark per testare le perfomance.
Durante la mia attività di tesi ho implementato diverse feature che colmando i problemi presenti, rendendo il sistema più stabile e ottimale come descritto nei capitoli precedenti.

Possibili sviluppi futuri sono principalmente la creazione di applicazioni mirate a modelli di business da presentare ad aziende interessate al prodotto e l'ampliamento di funzionalità offerte dal Vadalog Reasoner come l'integrazione di ulteriori storage con cui interagire, l'implementazione di nuovi algoritmi di join per ottimizzare scenari differenti, supporto a nuovi tipi di dato strutturati e tanto altro.

Bibliografia

[1] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *VLDB*, 9(3):108–119, 2015.

[2] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Basi di dati: modelli e linguaggi di interrogazione (seconda edizione)*. McGraw-Hill, 2006.

[3] J.-F. Baget, M. Leclère, M.-L. Mugnier, S. Rocher, and C. Sipieter. Graal: A toolkit for query answering with existential rules. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 328–344. Springer, 2015.

[4] J.-F. Baget, M. Leclère, M.-L. Mugnier, and E. Salvat. On rules with existential variables: Walking the decidability line. *Artificial Intelligence*, 175(9-10):1620–1654, 2011.

[5] L. Bellomarini, G. Gottlob, A. Pieris, and E. Sallinger. Swift logic for big data and knowledge graphs. 2017.

[6] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. *VLDB*, 8(6):690–701, 2015.

[7] A. Calı, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res*, 48:115–174, 2013.

[8] A. Cali, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. 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.

[9] A. Cali, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128, 2012.

[10] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, M. Rodriguez-Muro, R. Rosati, M. Ruzzi, and D. F. Savo. The mastro system for ontology-based data access. *Semantic Web*, 2(1):43–53, 2011.

[11] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer Science & Business Media, 2012.

[12] A. K. Chandra and M. Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM*, 14(3):671–677, 1985.

[13] CodeMirror. `https://codemirror.net/`.

[14] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *CSUR*, 33(3):374–425, 2001.

[15] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[16] T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. Sellers. Oxpath: A language for scalable data extraction, automation, and crawling on the deep web. *VLDB*, 22(1):47–72, 2013.

[17] T. Furche, G. Gottlob, B. Neumayr, and E. Sallinger. Data wrangling for big data: Towards a lingua franca for data wrangling. 2016.

[18] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks!: llunatic goes open source. *VLDB*, 7(13):1565–1568, 2014.

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

[20] S. S. Huang, T. J. Green, and B. T. 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.

[21] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently computable datalog∃ programs. *KR*, 12:13–23, 2012.

[22] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *TOCL*, 7(3):499–562, 2006.

[23] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory rdf systems. In *AAAI*, pages 129–137, 2014.

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

[25] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. 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.

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

[27] sigmajs. `http://sigmajs.org/`.

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

[29] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

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

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

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

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

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