UNIVERSITÀ DEGLI STUDI ROMA TRE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Questo è il titolo della tesi

Laureando
Marco Faretra
Matricola 460573

Relatore
Prof. Paolo Atzeni

Correlatore
Ing. Luigi Bellomarini

Anno Accademico 2016/2017

Questa è la dedica

Ringraziamenti

Grazie a tutti

Introduzione

Negli ultimi tempi, la mole di dati all'interno delle aziende cresce quotidianamente, per questo motivo molte compagnie desiderano mantenere i propri dati in knowledge graph, che per l'utilizzo e la gestione di tale strumento si necessita di un Knowledge Graph Management System (KGMS).

Fin dagli anni 70, l'importanza della conoscenza è stata evidente, e l'idea di salvare conoscenza e di elaborarla per trarre nuova conoscenza esisteva già da allora. Il collo di bottiglia era la tecnologia di quei tempi, gli hardware erano troppo lenti, la memoria principale troppo piccola; i DBMS erano troppo lenti e rigidi; Non era presente un web dove un sistema esperto poteva acquisire dati; il machine learning e le reti neurali furono ridicolizzate e non riuscite.

Con il passare degli anni, l'avvento tecnologico ha subito una crescita radicale, l'hardware si è evoluto, le tecnologie dei database sono migliorate notevolmente, è presente un web con dati aperti, le aziende possono partecipare sui social networks. La ricerca ha portato ad una comprensione migliore di molti aspetti nell'elaborazione della conoscenza e reasoning su grandi quantità di dati.

A causa di tutto ciò, migliaia di grandi e medie aziende, desiderano gestire i propri knowledge graph e cercano adeguati KGMS. Inizialmente soltanto le grandi aziende ad esempio Google (che utilizza il proprio knowledge graph per il proprio motore di ricerca), Amazon, Facebook, ecc... ne possedevano uno, ma con il passare degli anni molte aziende medio/basse desiderano avere un knowledge graph aziendale privato, che contiene molti dati in forma di fatti, come ad esempio conoscenza su clienti, prodotti, prezzi, concorrenti, piuttosto che conoscenza di tutto il mondo da Wikipedia o altre fonti simili.

Un KGMS completo deve svolgere compiti complessi di reasoning, ed allo stesso tempo ottenere performance efficienti e scalabili sui Big Data con una complessità computazionale accettabile. Inoltre necessita di interfacce con i database aziendali, il web e il machine learning. Il core di un KGMS deve fornire un linguaggio per rappresentare la conoscenza e il resoning.

Vadalog rappresenta un sistema KGMS, che offre un motore centrale di reasoning principale ed un linguaggio per la gestione e l'utilizzo.
Il linguaggio Vadalog appartiene alla Famiglia Datalog±, che estende Datalog con quantificatori esistenziali nelle teste delle regole, nonché da altre caratteristiche ed allo stesso tempo limita la sua sintassi in modo da ottenere decidibilità e tracciabilità dei dati. [CGK13, CGP12, CGL$^+$10]
Datalog [ACPT06] è un linguaggio di interrogazione per basi di dati che ha riscosso un notevole interesse dalla metà degli anni ottanta, è basato su regole di deduzione.
Il core logico di Vadalog è in grado di processare tale linguaggio ed è in grado di eseguire task di reasoning ontologici e risulta computazionalmente efficiente, tale da soddisfare i requisiti già citati (Big Data, Web, Machine Learning, ...), esso ha accesso ad un repository di regole. Per dare un esempio consente l'aggregazione attraverso la somma, il prodotto, il massimo, ecc... anche in presenza di ricorsioni.
Esso fornisce anche degli strumenti che permettono il data analytics, l'iniezione di codice procedurale, l'integrazione con diverse tipologie di input (ad esempio database relazionali, file csv, ecc...).

L'obiettivo della mia Tesi è stato l'ampliamento di Vadalog con nuove features.
Di seguito un accenno delle funzionalità di cui mi sono principalmente occupato, che verrà descritto in maniera più approfondita nei prossimi capitoli:

- Implementazione di nuovi tipi di dato.
- Riscritture per ottimizzare i tempi di calcolo (ad esempio "Push Selection Down").
- Creazione di benchmark per effettuare test sulle performance.

- Supporto di nuove funzionalità (ad esempio, supporto ai csv, supporto alle funzioni arbitrarie, ecc...).
- Integrazione di codice procedurale all'interno del linguaggio Vadalog.

L'attività di Tesi è stata svolta presso il Laboratorio Basi di Dati, dell'Università degli Studi Roma Tre, in collaborazione con L'Università di Oxford.

Indice

Introduzione **iv**

Indice **vii**

Elenco delle figure **ix**

1 Vadalog Engine **1**

1.1 Proprietà di un KGMS 1

1.1.1 Linguaggio e sistema per il reasoning 1

1.1.2 Accesso e gestione dei Big Data 2

1.1.3 Inserimento di codice procedurale e di terze parti 3

1.2 Architettura 3

1.2.1 Reasoning 4

1.2.2 Architettura stream-based e tecniche per la gestione della cache . 5

1.2.3 Interfacce 8

2 Il linguaggio Vadalog **10**

2.1 Questa è una Sezione 10

2.1.1 Questa è una Sottosezione 10

3 Algoritmi **12**

4 Prove sperimentali **13**

5 Related Work **14**

Conclusioni e sviluppi futuri **15**

Bibliografia **16**

Elenco delle figure

1.1 Schema dell'architettura dell'Engine di Vadalog. 4
1.2 Esempio di query plan. 6
1.3 Funzionamento dell'architettura stream-based. 8
1.4 Sistemi per interfacciarsi con Vadalog Engine. 9

2.1 SPQR-tree di un grafo. (a) L'albero di allocazione della faccia esterna. (b) Il cammino notevole di cui si parla tanto nella Sezione 2.1. 10

Capitolo 1

Vadalog Engine

1.1 Proprietà di un KGMS

Un KGMS completo deve disporre di diversi requisiti, che elencheremo sulla base di tre categorie principali, descritte nei sottocapitoli 1.1.1, 1.1.2 ed 1.1.3

1.1.1 Linguaggio e sistema per il reasoning

Dovrebbe esistere un formalismo logico per esprimere fatti e regole, e di un engine per il reasoning che usa tale linguaggio che dovrebbe fornire le seguenti caratteristiche:

- Sintassi semplice e modulare: Deve essere semplice aggiungere e cancellare fatti e nuove regole. I fatti devono coincidere con le tuple sul database.
- Alta potenza espressiva: Datalog [CGT12, HGL11] è un buon punto di riferimento per il potere espressivo delle regole. Con una negazione molto lieve cattura PTIME [DEGV01].
- Calcolo numerico e aggregazioni: Il linguaggio deve essere arricchito con funzioni di aggregazione per la gestione di valori numerici.
- Probabilistic Reasoning: Il linguaggio dovrebbe essere adatto ad incorporare metodi di reasoning probabilistico e il sistema dovrebbe propagare probabilità o valori di certezza durante il processo di reasoning.

- Bassa complessità: Il reasoning dovrebbe essere tracciabile nella complessità dei dati. Quando possibile il sistema dovrebbe essere in grado di riconoscere e trarre vantaggio da set di regole che possono essere elaborate in classi di complessità a basso livello di spazio.

- Rule repository, Rule management and ontology editor: È necessario fornire una libreria per l'archiviazione di regole e definizioni ricorrenti, ed un'interfaccia utente per la gestione delle regole.

- Orchestrazione dinamica: Per applicazioni più grandi, deve essere presente un nodo master per l'orchestrazione di flussi di dati complessi.

1.1.2 Accesso e gestione dei Big Data

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

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

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

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

- 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 pubblicizz-

zati dai concorrenti) e integrandoli in database locali e scambiare dati con moduli e server web disponibili (API).

1.1.3 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.

1.2 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.
Le fonti esterne sono supportate e gestite attraverso dei trasduttori, ovvero adattatori intelligenti che consentono un'interazione attiva con le fonti durante il processo di reasoning.
Come indicato in figura 1.1, che rappresenta la nostra architettura di riferimento, la componente centrale di un KGMS è il suo motore di reasoning principale, che ha accesso ad un repository di regole. Sono stati raggruppati vari moduli che forniscono funzionalità pertinenti di accesso ai dati ed analisi. Come possiamo vedere ad esempio la possibilità di accedere a RDBMS utilizzando il linguaggio standard SQL, nonché l'accesso ai NoSQL Stores attraverso delle API, ecc...

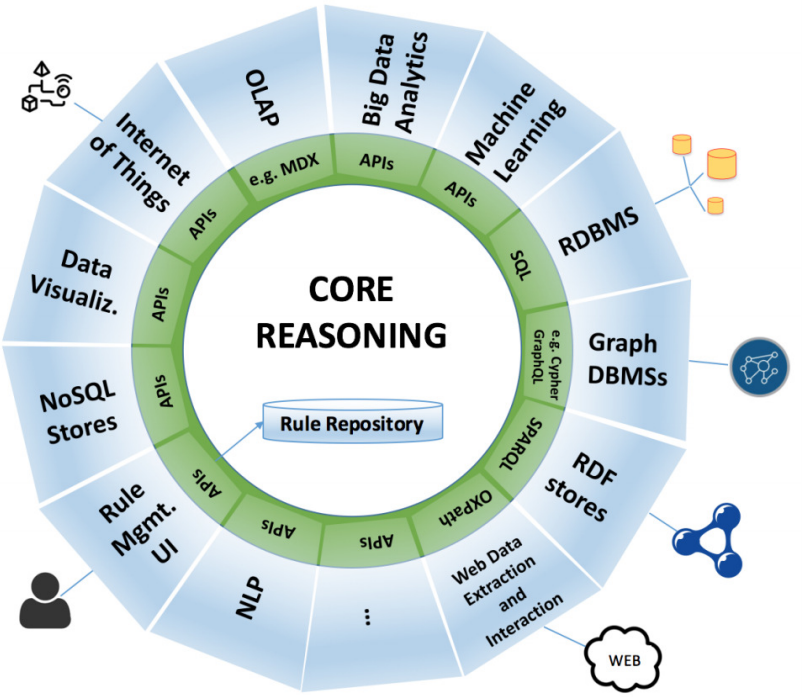


Figura 1.1: Schema dell'architettura dell'Engine di Vadalog.

1.2.1 Reasoning

Il linguaggio Vadalog estende la famiglia di linguaggi Datalog±, che chiameremo anche Warded Datalog±, 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).
Ne parleremo in maniera più approfondita nel Capitolo 2.

In questa sezione viene mostrato il processo di reasoning, ovvero come il sistema sfrutta le proprietà chiave di Warded Datalog±.

Il sistema Vadalog, fa un uso piuttosto alto di ricorsione, quindi è probabile che ci si ritrovi in un ciclo infinito all'interno del grafo di esecuzione (grafo contenente tutti i nodi per la ricerca della soluzione, partendo dai nodi di output ai nodi di input), si utilizza quindi una strategia di *Termination Control*.
Tale strategia, rileva ridondanza, ovvero la ripetizione di un nodo più volte, il prima possibile combinando il tempo di compilazione e tecniche a runtime.
A tempo di compilazione, grazie alla wardedness, che limita l'interazione tra i labeled null, l'engine riscrive il programma in modo che i joins con specifici valori di labeled null non si verificheranno mai (Harmful Join elimination).
A runtime, il sistema adotta una tecnica di potatura ottimale di ridondanza e rami che non terminano mai (per la ricorsione), questa tecnica è strutturata in due parti **detection** e **potatura**.
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 potatura, 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.

1.2.2 Architettura stream-based e tecniche per la gestione della cache

Il sistema Vadalog, 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 nel sottocapitolo 1.2.1, che garantiscono la risoluzione e la ridondanza.
Da un insieme di regole Vadalog, viene generato un *query plan* (cioè un grafo con un nodo per ogni regolae 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 1.2 viene rappresentato un esempio di query plan di un semplice programma Vadalog.

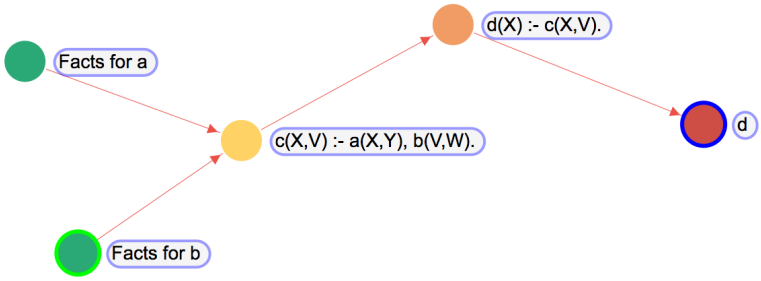


Figura 1.2: Esempio di query plan.

Il sistema Vadalog utilizza un approccio stream-based (o approccio a pipeline). Tale approccio [Wik] presume di avere i dati da elaborare organizzati in gruppi (stream) e che questi possano essere elaborati applicandone una serie di operazioni, spesso tali operazioni vengono elaborate tramite l'utilizzo di strutture a pipeline, e per ridurre i tempi vengono spesso utilizzate delle cache.
Nel nostro sistema i fatti sono attivamente richiesti dai nodi di output ai loro predecessori e così via, fino ad arrivare ai nodi di input, che ricavano i fatti dalla sorgente dati. L'approccio stream è essenziale per limitare il consumo di memoria, in modo che il sistema è 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 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 1.3, 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 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.

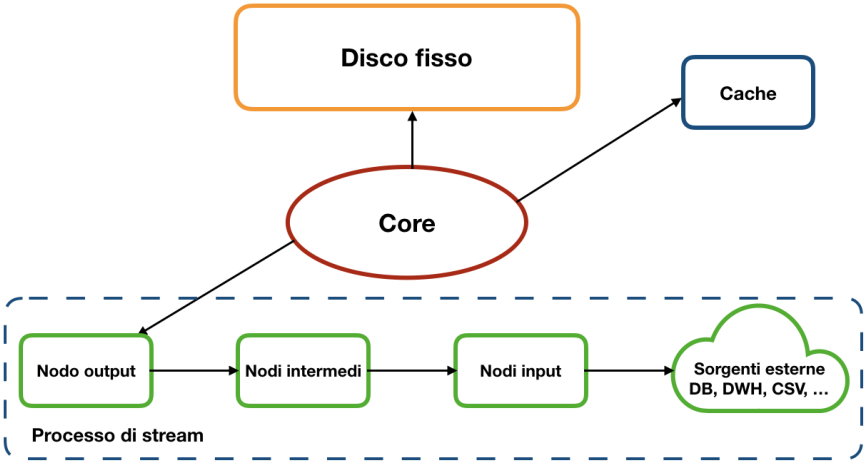


Figura 1.3: Funzionamento dell'architettura stream-based.

Per i join il sistema Vadalog adotta un estensione del Nested Loop Join, adatto per l'approccio stream-based ed efficiente in combinazione con le cache locali degli operandi.
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.

1.2.3 Interfacce

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

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

Come descritto in Figura 1.4 sono presenti due controller, uno per la gestione delle chiamate rest (VadaRestController), ed uno per la gestione delle chiamate Java (LibraryController), che si interfacciano con un unico controller che gestisce le chiamate pervenute da entrambi.

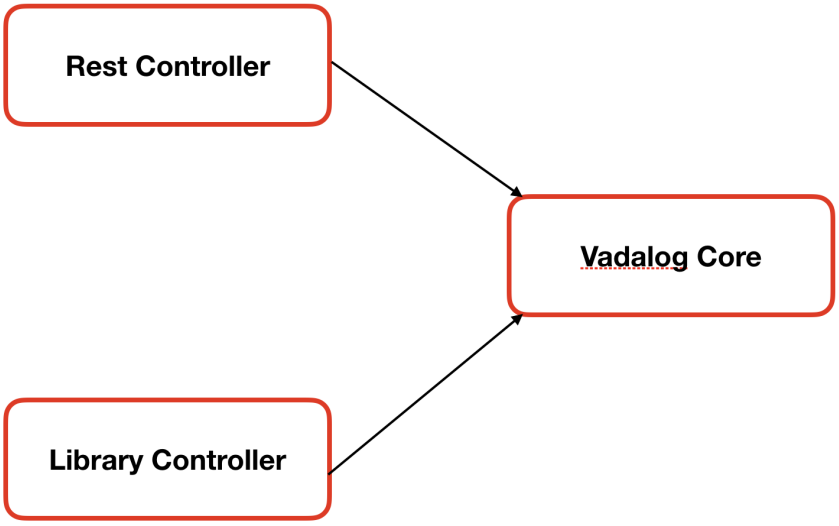


Figura 1.4: Sistemi per interfacciarsi con Vadalog Engine.

Capitolo 2

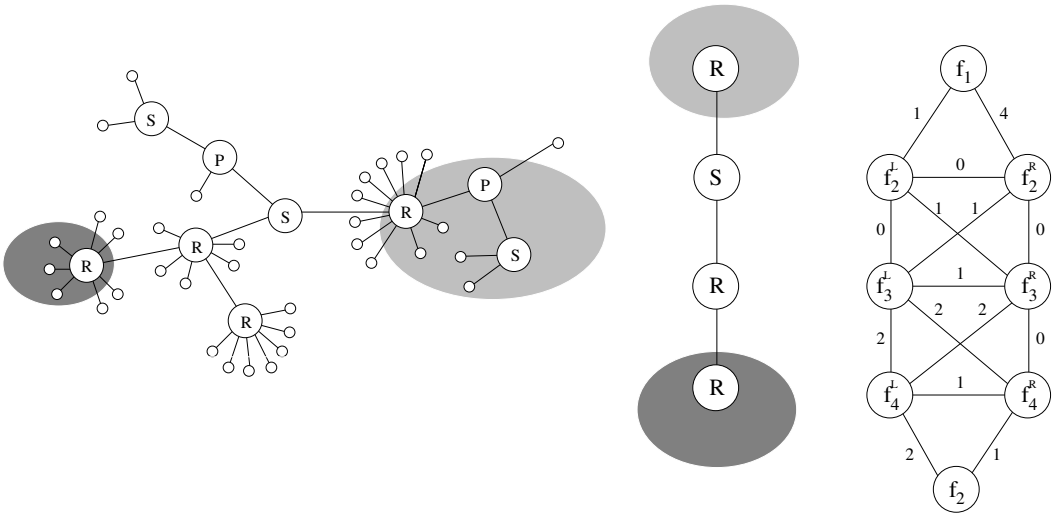
Il linguaggio Vadalog

2.1 Questa è una Sezione

Prova di testo di capitolo. Vorrei citare qui tutta l'opera omnia di [**?**, **?**, **?**, **?**].

2.1.1 Questa è una Sottosezione

Ancora del testo



(a) (b)

Figura 2.1: SPQR-tree di un grafo. (a) L'albero di allocazione della faccia esterna. (b) Il cammino notevole di cui si parla tanto nella Sezione 2.1.

Come si evince dalle Figure 2.1.a e 2.1.b non si capisce molto.

Capitolo 3

Algoritmi

Capitolo 4

Prove sperimentali

Capitolo 5

Related Work

Conclusioni e sviluppi futuri

La tesi è finita

Bibliografia

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

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

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

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

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

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

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

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

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

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

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