Università della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXVI CICLO

Settore Disciplinare INF/01 – INFORMATICA

Tesi di Dottorato

Tecniche per la valutazione distribuita di programmi logici

Rosamaria Barilaro

Supervisori
Prof. Nicola Leone
Prof. Giorgio Terracina

Coordinatore
Prof. Nicola Leone

A.A. 2013 – 2014

Università della Calabria

Dipartimento di Matematica

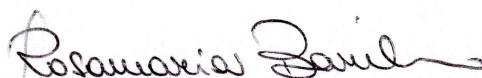
Dottorato di Ricerca in Matematica ed Informatica

XXVI CICLO

Settore Disciplinare INF/01 – INFORMATICA

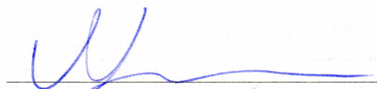
Tesi di Dottorato

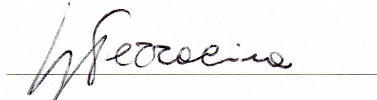
Tecniche per la valutazione distribuita di programmi logici



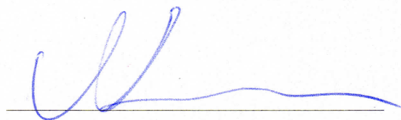
Rosamaria Barilaro

Supervisori

Prof. Nicola Leone 

Prof. Giorgio Terracina 

Coordinatore

Prof. Nicola Leone 

A.A. 2013 – 2014

Ringraziamenti

Grazie *in primis* al prof. Nicola Leone, coordinatore del dottorato e mio supervisore, che ha creduto in me incoraggiandomi nei momenti di difficoltà ed ha reso possibile il raggiungimento di questo traguardo.

Desidero poi ringraziare il mio supervisore prof. Giorgio Terracina, che mi ha guidato con pazienza in questi anni, per la grande disponibilità e cortesia dimostratemi, e per tutto l'aiuto fornito durante la stesura di questo lavoro.

Il mio ringraziamento va anche al prof. Francesco Ricca, per il suo autorevole aiuto su alcuni aspetti della ricerca.

Ringrazio inoltre Adele e Claudio per la loro competente disponibilità e tutti i colleghi con cui ho condiviso questo percorso.

Infine ma non da ultimo, desidero ringraziare con tutto il cuore la mia famiglia di origine e Leonardo per non avermi mai negato il loro affetto ed il loro sostegno in questi anni e per l'amorevole pazienza dimostrata nella fase di preparazione di questa tesi, che dedico loro con profonda riconoscenza.

Abstract

Recent developments in IT, and in particular the expansion of networking technologies, have made quite common the availability of software architectures where data sources are distributed across multiple (physically-different) sites. As a consequence, the number of applications requiring to efficiently query and *reason* on natively distributed data is constantly growing.

In this thesis we focus on the context in which it is necessary to combine data natively resides on different, autonomous and distributed sources and it is appropriate to deal with reasoning task to extract knowledge from the data, via deductive database techniques [1].

The aim is distributed evaluation of logic programs through an optimization strategy that minimizes the cost of the local process and data transmission.

We considered that a single logic rule can be seen as a conjunctive query (possibly with negation), whose result must be stored in the head predicate.

Then, starting from the conjunctive query optimization techniques, the idea is to extend the best results of these to evaluation of logic programs.

In this context the methods based on structural analysis of the queries seem particularly promising. Indeed, logical rules often contain multiple interactions interactions among join variables [2].

In the case of simple queries (acyclic) there are several algorithms that ensure execution time with a polynomial upper bound. Structural methods [3, 4, 5, 6] attempt to propagate the good results of acyclic queries to larger classes of these, whose structure is cyclic, but with a low "degree of cyclicity".

The Hypertree Decomposition technique [6] appears to be the most powerful since generalizes strongly all other structural methods and guarantees improved response times for each class of queries. Decomposition can be interpreted as an execution plan for the query, which first requires the evaluation of the join associated with each cluster, and then requires the processing of the resulting join tree using a bottom-up approach.

We used a weighted extension of Hypertree Decomposition [7] that combine structural analysis with evaluation of relevant quantitative information about the data, such as the number of tuples in relations, the selectivity of attributes and so on, and calculates minimum decompositions w.r.t. a cost function. We suitably modified this method in order to estimate the cost of data transmission between different sites resulting from the distribution of the sources and the correct evaluation of negation in rule bodies.

According decomposition the query is transformed into a (tree-like) set of sub-queries which also allows the parallel evaluation of independent sub-query. We used parallel techniques combined with techniques for query optimization.

We have adopted DLV^{DB} [8, 9, 10, 11] as core reasoning engine, which allows to evaluate logic programs directly on database and combines appropriately expressive power of logic programming systems and the efficient data management of DBMSs. The interaction with databases is achieved by means ODBC connections, therefore, in case of distributed computing on network it allows to transparently access different sources and to express very simply distributed queries.

We have implemented a prototype that we used to conduct the experiments. The preliminary results are very encouraging and showed the validity of the approach.

Sommario

I recenti sviluppi nel settore dell'Information Technology ed, in particolare, la capillare diffusione delle tecnologie di rete, hanno reso disponibili nuove architetture software in cui i dati sono distribuiti su più siti (fisicamente differenti). Di conseguenza, il numero di applicazioni che necessitano di eseguire efficientemente query effettuando al contempo processi di ragionamento su dati distribuiti in modo nativo è in costante aumento.

In questa tesi poniamo l'attenzione al contesto in cui sia necessario combinare tra loro dati residenti in modo nativo su diverse sorgenti autonome e distribuite ed in cui sia opportuno implementare meccanismi di ragionamento per estrarre conoscenza dai dati e realizzare processi di inferenza, così come avviene nei sistemi di basi di dati deduttive [1].

L'obiettivo è la valutazione distribuita di programmi logici attraverso una strategia di ottimizzazione che minimizzi il costo del processo locale e della trasmissione dei dati.

L'analisi del problema nasce dall'osservazione che una singola regola logica può essere vista come una query congiuntiva (eventualmente con negazione) il cui risultato deve essere memorizzato nel predicato della testa. Partendo, quindi, dalle tecniche di ottimizzazione di query congiuntive, l'idea è di tentare di estenderne i migliori risultati alla valutazione di programmi logici. In questo contesto risultano particolarmente promettenti tecniche basate sull'analisi delle proprietà strutturali delle interrogazioni. Infatti, nel caso di regole logiche, esistono spesso molteplici interazioni tra le variabili dei predicati in esse contenute [2].

Nel caso di interrogazioni semplici (acicliche) esistono numerosi algoritmi che garantiscono un tempo di esecuzione con un upper bound polinomiale. I metodi strutturali [3, 4, 5, 6] tentano di propagare i buoni risultati relativi alle query acicliche a classi più ampie, la cui struttura è ciclica, ma con un basso "grado di ciclicità".

La tecnica basata sulla nozione di Hypertree Decomposition [6] sembra essere tra queste quella più potente, in quanto generalizza fortemente tutti gli altri metodi strutturali e garantisce per ogni classe di interrogazioni migliori tempi di risposta. La decomposizione può essere interpretata come un vero e proprio piano di esecuzione per la query, che richiede dapprima la valutazione dei join associati a ciascun cluster e quindi il processamento del join tree risultante utilizzando un approccio bottom-up.

Nel nostro approccio, è stata utilizzata un'estensione pesata di Hypertree Decomposition [7] che, all'analisi prettamente strutturale aggiunge la valutazione di rilevanti informazioni quantitative sui dati, come dimensione delle relazioni e selettività degli attributi, e calcola decomposizioni minime rispetto ad una

funzione di costo. Abbiamo opportunamente modificato tale decomposizione al fine di stimare il costo di trasmissione dei dati fra siti diversi derivante dalla distribuzione delle sorgenti e per la corretta valutazione di atomi negati.

In accordo alla decomposizione la query viene trasformata in un albero di sotto-query che consente anche la valutazione parallela di sotto-query indipendenti. Abbiamo utilizzato tecniche di valutazione parallela combinate a tecniche di ottimizzazione di query. Abbiamo adottato DLV^{DB} [8, 9, 10, 11] quale motore di ragionamento, che effettua la valutazione dei programmi logici direttamente sulla base di dati, e che è in grado di combinare in maniera adeguata il potere espressivo dei sistemi di programmazione logica e l'efficiente gestione dei dati propria dei DBMS. L'interazione con le basi di dati è realizzata per mezzo di connessioni ODBC, pertanto, nel caso di elaborazione distribuita dei dati sulla rete consente di accedere in modo pressoché trasparente a sorgenti diverse e di esprimere in maniera molto semplice interrogazioni distribuite.

Abbiamo implementato un prototipo utilizzato per condurre gli esperimenti. I risultati preliminari sono stati particolarmente incoraggianti ed hanno dimostrato la validità dell'approccio.

Indice

1 Introduzione **1**

2 Nozioni preliminari **5**

2.1 Answer Set Programming 5

2.1.1 Programmazione Logica 8

2.1.2 Definizione formale della ASP 10

2.1.3 Rappresentazione della conoscenza e Ragionamento in ASP 14

2.1.4 Estensioni al linguaggio 18

2.1.5 Applicazioni 22

2.2 Hypertree Decomposition 23

2.2.1 Database e query acicliche 23

2.2.2 Query Decomposition 25

2.2.3 Hypertree Decomposition 26

2.2.4 Hypertree Decomposition Normal Form (NF) 30

2.2.5 Query Decomposition e Query Plan 31

2.2.6 Weighted Hypertree Decomposition 32

2.2.7 Un modello di costo 35

3 Tecnologie utilizzate **41**

3.1 DLV^{DB} 41

3.1.1 Esecuzione in memoria centrale 42

3.1.2 Esecuzione in memoria di massa 44

3.1.3 Architettura del sistema 47

3.2 DLV Wrapper 52

3.2.1 Panoramica sul DLV Wrapper Package 54

3.2.2 Architettura interna. 54

4 Descrizione della tecnica **61**

4.1 Unfolding delle regole 62

4.2 Ottimizzazione inter-componenti 67

4.3 Ottimizzazione intra-componente 69

4.3.1 Generazione statistiche 70

4.3.2 Tecnica di decomposizione 72

4.3.3 Creazione ed esecuzione del SubPrograms Tree 76

4.3.4 Ottimizzazione “bulk copy” 77

4.3.5 Ottimizzazione “caching” 78

5 Letteratura correlata **79**
5.1 Ottimizzazione di query distribuite 79
5.1.1 Distributed INGRES Algorithm 82
5.1.2 R* algorithm 84
5.1.3 SDD-1 algorithm 85
5.1.4 AHY algorithm 86
5.2 Data Integration 88
5.2.1 Mapping Global as View (GAV) 92
5.2.2 Mapping Local as View (LAV) 95
5.2.3 Nuove architetture per sistemi di integrazione dati 96
5.3 Valutazione parallela di programmi logici 99

6 Applicazioni ed Esperimenti **101**
6.1 Data integration 101
6.1.1 Esperimenti 102
6.2 Ontology based Data Access distribuito 105
6.2.1 Esperimenti 108
6.3 Multi-Context Systems relazionali 109
6.3.1 Esperimenti 113

7 Conclusioni **115**

Bibliografia **117**

Elenco delle tabelle

6.1 Test su programmi logici. 103
6.2 Risultati dei test su MCS. 114

Elenco delle figure

2.1 Join tree per l'ipergrafo $H(Q_0)$ 24
2.2 2-width decomposition per la query Q_1 26
2.3 2-width decomposition per l'ipergrafo H_2 28
2.4 Atom representation dell'hypertree decomposition di Figura 2.2.4 29
2.5 Join tree per la query Q'_2 31
2.6 Algoritmo minimal-k-decomp 34
2.7 Stime utilizzate nel modello di costo per le operazioni 38

3.1 DLV^{DB} Grammatica per le direttive ausiliarie 45
3.2 DLV^{DB} Esempio di direttive ausiliarie 47
3.3 DLV^{DB} Architettura del sistema 48
3.4 DLV^{DB} Algoritmo Semi-Naïve differenziale 52
3.5 DLV Wrapper: Pipeline di esecuzione 55

4.1 Dependency Graph del programma P 68
4.2 Grafo di esecuzione parallela dei moduli del programma P 69
4.3 Hypertree Decompositions: (a) etichettatura statica delle sorgenti (b) etichettatura alternativa delle sorgenti. 73

6.1 Risultati dei test su programmi logici. 104
6.2 Ontology-based Data Access distribuito: (a) Framework e (b) Architettura del sistema. 107
6.3 Esecuzione delle query Ontology-based: riscritture di (a) Presto e (b) Requiem. 108

Capitolo 1

Introduzione

La rapida evoluzione dei sistemi informatici e la loro diffusione in molti settori commerciali, statali e di ricerca, ha determinato, negli ultimi anni un vertiginoso aumento delle sorgenti informative, generalmente autonome, eterogenee e largamente distribuite, ponendo nuove sfide nella gestione delle informazioni.

In tale contesto, i sistemi di gestione di basi di dati distribuite (DDBMS) attualmente in commercio, soddisfano solo parzialmente l'ambizioso obiettivo di integrazione in domini applicativi complessi, non supportando meccanismi di ragionamento necessari per estrarre conoscenza dai dati e realizzare sofisticati processi di inferenza.

Le capacità di ragionamento necessarie a tal fine possono essere fornite dai sistemi basati su linguaggi logici [12, 13, 14, 15], il cui recente sviluppo ha rinnovato l'interesse nel campo del ragionamento non-monotono e della programmazione logica dichiarativa per la risoluzione di molti problemi in differenti aree applicative. Da ciò la possibilità di fornire funzionalità di inferenza e ragionamento richieste dalle nuove aree di applicazione che interessano i sistemi di basi di dati.

Tuttavia, il limite di questi sistemi nella gestione di grandi quantità di dati distribuiti è evidente poiché l'elaborazione viene effettuata direttamente in memoria centrale le cui dimensioni possono risultare insufficienti per grandi volumi di dati su cui applicare complessi algoritmi di inferenza logica. Da ciò la necessità di tecniche in grado di combinare in maniera adeguata il potere espressivo dei sistemi di programmazione logica e l'efficiente gestione dei dati propria dei DBMS. Tale significativa sintesi è caratteristica del sistema DLV^{DB} [8, 11], progettato come estensione del sistema DLV, che combina l'esperienza (maturata nell'ambito del progetto DLV) nell'ottimizzare programmi logici con le avanzate capacità di gestione delle basi di dati implementate nei DBMS esistenti. Risulta, pertanto, oltremodo efficiente in applicazioni che necessitino di valutare programmi complessi, manipolando grandi quantità di dati.

L'interazione con le basi di dati è realizzata per mezzo di connessioni ODBC, pertanto, nel caso di elaborazione distribuita dei dati sulla rete consente di accedere in modo pressoché trasparente a sorgenti eterogenee in termini di schema globale e di esprimere quindi in maniera estremamente semplice interrogazioni distribuite.

Il contributo della tesi si colloca nell'ambito dell'integrazione dei dati attraverso la programmazione logica con l'obiettivo di effettuare la valutazione dei programmi logici direttamente sui database sorgente attraverso una strategia di ottimizzazione che minimizza il costo del processo locale e della trasmissione dei dati.

Per l'ottimizzazione di ogni singola regola logica abbiamo sfruttato un'estensione pesata di Hypertree Decomposition [7] che, all'analisi prettamente strutturale aggiunge la valutazione di rilevanti informazioni quantitative sui dati, come dimensione delle relazioni e selettività degli attributi, e calcola decomposizioni minime rispetto ad una funzione di costo. Abbiamo opportunamente modificato tale metodi di decomposizione al fine di stimare il costo di trasmissione dei dati fra siti diversi derivante dalla distribuzione delle sorgenti e per la corretta valutazione di atomi negati.

Abbiamo sviluppato un algoritmo di riscrittura della query ottimizzata posta sullo schema globale in termini di sotto-query poste sulle sorgenti. Le sotto-query, infine, vengono processate in parallelo secondo il piano di esecuzione individuato dall'ottimizzazione, con l'obiettivo di minimizzare il tempo globale di risposta.

Abbiamo sfruttato tecniche di parallelizzazione al fine di migliorare l'efficienza nella valutazione dei programmi. In particolare, abbiamo integrato nel nostro approccio il parallelismo in tre diversi momenti della computazione.

Il primo livello di parallelismo viene attuato nell'esecuzione dei moduli, che consistono in una porzione del programma in ingresso indotta dal Grafo delle Dipendenze: è particolarmente utile quando si trattano programmi contenenti parti che sono, in qualche modo, indipendenti.

Il secondo livello permette la valutazione parallela delle regole all'interno di un determinato modulo: esso è particolarmente efficace quando è elevato il numero di regole nei moduli.

Il terzo, per ogni singola regola, permette la valutazione parallela dei sotto-programmi ottenuti dal processo di decomposizione strutturale.

I principali contributi di questa tesi possono essere riassunti come segue:

- Abbiamo studiato nuove tecniche per la valutazione distribuita di generici programmi Datalog, in grado di integrare informazioni quantitative sui dati con analisi prettamente strutturali della query;
- Abbiamo adattato tecniche di riscrittura di query basate su "unfolding" [1] allo scopo di individuare l'insieme minimale di dati sorgente effettivamente necessari al calcolo della risposta alla query e di ristrutturare la query utente per generare sotto-query su cui la tecnica sviluppata risulta essere particolarmente efficace;
- Abbiamo investigato ed implementato tecniche di esecuzione parallela dei programmi, in grado di ridurre significativamente i tempi di risposta;
- Abbiamo elaborato tecniche di trasferimento dei dati tra origini eterogenee che consentono di effettuare una gestione più veloce ed efficiente delle copie;
- Abbiamo indagato nuovi scenari di applicazione della tecnica nell'area del "Ontology-Based Query Answering" e dei "Multi-Contents Systems", discutendo i risultati ottenuti.

La tesi è strutturata nel modo seguente:

- nel Capitolo 2 introduciamo nozioni preliminari alla comprensione dei capitoli successivi. In particolare descriviamo il formalismo dell'Answer Set Programming, fornendo una descrizione della sintassi e della semantica ed una discussione sulle sue estensioni e sulle possibili applicazioni. Di seguito presentiamo i principali risultati relativi alle query acicliche e descriviamo nel dettaglio la tecnica di decomposizione basata su hypertree. Illustriamo un'estensione pesata di Hypertree Decomposition che considera anche informazioni quantitative sui dati ed il relativo algoritmo per il calcolo di decomposizioni minime rispetto ad una funzione di costo. Tale decomposizione, opportunamente modificata al fine di stimare il costo di trasmissione dei dati fra siti diversi derivante dalla distribuzione delle sorgenti, viene utilizzata per la valutazione del programma logico distribuito e la generazione del query plan;

- nel Capitolo 3 descriviamo le tecnologie e gli strumenti utilizzati per la realizzazione del progetto. Ci soffermiamo, infatti, sulla descrizione del sistema DLV^{DB}, una versione del sistema DLV specifica per l'esecuzione di programmi su DMBS, utilizzata quale motore di inferenza nella valutazione dei programmi e sul Wrapper Java, sviluppato presso l'Università della Calabria, in grado di gestire l'interazione tra la DLP e programmi scritti in Java;

- il principale contributo della tesi è presentato nel Capitolo 4, in cui descriviamo nel dettaglio la tecnica progettata. Illustriamo, dandone adeguata motivazione, le scelte effettuate e le limitazioni imposte al sistema. Descriviamo l'algoritmo di decomposizione della query distribuita posta sullo schema globale e la riscrittura in termini di sotto-query sulle sorgenti, nonchè le ottimizzazioni implementate nel sistema;

- nel Capitolo 5 analizziamo la letteratura correlata, discutendo i concetti fondamentali relativi all'ottimizzazione di query distribuite ed analizzando nel dettaglio le problematiche relative ai costi di esecuzione del processo locale e della trasmissione dei dati, nonchè i più diffusi algoritmi di ottimizzazione di query distribuite implementati nei DDBMS. Discutiamo, altresì, delle diverse soluzioni presenti in letteratura in ambito di integrazione dati, analizzando le caratteristiche architetturali dei diversi approcci; Presentiamo, altresì, alcuni approcci alla parallelizzazione di programmi logici, analizzando caratteristichei e potenzialità.

- nel Capitolo 6 mostriamo l'applicazione della nostra tecnica in contesti eterogenei, discutendo i risultati della sperimentazione;

- infine, il Capitolo 7 contiene una sintesi del lavoro svolto e delle problematiche affrontate.

Capitolo 2

Nozioni preliminari

La prima parte del capitolo illustra i concetti fondamentali dell'Answer Set Programming, un paradigma di programmazione dichiarativo basato sul ragionamento non-monotono e la programmazione logica.

Nella seconda parte del capitolo presentiamo i concetti fondamentali relativi alle query congiuntive e riportiamo gli importanti risultati ottenuti sulla particolare sottoclasse delle interrogazioni congiuntive acicliche. Presentiamo e confrontiamo, inoltre, le tecniche di decomposizione strutturale Query Decomposition e la più recente Hypertree Decomposition. Introduciamo al contempo un'efficiente strategia di valutazione per interrogazioni aventi hypertree-width limitata, ed un'estensione weighted del concetto di Hypertree Decomposition, in grado di sfruttare anche eventuali dati quantitativi e statistiche di catalogo associati ad un'interrogazione. La sezione si conclude illustrando un algoritmo polinomiale in grado di calcolare decomposizioni minime rispetto ad una particolare funzione di peso, che corrispondono a piani di esecuzione ottimi in base al modello di costo utilizzato.

2.1 Answer Set Programming

Le applicazioni software, ormai, hanno pervaso le nostre vite; infatti, al giorno d'oggi, molti problemi reali sono trattati in modo automatizzato. Tuttavia, non tutti i problemi possono essere risolti in maniera semplice da un computer, alcuni di essi hanno un'elevata complessità intrinseca. Trovare metodi efficienti e corretti per risolverli non è semplice. L'ingegnerizzazione dei software tradizionali si focalizza su un approccio algoritmico ed imperativo, in cui al computer viene detto praticamente quali passi deve eseguire per risolvere un dato problema. Trovare buoni algoritmi per problemi difficili richiede abilità e conoscenze, tuttavia spesso non risulta essere un task semplice. Per questo potrebbe essere utile rivolgersi ad un esperto, anche se ci potrebbero essere comunque dei problemi: per esempio, quando le specifiche del problema variano anche di poco, magari perché vengono fuori alcune alcune informazioni addizionali sulla natura del problema, spesso sono necessarie delle importanti reingegnerizzazioni. La limitazione principale di tale approccio sta nel fatto che la conoscenza del pro-

blema e delle sue soluzioni è stata rappresentata implicitamente attraverso la specifica di un particolare modo di risolvere il problema piuttosto che attraverso la specifica del problema stesso. Il caso degli aggiornamenti di rappresentazioni viene comunemente chiamato "tolleranza di elaborazione".

Un'alternativa che si adatta bene alla tolleranza di elaborazione è la cosiddetta programmazione dichiarativa. In questo approccio, il problema e le sue soluzioni sono specificate in maniera esplicita. In particolare, anziché specificare come una soluzione deve essere ottenuta, in questo contesto vengono espresse solo le caratteristiche che devono avere il problema e le sue soluzioni. Metodi come questo vengono sicuramente naturali nella comunità scientifica, ma anche nella vita di tutti i giorni. Infatti, prima di provare a risolvere un problema, di solito lo studiamo e cerchiamo di capire come sono fatte le sue soluzioni prima di cercare un metodo per calcolarne una. Uno dei primi ad introdurre questo metodo nell'informatica è stato John McCarthy negli anni '50 [16]. Egli ha anche ipotizzato che il linguaggio più naturale per specificare problemi fosse la logica, in particolare la logica dei predicati.

Infatti, la logica è un candidato eccellente per la programmazione dichiarativa: fornisce un formalismo semplice ed astratto, ed inoltre si presta bene ad una eventuale automazione. Analogamente ad una macchina astratta od elettronica che può eseguire un modello imperativo (un algoritmo) per ottenere una soluzione del problema modellato, la logica computazionale ha prodotto dei tools che, data la specifica dichiarativa espressa in logica di un certo problema, consentono di ottenere le soluzioni in via del tutto automatica. Infatti, molta gente al giorno d'oggi usa questo metodo per la risoluzione dei problemi: le query su una base di dati relazionale insieme agli schemi di basi di dati sono specifiche dichiarative delle soluzioni che i risultati della query forniscono. Inoltre, il più diffuso linguaggio per l'interrogazione delle basi di dati, l'SQL, è fondamentalmente una logica dei predicati scritta in un modo particolare [17].

Tuttavia, si vuole andare oltre le basi di dati per come usate oggi. E' stato dimostrato che le basi di dati relazionali ed i linguaggi di interrogazione come l'SQL possono rappresentare solo problemi semplici. Per esempio, problemi come trovare il percorso più conveniente per la visita di un certo numero di città, o riempire un contenitore con oggetti di dimensioni differenti in modo da massimizzare il valore trasportato nel contenitore, sono problemi tipici che probabilmente non possono essere risolti usando l'SQL. Potrebbe sembrare inusuale utilizzare in questo contesto la parola "probabilmente", ma alla base di questa congettura c'è uno dei più famosi problemi aperti dell'informatica – capire se P è uguale ad NP. Queste ultime sono classi di complessità; fondamentalmente ogni problema ha una complessità intrinseca che si basa sulla quantità di risorse richieste per risolverlo su un modello di macchina standard, espressa in termini di dimensione dell'input. P è definita come la classe dei problemi che richiedono al massimo una quantità di risorse temporali esprimibile come un polinomio sulla dimensione dell'input (che è variabile). NP è una lieve alterazione di P, dove invece di un modello di macchina deterministica viene usato un modello di macchina non deterministica. Una macchina non deterministica è un concetto piuttosto inusuale: invece di eseguire comandi uno per volta, andando sempre da uno stato macchina all'altro, una macchina non deterministica si può trovare in due o più stati (allo stesso tempo) dopo aver eseguito un comando. In un certo senso, questo significa che la macchina ha la possibilità di memorizzare e lavorare con un numero illimitato di stati macchina ad ogni passo. Intuitivamente, ci

si aspetterebbe che una macchina deterministica sia profondamente diversa da una non deterministica, e che la non deterministica risolva più problemi sotto gli stessi vincoli di tempo. Tuttavia, finora nessuno è riuscito a dimostrare in maniera convincente né che P è diversa da NP, né che le due classi coincidono. Comunque, intuitivamente ci si aspetterebbe che siano diverse, ed in molti hanno dimostrato che se P ed NP coincidessero dei risultati molto meno intuitivi ne seguirebbero.

La Programmazione Logica rappresenta un tentativo di utilizzo della programmazione dichiarativa con la logica che va oltre i problemi in P, e così oltre le basi di dati tradizionali. Il costrutto principale nella programmazione logica è la regola, un'espressione del tipo *Head* ← *Body*, dove *Body* è una congiunzione logica che potenzialmente potrebbe includere la negazione, ed *Head* o è una formula atomica o una disgiunzione logica. Queste regole possono essere viste come delle formule logiche (← denota una implicazione), il cui significato speciale dice che ciascuna *Head* è definita dalla regola alla quale appartiene. Agli inizi (come descritto nel paragrafo successivo), la programmazione logica in realtà ha tentato di diventare un "linguaggio di programmazione" su larga scala. Il suo linguaggio più famoso, PROLOG [18], ha cercato di raggiungere questo obiettivo, ma per poterci giungere ha dovuto rinunciare alla piena dichiaratività. Per esempio, nelle regole PROLOG l'ordine all'interno del *Body* è importante, così come l'ordine tra le regole (in particolare per la terminazione). Inoltre, PROLOG ha anche un numero non indifferente di costrutti non logici.

L'Asnwer Set Programming (ASP) è una branca della programmazione logica, che non aspira a creare un linguaggio general-purpose. A tal riguardo, è influenzata dai linguaggi per le basi di dati, visto che nemmeno questi sono linguaggi general-purpose ma sono sufficienti per una particolare classe di problemi. ASP fa comunque uno sforzo per allargare la classe dei problemi esprimibili dal linguaggio. Mentre, come già detto in precedenza, probabilmente in SQL non si possono esprimere problemi NP difficili, in ASP si può fare. In realtà, in ASP si possono esprimere tutti i problemi nella classe di complessità Σ_2^P e nella classe ad essa complementare Π_2^P, che sono delle classi simili ad NP, ma probabilmente piuttosto maggiori (quantomeno uguali).

In ASP, il costrutto regola *Head* ← *Body* (dove *Head* può essere una disgiunzione) si legge come una formula di logica non-monotona, piuttosto che di logica classica. Le logiche non-monotone rappresentano uno sforzo per formulare una logica di senso comune che adatta la semantica della logica in modo tale che questa corrisponda meglio al ragionamento di tutti i giorni, il quale è caratterizzato dalla presenza di conoscenza incompleta, ragionamenti ipotetici ed assunzioni di default. Si può dimostrare che le logiche non-monotone si adattano a questo contesto molto meglio delle logiche classiche.

Ricapitolando, ASP è un formalismo che è emerso dalla programmazione logica. La sua principale caratteristica rappresentativa è costituita dalle regole, le quali sono interpretate secondo i principi di senso comune. Essa consente la specifica puramente dichiarativa di una vasta classe di problemi, generalizzando l'approccio dichiarativo delle basi di dati. In ASP, si può scrivere un programma (un insieme di regole), che rappresenta un problema da risolvere. Questo programma, insieme a qualche input, il quale è espresso a sua volta attraverso un insieme di regole, ha una serie di soluzioni (o anche nessuna) che corrispondono alle soluzioni del problema modellato. Il termine Answer Set è stato coniato proprio perché queste soluzioni, di solito, sono degli insiemi.

Riguardo alla terminologia, la ASP a volte viene usata in senso molto più ampio, riferendosi a qualsiasi formalismo dichiarativo che rappresenta le soluzioni come degli insiemi. Comunque, il significato più frequente, è adottato in questo discussione, risale a [19]. Inoltre, dato che la ASP è uno dei rami più importanti della programmazione logica all'interno del quale le teste delle regole possono essere disgiuntive, a volte il termine Disjunctive Logic Programming (DLP) può essere ritrovato in luogo della ASP. Ancora, altri termini che si usano spesso al posto della ASP sono A-Prolog e Stable Logic Programming.

2.1.1 Programmazione Logica

Le radici della Answer Set Programming si ritrovano prevalentemente nella Programmazione Logica, nel ragionamento non-monotono e nelle basi di dati. In questo paragrafo ci limiteremo a dare una descrizione generale sulla storia della Programmazione Logica dalla prospettiva della Answer Set Programming. Dunque, non arriveremo a discutere di alcuni importanti aspetti della Programmazione Logica, quali la Constraint Logic Programming [20] o la Abductive Logic Programming [21].

Come citato in precedenza, probabilmente il primo a suggerire la logica, ed in particolare la logica dei predicati, come un linguaggio di programmazione fu John McCarthy negli anni '50 [16]. L'esempio motivante di McCarthy era costituito da un'applicazione di Intelligenza Artificiale e coinvolgeva il planning come task principale, un'agenda in continua elaborazione [22].

Gli sviluppi nel campo della Logica Computazionale, in particolare la specifica del principio di Risoluzione ed Unificazione come un metodo computazionale introdotto da J. Alan Robinson nel 1965 [23], hanno fatto da catalizzatore per la nascita della Programmazione Logica. Questo sviluppo alla fine si è veramente diffuso solo quando un sistema reale, Prolog, sviluppato intorno ad Alain Colmerauer a Marsiglia, divenne disponibile [18]. In realtà, anche altri sistemi molto più ristretti erano stati rilasciati prima, ma Prolog rappresentò il vero passo in avanti della Programmazione Logica.

Uno dei primi sostenitori di quello che sarebbe diventato noto come il paradigma della Programmazione Logica fu Robert Kowalski, che fornì le basi filosofiche del paradigma della Programmazione Logica in alcuni dei suoi lavori, per esempio in [24] e [25]. Kowalski collaborò con Colmerauer su Prolog, inoltre dal suo gruppo di ricerca ad Edimburgo furono sviluppate alcune implementazioni alternative a Prolog. Ci fu anche uno sforzo notevole per cercare di trovare uno standard sul linguaggio, il quale sarebbe divenuto noto come Edinburgh Prolog e fu usato come specifica de facto di Prolog per molti anni fino alla definizione dell'ISO Prolog nel 1995 [26].

Tuttavia, la Programmazione Logica, e Prolog in particolare, furono ispirati dalla Logica del Primo Ordine classica, anche se questi tipi di logica non coincidono esattamente. Inizialmente le differenze non erano molto chiare. Il primo sforzo fatto col fine di fornire una definizione formale della semantica della Programmazione Logica è da ricondurre a Kowalski, il quale insieme a Maarten van Emden diede in [27] una semantica basata sul punto fisso di alcuni operatori per una classe ristretta di programmi logici (programmi Horn, chiamanti anche programmi positivi). Questa semantica del punto fisso essenzialmente coincide con quella dei modelli minimali di Herbrand e con quella per il query answering su programmi Horn basata su risoluzione. La caratteristica principale che man-

ca ai programmi Horn è la negazione – tuttavia Prolog aveva un operatore di negazione.

Infatti, la ricerca di una semantica adeguata nello spirito dei modelli minimali per programmi che contengono la negazione si è rivelata essere tutt'altro che semplice. Un primo sforzo fu fatto da Keith Clark in [28], definendo una trasformazione dei programmi in formule di logica classica, che vengono poi interpretate usando la semantica dei modelli classici. In particolare, la semantica ottenuta non coincide con quella dei modelli minimali su programmi positivi. Più o meno nello stesso periodo, Raymond Reiter formulò la Closed World Assumption in [29], che può essere vista come la base filosofica del trattamento della negazione. Un'ulteriore pietra miliare nella ricerca della semantica per programmi con negazione è stata la definizione di quella che più tardi sarebbe divenuta nota come la semantica dei modelli perfetti per programmi la cui negazione è stratificabile [30, 31]. L'idea base della stratificazione sta nel fatto che il programma può essere partizionato in sottoprogrammi (strati) tali che le regole di ciascuno strato contengono predicati negati solo se essi sono definiti in altri strati. In tal modo, è possibile valutare il programma valutando separatamente le sue partizioni cosicché un certo "strato" venga processato solo quando quelli da cui dipende (negativamente) saranno stati già processati.

Comunque un importante passo in avanti, anche se è ovvio che non tutti i programmi logici sono stratificati. In particolare, i programmi negativamente ricorsivi non sono mai stratificati, ed il problema di assegnare una semantica ai programmi non stratificati restava ancora aperto. Ci furono essenzialmente due approcci per trovare delle definizioni adeguate: nel primo si rinunciava alla definizione classica di modelli che assegnano solo due valori di verità, e si introduceva un terzo valore che rappresentava intuitivamente un valore sconosciuto. Questo approccio richiedeva una definizione piuttosto differente, perché nell'approccio a due valori si poteva dare una definizione solo per i valori positivi dicendo implicitamente che tutti gli altri costrutti sono considerati negativi. Per esempio, nei modelli minimali si cerca di minimizzare gli elementi veri affermando implicitamente che tutti gli elementi non contenuti nel modello minimale saranno falsi. Invece, con 3 valori di verità questa strategia non può più essere applicata, visto che gli elementi che non sono veri possono essere o falsi o indefiniti. Per risolvere questo problema, Allen Van Gelder, Kenneth Ross, e John Schlipf introdussero la nozione di unfounded sets, col fine di definire quali elementi del programma dovrebbero essere sicuramente falsi. Combinando le tecniche esistenti per la definizione dei modelli minimali con gli unfounded sets [32], essi hanno definito la nozione di well-founded model. In tal modo, ogni programma dovrebbe avere ancora un singolo modello, così come c'è un modello minimale unico per programmi positivi ed un modello perfetto unico per programmi stratificati.

Il secondo approccio consiste nel vedere i programmi logici come formule di logiche non-monotone [33] piuttosto che come formule di logica classica (con l'aggiunta di un criterio di minimalità), e come corollario, rinunciando alla proprietà di unicità del modello. Tra i primi a concretizzare questo approccio possiamo citare Michael Gelfond, il quale propose in [34] di considerare i programmi logici come formule di logica autoepistemica, Nicole Bidoit e Christine Froidevaux, che proposero in [35] di considerare i programmi logici come formule di logica di default. Entrambi questi sviluppi sono stati raccolti da Michael Gelfond e Vladimir Lifschitz, che in [36] definirono la nozione di modelli stabili, la quale

è stata ispirata dalle logiche non-monotone anche se non fa riferimento esplicito ad esse ma, piuttosto, si basa su una ridotta che emula in maniera efficace l'inferenza non-monotona. E fu proprio questa formulazione sorprendentemente semplice, che non richiedeva alcuna conoscenza pregressa sulle logiche non classiche, che divenne molto nota. A differenza dei modelli well-founded, poteva esistere nessuno, uno o anche più modelli stabili per ogni programma. Tuttavia, well-founded e modelli stabili sono strettamente correlati. Ad esempio, infatti, il modello well-founded di un programma è contenuto in ogni modello stabile [37]. Inoltre, entrambi gli approcci coincidono con i modelli perfetti su programmi stratificati.

Un'altra linea di ricerca decisamente ortogonale riguarda l'uso della disgiunzione nelle teste delle regole. Questo costrutto è molto utile, perché consente di rappresentare delle definizioni non deterministiche dirette. Prolog e molti altri linguaggi di programmazione logica tradizionalmente non forniscono tale caratteristica, essendo ristretti alle cosiddette regole definite. Jack Minker è stato uno dei pionieri e dei primi sostenitori della disgiunzione nei programmi. In [38] egli formulò la Generalized Closed World Assumption, che fornisce un esempio ed una semantica intuitiva per i programmi logici disgiuntivi. Questo concetto è stato elaborato nel corso degli anni, ed ha dato luogo alla nozione di Extended GCWA introdotta in [39]. Alla fine, anche la semantica dei modelli stabili è stata estesa ai programmi disgiuntivi [40] mediante una semplice alterazione della definizione originale data in [36]. Dall'altro lato, definire un'estensione dei modelli well-founded per programmi disgiuntivi rimane tuttora una questione controversa con varie definizioni rivaleggianti, [41].

Il passo finale verso la Answer Set Programming nel senso più tradizionale del termine è costituita dall'aggiunta di un secondo tipo di negazione, che ha un significato più classico della negation as failure. Combinando questa caratteristica con i modelli stabili disgiuntivi di [40] si arriva alla definizione degli answer set in [19].

2.1.2 Definizione formale della ASP

Nel prosieguo forniremo una definizione formale della sintassi e della semantica della Answer Set Programming nello spirito di [19], cioè la programmazione logica disgiuntiva che prevede due tipi di negazione (indicati come strong negation e negation as failure) sotto la semantica degli answer set.

Sintassi

Seguendo una convenzione risalente a Prolog, le stringhe che iniziano con lettera maiuscola denotano delle variabili logiche, mentre quelle che cominciano per lettera minuscola denotano delle costanti. Un *termine* è o una variabile o una costante. Si noti che di solito in ASP i simboli di funzione non vengono considerati.

Un *atomo* è un'espressione del tipo $p(t_1, ..., t_n)$, dove p è un predicato di arità n e $t_1, ..., t_n$ sono dei termini. Un *letterale classico* l può essere o un atomo p (in questo caso si dice che il letterale è positivo), oppure un atomo negato $\neg p$ (in questo caso si dice che è negativo). Un *negation as failure literal* l può essere del tipo l oppure *not* l, dove l è un letterale classico; nel primo caso l è

positivo, e nel secondo caso l è *negativo*. Se non diversamente specificato, con la parola *letterale* intendiamo un letterale classico.

Dato un letterale classico l, il suo letterale *complementare* $\neg\, l$ è definito come $\neg\, p$ se $l = p$ e come p se $l = \neg\, p$. Un insieme L di letterali si dice *consistente* se, per ogni letterale $l \in L$, il suo letterale complementare non è contenuto in L.

Una *regola disgiuntiva* (*regola* per brevità) r è un costrutto

$$(1)\ \mathrm{a}_1 \vee ... \vee \mathrm{a}_n \leftarrow \mathrm{b}_1, \ldots, \mathrm{b}_k, \text{not } \mathrm{b}_{k+1}, \ldots, \text{not } \mathrm{b}_m.$$

dove $a_1, \ldots, a_n, b_1, ..., b_m$ sono letterali classici ed $n \geq 0$, $m \geq k \geq 0$. La disgiunzione $\mathrm{a}_1 \vee ... \vee \mathrm{a}_n$ è chiamata la *testa* di r, mentre la congiunzione b_1, ... ,b_k, not b_{k+1}, ... , not b_m è indicata come il *corpo* di r. Una regola senza letterali in testa (cioè $n = 0$) solitamente è chiamata *integrity constraint*. Una regola che ha esattamente un solo letterale in testa (cioè $n = 1$) è chiamata *regola normale*. Se il corpo è vuoto (cioè $k = m = 0$), la regola è chiamata anche fatto ed, in questo caso, il simbolo "$\leftarrow$" viene generalmente omesso.

Se r è una regola del tipo (1), allora $H(r) = \{a_1, ..., a_n\}$ è l'insieme dei letterali della testa mentre $B(r) = B^+(r) \cup B^-(r)$ è l'insieme dei letterali del corpo, dove $B^+(r)$(il *corpo positivo*) è $\{b_1, ..., b_k\}$ mentre $B^-(r)$ (il *corpo negativo*) è $\{b_{k+1}, ..., b_m\}$.

Un *programma* ASP P è definito come un insieme finito di regole. Un programma not-free P (cioè tale che $\forall r \in P : B^-(r) = \emptyset$) è chiamato *positivo* oppure *Horn* (nei programmi positivi la negation as failure non appare, mentre la strong negation potrebbe esserci), ed un programma $\vee$-free P (cioè tale che $\forall r \in P : |H(r)| \leq 1$) è chiamato *programma logico normale*.

In ASP, le regole dei programmi di solito devono essere *safe*. Le cause dell'adozione della safety in questo contesto si possono ricondurre al campo delle basi di dati, dove la safety è stata introdotta come un mezzo per garantire che le query (i programmi nel caso della ASP) non dipendessero dall'universo (l'insieme delle costanti) considerato. Per esempio, un fatto $p(X)$. dà origine alla veridicità di $p(a)$ quando viene considerato l'universo $\{a\}$, mentre dà origine alla veridicità di $p(a)$ e $p(b)$ quando viene considerato l'universo $\{a, b\}$. I programmi safe non soffrono questo problema quando vengono considerate quantomeno le costanti che appaiono nel programma [17].

Una regola si dice *safe* se ciascuna delle sue variabili appare anche in almeno uno dei letterali positivi del suo corpo. Un programma ASP è *safe* se ciascuna delle sue regole è *safe*, e nel seguito faremo riferimento soltanto a programmi *safe*.

Un termine (un atomo, una regola, un programma, ecc.) si dice *ground*, se in esso non appare alcuna variabile. A volte un programma ground viene anche detto programma *proposizionale*.

Esempio 2.1.1. Si consideri il seguente programma:

r_1: a(X) $\vee$ b(X) $\leftarrow$ c(X,Y), d(Y), not e(X).

r_2: $\leftarrow$ c(X,Y), k(Y), e(X), not b(X).

r_3: m $\leftarrow$ n, o, a(1).

r_4: c(1,2).

r_1 è una regola disgiuntiva con $H(r_1) = \{a(X), b(X)\}$, $B^+(r_1) = \{c(X,Y), d(Y)\}$, e $B^-(r_1) = \{e(X)\}$.

r_2 è un integrity constraint con $B^+(r_2) = \{c(X,Y), k(Y), e(X)\}$, e $B^-(r_2) = \{b(X)\}$.

r_3 è una regola ground, positiva e non-disgiuntiva con $H(r_3) = \{m\}$, $B^+(r_3) = \{n,o,a(1)\}$, e $B^-(r_3) = \emptyset$.
Infine, r_4 è un fatto (si noti che il simbolo $\leftarrow$ è omesso). Inoltre, tutte le regole del programma sono safe. □

Semantica

Di seguito forniremo la semantica dei programmi ASP, la quale è basata sulla semantica degli answer set originariamente definita da Gelfon e Lifschitz in [19]. Tuttavia, a differenza di quella originale qui verranno considerati solo gli answer set consistenti, dato che ormai oggi è pratica diffusa considerare solo quelli.

Notiamo che in ASP si assume la disponibilità di alcuni predicati preinterpretati, come per esempio =, <, >. Nonostante ciò, è anche possibile definire questi predicati esplicitamente come dei fatti, cosicché essi non vengano trattati in un modo particolare.

Universo di Herbrand e Base di Letterali. Per ciascun programma P, l'*universo di Herbrand*, denotato da U_P, è formato dall'insieme di tutte le costanti che appaiono in P. Se non ci sono costanti in P allora U_P è formato da una costante arbitraria (in realtà, dato che il linguaggio non contiene simboli di funzione e dato che le regole devono essere safe, questa costante extra non è necessaria; tuttavia, in questo contesto abbiamo scelto di mantenere la definizione classica per evitare di creare confusione). La *base di letterali di Herbrand* B_P è data dall'insieme di tutti i letterali (classici) ground costruibili a partire dai simboli di predicato che appaiono in P e dalle costanti in U_P (si noti che, per ogni atomo p, B_P contiene anche il letterale strongly negated $\neg\, p$).

Esempio 2.1.2. Si consideri il seguente programma:
P_0 = {
r_1: a(X) $\vee$ b(X) $\leftarrow$ c(X,Y).
r_2: e(X) $\leftarrow$ c(X,Y), not b(X).
r_4: c(1,2).
}
allora l'universo è $U_{P0} = \{1,2\}$, mentre la base è $B_{P0} = \{$*a(1), a(2), b(1), b(2), e(1), e(2), c(1,1), c(1,2), c(2,1), c(2,2),* $\neg$ *a(1),* $\neg$ *a(2),* $\neg$ *b(1),* $\neg$ *b(2),* $\neg$ *e(1),* $\neg$ *e(2),* $\neg$ *c(1,1),* $\neg$ *c(1,2),* $\neg$ *c(2,1),* $\neg$ *c(2,2)*$\}$. □

Istanziazione Ground. Per ogni regola r, *Ground(r)* denota l'insieme delle regole ottenute rimpiazzando ciascuna variabile in r con le costanti in U_P in ogni modo possibile. Per ogni programma P, la sua istanziazione è data dall'insieme *Ground(P)* $= \cup_{r \in P}$ *Ground(r)*. Si noti che per programmi proposizionali vale la seguente affermazione: $P =$ *Ground(P)*.

Esempio 2.1.3. Si consideri nuovamente il problema P_0 dell'Esempio 2.1.2 La sua istanziazione ground è:
Ground(P_0) = {
}
Si noti che l'atomo *c(1,2)* era già ground in P_0, mentre le regole $g_1, ..., g_4$ (rispettivamente $g_5, ..., g_8$) sono ottenute rimpiazzando le variabili in r_1 (rispettivamente r_2) con le costanti in U_{P0}. □

g1: a(1) ∨ b(1) ← c(1,1).	g2: a(1) ∨ b(1) ← c(1,2).
g3: a(2) ∨ b(2) ← c(2,1).	g4: a(2) ∨ b(2) ← c(2,2).
g5: e(1) ← c(1,1), not b(1).	g6: e(1) ← c(1,2), not b(1).
g7: e(2) ← c(2,1), not b(2).	g8: e(2) ← c(2,2), not b(2).
g9: c(1,2).	

Answer Sets. Per ogni programma P, i suoi answer set sono definiti in due passi usando la sua istanziazione ground *Ground(P)*: prima sono definiti gli answer set dei programmi disgiuntivi positivi, poi quelli dei programmi generali sono definiti mediante una riduzione a programmi disgiuntivi positivi ed una condizione di stabilità.

Un'interpretazione I è un insieme consistente (dove un insieme $I \subseteq B_P$ è *consistente* se per ogni letterale classico positivo tale che $l \in I$ si può dire che $\neg\ l \in I$) di letterali ground classici $I \subseteq B_P$ rispetto ad un programma P. Una interpretazione consistente $X \subseteq B_P$ è detta *chiusa sotto* P (dove P è un programma datalog disgiuntivo positivo) se per ogni $r \in$ *Ground(P), H(r)* $\cap X \neq \emptyset$ ogni volta che $B(r) \subseteq X$. Una interpretazione chiusa sotto P è indicata anche come un *modello* di P. Un'interpretazione $X \subseteq B_P$ è un *answer set* per un programma disgiuntivo positivo P, se è minimale (rispetto all'inclusione degli insiemi) tra tutte le interpretazioni (consistenti) che sono chiuse sotto P.

Esempio 2.1.4. Il programma positivo $P_1 = \{a \vee \neg\ b \vee c.\}$ ha come answer set $\{a\}$, $\{\neg\ b\}$, e $\{c\}$; si noti che essi sono minimali e corrispondono ai possibili modi di soddisfare la disgiunzione. L'estensione $P_2 = P_1 \cup\ \{\leftarrow a.\}$ ha come answer set $\{\neg\ b\}$ e $\{c\}$, dato che, confrontando P_1 e P_2, il constraint aggiuntivo non è soddisfatto dall'interpretazione $\{a\}$. Inoltre, il programma positivo $P_3 = P_2 \cup\ \{\neg\ b \leftarrow c.; c \leftarrow \neg\ b.\}$ ha come unico answer set $\{\neg\ b,c\}$ (infatti, l'altra interpretazione consistente chiusa $\{a,\neg\ b,c\}$ non è minimale). Mentre è semplice verificare che $P_4 = P_3 \cup\ \{\leftarrow c\}$ non ha answer set. □

La *ridotta* o *trasformata di Gelfond-Lifschitz* di un programma ground P rispetto ad un set $X \subseteq B_P$ è un programma positivo ground P^X, ottenuto apportando a P le seguenti modifiche:

- cancella tutte le regole $r \in P$ per le quali $B^-(r) \cap X \neq \emptyset$;
- cancella il corpo negativo dalle restanti regole.

Un *answer set* di un programma P è un insieme $X \subseteq B_P$ tale che X è un answer set di *Ground(P)*^{X}.

Esempio 2.1.5. Per il programma ground negativo $P_5 = \{a \leftarrow not\ b\}$, $A = \{a\}$ è l'unico answer set, visto che $P_5^A = \{a.\}$. Per esempio per $B = \{b\}$, $P_5^B = \emptyset$, e così B non è un answer set. □

Esempio 2.1.6. Si consideri nuovamente P_0 dell'Esempio 2.1.2, la cui istanziazione ground *Ground*(P_0) è stata riportata nell'Esempio 2.1.3. Un modo naïve di calcolare gli answer set di P_0 è considerare tutte le possibili interpretazioni e poi verificare quale di esse è un answer set di *Ground*(P_0). Per esempio, si consideri l'interpretazione $I_0 = \{c(1,2),a(1),e(1)\}$, la ridotta *Ground*$(P_0)^{I0}$ contiene le regole g_1, g_2, g_3, g_4, g_9 più *e(1) ← c(1,1), e(1) ← c(1,2), e(2) ← c(2,1)* ed *e(2) ←*

c(2,2), ottenute cancellando i letterali negativi rispettivamente da g_5, g_6, g_7 e g_8. Possiamo, quindi, verificare che I_0 è un answer set per $Ground(P_0)^{I0}$ e, di conseguenza, anche un answer set per $Ground(P_0)$. La ridotta $Ground(P_0)^{I1}$ contiene le regole g_1, g_2, g_3, g_4, g_9 più entrambe *e(2)* $\leftarrow$ *c(2,1)* ed *e(2)* $\leftarrow$ *c(2,2)* (si noti che entrambe g_5 e g_6 vengono eliminate perché *b(1)* $\in I_1$). I_1 non è un answer set di $Ground(P_0)^{I1}$ perché {*c(1,2),b(1)*}$\subset I_1$. Di conseguenza I_1non è un answer set di P_0. Si può verificare che P_0 ha due answer set: I_0 e {*c(1,2),b(1)*}.
□

2.1.3 Rappresentazione della conoscenza e Ragionamento in ASP

La ASP è stata sfruttata in diversi domini che variano dalle classiche basi di dati deduttive all'intelligenza artificiale. La ASP può essere usata per codificare problemi in maniera del tutto dichiarativa; infatti, il potere delle regole disgiuntive consente di esprimere problemi che sono più complessi di NP, e la separazione (opzionale) di un programma non-ground fissato dalla base di dati in input consente di ottenere soluzioni uniformi al variare delle istanze.

Più nel dettaglio, molti problemi di complessità relativamente elevata possono essere risolti in maniera del tutto naturale seguendo il paradigma di programmazione "Guess&Check", originariamente introdotto in [42] e rifinito in [43]. L'idea che sta alla base di questo metodo può essere riepilogata come segue: una base di dati di fatti è usata per specificare un'istanza di un problema, mentre un insieme di regole (solitamente disgiuntive), chiamato "parte di guessing", è usato per definire lo spazio di ricerca; le soluzioni sono poi identificate nello spazio di ricerca da un altro insieme di regole (opzionali), chiamate "parte di checking", che impongono alcuni vincoli di ammissibilità. Fondamentalmente, gli answer set di un programma, che combina una base di dati in input con una parte di guessing, rappresentano le "soluzioni candidate"; quei candidati sono poi filtrati aggiungendo la parte di checking, la quale garantisce che gli answer set del programma risultante rappresentino precisamente le soluzioni ammissibili per l'istanza in input. Per afferrare l'intuizione che sta dietro il ruolo di entrambe le parti di guessing e checking, si consideri il seguente esempio.

Esempio 2.1.7. Si supponga di voler partizionare un insieme di persone in due gruppi, evitando che padri e figli appartengano allo stesso gruppo. Seguendo il paradigma Guess&Check, useremo una regola disgiuntiva per "guessare" tutti i possibili assegnamenti delle persone ai vari gruppi come segue:

group(P,1) $\vee$ *group(P,2)* $\leftarrow$ *person(P).*

Per capire cosa fa questa regola si consideri una semplice istanza del problema, dove ci sono due persone: *joe* e suo padre *john*. Questa istanza è rappresentata da tre fatti

person(john). person(joe). father(john,joe).

Si può verificare che gli answer set del programma risultante (fatti più regola disgiuntiva) corrispondono a tutti i possibili assegnamenti delle due persone ai due gruppi:

{*person(john), person(joe), father(john,joe), group(john,1), group(joe,1)*}
{*person(john), person(joe), father(john,joe), group(john,1), group(joe,2)*}
{*person(john), person(joe), father(john,joe), group(john,2), group(joe,1)*}
{*person(john), person(joe), father(john,joe), group(john,2), group(joe,2)*}

Tuttavia, si vogliono scartare quegli assegnamenti dove padre e figlio appartengono allo stesso gruppo. A questo scopo aggiungiamo la parte di checking esprimendo il seguente constraint:

$\leftarrow$ *group(P1,G),group(P2,G),father(P1,P2).*

Gli answer set del programma ottenuto sono quelli che ci aspettavamo, dove la parte di checking ha agito come una sorta di filtro:

{*person(john), person(joe), father(john,joe), group(john,1), group(joe,2)*}

{*person(john), person(joe), father(john,joe), group(john,2), group(joe,1)*} □

Di seguito illustreremo l'uso della ASP come un tool per la rappresentazione della conoscenza e per il ragionamento attraverso degli esempi. In particolare, prima tratteremo un problema che ha motivato le applicazioni di basi di dati deduttive classiche; poi sfrutteremo il paradigma di programmazione "Guess&Check" per mostrare come possono essere codificati in ASP un numero sostanziale di problemi famosi più difficili.

Raggiungibilità. Dato un grafo diretto finito $G = (V, A)$, vogliamo calcolare tutte le coppie di nodi $(a, b) \in V \times V$ tali che b è raggiungibile da a attraverso un una sequenza non vuota di archi in A. In altre parole, il problema equivale a calcolare la chiusura transitiva della relazione A.

Il grafo in input è codificato assumendo che A è rappresentato attraverso la relazione binaria *arc(X,Y)*, dove un fatto *arc(a,b)* dice che G contiene un arco da a a b, cioè $(a, b) \in A$; mentre l'insieme dei nodi V non è esplicitamente rappresentato visto che i nodi che appaiono nella chiusura transitiva sono implicitamente dati da questi fatti.

Il programma seguente, poi, definisce una relazione *reachable(X,Y)* che contiene tutti i fatti *reachable(a,b)* tali che b è raggiungibile da a attraverso degli archi del grafo in input G.

r_1: *reachable(X,Y)* $\leftarrow$ *arc(X,Y).*

r_2: *reachable(X,Y)* $\leftarrow$ *arc(X,U), reachable(U,Y).*

La prima regola asserisce che il nodo Y è raggiungibile dal nodo X se c'è un arco nel grafo da X a Y, mentre la seconda regola rappresenta la chiusura transitiva asserendo che il nodo Y è raggiungibile dal nodo X se esiste un nodo U tale che U è direttamente raggiungibile da X (c'è un arco da X a U) ed Y è raggiungibile da U.

Per esempio, si cosideri il grafo rappresentato dai seguenti fatti:

arc(1,2). arc(2,3). arc(3,4).

L'unico answer set del programma riportato sopra insieme a questi tre fatti è {*reachable(1,2), reachable(2,3), reachable(3,4), reachable(1,3), reachable(2,4), reachable(1,4), arc(1,2), arc(2,3), arc(3,4)*}

I primi tre letterali riportati sono dedotti attraverso la regola r_1, mentre gli altri letterali contenenti il predicato *reachable* sono dedotti usando la regola r_2.

Di seguito descriveremo l'utilizzo del paradigma di "Guess&Check".

Hamiltonian Path. Dato un grafo diretto finito $G = (V, A)$ ed un nodo $a \in V$ del grafo, esiste un path G che inizia in a e passa attraverso ogni nodo in V esattamente una volta

Questo è un classico problema NP-Completo definito nella teoria dei grafi. Si supponga che il grafo G sia rappresentato da fatti definiti sui predicati *node* (unario) ed *arc* (binario), ed il nodo iniziale a sia indicato mediante il

predicato *start* (unario). Allora, il programma seguente risolve il problema dell'*Hamiltonian Path*:

r_1*: inPath(X,Y) ∨ outPath(X,Y) ← arc(X,Y).*
r_2*: reached(X) ← start(X).*
r_3*: reached(X) ← reached(Y),inPath(Y,X).*
r_4 *:← inPath(X,Y), inPath(X,Y 1),Y <> Y 1.*
r_5 *:← inPath(X,Y), inPath(X1,Y),X <>X1.*
r_6 *:← node(X), not reached(X), not start(X).*

La regola disgiuntiva (r_1) genera un sottoinsieme S di archi, che sono quelli che faranno parte del path, mentre il resto del programma verifica se S costituisce un Hamiltonian Path. A tal proposito, viene definito un predicato ausiliario *reached*, il quale specifica l'insieme dei nodi che sono raggiunti a partire dal nodo iniziale. Questa parte somiglia molto al problema della raggiungibilità, ma qui la transitività è definita solo sul predicato guessato *inPath* mediante la regola r_3. Si noti che *reached* è determinato completamente dal guess per *inPath*, non è necessario alcun guess aggiuntivo.

Nella parte di checking, i primi due constraint (cioè, r_4 ed r_5) assicurano che l'insieme degli archi S selezionato da *inPath* soddisfi i requisiti che ogni Hamiltonian Path dovrebbe rispettare, cioè: *(i)* non ci devono essere due archi che partono dallo stesso nodo, e *(ii)* non ci devono essere due archi che terminano nello stesso nodo. Il terzo constraint impone che tutti i nodi del grafo siano raggiunti dal nodo iniziale nel sottografo indotto da S.

Consideriamo ora un programma alternativo P'_{hp}, che risolve sempre il problema dell'Hamiltonian Path, ma che intreccia la raggiungibilità con il guess:

r_1*: inPath(X,Y) ∨ outPath(X,Y) ← reached(X), arc(X,Y).*
r_2*: inPath(X,Y) ∨ outPath(X,Y) ← start(X), arc(X,Y).*
r_3*: reached(X) ← inPath(Y,X).*
r_4 *:← inPath(X,Y), inPath(X,Y 1),Y <> Y 1.*
r_5 *:← inPath(X,Y), inPath(X1,Y),X <>X1.*
r_6 *:← node(X), not reached(X), not start(X).*

Le due regole disgiuntive (r_1 ed r_2), insieme alla regola ausiliaria r_3, generano un sottoinsieme S di archi, che sono quelli che faranno parte del path, mentre il resto del programma verifica se S costituisce un Hamiltonian Path. In questo contesto, *reached* è definito in modo differente. Infatti, *inPath* è già definito in un modo che solo gli archi raggiungibili dal nodo iniziale saranno guessati. Il resto della parte di checking è uguale a quella di P_{hp}.

Numeri di Ramsey. Nell'esempio precedente, abbiamo visto come un problema di ricerca può essere codificato in un programma ASP i cui answer set coincidono con le soluzioni del problema. Ora costruiremo un programma i cui answer set certificano che una certa proprietà non è soddisfatta, cioè la proprietà è verificata se e solo se il programma non ha answer set. Successivamente applicheremo lo schema di programmazione di cui sopra al ben noto problema della teoria dei numeri e dei grafi.

Il numero di Ramsey $R(k,m)$ è il più piccolo intero n tale che, in qualunque modo noi coloriamo gli archi del grafo completo bidirezionale (clique) con n nodi usando due colori, il rosso ed il blu, c'è una clique rossa con k nodi (una k-clique rossa) oppure una clique blu con m nodi (una m-clique blu).

I numeri di Ramsey esistono per tutte le coppie di interi positivi k ed m [44]. Successivamente mostreremo un programma P_{ra} che ci consente di decidere se

un dato intero p non è il numero di Ramsey *R(3,4)*, come descritto qui sotto. Sia F_{ra} la collezione dei fatti per i predicati in input *node* ed *arc*, i quali codificano un grafo completo con n nodi. P_{ra} è il programma seguente:

r_1*: blue(X,Y)* $\vee$ *red(X,Y)* $\leftarrow$ *arc(X,Y).*

r_2 :$\leftarrow$ *red(X,Y), red(X,Z), red(Y,Z).*

r_3 :$\leftarrow$ *blue(X,Y), blue(X,Z), blue(Y,Z), blue(X,W), blue(Y,W), blue(Z,W).*

Intuitivamente, la regola disgiuntiva r_1 genera un colore per ciascun arco. Il primo constraint (r_2) elimina le colorazioni contenenti una clique rossa (cioè, un grafo completo) con 3 nodi, ed il secondo constraint (r_3) elimina le colorazioni contenenti una clique blu con 4 nodi. Il programma $P_{ra} \cup F_{ra}$ ha un answer set se e solo esiste una colorazione degli archi del grafo completo su n nodi che non contiene clique rosse di dimensione 3 e clique blu di dimensione pari a 4. Quindi, se c'è un answer set per un particolare n, allora n non è *R(3,4)*, cioè n $<$*R(3,4)*. D'altra parte, se $P_{ra} \cup F_{ra}$ non ha answer set allora $n \geq$ *R(3,4)*. Così, il numero di Ramsey *R(3,4)* sarà il più piccolo n tale per cui non viene trovato alcun answer set.

Strategic Companies. Negli esempi considerati finora la complessità dei problemi era limitata al massimo al primo livello della Gerarchia Polinomiale [45] (in NP oppure co-NP). Ora dimostreremo che possono essere codificati in ASP anche dei problemi più complessi, che appartengono al secondo livello della Gerarchia Polinomiale. A questo scopo, ora considereremo un problema di rappresentazione della conoscenza, ispirato ad una situazione comune al mondo degli affari, che è conosciuto sotto il nome *Strategic Companies* [46].

Si supponga di avere un insieme $C = \{c_1, ..., c_m\}$ di compagnie c_i controllate da una holding ed un set $G = \{g_1, ..., g_n\}$ di oggetti. Per ogni c_i si ha un set $G_i \subseteq G$ di oggetti prodotti da c_i ed un insieme $Oi \subseteq C$ di compagnie che controllano (possiedono) c_i. O_i è chiamato il controlling set di c_i. Questo controllo può essere visto come una maggioranza di azioni; le compagnie non in C, che non verranno modellate in questo contesto, potrebbero avere a loro volta delle quote in altre compagnie. Si noti che, in generale, una compagnia potrebbe avere più di un controlling set. La holding produce tutti gli oggetti in G tali che $G = \cup_{ci \in C} G_i$.

Un sottoinsieme di compagnie $C' \subseteq C$ è un insieme *production-preserving* se verifica le seguenti condizioni: *(i)* Le compagnie in C' producono tutti gli oggetti in G tali che $\cup_{ci \in C'} G_i = G$. *(ii)* Le compagnie in C' sono chiuse sotto la relazione di controllo, cioè se $O_i \subseteq C\prime$ per qualche i = *1,...,m* allora deve valere che $c_i \in C'$.

Un sottoinsieme minimale C', che è prodution-preserving, è chiamato *strategic set*. Una compagnia $c_i \in C$ è chiamata *strategic* se appartiene a qualche strategic set di C.

Questa nozione è rilevante quando le compagnie sono vendute. Infatti, intuitivamente, vendere qualche compagnia non-strategic non riduce il potere economico della holding. Calcolare le strategic companies è un problema la cui complessità appartiene al secondo livello della Gerarchia Polinomiale [46].

Nel seguito considereremo un'impostazione semplificata come quella presentata in [46], dove ogni oggetto è prodotto al più da due compagnie (per ogni $g \in G\ |\{c_i\ |g \in G_i\}| \leq$ *2*) e ciascuna compagnia può essere controllata contemporaneamente da al più altre tre compagnie, cioè $|O_i| \leq$ *3* per i = *1,...,m*. Si assuma che per una data istanza di Strategic Companies, F_{st} contenga i seguenti fatti:

- *company(c)* per ogni $c \in C$,
- *prod_by(g,*c_j, c_k*)*, se $\{c_i \mid g \in G_i\} = \{c_j, c_k\}$, dove c_j and c_k potrebbero anche coincidere,
- *contr_by(*c_i, c_k, c_m, c_n*)*, se $c_i \in C$ ed $Oi = \{c_k, c_m, c_n\}$, dove c_k, c_m, e c_n non sono necessariamente distinte.

Ora presenteremo un programma P_{st} che caratterizza questo problema difficile usando solo due regole:

r_1*: strat(Y)* $\vee$ *strat(Z)* $\leftarrow$ *prod_by(X,Y,Z).*

r_2*: strat(W)* $\leftarrow$ *contr_by(W,X,Y,Z), strat(X), strat(Y), strat(Z).*

In questo contesto *strat(X)* significa che una compagnia X è una strategic company. La parte di guessing del programma è formata dalla regola disgiuntiva r_1, mentre la parte di checking consiste della regola normale r_2. Il programma P_{st} è sorprendentemente succinto, soprattutto considerando che Strategic Companies è un problema difficile.

Il programma P_{st} sfrutta la minimizzazione inerente alla semantica degli answer set per verificare se un insieme di compagnie candidato C' che produce tutti gli oggetti e che preserva il controllo è anche minimale rispetto a questa proprietà.

La regola di guessing r_1 intuitivamente seleziona una tra le compagnie c_1 e c_2 che producono un certo oggetto g, che è descritto da *prod_by(g,*c_1, c_2*)*. Se non ci fossero informazioni di controllo, la minimalità degli answer set assicurerebbe naturalmente che gli answer set di $F_{st} \cup$ $\{$*r1*$\}$ corrispondano ai strategic set; non ci sarebbe bisogno di alcun controllo aggiuntivo. Tuttavia, nel caso in cui le informazioni di controllo siano disponibili, la regola r_2 verifica che non sia venduta alcuna compagnia controllata da altre compagnie nello strategic set, richiedendo semplicemente che questa compagnia debba essere strategica a sua volta. La minimalità degli strategic set è automaticamente assicurata dalla minimalità degli answer set.

Gli answer set di $F_{st} \cup P_{st}$ hanno una corrispondenza uno ad uno con gli answer set della holding descritta in F_{st}; una compagnia c è quindi strategica se e solo se *strat(c)* è in qualche answer set di $F_{st} \cup P_{st}$. E' importante notare che il "constraint" di checking r_2 interferisce con la regole di guessing r_1: applicare r_2 può "rovinare" l'answer set minimale generato da r_1. Per esempio, si supponga che la parte di guessing dia luogo alla regola ground

r_3*: strat(c1)* $\vee$ *strat(c2)* $\leftarrow$ *prod by(g,c1,c2)*

e che il fatto *prod_by(g,c1,c2)* sia dato in F_{st}. Ora si supponga che la regola sia soddisfatta dalla parte di guessing rendendo *strat(c1)* vero. Se, tuttavia, nella parte di checking viene applicata un'istanza della regola r_2, la quale deriva *strat(c2)*, allora l'applicazione della regola r_3 da cui deriva *strat(c1)* è invalidata, visto che la minimalità dell'answer set implica che la regola r_3 non possa giustificare la veridicità di *strat(c1)* se un altro atomo della testa è vero.

2.1.4 Estensioni al linguaggio

Il lavoro sulla ASP è partito dalle regole standard ma, relativamente presto, sono state sviluppate delle implementazioni che estendono il linguaggio base. Le più importanti estensioni possono essere raggruppate in due classi principali:

- *Costrutti di ottimizzazione*
- *Aggregati*

Costrutti di ottimizzazione. Il linguaggio ASP base può essere usato per risolvere problemi di ricerca complessi, ma, nativamente, non fornisce costrutti per la specifica di problemi di ottimizzazione (cioè problemi dove qualche funzione obiettivo deve essere minimizzata o massimizzata). Due estensioni della ASP sono state concepite per i problemi di ottimizzazione: *Weak Constraint* [43] e *Optimize Statement* [12].

Nel linguaggio base i constraint sono regole con la testa vuota, e rappresentano una condizione che *deve* essere soddisfatta, e per questa ragione sono chiamati anche *strong* constraint. Al contrario degli strong constraint, i *weak constraint* ci consentono di esprimere delle preferenze, cioè condizioni che *dovrebbero* essere soddisfatte. Quindi, essi possono anche essere violati. La loro semantica cerca di minimizzare il numero di istanze di weak constraint violate. In altre parole, la presenza degli strong constraint modifica la semantica di un programma scartando tutti i modelli che non soddisfano alcuni di essi; mentre i weak constraint identificano delle soluzioni approssimate, cioè quelle in cui sono soddisfatti più (weak) constraint possibili. Da un punto di vista sintattico, un weak constraint è come praticamente uno strong constraint dove il simbolo di implicazione $\leftarrow$ è sostituito da $<\sim$. Il significato informale di un weak constraint $<\sim B$ *è* "prova a falsificare B", oppure "B dovrebbe essere preferibilmente falso". In aggiunta, per un certo weak constraint potrebbero essere specificati, racchiusi tra parentesi quadre alla fine del constraint, un peso ed un livello di priorità (mediante interi o variabili). Se non specificati, si assume che il weak constraint abbia peso 1 e livello di priorità 1, rispettivamente.

In questo caso, siamo interessati agli answer set che minimizzano la somma dei pesi dei weak constraint violati (non soddisfatti) nel livello di priorità più elevato e, tra questi, quelli che minimizzano la somma dei pesi dei weak constraint violati nel livello di priorità inferiore, e così via. In altre parole, gli answer set sono considerati in ordine lessicografico secondo i livelli di priorità in base alla somma dei pesi dei weak constraint violati. Quindi, valori più alti per i pesi ed i livelli di priorità conferiscono ai weak constraint una importanza maggiore (per esempio i più importanti constraint sono quelli che hanno il peso più alto tra quelli con il livello di priorità più elevato).

Per esempio si consideri il Traveling Salesman Problem (TSP). Il problema TSP è una variante del problema dell'Hamiltonian Cycle considerato in precedenza, che cerca di trovare il ciclo Hamiltoniano più breve (di costo minimale) in un grafo diretto *pesato*. Questo problema può essere risolto adattando la codifica del problema del ciclo Hamiltoniano vista nel Paragrafo 2.1.3 alla gestione dei pesi sugli archi, aggiungendo un solo weak constraint.

Si supponga nuovamente che il grafo G sia specificato dai predicati *node* (unario) ed *arc* (ternario), e che il nodo iniziale venga evidenziato mediante il predicato *start* (unario).

Quindi, di seguito riportiamo il programma ASP con i weak constraint che risolve il problema TSP:

r_1: $inPath(X,Y) \vee outPath(X,Y) \leftarrow arc(X,Y).$
r_2: $reached(X) \leftarrow start(X).$
r_3: $reached(X) \leftarrow reached(Y), inPath(Y,X).$

r_4 :← *inPath(X,Y,_),inPath(X,Y 1,_),Y <>Y 1.*
r_5 :← *inPath(X,Y,_), inPath(X1,Y,_), X <>X1.*
r_6 :← *node(X), not reached(X).*
r_7*: <∼ inPath(X,Y,C) [C,1].*

L'ultimo weak constraint (r_7) dice che sarebbe preferibile evitare di prendere nel path archi i cui costi sono elevati. Tale constraint ha come effetto quello di selezionare quegli answer set per i quali il costo totale degli archi selezionati dal predicato inPath (tale costo totale coincide con la lunghezza del path) è il minore (cioè il path più corto).

La codifica del problema TSP riportata sopra è un esempio di pattern di programmazione "guess, check and optimize", che estende il "guess and check" originale (vedi Paragrafo 2.1.3) aggiungendo una "parte di ottimizzazione" che contiene principalmente dei weak constraint. Nell'esempio di cui sopra, la parte di ottimizzazione contiene solo il weak constraint r_7.

Gli *optimize statement* sono sintatticamente molto più semplici. Essi assegnano dei valori ad un insieme di letterali ground e, di conseguenza, selezionano quegli answer set per i quali la somma dei valori assegnati ai letterali veri (rispetto all'answer set stesso) è massimale o minimale. Non è difficile notare che i Weak Constraint riescono ad emulare gli Optmize Statement, ma non il contrario.

Aggregati. Ci sono alcune semplici proprietà, che ricorrono spesso nelle applicazioni del mondo reale, che non possono essere codificate in modo semplice e naturale in ASP. In special modo le proprietà che richiedono l'uso degli operatori aritmetici (come per esempio la somma, il conteggio o il massimo) su un insieme che soddisfa certe condizioni, se si vuole restare confinati al linguaggio ASP classico, richiedono delle codifiche piuttosto pesanti e complicate (che necessitano spesso di una relazione di ordinamento "esterno" sui termini).

Osservazioni simili sono state fatte anche in alcuni domini correlati, specialmente nei sistemi per le basi di dati. Tali osservazioni hanno portato alla definizione delle funzioni di aggregazione. In particolar modo nei sistemi per le basi di dati questo concetto è ormai pienamente integrato sia da un punto di vista pratico che teorico. Quando i sistemi per l'ASP hanno iniziato a diffondersi e ad essere utilizzati in applicazioni reali, è diventato subito chiaro che gli aggregati erano necessari anche qui. Per primi sono stati introdotti i constraint sulla cardinalità e sul peso [12], che sono casi speciali di aggregati. Tuttavia, in generale potrebbe essere utile avere anche altri aggregati (come il minimo, massimo o la media), e non è chiaro come generalizzare il framework dei constraint sul peso e sulla cardinalità in modo da consentire al programmatore di utilizzare degli aggregati arbitrari. Per colmare questa lacuna, l'ASP è stata estesa attraverso degli atomi speciali che gestiscono delle funzioni di aggregazione [47]. Intuitivamente, una funzione di aggregazione può essere vista come una funzione (possibilmente parziale) che mappa dei multiset di costanti su una costante.

Una *funzione di aggregazione* è della forma $f(S)$, dove S è un insieme di termini del tipo $\{$ *Vars : Conj* $\}$, dove *Vars* è una lista di variabili mentre *Conj* è una congiunzione di atomi standard, ed f è un simbolo di funzione di aggregazione.

Le funzioni di aggregazione più comuni calcolano il numero di termini, la somma di interi non-negativi, ed il minimo/massimo termine in un set.

Gli aggregati sono particolarmente utili quando devono essere trattati dei problemi del mondo reale. Si consideri il seguente esempio di applicazione: bisogna formare un team di progetto a partire da un insieme di impiegati secondo le seguenti specifiche:

1. Nel team ci devono essere almeno un certo numero di skill diversi.
2. La somma dei salari degli impiegati che lavorano nel team non deve superare il budget dato.

Si supponga che i nostri impiegati siano dati da un certo insieme di fatti del tipo *emp(EmpId,Skill,Salary)*; il numero minimo di skill diversi ed il budget sono specificati dai fatti nSkill(N) e budget(B). Codifichiamo ciascuna delle proprietà introdotte finora mediante un atomo aggregato e la imponiamo mediante un integrity constraint:

r_1: *in(I)* $\vee$ *out(I)* : $-$*emp(I,Sk,Sa).*

r_3 :$\leftarrow$ *nSkill(M),not #count*{*Sk : emp(I,Sk,Sa),in(I)*}*>= M.*

r_4 :$\leftarrow$ *budget(B),not #sum*{*Sa,I : emp(I,Sk,Sa),in(I)*}*<= B.*

Intuitivamente, la regola disgiuntiva “indovina” se un impiegato viene incluso nel team o meno, mentre i due constraint hanno una corrispondenza uno ad uno con i requisiti. Infatti, la funzione #count conta il numero di impiegati nel team, mentre #sum somma i salari degli impiegati che fanno parte del team. Si noti che grazie agli aggregati, la traduzione delle specifiche è molto più semplice e lineare.

Altre estensioni. Il linguaggio ASP è stato esteso anche in altre direzioni, col l'obiettivo di soddisfare i requisiti dei differenti domini applicativi; quindi, esiste un certo numero di linguaggi interessanti che fondano le proprie radici nella ASP. Per esempio, la ASP è stata usata per definire ed implementare alcuni *linguaggi per l'azione* (cioè linguaggi concepiti per trattare azioni e cambiamenti) come K [48] e $\mathcal{E}$ [49]. E' stato introdotto anche un linguaggio di logica, *ID-Logic* [50], per trattare la logica classica con definizioni induttive (che corrispondono semanticamente a regole logiche). Altre estensioni della ASP sono state concepite per trattare le *Ontologie* (cioè modelli astratti di un dominio complesso). In particolare, in [51] è stato proposto un linguaggio basato su ASP per la specifica di ontologie ed il ragionamento ontologico. Tale linguaggio estende ASP per poter gestire entità complesse del mondo reale come classi, oggetti, componenti di oggetti, assiomi e tassonomie. In [52] è stata introdotta anche una semantica basata sull'assunzione di mondo aperto per programmi ASP. Inoltre, in [53] è stata elaborata una estensione di ASP, chiamata *HEX-Programs*, che supporta atomi di ordine più elevato così come atomi esterni. Atomi esterni consentono di integrare sorgenti esterne di computazione in un programma logico. Così, gli HEX-Programs sono utili in diversi task, tra cui il meta-reasoning, il data type manipulation ed il ragionamento su ontologie rappresentate in Description Logics (DL) [54]. I *template predicate* sono stati introdotti in [55]. I template predicate sono predicati intensionali speciali definiti attraverso sottoprogrammi generici riutilizzabili, che sono stati concepiti per facilitare l'implementazione e per migliorare la leggibilità e la compattezza dei programmi. Infine, sono stati studiati anche dei programmi nidificati, che consentono la rappresentazione di espressioni logiche nidificate.

2.1.5 Applicazioni

L'Answer Set Programming è stata applicata con successo in molte aree, tra cui:

- *Information integration.* L'ASP è stata usata per supportare il consistent query answering nei sistemi di information integration usando il cosiddetto approccio Global-as-View [56, 57, 58], anche in presenza di inconsistenze e incompletezza sui dati.
- *Configuration and Verification management.* In product configuration [59], l'ASP è stata utilizzata come semantica dichiarativa che fornisce definizioni formali per concetti rilevanti nella configurazione di prodotti, che includono i modelli di configurazione, i requisiti e le configurazioni valide. Ed, in particolare, nel campo della configurazione del software, è stato implementato usando l'ASP un prototipo che configura l'intero sistema Debian Linux.
- *Knowledge Management.* L'ASP ha un enorme potenziale nell'impiego nell'area della gestione della conoscenza e delle tecnologie semantiche.
- Un sistema basato sull'ASP per la rappresentazione di ontologie ed il reasoning, chiamato OntoDLV [51], è impiegato in molte applicazioni del mondo reale che variano dall'e-learning alle ontologie enterprise alle applicazioni basate sugli agenti. In [60] è stato presentato un approccio basato sull'ASP al problema del riconoscimento e dell'estrazione di informazioni da documenti non strutturati. Mentre, in [61, 62] è stato introdotto un sistema per la classificazione dei contenuti, chiamato OLEX, che sfrutta l'ASP per estrarre metadati sui concetti e sulla semantica dai documenti.
- *Security engineering.* In [63] viene dimostrato come i protocolli di sicurezza possono essere specificati e verificati in maniera efficiente combinando il ragionamento sulle azioni nella programmazione logica. In particolare, sono stati modellati due casi di studio significativi nella verifica dei protocolli: il protocollo Needham-Schroeder public-key classico, ed il protocollo Aziz-Diffie key agreement per le comunicazioni mobili.

Inoltre, si possono ritrovare in letteratura altre applicazioni in varie aree, inclusa quella delle aste [64], dello scheduling [65], la policy description [66], workflow management [67], outlier detection [68], linguistica [69], sistemi multi agente [70, 71, 72], ed E-learning [72].

In conclusione, l'ASP è un tool molto interessante per la rappresentazione della conoscenza ed il ragionamento, e grazie all'applicabilità delle implementazioni degli ASP solvers ai problemi del mondo reale, si sta diffondendo anche in molte applicazioni industrialmente rilevanti.

Vale la pena notare che, i sistemi ASP al momento sono lontani dall'essere sfruttabili in maniera semplice per lo sviluppo di applicazioni a livello industriale; e, come ogni linguaggio di programmazione, l'ASP ha bisogno di tool e metodologie di sviluppo per facilitare e migliorare il processo di implementazione. Al momento, il campo dell'ingegneria del software per l'ASP è stato già fissato dalla comunità dell'ASP [73], ed è in continua evoluzione. Infatti, sia le metodologie (vedi Paragrafo 2.1.3) che i tool prototipi sono già disponibili [74, 55, 73, 75, 76, 51].

2.2 Hypertree Decomposition

2.2.1 Database e query acicliche

Un schema relazionale R consiste di un nome (nome della relazione) r e di una lista ordinata e finita di attributi. Ad ogni attributo A dello schema è associato un $Dom(A)$ numerabile di valori atomici. Un'istanza di relazione, o più semplicemente relazione, sullo schema $R = (A_1, \ldots, A_k)$ è un insieme finito del prodotto cartesiano $Dom(A_q) \times \cdots \times Dom(A_k)$. Gli elementi delle relazioni sono detti *tuple*. Un database schema DS consiste di un insieme finito di relazioni. Un'istanza di database, o più semplicemente *database*, **DB** su un database schema $DS = \{R_1, \ldots, R_m\}$ consiste delle relazioni $r_1, \ldots, r_m$ sugli schemi $R_1, \ldots, R_m$, rispettivamente, e di un universo finito:

$$U \subseteq \cup_{R_i(A_1^i, \cdots, A_{ki}^i) \in DS}(Dom(A_1^i) \cup \cdots Dom(A_{ki}^i))$$

tale che tutti i dati contenuti in **DB** provengano da U. Nel prosieguo, si utilizza la convenzione standard [17, 2] di identificare un'istanza di un database relazionale con una teoria logica di fatti ground. Una tupla $\langle a_1, \cdots, a_k \rangle$, appartenente ad una relazione r, è identificata con l'atomo ground $r(a_1, \cdots, a_k)$. il fatto che una certa tupla $\langle a_1, \cdots, a_k \rangle$ appartenga ad una relazione r di un database **DB** è denotato semplicemente come $r(a_1, \cdots, a_k) \in \mathbf{DB}$. Una query congiuntiva (rule-based) Q su un database schema $DS = \{R_1, \ldots, R_m\}$ consiste di una regola nella forma

$$Q: \quad ans \leftarrow r_1(u_1) \wedge \ldots r_n(u_n)$$

dove $n \geq 0$; $r_1, \ldots, r_n$ sono nomi di relazioni (non necessariamente distinti) di DS; e $\mathbf{u}_1, \ldots, \mathbf{u}_n$ sono liste di termini (cioè variabili o costanti) di lunghezza appropriata. L'insieme delle variabili presenti in Q è denotato da $var(Q)$, mentre l'insieme di atomi contenuti nel corpo di Q è indicato da $atoms(Q)$. Similmente, per ogni atomo $A \in atoms(Q)$, $var(A)$ indica l'insieme di variabili che compaiono in A; per ogni set di atomi $R \subseteq atoms(Q)$, si definisce $var(R) = \cup_{A \in R} var(A)$.

La risposta a Q definita su un database **DB** con universo associato U, consiste di una relazione *ans*, la cui arità è uguale alla lunghezza di $\mathbf{u}$, così definita. La relazione *ans* contiene tutte le tuple $\mathbf{u}\theta$ tale che θ: $var(Q) \rightarrow U$ è una sostituzione che sostituisce ogni variabile in $var(Q)$ con un valore di U tale che per $1 \leq i \leq n$, $r_i(\mathbf{u}_i)\ \theta \in \mathbf{DB}$. (Per un atomo A, $A\theta$ indica l'atomo ottenuto da A sostituendo uniformante $\theta(X)$ per ogni variabile X contenuta in A).

La query congiuntiva Q è una *Boolean conjunctive query* (BCQ) se l'atomo di testa $ans(\mathbf{u})$ non contiene variabili ed è semplicemente un atomo proposizionale. La valutazione della query Q è *true* se esiste una sostituzione θ tale che $1 \leq i \leq n$, $r_i(\mathbf{u}_i)\ \theta \in \mathbf{DB}$; altrimenti la valutazione su Q è false. Una query congiuntiva Q può essere rappresentata come un'interrogazione SELECT-PROJECT-JOIN definita nell'ambito dell'algebra relazionale, o da semplici query SQL del tipo:

$$SELECT\ R_{i1}.A_{j1}, \ldots, R_{ik}.A_{jk}\ FROM\ R_1, \cdots, R_n\ WHERE\ cond$$

dove *cond* è una congiunzione di condizioni nella forma $R_i.A = R_j.B$ o$R_i.A = c$, dove c è una costante.

Se Q è una query congiuntiva, si definisce l'ipergrafo $H(Q) = (V, E)$ associato a Q come segue. Il set di vertici V, indicato da $var(H(Q))$, consiste di tutte le variabili che compaiono in Q. Il set E, denotato da $edges(H(Q))$, contiene per ogni atomo $r_i(\mathbf{u}_i)$ nel corpo di Q un hyperedge che consiste di tutte le variabili che compaiono in $\mathbf{u}_i$. Si noti che la cardinalità di $edges(H(Q))$ può essere più piccola di quella di $atoms(Q)$, in quanto due atomi con esattamente le stesse variabili danno origine al medesimo hyperedge in $edges(H(Q))$. Una query Q è aciclica [77, 78] se e solo se il suo ipergrafo $H(Q)$ è aciclico o, equivalentemente, se se ha una join forest. Una join forest per l'ipergrafo $H(Q)$ è una foresta G il cui insieme di vertici V_G è l'insieme di $edges(H(Q))$ e ,per ogni paio di hyperedge $h_1$ed h_2 in V_G aventi variabili in comuni (cioè tali che $h_1 \cap h_2 \neq \emptyset$), risultano soddisfatte le seguenti condizioni:

- $h_1$ed h_2 appartengono alla stessa componente connessa di G
- tutte le variabili condivise da $h_1$ed h_2 compaiono in ogni vertice sul (unico) path da h_1 a h_2

Se G è un albero, allora è chiamato *join tree* per $H(Q)$; una query Q è aciclica se ha un join tree [77, 78].

Esempio 2.2.1. Si consideri la query congiuntiva Q_0 seguente:

$$ans \leftarrow r(Y, Z) \wedge g(X, Y) \wedge s(Y, Z, U) \wedge h(Z, U, W) \wedge t(Y, Z) \wedge w(Z, U)$$

□

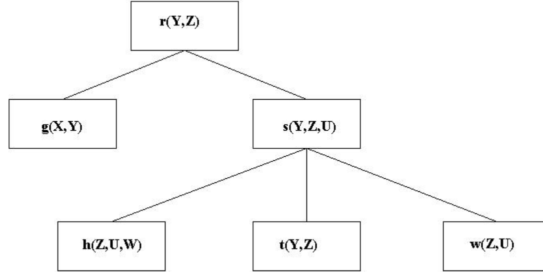


Figura 2.1: Join tree per l'ipergrafo $H(Q_0)$

La query è aciclica; il join tree associato all'ipergrafo$H(Q_0)$ è mostrato in Figura 2.2.1

Intuitivamente, la valutazione efficiente delle query acicliche è dovuta al fatto che possono essere valutate processando i relativi join tree utilizzando una strategia bottom up basata su upward semijoin, riducendo la dimensione delle relazioni intermedie (che potrebbero diventare esponenziali nel caso in cui venissero eseguite join regolari). Le query acicliche godono di interessanti proprietà computazionali, tra cui:

- Il problema BCQ di valutare una query booleana congiuntiva può essere risolto efficientemente se l'input è aciclico. Yannakakis ha proposto un algoritmo polinomiale (sequenziale) per la valutazione di query booleane acicliche. Inoltre, ha mostrato che è possibile rispondere a query congiuntive non booleane in tempo polinomiale nella dimensione combinata dell'istanza input e della relazione di output [79]

- E' stato mostrato che la risposta a query acicliche è altamente parallelizzabile, ed il problema è completo per la classe LOGCFL[80]. Efficienti algoritmi per la valutazione di query booleane e non sono stati proposti in [80] e [81]. Gli algoritmi in questione girano su macchine parallele che realizzano il cosiddetto inter-operation parallelism [82], cioè macchine che eseguono differenti operazioni relazionali in parallelo. Questi algoritmi possono essere implementati per rispondere efficientemente a query acicliche in contesti distribuiti.

- L'aciclicità è efficientemente riconoscibile, ed il calcolo di un join tree è efficientemente realizzabile: è possibile decidere se un ipergrafo è aciclico in tempo lineare [83] ed il problema appartiene alla classe L (deterministic logspace)

L'aciclicità è una proprietà chiave per la risoluzione polinomiale di problemi che in generale sono NP-hard come ad esempio BCQ[84] ed altri problemi equivalenti Conjunctive Query Containment [3, 85], Constraint satisfaction[86, 87].

2.2.2 Query Decomposition

In questa sezione illustriamo i concetti di query width e query decomposition. La definizione proposta è una modifica di quella originariamente introdotta da Chekuri e Rajarman in [3]; per ogni query congiuntiva Q, infatti, non è considerato l'atomo $head(Q)$, così come le costanti eventualmente presenti in Q.

Definizione 1. Una *query decomposition* per una query congiuntiva Q è la coppia $\langle T, \lambda \rangle$, dove $T = (N, E)$ è un albero, e λ è una funzione di etichettatura che associa ad ogni vertice $p \in N(p)$ un insieme $\lambda(p) \subseteq (atoms(\text{Q}) \cup var(\text{Q}))$, in modo che le seguenti condizioni risultino soddisfatte:

1. per ogni atomo A di Q, esiste un vertice $p \in N(p)$ tale che $A \in \lambda(p)$

2. per ogni atomo A di Q, l'insieme $\{p \in N | A \in \lambda(p)\}$ induce un subtree di T (connesso)

3. per ogni variabile $Y \in var(Q)$, l'insieme

   $$\{p \in N | Y \in \lambda(p)\} \cup \{p \in N | Y \text{ } e \text{ } contenuto \text{ } in \text{ } qualche \text{ } atomo \text{ } A \in \lambda(p)\}$$

   induce un subtree (connesso) di T

La *width* della query decomposition $\langle T, \lambda \rangle$ è definita come $max_{p \in N} |\lambda(p)|$. La *query-width* $qw(Q)$ di Q è definita come la minima width su tutte le sue query decomposition. Una query decomposition è *pura* se, per ogni vertice $p \in N(p)$, $\lambda(p) \subseteq (atoms(\text{Q}))$. Si osservi che la condizione 3 della definizione 1 è analoga

alla connectedness condition dei join tree e perciò è indicata, analogamente, come Connectedness Condition.

Esempio 2.2.2. Si consideri la seguente query Q_1:

$$ans \leftarrow s(Y, Z, U) \wedge g(X, Y) \wedge s(Z, U, W) \wedge t(Z, X) \wedge t(Y, Z)$$

Q_1 è una query ciclica, e la sua query width è pari a 2. Una 2-width decomposition per Q_1 è mostrata in Figura 2.2.2. Si noti che tale decomposizione è pura.

□

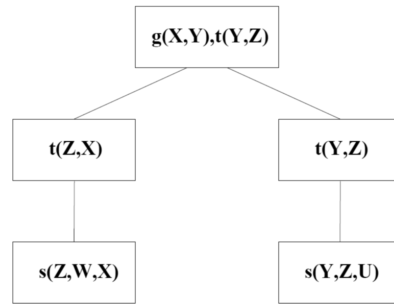


Figura 2.2: 2-width decomposition per la query Q_1

Proposizione 2.2.3. *Sia Q una query congiuntiva e $\langle T, \lambda \rangle$ una c-width query decompostion di Q.*
Risulta:

- *esiste una c-width decompositon $\langle T, \lambda' \rangle$di Q pura*
- *$\langle T, \lambda' \rangle$è calcolabile a partire da $\langle T, \lambda \rangle$ in logspace [88]*

Dalla proposizione 2.2.3 risulta che, per ogni query congiuntiva Q, $qw(Q) \leq k$ se e solo se Q ha una c-width decomposition pura, per qualche $c \leq k$. Le query con *k-bounded-width* sono quelle query la cui query-width è limitata da un fattore costante $k > 0$. La nozione di bounded query-width generalizza la nozione di aciclicità [79]. Risulta, infatti, che le query congiuntive acicliche sono esattamente le query congiuntive con query-width pari ad 1, perché ogni join tree è una query decomposition di width 1. Le query con width limitata condividono con le query acicliche importanti proprietà computazionali: il problema BCQ può essere efficientemente risolto su query con query-width limitata, se una k-width query decomposition della query è fornita come input addizionale. Chekuri e Rajarman hanno fornito un algoritmo polinomiale per questo problema [3]; mentre Gottlob et al.[6] hanno precisato che il problema è LOGCFL-completo. Sfortunamente, a differenza dell'aciclicità, non è conosciuto alcun metodo efficiente per il controllo di query width limitate. Infatti, risulta che decidere se una query congiuntiva ha una query decomposition con query-width limitata è NP-completo [89].

2.2.3 Hypertree Decomposition

Un *hypertree* per un ipergrafo Hè una tripla $\langle T, \chi, \lambda \rangle$, dove $T = (N, E)$ è un albero con radice fissata, ed χ e λ sono funzioni di etichettatura che associano ad ogni vertice $p \in N(p)$ i due insiemi $\chi(p) \subseteq var(H)$ e $\lambda(p) \subseteq edges(H)$. La *width* di un hypertree è definita come la massima cardinalità tra quelle degli insiemi $\lambda(p)$ associati, i.e., $max_{p \in N} |\lambda(p)|$. Si denotano con $vertices(T)$ e $root(T)$, l'insieme dei vertici N di T e la radice, rispettivamente, di T. Per ogni $p \in vertices(T)$, T_p indica il subtree di T con radice in p. Se T' è un subtree di T, si definisce $\chi(T') = \cup_{v \in vertices(T')} \chi(v)$

Definizione 2. Una *generalized hypertree decomposition* [90] di un ipergrafo H è un hypertree $HD = \langle T, \chi, \lambda \rangle$ per H che soddisfa le seguenti condizioni:

1. Per ogni edge $h \in edges(H)$, tutte le sue variabili compaiono assieme in qualche vertice del decomposition tree, ovvero, esiste un vertice $p \in vertices(T)$ tale che $h \subseteq \chi(p)$ (si dice che p *copre* h)

2. Per ogni variabile $Y \in var(H)$, l'insieme $\{p \in vertices(T) | Y \in \chi(p)\}$ induce un sottoalbero (connesso) di T (*Connectedness Condition*)

3. Per ogni vertice $p \in vertices(T)$, le variabili in χ dovrebbero appartenere agli iperarchi contenuti in λ, ovvero, $\chi(p) \subseteq \lambda(p)$

   Una *hypertree decomposition* è una generalized hypertree decomposition che soddisfa la seguente condizione aggiuntiva:

4. Per ogni vertice $p \in vertices(T)$, $var\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$(*Special Descendant Condition*)

La *hypertree-width* $hw(H)$ (rispettivamente *generalized hypertree-width* $ghw(H)$) di H è la width minima tra tutte le sue hypertree decomposition (rispettivamente generalized hypertree decomposition). Un edge $h \in edges(H)$ è fortemente coperto in *HD* se esiste un vertice $p \in vertices(T)$ tale che $var(h) \subseteq \chi(p)$ e $h \in \lambda(p)$. In questo caso si dice che p copre fortemente h. Una decomposizione *HD* di un ipergrafo H è una *complete decomposition* di H se ogni edge di H è fortemente coperto in *HD*.

Le nozioni di hypertree width e di generalized hypertree width sono generalizzazioni del concetto di aciclicità, in quanto gli ipergrafi aciclici sono esattamente quegli ipergrafi con hypertree-width e generalized hypertree-width pari ad uno. In particolare, le classi di query congiuntive aventi (generalized) hypertree width limitata hanno le stesse importanti proprietà computazionali delle query acicliche [6]. La definizione di generalized hypertree decomposition, in una prima analisi, sembra suggerire un semplice clustering degli hyperedge (cioè gli atomi della query) in modo che la classica connectedness condition del join tree risulti garantita. In realtà, una generalized hypertree decomposition può discostarsi da questo principio in due modi distinti:

- Un hyperedge già utilizzato in un cluster può essere riutilizzato in un altro
- Alcune variabili presenti in un hyperedge riutilizzato non devono soddisfare alcuna condizione

Al fine di chiarire meglio la nozione di generalized hypertree decomposition, si considerino le funzioni di etichettatura associate ad ogni vertice p dell'hypertree: l'insieme degli hyperedge $\lambda(p)$, e il set delle *effettive* variabili $\chi(p)$, che devono soddisfare la connectedness condition. Si noti che tutte le variabili che compaiono negli hyperedge contenuti in $\lambda(p)$ ma che non sono incluse in $\chi(p)$ sono in effetti ininfluenti, e non devono soddisfare la connectedness condition richiesta. Alla luce di ciò, la funzione χ gioca un ruolo determinante per determinare un riarrangiamento in formato join tree delle connessioni sulle variabili. Oltre alla connectedness condition, il riarrangiamento deve rispettare la fondamentale condizione 1 della definizione 2: ogni hyperedge (cioè ogni atomo della query) deve essere adeguatamente considerato nella decomposizione. Poiché le variabili rilevanti sono quelle contenute nelle etichette χ dei vertici della decomposizione, le etichette λ hanno il compito di coprire solo queste variabili (condizione 3) con il minor numero di hyperedge possibile. Infatti, come detto, la width della decomposizione è determinata dall' etichetta λ che contiene il maggior numero di nodi. Questo è il risultato più importante dell'approccio, e deriva dalle specifiche proprietà dei problemi basati su ipergrafi, in cui gli hyperedge giocano spesso un ruolo determinante.

Esempio 2.2.4. Si consideri la seguente query congiuntiva Q_2:

$$ans \leftarrow a(S,X,X',C,F) \wedge b(S,Y,Y',C',F') \wedge c(C,C',Z) \wedge d(X,Z) \wedge e(Y,Z)$$
$$\wedge f(F,F',Z') \wedge g(X',Z') \wedge h(Y',Z') \wedge j(J,X,Y,X',Y')$$

Sia H_2l'ipergrafo associato a Q_2. Essendo H_2 ciclico, si ha che $hw(H_2) > 1$.

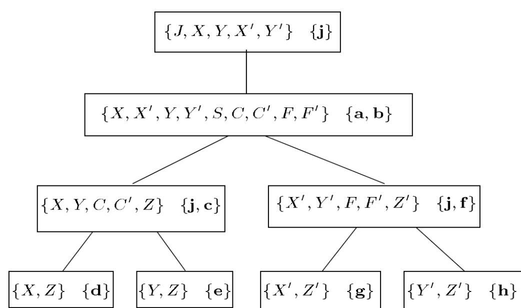


Figura 2.3: 2-width decomposition per l'ipergrafo H_2

In Figura 2.2.4 è mostrata una decomposizione (completa) HD_2 per l'ipergrafo H_2 con width pari a 2, e quindi $hw(H_2) = 2$, essendo esclusi i valori inferiori a 2 per gli ipergrafi aciclici.

□

In figura 2.2.3 è mostrata una rappresentazione alternativa della medesima decomposizione, chiamata *atom* (o *hyperedge*) representation [6]: ogni nodo p nell'albero è etichettato con l'insieme degli atomi in $\lambda(p)$; $\chi(p)$è l'insieme delle variabili, a meno delle variabili anonime "_", che appaiono in tali hyperedge.

In una simile rappresentazioni le occorrenze delle variabili anonime prendono il posto delle variabili in $var(\lambda(p)) - \chi(p)$.

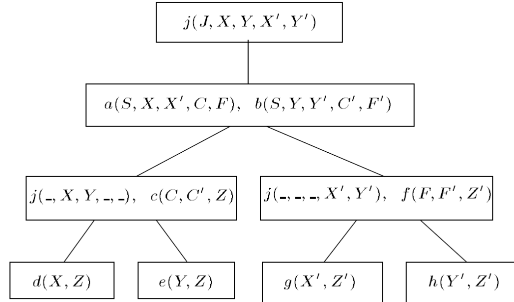


Figura 2.4: Atom representation dell'hypertree decomposition di Figura 2.2.4

Sia k un numero intero fissato. Un'istanza I del problema CQ ha una k-bounded (generalized) hypertree-width se risulta $(g)hw(H\ (I)) \leq\ k$. Una classe di query ha una bounded (generalized) hypertree-width se esiste un $k\ \geq$ 1 tale che tutte le istanze nella classe hanno k-bounded (generalized) hypertree-width. L'intrattabilità della nozione di generalized hypertree width è stata recentemente chiarita da [91], che ne hanno mostrato la natura NP-hard già per k= 3.

Per un qualsiasi $k\ \geq 1$, decidere se un dato ipergrafo ha una hypertree-width al più pari a k è, invece, in LOGCFL, e perciò, risulta un problema trattabile ed altamente parallelizzabile. Dualmente, il corrispondente problema di individuare una k-bounded hypertree decomposition è in L^{LOGCFL}[6]. La trattabilità del problema sembra legata alla Special Descendant Condition (condizione 4 della definizione 2).

Si consideri un vertice p di una hypertree decomposition e un hyperedge $h \in \lambda(p)$ tale che alcune variabili $\bar{X} \subseteq h$ compaiano nell'etichetta χ di qualche vertice nel subtree T_p con radice in p. In base a tale condizione, queste variabili devono necessariamente comparire anche in $\chi(p)$. Intuitivamente, questo si traduce nel fatto che bisogna considerare le variabili contenute in $\bar{X}$ a questo punto del decomposition tree, se si vuole inserire h in $\lambda(p)$. Per esempio, come conseguenza immediata di tale condizione, per la radice r di una qualsivoglia hypertree decomposition deve sempre risultare $\chi(r) = var\ (\lambda(r))$. Una volta che un hyperedge è stato coperto da qualche vertice del decomposition tree, ogni sottoinsieme delle sue variabili può essere utilizzato liberamente per decomporre i rimanenti cicli dell'ipergrafo. In sostanza, nelle generalized hypertree decomposition, è possibile scegliere come variabili rilevanti un qualsivoglia sottoinsieme delle variabili che compaiono in λ, senza alcuna limitazione; nelle hypertree decomposition, invece, è possibile scegliere le variabili rilevanti solo in modo da garantire che la descendant condition sia soddisfatta. Alla luce di ciò, la nozione di hypertree-width è più potente della (intrattabile) nozione di query-width, ma meno generale della nozione (probabilmente intrattabile) di generalized hypertree-width, che rappresenta la nozione più generale. Si consideri, a titolo d'esempio, la decomposizione mostrata in figura 2.2.3. Le variabili

appartenenti all'hyperedge corrispondente all'atomo j in H_2, sono incluse tutte insieme solo nella root della decomposizione, mentre è possibile osservare due differenti sottoinsiemi di questo hyperedge nel resto della decomposizione. Si osservi che la descendant condition è soddisfatta. Si consideri il vertice del livello 2, sulla sinistra: le variabili J,X' ed Y' non sono incluse nell'etichetta χ del vertice (sono sostituite dalla variabile anonima "_"), ma non devono più occorrere nel subtree con radice nel vertice. D'altro canto, se si fosse imposto di prendere tutte le variabili che compaiono negli atomi del decomposition tree, non sarebbe stato possibile individuare una decomposition di width 2. In realtà, j è l'unico atomo che può contenere le coppie X,Y ed X',Y', e non può più essere usato interamente, in quanto la variabile J non può occorrere al di sotto del vertice etichettato con a e b, altrimenti risulterebbe violata la connectedness condition (condizione 3 della definizione 2). La generalized hypertree-width per l'ipergrafo H_2 è pari a 2, esattamente come la hypertree width, ma in generale potrebbe essere anche minore. Un'interessante studio condotto da Alder et al. [92], mostra che la differenza tra le due nozioni di width è definita all'interno di un fattore costante: per un qualsiasi ipergrafo H, risulta $ghw(H) \leq hw(H) \leq 3ghw(H) + 1$. Segue immediatamente che una classe di ipergrafi ha una bounded generalized hypertree-width se e solo se ha una bounded hypertree-width, e pertanto le due nozioni identificano lo stesso insieme di classi trattabili.

2.2.4 Hypertree Decomposition Normal Form (NF)

E' stato osservato che, in accordo alla definizione 2, un ipergrafo può avere alcune hypertree decomposition indesiderate, con un numero elevato di vertici m nel decomposition tree. Per esempio, una decomposizione può contenere due vertici con esattamente le stesse etichette. Al fine di evitare queste ridondanze, è possibile definire una forma normale per le hypertree decomposition. Prima di introdurre la nuova definizione, è necessario fornire alcune importanti definizioni. Sia H un ipergrafo, e sia $V \subseteq var(H)$ un insieme di variabili X,$Y \in var(H)$. X è $[V]$-*adjacent* (adiacente) ad Y se esiste un arco $h \in edges(H)$ tale che $\{X, Y\} \subseteq (var(h) - V)$. Un $[V]$-path (cammino) π da X a Y consiste in una sequenza $X = X_0, \ldots, X_h = Y$ di variabili ed una sequenza di hyperedge $A_0, \ldots, A_{h-1}$ ($h \geq 0$) tali che X_i è $[V]$-adjacent con X_{i+1} e $\{X_i, X_{i+1}\} \subseteq var(A_i)$, per ogni $i \in [0 \ldots h-1]$. Un insieme $W \subseteq var(H)$ è $[V]$-*connected* (connesso) se $\forall X, Y \in W$esiste un $[V]$-path da X a Y. Una $[V]$-*component* (componente) è un insieme non vuoto massimale di variabili $W \subseteq (var(Q) - V)$ che sia $[V]-$*connected*. Sia $HD = \langle T, \chi, \lambda \rangle$ un hypertree di H e V$\subseteq$ *var* (H) un insieme di variabili. Si definisce $vertices(V, H) = \{p \in vertices(T) | \chi(p) \cap V \neq \emptyset\}$.

Definizione 3. Una hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ di una query congiuntiva Q è in *normal form* (*NF*) [93] se per ogni vertice $r \in vertices(T)$ e per ogni figlio s di r risultano soddisfatte le seguenti condizioni:

1. esiste esattamente una $[r]$-component C$_r$ tale che $X(T_s) = C_r \cup (\chi(s) \cap \chi(r))$

2. $\chi(s) \cap C_r \neq \emptyset$,dove C$_r$ è la $[r]$-component che soddisfa la condizione 1

3. $var(\lambda(s)) \cap \chi(r) \subseteq \chi(s)$

Le hypertree decomposition in normal form con width al più pari a k possono essere calcolate in tempo polinomiale nella dimensione di un dato ipergrafo H (ma esponenziale nel parametro k). Il numero di vertici m non può superare in H il numero delle variabili, e tipicamente risulta molto più piccolo. Inoltre, risulta che H ha una hypertree decomposition con width w se e solo se ha una normal form hypertree decomposition della stessa width w [88].

2.2.5 Query Decomposition e Query Plan

In questa sezione descriviamo l'idea base per utilizzare (generalized) hypertree decomposition per rispondere a query congiuntive. Sia $k \geq 1$ una costante fissata, Q una query congiuntiva definita su un database **DB**, e $HD = \langle T, \chi, \lambda \rangle$ una generalized hypertree decomposition di Q di width $w \leq k$. La procedura di valutazione si compone di due fasi:

1. per ogni vertice $p \in vertices(T)$, si calcoli la join tra le relazioni che appaiono insieme nell'etichetta $\lambda(p)$, e si proietti il risultato sulle variabili definite in $\chi(p)$. Alla fine di questa fase, la query Q', data dalla congiunzione di tutti i risultati intermedi, è una query congiuntiva aciclica, equivalente a Q. Il decomposition tree T associato a Q rappresenta un join tree per Q'.

2. si valuti Q', e quindi Q, utilizzando un qualsivoglia algoritmo per query acicliche, come ad esempio l'algoritmo di Yannakakis.

In Figura 2.2.5 è mostrato il join tree JT_2 ottenuto dopo la fase 1, a partire dalla query Q_2 dell'esempio 2.2.4, e dalla generalized hypertree decomposition di Figura 2.2.3.

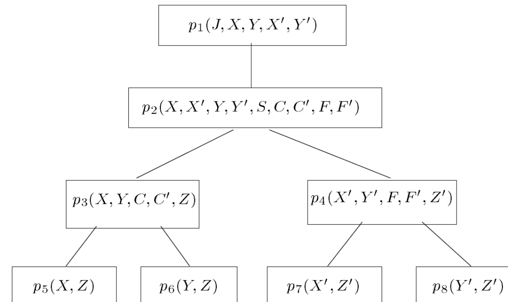


Figura 2.5: Join tree per la query Q'_2

Si consideri, ad esempio, la costruzione del vertice etichettato con p_3. Il vertice deriva dalla join degli atomi j e c (contenuti nel corrispondente vertice in Figura 2.2.3), e dalla seguente proiezione sulle variabili X,Y,C,C', e Z (appartenenti all'etichetta χ di quel vertice). Per costruzione, JT_2 soddisfa la connectedness condition. Perciò, la congiunzione degli atomi relativi ai vertici del tree è una query aciclica e JT_2 rappresenta una dei suoi join tree. La valutazione di tale query restituisce il medesimo risultato della query Q_2 [6].

Il passo 1 è realizzabile in tempo $O(m|r_{max}|^w)$, dove m rappresenta il numero di vertici di T, e r_{max} è la relazione del **DB** avente la dimensione più grande. Nel caso di query booleane, il costo della fase 2 dell'algoritmo non impiega più tempo di quello relativo alla fase 1, che pertanto risulta un upper bound per l'intero processo di valutazione della query. Per query non booleane, l'algoritmo di Yannakakis lavora in tempo polinomiale nella dimensione combinata dell'input e dell'output, e perciò è necessario aggiungere al costo un termine dipendente dalla risposta alla data query (che può essere esponenziale nella dimensione dell'input). Nel caso della query Q_2, ad esempio, l'upper bound è dato da $O(7|r_{max}|^2 log|r_{max}|)$, mentre il tipico tempo di valutazione per algoritmi di query answering che non indagano le proprietà strutturali è $O(|r_{max}|^7)$ Segue che, per ogni $k \geq 1$, si può rispondere alla classe di tutte le query aventi k-bounded hypertree width in tempo polinomiale, nella dimensione combinata dell'input e dell'output per query non booleane. Inoltre, data una query Q, sia la computazione di un hypertree decomposition *HD* di width al più k per $H(Q)$, che la successiva risposta all'interrogazione Q esaminando *HD* sono entrambi task polinomiali. Come precedentemente accennato, non è conosciuto alcun algoritmo polinomiale per individuare query con k-bounded generalized hypertree width; la ricerca è comunque particolarmente attiva per individuare euristiche efficaci per il calcolo di tali hypertree decomposition [86, 94, 91].

2.2.6 Weighted Hypertree Decomposition

Come descritto nella precedente sezione, data una query Q definita su un database DB ed una k-width decomposition *HD* per Q, con k piccolo, esiste un algoritmo polinomiale per rispondere a Q, mentre in generale il problema è NP-hard e tutti gli algoritmi disponibili richiedono, nel caso peggiore, tempo esponenziale. *HD*, in realtà, non è solo una rappresentazione teorica fine a stessa, ma rappresenta uno strumento concreto per valutare l'interrogazione. Infatti, i due passi necessari alla valutazione di Q rappresentano un vero e proprio piano di esecuzione, sebbene non completamente specificato. In particolare, non è scelto alcun tipo di algoritmo (merge, nested-loop, etc.) per la realizzazione fisica delle join previste dalla strategia di valutazione. Questa fase di ottimizzazione fisica, in realtà, può essere realizzata in maniera molto semplice, ricorrendo alle ben note tecniche di analisi della teoria dei database. Il vero problema consiste nell'individuare, prima dell'ottimizzazione fisica, una decomposizione di $H(Q)$. Infatti, in generale esiste un numero esponenziale di decomposizioni per un dato ipergrafo. Ognuna di queste corrisponde ad un opportuno modo di creare i cluster per gli atomi e di riarrangiarne e le connessioni. Fin tanto che è garantito un upper bound polinomiale, la ricerca potrebbe essere indirizzata solo all'individuazione di decomposizioni con width minima. Tuttavia, in applicazioni reali questa strategia potrebbe rivelarsi non completamente soddisfacente. In particolare, per query definite su un database **DB**, non si può prescindere dal considerare le informazioni contenute all'interno database stesso. Ad un'analisi prettamente strutturale è necessario, pertanto, aggiungere la valutazione di rilevanti dati quantitativi, come dimensione delle relazioni, selettività degli attributi, etc.

Decomposizioni minime. Formalmente, dato un ipergrafo H , una *hypertree weighting function* (*HWF*) ω_H è una qualsiasi funzione polinomiale che

associa ogni generalized hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ di H ad un numero reale, detto *weight* di *HD*. Un semplice esempio di HWF è dato dalla funzione ω_H^w $(HD) = \max_{p \in vertices(T)} |\lambda(p)|$, che pesa una decomposizione *HD* solo sulla base del vertice "peggiore", ovvero del vertice che presenta la più grande cardinalità dell'insieme λ associato, che determina anche la width della decomposizione. In molte applicazioni, è possibile utilizzare criteri di valutazione sostitutivi alla width minima, al fine di individuare hypertree decomposition significative. Si potrebbe pensare, ad esempio, di minimizzare il numero di vertici aventi la più grande width w e, per decomposizioni aventi il medesimo numero di tali vertici, minimizzare il numero di vertici aventi width pari a $w - 1$, e proseguire in tal modo. A tal fine, è possibile definire la HWF ω_H^{lex} $(HD) = \sum_{i=1}^{w} |\{p \in N \ tale \ che \ |\lambda(p)| = i\}| \times B^{i-1}$, dove N= *vertices*(T), B = $|edges(H) + 1|$, e w è la width di *HD*.

Sia $k \geq 0$ un numero fissato e H un ipergrafo. La classe kHD_H(resp., $kNFD_H$) è definita come l'insieme di tutte le hypertree decomposition (rispettivamente normal form hypertree decomposition) di H aventi width al più k.

Definizione 4. Sia H un ipergrafo, ω_H una weighting function,e C_H una classe di generalized hypertree decomposition di H. Allora, una decomposizione *HD* $\subseteq C_H$ è minima *rispettoa* ω_H e C_H , denotata con $[\omega_H, C_H]$-*minimal*, se non esiste un *HD'* $\in C_H$ tale che ω_H *(HD')* $< \omega_H$ *(HD)* [7]. Ad esempio, le $[\omega_H^w$, $kHD_H]$-*minimal* decomposition sono esattamente le k-bounded hypertree decomposition aventi la minima width possibile, mentre le $[\omega_H^{lex}$, $kHD_H]$-*minimal* decomposition sono solo un sottoinsieme di queste, corrispondente alle decomposizioni lessicograficamente minime, descritte in precedenza. Non è difficile mostrare che, per weighting function generali, la computazione di decomposizioni minime è un problema difficile, anche considerando solo k-bounded decomposition [7]. E' necessario restringere il campo d'azione a HWF più semplici. Sia $\langle R^+, \oplus, \min, \perp, +\infty \rangle$ un semi anello, in cui $\oplus$ è un operatore strettamente binario, commutativo, ed associativo, $\perp$ è l'elemento neutro per $\oplus$ (e.g., 0 per $+$, 1 per $\times$, etc.) e elemento assorbente per min, e min distribuito su $\oplus$. Data una funzione g ed un insieme di elementi $S = \{p_1, \cdots, p_n\}$, si denota con $\oplus_{p_i \in S} g(p_i)$ il valore di $g(p_1) \oplus \cdots \oplus g(p_n)$.

Definizione 5. Sia H un ipergrafo. Una *tree aggregation function* (TAF) [7] è una qualsivoglia weighting function della forma:

$$F_H^{\oplus,v,e} \ (HD) = \bigoplus_{p \in N} (\omega_H \ (p) \oplus \bigoplus_{(p,p') \in E} e_H \ (p,p'))$$

associando ad R^+ il valore dell'hypertree decomposition $HD = \langle (N, E), \chi, \lambda \rangle$, dove ω_H $:N \mapsto R^+$ ed e_H : $N \times N \mapsto R^+$ sono due funzioni polinomiali che valutano i vertici e gli hyperedge di un hypertree, rispettivamente. Data una query Q definita su un database **DB**, sia $HD = \langle T, \chi, \lambda \rangle$ una hypertree decomposition in normal form per $H(Q)$. Per ogni vertice p di T, sia $E(p)$ l'espressione relazionale $E(p) = \bowtie_{h \in \lambda(p)} \prod_{\chi(p)} rel(h)$, cioè il join di tutte le relazioni nel **DB** corrispondenti agli hyperedge contenuti in $\lambda(p)$, proiettato opportunamente sulle variabili di $\chi(p)$.

Dato un nodo incoming p' di p nella decomposizione *HD*, si definiscono le seguenti funzioni:

- $v^*_{H(Q)}(p)$ stima il costo di valutazione della funzione $E(p)$
- $e^*_{H(Q)}(p, p')$ stima il costo di valutazione della semi-join tra $E(p)$ ed $E(p')$

Sia $cost_{H(Q)}$ la TAF $F^{+,v^*,e^*}_{H(Q)}$, determinata dalle funzioni appena definite. Intuitivamente, $cost_{H(Q)}$ pesa le hypertree decomposition dell'ipergrafo $H(Q)$ in modo che le decomposizioni minime equivalgano a piani ottimali di esecuzione della query sul database **DB.** Si noti che può essere implementato per v^* ed e^* un qualsivoglia metodo per calcolare le stime della valutazione delle operazioni algebriche a partire dalle informazioni quantitative contenute nel **DB** (e.g, cardinalità delle relazioni, selettività degli attributi, etc.) Ovviamente, tutte queste potenziali weigthing function, non potrebbero essere utilizzate senza un adeguato algoritmo polinomiale per il calcolo delle hypertree decomposition minime. Nell'affrontare il calcolo di tali decomposizione, svolge un ruolo determinate il tipo di decomposizione alla quale si è interessati. Risulta, infatti, che il calcolo delle decomposizioni minime richieste, considerando una qualsiasi tree aggregation function, è un problema trattabile nel momento in cui si lavora con decomposizioni in forma normale, mentre è stato provato che il problema è ancora NP-hard considerando l'intera classe di hypertree decomposition con width limitata. Un algoritmo polinomiale per il problema, minimal$-k-$decomp, è illustrato in Figura 2.2.6.

L'algoritmo minimal-k-decomp [95] memorizza un grafo bipartito pesato CG, chiamato *Candidates Graph*, che mantiene tutte le informazioni necessarie al calcolo della decomposizione richiesta. I nodi del grafo sono suddivisi in due insiemi N_{sub} ed N_{sol}, che rappresentano i sottoproblemi da risolvere e le loro soluzioni, rispettivamente. I nodi in N_{sub} hanno la forma (R, C), dove R è un insieme di al più k edge di H, chiamato k-vertice, e C è una $[R]$-*component.* Inoltre, esiste un nodo speciale $(\emptyset, var(H))$ che rappresenta l'intero problema. I nodi in N_{sol} hanno la forma (S, C'), dove S è un k-vertice, C' è una componente da decomporre, $var(S) \cap C' \neq \emptyset$, e, $\forall h \in S$, $h \cap var(edges(C')) \neq \emptyset$. Intuitivamente, questo nodo potrebbe essere la radice di una hypertree decomposition per il sub-hypergraph indotto da $var(edges(C'))$. Il nodo (S, C') ha un arco che punta a tutti i nodi della forma $(R', C') \in N_{sub}$ per i quali rappresenta una soluzione candidata, ovvero, per il quale risulta $var(edges(C')) \cap var(R') \subseteq var(S)$. Inoltre, il nodo ha un certo numero di archi entranti provenienti da nodi della forma $(S, C'') \in N_{sub}$, per ogni $[S]$-*component* C'' che è inclusa in C', poiché ognuno di tali nodi rappresenta un sottoproblema di (S, C') (o, più precisamente, di ogni $(R', C') \in N_{sub}$ al quale il nodo (S, C') è connesso). Per ogni nodo $p \in N_{sol}$ è inizialmente settato $weight(p) := v_H(p)$. Successivamente, poiché $\oplus$ è associativo, commutativo, e chiuso, è possibile aggiornare questo peso settando $weight(p) := weight(p) \oplus \; e_H(p,p')$, non appena si conosce un qualsiasi nodo p' discendente di p nell'albero di decomposizione. Tali discendenti sono ottenuti filtrando opportunamente i nodi che sono connessi con i suoi incoming node in N_{sub}, corrispondenti ai suoi sottoproblemi. Se un nodo $q \in N_{sub}$ non ha soluzioni candidate,cioè se $incoming(q) = \emptyset$, allora non è risolvibile. Tale informazione può essere immediatamente utilizzata per rimuovere tutti i nodi uscenti dal nodo in questione, per i quali esso era un sottoproblema. D'altra parte, se

```
Input: A hypergraph ℋ, a tree aggregation function F_ℋ^{⊕,v,e}.
Output: An [F_ℋ^{⊕,v,e}, kNFD_ℋ]-minimal hypertree decomposition of ℋ, if any; otherwise, failure.
Var  CG = (N_sol ∪ N_sub, A, weight): weighted directed bipartite graph;
     HD = ⟨(N_sol, E), χ, λ⟩: hypertree of ℋ;
Begin
  (* Build the Candidates Graph *)
  N_sub := {(∅, var(ℋ))} ∪ {(R, C) | R is a k-vertex and C is an [var(R)]-component};
  N_sol := {(S, C) | S is a k-vertex, C is any [var(R)]-component (for some R), var(S) ∩ C ≠ ∅ and,
              ∀h ∈ S, h ∩ var(edges(C)) ≠ ∅};
  A := ∅;
  For each (R, C) ∈ N_sub Do
     For each (S, C) ∈ N_sol such that var(edges(C)) ∩ var(R) ⊆ var(S) Do
        Add an arc from (S, C) to (R, C) in A;
        For each (S, C') ∈ N_sub s.t. C' ⊆ C Do (* Connect its subproblems *)
           Add an arc from (S, C') to (S, C) in A;
  (* Evaluate the Candidates Graph *)
  For each p = (S, C) ∈ N_sol Do
     λ(p) := S; χ(p) := var(edges(C)) ∩ var(S), and weight(p) := v_ℋ(p);
  weighted := {p ∈ N_sol | incoming(p) = ∅};
  toBeProcessed := N_sub;
  While toBeProcessed ≠ ∅ Do
     Extract a node q from toBeProcessed such that incoming(q) ⊆ weighted;
     If incoming(q) = ∅ Then (* no way to solve subproblem q *)
        Remove all p' ∈ outcoming(q);
     Else (* success, for this subproblem *)
        For each p' ∈ outcoming(q) Do
           weight(p') := weight(p') ⊕ min_{p∈incoming(q)}(weight(p) ⊕ e_ℋ(p', p));
           If incoming(p') ∩ toBeProcessed = ∅ Then
              Add p' to weighted;
  EndWhile (* toBeProcessed *)
  (* Identify a minimal weighted hypertree decomposition (if any) *)
  If incoming((∅, var(ℋ))) = ∅ Then
     Output failure;
  Else
     E := ∅; (* the tree of the decomposition has no edges, initially *)
     Choose a minimum-weighted p ∈ incoming((∅, var(ℋ))); (* select *)
     Select-hypertree(p);
     Remove all isolated vertices in the tree (N_sol, E);
     Output HD;
End.
-----------------------------------------------------------------------
Procedure Select-hypertree(p ∈ N_sol)
  For each q ∈ incoming(p) Do
     Choose a minimum-weighted p' ∈ incoming(q); (* select *)
     Add the edge {p, p'} to E;
     Select-hypertree(p');
EndProcedure;
```

Figura 2.6: Algoritmo minimal-k-decomp

il nodo ha qualche candidato, quando tutti questi sono stati completamente valutati, può propagare questa informazione a tutti i suoi nodi outcoming. A tal punto, poiché min è distribuito su $\oplus$, è possibile selezionare come sua soluzione il suo incoming node a peso minimo in N_{sol}. Nel momento in cui tutti i nodi sono stati processati, le informazioni memorizzate nel grafo pesato CG sono sufficienti per calcolare ogni hypertree decomposition minima di H in normal form con width al più k, se ne esiste qualcuna. Uno di questi hypertree è infine selezionato dalla routine *Select-hypertree*. Ulteriori informazioni sull'algoritmo sono presenti in [7]. L'algoritmo è stato concretamente implementato nel prototipo cost$-k-$decomp, disponibile presso [92], che calcola una decomposizione minima rispetto alla funzione $cost_{H(Q)}$ ed alla classe delle k-bounded hypertree decomposition in forma normale.

2.2.7 Un modello di costo

Il costo reale del processamento di una decomposizione non si può, in generale, misurare esattamente senza eseguire tutte le operazioni necessarie. L'utilizzo di modelli per effettuare delle stime è dunque un passo obbligatorio per stabilire un confronto quantitativo tra i diversi piani di valutazione.

I modelli di costo nei DBMS commerciali considerano; indici, dimensioni dei blocchi, dimensioni della cache, tempo di CPU, tempo di spostamento della testina su disco, etc.

Prendere in considerazione tutti questi parametri è fisici è sicuramente al di fuori degli scopi di questo lavoro e complicherebbe soltanto la descrizione degli aspetti più rilevanti del lavoro. Si è deciso pertanto di utilizzare un modello di costo più semplice, basato sul numero di tuple delle relazioni in input e delle relazioni intermedie generate durante il processamento. Sebbene non ideale, una tale stima per il costo è ragionevole ed ampiamente utilizzata in letteratura. E' stato inoltre osservato che, se la valutazione di un piano è n secondo tale modello di costo, allora il costo del miglior metodo possibile sarà limitato superiormente da $O(n \cdot log\ n)$ e non potrà sicuramente essere meno di $\Omega(n)$.

Intuitivamente, l'uso dell'euristica che seleziona i piani che minimizzano la somma delle tuple delle relazioni coinvolte è giustificato dalla dipendenza lineare che sussiste tra la dimensione fisica e la cardinalità di una relazione. La riduzione della dimensione fisica si traduce in una diminuzione delle operazioni di I/O necessarie a leggere/scrivere le relazioni sul disco. E' facile capire che la diminuzione degli accessi al disco implica un miglioramento naturale nel tempo d'esecuzione degli algoritmi che eseguono il piano di valutazione fisico determinato dall'ottimizzatore.

Infatti, se si riesce a ridurre significativamente il numero complessivo delle tuple coinvolte si può dunque influenzare significativamente il costo effettivo del piano.

Alla stregua di tali considerazioni, è possibile definire brevemente il costo associato ad una decomposizione $HD = \langle T, \chi, \lambda \rangle$ per un'interrogazione Q come la somma delle cardinalità delle relazioni di base, associate ai vertici di T, e delle relazioni intermedie generate per valutare Q.

Si noti che le operazioni algebriche utilizzate nei piani di valutazione a cui si è interessati sono join multiple (associate ai vertici delle decomposizioni) e semijoin multiple (associate agli archi tra padre e figli). Si esprimerà un particolare ordinamento nell'esecuzione di tali operazioni attraverso una espressione di join definita come segue.

Espressione di join e costo associato. Una espressione di join è un'espressione dell'algebra relazionale in cui occorrono semplicemente: relazioni come operandi, operatori di join ($\bowtie$) come operatori binari, e parentesi. Una espressione di join indicata con $E(\bowtie R)$ specifica un piano di valutazione parziale per la join multipla $\bowtie R$.

Il costo associato ad un'espressione di join E, denotato con con *cost(E)*, è definito come la somma delle cardinalità delle relazioni intermedie prodotte dagli operatori binari di join. Si noti che non abbiamo considerato in *cost(E)* le relazioni di input (gli operandi) di E, che saranno comunque tenute in conto diversamente nelle definizioni che seguiranno.

Esempio 2.2.5. Si vuole valutare la join multipla di quattro relazioni i cui schemi sono:

$$R_1(A, B), R_2(B, C), R_3(C, D), R_4(D, A)$$

Una possible espressione di join che suggerisce come effettuare tale valutazione è:

$$E = (R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$$

Quest'espressione relazionale indica che devono essere valutate inizialmente ed indipendentemente:

$$(R_1 \bowtie R_2) e (R_3 \bowtie R_4)$$

Ed infine utilizzare il risultato intermedio delle due per valutare la join complessiva. Se si denota con |R| la cardinalità della relazione R, il costo di tale ordinamento è espresso da:

$$cost(E) = |R_1 \bowtie R_2| + |R_3 \bowtie R_4| + |(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)|$$

cioè è costituito, secondo quanto detto, dalle cardinalità delle sole relazioni generate. □

Il metodo prevalentemente usato per l'identificazione di un ordinamento ottimale secondo un certo modello di costo è basato sulla programmazione dinamica. Questo è in sintesi un algoritmo che enumera tutte le possibili espressioni (piani) di join in modo tale da poterne confrontare i rispettivi costi. Se si considerano join tra più di 5 o 6 relazioni, l'uso di questo metodo, però, diventa proibitivo. Si ricorre dunque ad altre strategie che riducendo lo spazio di ricerca non garantiscono l'ottimalità della soluzione, ma con buona probabilità possono guidare ad un risultato vicino all'essere ottimale. Recenti algoritmi di enumerazione si basano anche su tecniche ibride dette di programmazione dinamica iterativa. L'idea è quella di applicare diverse volte la programmazione dinamica per trattare separatamente diverse parti del processo di ottimizzazione.

Criterio della selettività della join. Per risolvere il compromesso tra complessità dell'algoritmo di ordinamento da utilizzare e qualità della soluzione da ottenere, è stato stabilito di impiegare un'euristica molto ben conosciuta e sfruttata: il criterio della selettività della join. Come in tutte le euristiche, lo svantaggio principale è rappresentato da una riduzione dello spazio di ricerca che non preserva lo spazio delle soluzioni ottime. In particolare, per il criterio della selettività, il sotto-spazio di ricerca consiste in tutte quelle espressioni di join che prevedono la generazione di una sola relazione intermedia per volta, quindi sono impiegabili efficacemente sia il pipelining che gli eventuali percorsi d'accesso definiti per le relazioni in input. Infatti lo spazio delle cosiddette espressioni left-deep è particolare perché in ogni soluzione viene coinvolta una nuova relazione di input per ogni operazione intermedia.

Questo criterio, utilizzato nell'algoritmo cost-k-decomp, ha l'obiettivo consiste nel minimizzare, per ogni relazione intermedia da generare, un'indice di selettività della join.

L'indice di selettività della join, per una relazione R_i rispetto ad una data relazione R_j, è definito come:

$$\sigma_i \triangleq \frac{|R_i \bowtie R_j|}{R_j}$$

σ_i è effettivamente una stima dell'incremento di cardinalità dovuto alla join.

Per applicare l'euristica della selettività della join è, quindi, necessario stimare la cardinalità dell'output per ognuna delle operazioni di join intermedie.

Saranno necessarie anche le stime per la semijoin e la proiezione, per cui, a seguire, diamo un elenco di tutte le formule sfruttate (ed implementate in apposite funzioni di *cost-k-decomp*) per stimare queste operazioni.

Se R è una relazione, si indica con $|R|$ la sua cardinalità e con $V(A, R)$ la selettività del suo attributo A. Nella figura 2.2.7 sono presentate le formule necessarie a stimare la cardinalità e la selettività[1] degli attributi delle operazioni algebriche considerate a partire dalla conoscenza di tali informazioni sulle relazioni in input.

E' importante premettere, comunque, che queste stime si fondano sulle seguenti assunzioni di semplificazione:

Inclusione degli insiemi di valori. Se $V(A, R) \leq V(A, S)$ allora ogni possibile valore per l'attributo A in R è anche un possibile valore per l'attributo A in S.

Conservazione degli insiemi di valori. Se $A \in$*attr(S)* non è un attributo di join, cioè , $A \notin$*attr(R)*$\cap$*attr(S)*, allora $V(A, R \bowtie S) = V(A, S)$. In altre parole, gli attributi che non intervengono nella join non mutano la loro selettività.

$\left\|R \bowtie S\right\| = \frac{\|R\| \cdot \|S\|}{\prod_{A \in attr(R) \cap attr(S)} \max\{V(A,R), V(A,S)\}}$ $V(A, R \bowtie S) = \begin{cases} \min\{V(A,R), V(A,S)\}, A \in attr(R) \cap attr(S) \\ V(A,R), A \in attr(R) - attr(S) \\ V(A,S), A \in attr(S) - attr(R) \end{cases}$	**JOIN**
$\left\|R \ltimes S\right\| = \min\left\{\|R\|, \frac{\|R\| \cdot \|S\|}{\prod_{A \in attr(R) \cap attr(S)} \max\{V(A,R), V(A,S)\}}\right\}$ $V(A, R \ltimes S) = \begin{cases} \min\{V(A,R), V(A,S)\}, A \in attr(R) \cap attr(S) \\ V(A,R), A \in attr(R) - attr(S) \end{cases}$	**SEMIJOIN**
$\left\|\pi_{A_{i_1},\ldots,A_{i_k}}(R)\right\| = \min\left\{\frac{1}{2}\|R\|, V(A_{i_1}, R) \cdot \ldots \cdot V(A_{i_k}, R)\right\}$ $V(A_{i_j}, \pi_{A_{i_1},\ldots,A_{i_k}}(R)) = V(A_{i_j}, R)$	**PROIEZIONE**

Figura 2.7: Stime utilizzate nel modello di costo per le operazioni

Considerate due relazioni R ed S di cui si conosce cardinalità e selettività degli attri-buti, seguono le espressioni impiegate nel modello di costo adottato per stimare tali informazioni per le relazioni:

$$R \bowtie S, R \ltimes S \text{ e } \pi_{A_{i1},\ldots,A_{ik}}(R)$$

Costo di una hypertree decomposition. E' possibile, a questo punto, definire un modello di costo per le hypertree decomposition. Nonostante la sua

[1] Numero dei valori distinti di un attributo

semplicità, come precedentemente osservato nelle considerazioni iniziali, la somma delle cardinalità delle relazioni intermedie costituisce una buona misura del costo effettivo di un piano di valutazione.

In questa sezione vengono introdotte una serie di definizioni che consentono di stabilire inizialmente il costo associato ad un vertice, poi quello associato ad un sottoalbero ed infine quello di una hypertree decomposition complessiva, cioè del piano di valutazione ad essa associato secondo il modello di costo proposto. Si nota che ogni definizione è effettivamente data in una forma più generale che prescinde dalla particolare tecnica utilizzata per calcolare l'ordinamento delle join multiple associate ai vertici. Queste rappresentano in generale l'aliquota più onerosa del costo complessivo di una decomposizione. In aggiunta, ogni definizione è poi specializzata al caso in cui si utilizza l'euristica per il piano di valutazione delle join.

Definizione 6. (Costo associato ad un vertice)

Sia $HD = \langle T, \chi, \lambda \rangle$ una hypertree decomposition per un'interrogazione congiuntiva di una base di dati **DB** della quale si hanno informazioni quantitative sulla cardinalità e selettività degli attributi delle relazioni.

Inoltre, dato $p \in$ *vertices(T)* si consideri la seguente relazione ad esso associata:

$$J_p \triangleq \underset{A \in \lambda(p)}{\bowtie} \pi_{\chi(p)}(R_A)$$

Il costo del vertice è definito attraverso la somma di tre aliquote:

$$cost\,(p) \triangleq \sum_{A \in \lambda(p)} |R_A| + \sum_{A \in \lambda(p): var(A) - \chi(p) = \varnothing} \left|\pi_{\chi(p)}(R_A)\right| + cost(E\,(J_p))$$

dove $cost(E(J_p))$ rappresenta il costo della espressione di join $E(J_p)$ utilizzata per valutare J_p. Le relazioni in input per $E(J_p)$ appartengono a $\left\{\pi_{\chi(p)}\,(R_A) : A \in \lambda(p)\right\}$.

Questa definizione stabilisce che il costo associato ad un vertice è dato innanzitutto dalle relazioni[2] associate agli atomi in $\lambda(p)$, poi subisce un incremento dovuto alle sole relazioni che non hanno gli attributi contenuti tutti in $\chi(p)$, quindi necessitano di essere proiettate, ed infine si aggiunge l'aliquota più critica che deriva dal particolare ordinamento scelto per effettuare le operazioni di join tra le relazioni di $\lambda(p)$.

Definizione 7. (Costo associato ad un sottoalbero)

Sia $HD = \langle T, \chi, \lambda \rangle$ una hypertree decomposition per un'interrogazione congiuntiva di una base di dati **DB** della quale si hanno informazioni quantitative sulla cardinalità e selettività degli attributi delle relazioni.

Inoltre, dati $p \in$ *vertices(T)* e T_p sottoalbero di T radicato in p, si consideri la seguente relazione associata a p:

$$S_p \triangleq \begin{cases} J_p & \forall p \in leaves(T_p) \\ J_p \underset{c \in children(p)}{\ltimes} \pi_{\chi(p)}(S_c) & altrimenti \end{cases}$$

[2]Si sottintende sempre la cardinalità

Il costo di T_p è definito attraverso la somma di quattro aliquote:

$$cost\,(T_p) \triangleq cost\,(p) + \sum_{c \in children(p)} cost(T_c) +$$
$$+ \sum_{c \in children(p) : \chi(c) = \chi(p) \neq \varnothing} \left|\pi_{\chi(p)}(S_c)\right| + cost(E\,(S_p))$$

dove *cost(E(*S_p*))* rappresenta il costo della espressione di join $E(S_p)$ utilizzata per valutare S_p. Le relazioni in input per $E(S_p)$ appartengono a $\{J_p\} \cup \left\{\pi_{\chi(p)}\,(S_c) : c \in children(p)\right\}$.

Il costo associato ad un sottoalbero T_p si compone del costo del vertice p in cui è radicato più il costo di tutti i sottoalberi di p. A questo risultato deve essere aggiunta l'aliquota dovuta alle operazioni di semijoin tra p ed i suoi figli. Quest'ultima viene calcolata considerando prima le eventuali operazioni di proiezione dei figli su $\chi(p)$e poi le join (in realtà semijoin) tra p e questi risultati.

Definizione 8. (Costo associato ad una decomposizione)
Sia $HD = \langle T, \chi, \lambda \rangle$ una hypertree decomposition per un'interrogazione congiuntiva Q di una base di dati **DB** di cui si hanno informazioni quantitative sulla cardinalità e selettività degli attributi delle relazioni.
Il costo di HD è definito da:

$$cost(HD) \triangleq cost(T)$$

Si sottolinea che il modello proposto definisce il costo di una decomposizione in termini della stima delle tuple coinvolte nel processo di valutazione suggerito dall'algoritmo di Yannakakis. Inoltre si noti come le definizioni date siano effettivamente indipendenti dalle particolari tecniche utilizzate per stimare il costo di join e semijoin multiple.

Capitolo 3

Tecnologie utilizzate

In questo capitolo descriviamo le tecnologie e gli strumenti utilizzati per lo sviluppo del progetto. In particolare, ci soffermiamo sulla descrizione del sistema DLV^{DB}, una versione del sistema DLV specifica per l'esecuzione di programmi su DMBS, utilizzato quale motore di inferenza nella valutazione dei programmi e sul DLV Wrapper, sviluppato presso l'Università della Calabria, in grado di gestire l'interazione tra la DLP e programmi scritti in Java.

3.1 DLV^{DB}

DLV^{DB} [8, 9, 10, 11] è stato progettato come estensione del sistema DLV; esso combina l'esperienza (maturata nell'ambito del progetto DLV) nell'ottimizzare programmi logici con le avanzate capacità di gestione delle basi di dati implementate nei DBMS esistenti. DLV^{DB} possiede tutte le caratteristiche di elaborazione dati desiderabili da un DDS ma consente di supportare anche le funzionalità di ragionamento più avanzate dei sistemi logici fornendo così sostanziali miglioramenti sia nelle prestazioni relative alla valutazione dei programmi logici, sia nella facilità di gestione dei dati di input e di output possibilmente distribuiti su più database; il tutto nel contesto di un sistema logico ben collaudato. Ciò permette di applicare tale sistema in ambiti che necessitano sia di valutare programmi complessi, sia di lavorare su grandi quantità di dati. L'interazione con le basi di dati è realizzata per mezzo di connessioni ODBC che consentono di gestire in modo piuttosto semplice dati distribuiti su vari database in rete. L'architettura di DLV^{DB} è stata progettata in modo da poter consentire facilmente ulteriori estensioni che possano supportare le capacità più avanzate del linguaggio di DLV e prevede due tipologie di esecuzione:

- Esecuzione in memoria centrale, che carica dati presenti in tabelle di database eventualmente distribuiti ed esegue il programma logico in memoria centrale. Tale tipologia di esecuzione consentirà di sfruttare sia l'alta espressività di DLV, sia la capacità dei DBMS di memorizzare e recuperare efficientemente grandi quantità di dati; naturalmente, in questa modalità la quantità di dati su cui si potrà lavorare contemporaneamente sarà limitata dalla quantità di memoria centrale disponibile;

- Esecuzione diretta su database, che valuta i programmi logici direttamente sulla base di dati e consente di gestire grandi quantità di dati con un utilizzo minimo della memoria centrale.

3.1.1 Esecuzione in memoria centrale

L'esecuzione in memoria centrale di DLV^{DB} utilizza il modulo di istanziazione standard di DLV, ma consente di importare i fatti di input tramite viste (anche complesse) su database; inoltre consente di esportare l'output del programma (o solo parti di esso) in tabelle di database esterni.

Questa tipologia di esecuzione supporta pienamente il linguaggio di DLV e tutte le sue estensioni ed ottimizzazioni (ad esempio, vincoli forti e deboli, funzioni esterne, magic-set, ecc.); ciò è reso possibile dalla scelta progettuale di non modificare affatto la strategia di valutazione standard implementata in DLV, ma di introdurre semplicemente moduli per l'importazione/esportazione dei dati nel sistema.

L'input e l'output dei dati è consentito attraverso due comandi built-in nella sintassi di DLV, specificatamente i comandi #import ed #export.

Il comando #import. Legge le tuple da una specificata tabella di un database relazionale e le memorizza come fatti (programma EDB) con un nome di predicato *p* specificato dall'utente. Il nome degli atomi importati è impostato a *p*, e definisce una parte del programma EDB. Ulteriori predicati EDB possono essere aggiunti attraverso un file di testo. Dal momento che DLV supporta soltanto numeri interi senza segno e costanti il comando #import prevede un parametro che specifica una conversione di tipo per ogni colonna della tabella. La sintassi del comando #import è la seguente:

#import(databasename,username,password,query, predname, typeConv).

dove:

databasename è il nome dell'origine dati ODBC per il database di interesse;
username identifica l'utente che si connette al database (la stringa deve essere racchiusa da virgolette);
password è la stringa che specifica la password da utilizzare per l'accesso a databasename (la stringa deve essere racchiusa da virgolette);
query è una stringa che rappresenta lo statement SQL da eseguire per reperire i dati;
predname indica il nome del predicato nel programma in cui caricare i dati;
typeConv specifica le regole di conversione dei dati necessarie per convertire i tipi del database nei tipi di DLV.

La sintassi del parametro *typeConv* è la seguente: *type : Conv [, Conv].*, dove *type* è una stringa costante e *Conv* rappresenta uno tra i seguenti tipi di conversione:
U_INT : la colonna è convertita in un intero senza segno;
UT_INT: la colonna è troncata ad un intero senza segno;
UR_INT: la colonna è arrotondata ad un intero senza segno;

CONST: la colonna è convertita in una stringa senza virgolette;
Q_CONST: la colonna è convertita in una stringa con le virgolette.

Il numero di conversioni specificate deve essere identico al numero di colonne della tabella selezionata. Le stringhe convertite in CONST devono essere delle costanti valide di DLV (ad esempio non possono contenere spazi).

Un comando *#import* carica i dati dalla tabella una riga per volta, tramite la query SQL specificata dall'utente, e crea un atomo per ciascuna tupla selezionata. Il nome di ciascun atomo è impostato a *predname* ed è considerato come fatto, cioè parte del programma EDB. Altri predicati EDB possono essere aggiunti attraverso file di testo da inviare a DLV insieme al programma logico.

Il comando #export. Permette di esportare l'estensione di un predicato in un answer set su un database. Ogni atomo (per quel predicato) che è vero nell'answer set comporterà l'inserimento di una corrispondente tupla nel database.

Il comando *#export* ha due varianti. La prima ha la seguente sintassi:

#export(databasename, username, password, predname, tablename).

La seconda variante aggiunge un ulteriore parametro *"REPLACE where SQL-Condition"*, che consente di sostituire le tuple nella tabella in base alla condizione *"SQL-Condition"*. Ciò consente di aggiungere le tuple alla tabella senza creare conflitti nel caso in cui tali tuple violerebbero alcuni vincoli di integrità del database (ad esempio, duplicare valori per un attributo chiave).

In questo caso:

databasename è il nome dell'origine dati ODBC per il database di interesse;
username identifica l'utente che si connette al database (la stringa deve essere racchiusa da virgolette);
password è la stringa che specifica la password da utilizzare per l'accesso a databasename (la stringa deve essere racchiusa da virgolette);
predname identifica il nome del predicato da esportare nel database;
tablename indica la tabella in cui i dati devono essere copiati; essa deve essere già presente nel database, con il numero corretto di attributi;
"REPLACE where SQL-Condition" contiene le parole chiave *REPLACE* e *where* seguite da una *SQL-Condition* che indica le tuple che devono essere eliminate prima che avvenga l'esportazione; ciò può essere utile per eliminare quelle tuple che corrispondono a vincoli di integrità violati.

Esempio. Si assuma che un'agenzia di viaggi debba richiedere l'elenco di tutte le destinazioni raggiungibili da una certa compagnia aerea sia con i propri vettori, sia utilizzando accordi di code-share con altre compagnie.

Si supponga inoltre che i voli diretti di ciascuna compagnia siano memorizzati in una relazione *flight_rel(Id, From, To, Company)* del database *dbAirports*, mentre gli accordi di code-share siano memorizzati nella relazione *codeshare_rel(Company1, Company2, FlightId)* di un database esterno *dbCommercial*; se vi è un accordo di code-share tra la compagnia *c1* e la compagnia *c2* per il volo *FlightId*, vuol dire che *FlightId* è effettivamente realizzato tramite vettori di *c1*, ma può essere considerato anche realizzato da *c2*.

Infine, si assuma che, per ragioni di sicurezza, non è consentito alle agenzie di viaggio di accedere ai database *dbAirports* e *dbCommercial* e, pertanto, è

necessario memorizzare il risultato della computazione in una tabella *composed CompanyRoutes* di un terzo database *dbTravelAgency* predisposto appositamente per le agenzie di viaggio. Il programma datalog che può calcolare tutte le connessioni desiderate è il seguente:

destinations(From, To, Comp) :- flight(Id, From, To, Comp).
destinations(From, To, Comp) :- flight(Id, From, To, Comp), codeshare(C2, Comp, Id).
destinations(From, To, Comp) :- destinations(From, T2, Comp), destinations (T2, To, Comp).

Per poter utilizzare i dati residenti nei vari database introdotti precedentemente, è necessario far corrispondere il predicato *flight* con la relazione *flight_rel* di *dbAirports* ed il predicato *codeshare* con la relazione *codeshare_rel* di *dbCommercial*. Infine, è necessario far corrispondere il predicato *destinations* con la relazione *composedCompanyRoutes* di *dbTravelAgency*.

Pertanto, i comandi che è necessario aggiungere al precedente programma datalog sono:

#import(dbAirports,airportUser,airportPasswd , "SELECT FROM flight rel", flight, "type : U_INT, Q_CONST, Q_CONST, Q_CONST").
#import(dbCommercial,commUser,commPasswd , "SELECT FROM codeshare rel", codeshare, "type : Q_CONST, Q_CONST, U_INT").
#export(dbTravelAgency, agencyName, agencyPasswd, destinations, composed CompanyRoutes).

Di seguito illustriamo la tipologia di esecuzione direttamente in memoria di massa, con opportuno esempio.

3.1.2 Esecuzione in memoria di massa

Il sistema DLV^{DB} consente la valutazione di programmi logici normali stratificati direttamente in memoria secondaria, con i dati di input (fatti) memorizzati in database eventualmente distribuiti. Il sistema supporta sia la "true negation" che la "negation as failure" e tutti i predicati built-in attualmente previsti in DLV; infine, un aspetto molto importante riguarda il supporto di programmi contenenti funzioni di aggregazione (peraltro già introdotte anche in DLV) oltre che l'identificazione di eventuali violazioni di vincoli. Naturalmente, come accade in DLV, ciascun programma deve soddisfare specifiche regole di "safety" al fine di poterne garantire una valutazione corretta. DLV^{DB} è stato progettato per fornire tre peculiarità fondamentali:

- la capacità di valutare i programmi logici direttamente in memoria di massa, utilizzando una minima quantità di memoria centrale;
- la capacità di far corrispondere i predicati presenti nei programmi a viste, anche complesse, su tabelle di basi di dati;
- la possibilità di specificare in modo semplice ed intuitivo quali dati devono essere considerati l'input del programma e quali l'output.

```
Init-Section ::=
1.USEDB databaseName:username:password [System-Like]?.
System-Like ::=
LIKE [POSTGRES | ORACLE | DB2 | SQLSERVER | MYSQL]
Table-Definition (EDB definition)::=
2. USE tableName [( attribute [, attribute]*)]?
3. [AS ("SQL-Statement")]?
4. [FROM DatabaseName:UserName:Password]?
5. MAPTO predName [( type [, type] )]? [ALLOW_APPEND]?.
Table-Definition (IDB definition)::=
6. CREATE tableName [(attribute [, attribute]*)]?
7. MAPTO predName [( type [, type] )]?
8. [KEEP_AFTER_EXECUTION]?.
Table-Definition (Query definition)::=
9. QUERY tableName.
Final-Section ::=
10. DBOUTPUT DatabaseName:UserName:Password.
|
11. OUTPUT [Write-Option]? predName
12. [AS AliasTableName]?
13. IN DatabaseName:UserName:Password.
Write-Option :: =
14. APPEND
|
15. OVERWRITE
```

Figura 3.1: DLV^{DB} Grammatica per le direttive ausiliarie

Direttive ausiliarie. Come precedentemente evidenziato, DLV^{DB} è stato progettato principalmente per quelle applicazioni che operano su grandi quantità di dati memorizzati in database anche distribuiti. Pertanto, esso deve consentire all'utente di specificare una serie di direttive per gestire al meglio l'interazione del sistema con tali database.

Le *direttive ausialirie* devono essere scritte in un file separato con estensione "typ". Esse sono suddivise in tre sezioni. La *"Init Section"* consente di definire il working database, cioè il database in cui verrà fatta la valutazione del programma. La *"Table Definition Section"* consente di specificare i mapping tra i predicati del programma logico e le tabelle del database. Infine, la *"Final Section"* definisce le direttive per la memorizzazione di alcuni (o tutti) i risultati dell'esecuzione del programma.

La grammatica con cui l'utente può specificare tali direttive è mostrata in Figura 3.1.

Il comando *USEDB* (1.) consente di definire la connessione con il working database. Una possibile definizione di tale direttiva, quando il nome del database è *MyDatabase* (ad esempio su Postgres) e l'accesso è consentito all'utente *scott* con password *tiger* è il seguente:

USEDB MyDatabase:scott:tiger LIKE POSTGRES.

La direttiva *USEDB* è l'unica direttiva non opzionale.

La direttiva *LIKE* è opzionale ma fortemente raccomandata. Essa consente di selezionare lo specifico dialetto SQL ed ottimizzare l'interazione con il working database. Se non è specificata la direttiva *LIKE*, vengo utilizzati l'SQL e le funzionalità ODBC standard (la compatibilità totale è garantita con POSTGRES).

Attualmente i DBMS totalmente compatibili come working database sono:

- POSTGRESQL 8.0 e superiore
- MySQL 5.0
- ORACLE 10g
- SQL Server 2005
- DB2 UDB Versione 8.2

La direttiva *USE* va utilizzata per specificare un mapping tra una tabella che esiste nel database ed il predicato del programma. Si noti che la tabella può essere memorizzata in un database diverso da quello di lavoro (specificando l'opzione *[FROM DatabaseName]*). Quando viene specificata la clausola *[AS ("SQL-Statement")]* la tabella è aggiornata dal risultato dell'esecuzione di *"SQL-Statement'*. Nel caso in cui sia necessario aggiungere ad una tabella esistente nuove tuple derivate durante la computazione, è necessario specificare esplicitamente l'operazione di "append" attraverso la direttiva USE ... $ALLOW_APPEND$.

Con la direttiva *CREATE*, il sistema crea una tabella e definisce il mapping con il predicato. La tabella verrà creata nel working database. La direttiva alla riga 8., se presente, stabilisce che la tabella non venga eliminata alla fine della valutazione.

In entrambi i casi è necessario utilizzare l'opzione MAPTO per collegare la tabella al predicato corrispondente. Il numero di attributi della tabella deve chiaramente essere lo stesso dell'arità del predicato a cui essa viene associata. Nella definizione delle tabelle, viene richiesto di specificare i tipi di dato solo per consentire una corretta valutazione dei predicati built-in e delle funzioni di aggregazione.

L'opzione *"Query"* può essere utilizzata per specificare la tabella che dovrà memorizzare i risultati dell'eventuale query contenuta nel programma.

Nella *"Final Section"* la direttiva $DBOUTPUT$ consente di specificare il working database in cui il programma memorizza la computazione. Se è necessario memorizzare soltanto uno specifico predicato, è necessario utilizzare la direttiva 11. L'utente può scegliere se appendere i dati (tramite l'opzione *APPEND*) a quelli eventualmente già presenti, o se sovrascriverli (tramite l'opzione $OVERWRITE$).

E' importante sottolineare che tali direttive consentono di ottenere un'elevata flessibilità nel reperimento dei dati di input e nella memorizzazione dell'output.

Esempio. Si consideri lo scenario introdotto nell'esempio precedente dell'esecuzione in memoria centrale e si supponga che, a causa dell'enorme dimensione dei dati di input, si voglia eseguire il calcolo delle destinazioni raggiungibili direttamente in memoria di massa (cioè direttamente sul database). Per fare ciò,

```
USEDB dlvdb:myname:mypasswd.

USE flight_rel (Id, From, To, Company) FROM dbAirports:airportUser:airportPasswd
MAPTO flight (integer, varchar(255), varchar(255), varchar(255)).
USE codeshare_rel (Company1, Company2, FlightId) FROM dbCommercial:commUser:commPasswd
MAPTO codeshare (varchar(255), varchar(255), integer).

CREATE destinations_rel (From, To, Company)
MAPTO destinations (varchar(255), varchar(255), varchar(255)) KEEP_AFTER_EXECUTION.

OUTPUT  destinations AS composedCompanyRoutes IN dbTravelAgency:agencyName:agencyPasswd.
```

Figura 3.2: DLV^{DB} Esempio di direttive ausiliarie

è necessario utilizzare al posto dei comandi *#import* ed *#export* le direttive ausiliarie mostrate in Figura 3.2. Esse consentono di specificare i mapping tra i predicati del programma e le tabelle introdotte precedentemente. E' importante mettere in evidenza il fatto che il sistema potrà derivare automaticamente dei mapping di default ogniqualvolta ciò sia possibile (ad esempio quando una relazione ed un predicato hanno lo stesso nome) in modo da semplificare il più possibile la loro definizione.

3.1.3 Architettura del sistema

L'architettura del sistema DLV^{DB} è mostrata in Figura 3.3.

I riquadri etichettati con il simbolo DLV indicano i moduli implementati nel progetto DLV; gli altri, invece, sono i moduli che implementati specificatamente per DLV^{DB}.

Ciascun programma P dato in input al sistema è inizialmente analizzato dal *Parser* che codifica le regole del database intensionale (IDB) in un formato opportuno e predispone il database estensionale (EDB) dagli eventuali fatti introdotti direttamente nel programma. In assenza di tali fatti non viene costruita nessuna struttura dati in memoria principale.

Successivamente, l'*Optimizer* applica delle tecniche di riscrittura dei programmi logici al programma P ottenendo così un programma equivalente a P' ma che può essere istanziato in modo più efficiente e può produrre un programma ground più piccolo.

Il *Dependency Graph Builder* calcola il grafo delle dipendenze del programma P', le sue componenti connesse determinano l'ordine topologico di tali componenti.

Infine, il *DBInstantiator* riceve: *(i)* l'IDB e (eventualmente) l'EDB generati dal parser, *(ii)* il grafo delle dipendenze DG costruito dal Dependency Graph Builder e *(iii)* le direttive ausiliarie. Tale modulo valuta il programma P' attraverso una strategia di valutazione bottom-up basata sulla traduzione di ciascuna regola logica in uno statement SQL. Dato che i programmi in input sono limitati alla classe dei programmi normali stratificati, il *DBInstantiator* valuta completamente il programma e non è necessario applicare ulteriori moduli dopo la fase di grounding.

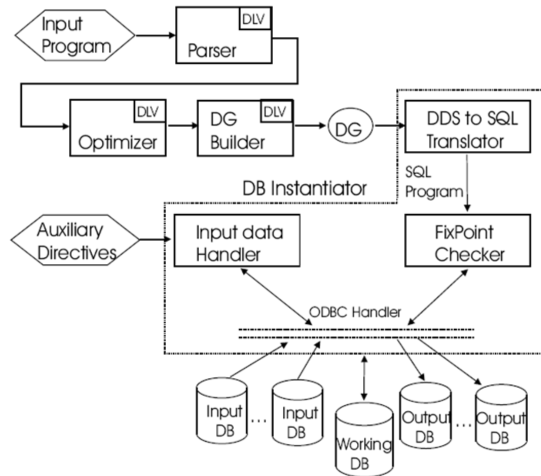


Figura 3.3: DLV^{DB} Architettura del sistema

Tutte le fasi della valutazioni saranno realizzate direttamente nel working database, attraverso l'esecuzione di statement SQL e senza caricare esplicitamente in memoria alcuno dei dati del programma (sarà il DBMS a scegliere cosa caricare in memoria in base alle proprie tecniche di ottimizzazione dell'accesso ai dati). Ciò consente a DLV^{DB} di essere completamente indipendente dalla dimensione dei dati in input e del programma generato.

L'interazione con i database, anche in questo caso, sarà realizzata tramite ODBC. Ciò consente da una parte di essere indipendenti dalle tipologie di DBMS utilizzabili, e dall'altra di gestire anche dati distribuiti su basi di dati accessibili tramite Internet.

Strategia di valutazione

La modalità di esecuzione in memoria centrale utilizza le tecniche di valutazione già presenti in DLV; al contrario, per la realizzazione della modalità di esecuzione diretta sul database è stato necessario definire una nuova strategia di valutazione. In particolare, la strategia adottata traduce il programma logico in input (eventualmente ottimizzato) in un insieme di statement SQL equivalenti e definisce un opportuno query plan tale che i dati ottenuti dalla sua esecuzione siano esattamente gli stessi di quelli che si otterrebbero dall'esecuzione in memoria centrale. In questa sezione focalizziamo l'attenzione sugli aspetti progettuali della definizione del query plan che implementa la strategia di valutazione; essa corrisponde ad un'implementazione differenziale del classico algoritmo Semi-Naïve.

In particolare, il grafo delle dipendenze DG costruito dal *Dependency Graph Builder* è inizialmente utilizzato per determinare l'ordine topologico delle varie componenti del programma; tali componenti sono quindi valutate una per volta, partendo dalle componenti più basse nell'ordine topologico, cioè quelle componenti la cui valutazione non dipende da altre. La valutazione di ciascuna componente viene iterativamente ripetuta finchè non è possibile derivare da es-

sa nuovi valori di verità, cioè se si raggiunge un punto fisso. A questo punto si passa alla valutazione della componente successiva nell'ordinamento.

In ciascuna iterazione, l'istanziazione di una regola consiste nell'esecuzione dello statement SQL corrispondente; uno degli obiettivi principali nell'implementazione di DLV^{DB} è proprio quello di associare un singolo statement SQL (non ricorsivo) a ciascuna regola del programma (sia essa ricorsiva o meno), senza strutture dati di supporto in memoria centrale per l'istanziazione.

Ciò è importante per sfruttare al meglio le tecniche di ottimizzazione presenti nei DBMS per la valutazione delle query SQL e per minimizzare i problemi di "memoria esaurita" causati dalle dimensioni limitate della memoria centrale e che spesso affliggono gli altri sistemi logici che lavorano in memoria centrale.

Nel seguito, tralasciando i dettagli implementativi della tecnica e concentrandosi sull'approccio, utilizziamo la notazione dell'algebra relazionale per descrivere il modo in cui viene determinato il query plan. Inoltre, per mettere in evidenza le differenze tra l'approccio differenziale utilizzato nel sistema ed il metodo Semi-Naïve classico introdurremo innanzitutto il metodo classico, evidenziandone alcune limitazioni ed i possibili i miglioramenti.

Si consideri un programma normale stratificato P ed il grafo delle dipendenze DG_P corrispondente; è possibile derivare da esso una sequenza ordinata di componenti $<P_{C1}, P_{C2}, ..., P_{Ch}>$tale che la valutazione della componente P_{Cg} ($1 \leq$ g $\leq$ h) dipende solo dai risultati ottenuti nella valutazione di componenti P_{Cf} tali che f <g.

L'algoritmo Semi-Naïve classico, quindi, considera quindi una componente di P per volta seguendo l'ordine determinato da $\mathrm{DG_P}$.

Il Semi-Naïve applicato a ciascuna componente può essere visto come un algoritmo a due fasi: la prima che valuta le regole non ricorsive; esse possono essere valutate completamente in una singola iterazione. La seconda considera le regole ricorsive ed utilizza un calcolo iterativo di punto fisso per la loro valutazione. In ciascuna iterazione esistono alcuni predicati *solved* per cui sono stati derivati tutti i valori di verità possibili, ed alcuni predicati *not completely solved* per i quali sono stati determinati solo alcuni valori di verità, mentre altri possono essere ottenuti da quelli calcolati fino a quella iterazione.

Sia $\mathrm{p_j}$ uno di tali predicati; in seguito indicheremo con $\Delta \mathrm{p_j^k}$ l'insieme di nuovi valori di verità ottenuti per $\mathrm{p_j}$ al passo k (e si definirà $\Delta \mathrm{p_j^k}$ il differenziale di $\mathrm{p_j}$).

Sia inoltre p:- $\varphi(\mathrm{p_1, p_2,...p_n, q_1,q_2,...q_m})$ una regola ricorsiva tale che φ è una formula del primo ordine, $\mathrm{p_1, p_2,...p_n}$, sono predicati mutuamente ricorsivi con p e $\mathrm{q_1,q_2,...q_m}$ predicati di base o derivati non mutuamente ricorsivi con p. L'algoritmo Semi-Naïve cerca di ridurre il numero di valori di verità determinati al passo k che erano già stati determinati in passi precedenti, attraverso la seguente formula:

$$\Delta\varnothing\left(p_1^k, \Delta p_1^k, .., p_n^k, \Delta p_n^k, q_1, .., q_m\right) = \varnothing\left(\left(p_1^k + \Delta p_1^k\right), \left(p_2^k + \Delta p_2^k\right), .., \left(p_n^k + \Delta p_n^k\right), q_1, .., q_m\right) -$$
$$\varnothing\left(p_1^k, p_2^k, .., p_n^k, q_1, q_2, .., q_m\right).$$

in cui, con un piccolo abuso di notazione vengono indicati con $\mathrm{p_j^k}$ i valori di verità calcolati per $\mathrm{p_j}$fino al passo k.

Pertanto, in ciascuna iterazione, l'algoritmo Semi-Naïve valuta solo il differenziale di φ piuttosto che l'intero φ. Tuttavia, come si vedrà tra breve, il Semi-Naïve esegue, meccanicamente, alcune operazioni che potrebbero essere evitate.

Si consideri la seguente regola in cui g ed h sono a loro volta predicati mutuamente ricorsivi con f:

f(X,Y):- f(X,Y), g(X,Y), h(X,Y).

In algebra relazionale, la valutazione di tale regola è equivalente alla formula:

$$F = F \bowtie G \bowtie H$$

e, all'iterazione k, il Semi-Naïve standard valuta la formula:

$$\begin{array}{rccccccl} \Delta F^k = & \Delta F^{k-1} & \bowtie & G^{k-1} & \bowtie & H^{k-1} & \cup & (a) \\ & F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-1} & \cup & (b) \\ & F^{k-1} & \bowtie & G^{k-1} & \bowtie & \Delta H^{k-1} & & (c) \end{array}$$

E' opportuno sottolineare che l'algoritmo utilizza tre tabelle differenti per ciascun atomo ricorsivo; infatti, data una generica relazione T, esso considera: (i) T^{k-1}, cioè l'insieme dei valori di verità calcolati per T fino all'iterazione *k-1*; (ii) ΔT^k, cioè i nuovi valori calcolati nell'iterazione k da T^{k-1} e (iii) ΔT^{k-1}. Ora, T^{k-1} può essere espressa come l'unione di due componenti: T^{k-2}e ΔT^{k-1}, cioè:

$$T^{k-1} \; = \; T^{k-2} \cup \Delta T^{k-1}$$

Questa osservazione, consente di riscrivere la formula precedente in:

$$\begin{array}{rlccccccl} \Delta F^k = & & & & & & & & \\ & \multirow{4}{*}{(a)} \left[\right. & \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & H^{k-2} & \cup & (1) \\ & & \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & (2) \\ & (a) & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & (3) \\ & & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & (4) \\ & & F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & (5) \\ & (b) & F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & (6) \\ & & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & (7) \\ & & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & (8) \\ & & F^{k-2} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & (9) \\ & (c) & F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & (10) \\ & & \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & (11) \\ & & \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & . & (12) \end{array}$$

in cui si possono evidenziare alcune operazioni di join ripetute più volte:
la join numero (2) è uguale alla join numero (11)
la join numero (3) è uguale alla join numero (7)
la join numero (4) è uguale alle join numero (8) e (12)
la join numero (6) è uguale alla join numero (10)

Queste osservazioni mettono in evidenza che, per l'esempio considerato, solo 7 join su 12 (cioè circa il 60%) sono effettivamente necessarie per valutare ΔF^k, cioè le join (1), (2), (3), (4), (5), (6) e (9). La situazione messa in evidenza in questo esempio può essere generalizzata ad un qualsiasi tipo di regole ricorsive; pertanto, in ciascuna iterazione del Semi-Naïve possono essere evitate un gran numero di operazioni di join seguendo un approccio differenziale. In particolare, si consideri la seguente organizzazione delle join inevitabili dell'esempio precedente:

$$\begin{array}{ccccccc} F^{k-2} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(9)} \\ F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(5)} \\ F^{k-2} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(6)} \\ \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & H^{k-2} & \cup & \textit{(1)} \\ \Delta F^{k-1} & \bowtie & G^{k-2} & \bowtie & \Delta H^{k-1} & \cup & \textit{(2)} \\ \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & H^{k-2} & \cup & \textit{(3)} \\ \Delta F^{k-1} & \bowtie & \Delta G^{k-1} & \bowtie & \Delta H^{k-1} & \cup & \textit{(4)} \end{array} \left\{ \begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \right.$$

Associando il simbolo 1 alle tabelle differenziali ed il simbolo 0 alle tabelle standard, è possibile ottenere la sequenza di join ottimizzata per una generica regola con r predicati ricorsivi, semplicemente enumerando i numeri binari da 1 a 2^r-1 ed utilizzando le tabelle standard e differenziali in corrispondenza dei simboli 0 e 1 così ottenuti.

Implementazione del Fix-Point Checker. La strategia di valutazione implementata nel sistema DLV^{DB} si basa su una versione differenziale del ben noto algoritmo Semi-Naïve. In questa sezione viene riportato l'algoritmo utilizzato per la sua implementazione. Esso assume che tutte le regole nel programma in input siano già state tradotte nei corrispondenti statement SQL. Sia P_{Cg} una componente di un programma P che dipende dai predicati $r_1,...,r_n$ risolti nelle componenti precedenti, che contiene i predicati non ricorsivi $q_1,...,q_m$ ed i predicati ricorsivi $p_1,...,p_n$. Siano inoltre $R_1,...,R_n$ (rispettivamente, $Q_1,...,Q_m$ e $P_1,...,P_n$) le relazioni corrispondenti ai predicati $r_1,...,r_n$ (rispettivamente, $q_1,...,q_m$ e $p_1,...,p_n$).

Nell'algoritmo mostrato in figura 3.4, la funzione EVAL valuta tutte le regole ricorsive che hanno q_i in testa eseguendo ciascuna delle query SQL corrispondenti e copiando i risultati di tali query nella relazione Q_i.

La funzione EVAL_DIFF implementa la valutazione differenziale delle regole ricorsive come descritto precedentemente; per ciascun predicato p_i essa valuta

```
Differential Semi-Naive(Input: R_1, ..., R_l. Output: Q_1, ..., Q_m, P_1, ..., P_n)
begin
        for i:=1 to m do // Evaluate non recursive predicates
(1)             Q_i = EVAL(q_i, R_1, ..., R_l, Q_1, ..., Q_m);
        for i:=1 to n do begin // Initialize recursive predicates
(2)             P_i^{k-2} = EVAL(p_i, R_1, ..., R_l, Q_1, ..., Q_m);
(3)             ΔP_i^{k-1} = P_i^{k-2};
        end;
        repeat
                for i:=1 to n do begin
(4)                     ΔP_i^k = EVAL_DIFF(p_i, P_1^{k-2}, ..., P_n^{k-2}, ΔP_1^{k-1}, ..., ΔP_n^{k-1}, R_1, ..., R_l, Q_1, ..., Q_m);
(5)                     ΔP_i^k = ΔP_i^k − P_i^{k-2} − ΔP_i^{k-1};
                end;
                for i:=1 to n do begin
(6)                     P_i^{k-2} = P_i^{k-2} ∪ ΔP_i^{k-1};
(7)                     ΔP_i^{k-1} = ΔP_i^k;
                end;
        until ΔP_i^k = ∅, ∀i 1 ≤ i ≤ n;
        for i:=1 to n do
(8)             P_i = P_i^{k-2};
end.
```

Figura 3.4: DLV^{DB} Algoritmo Semi-Naïve differenziale

i nuovi valori di verità per $\mathrm{p_i}$ all'iterazione k a partire dai valori calcolati fino all'iterazione k-2 e dai nuovi valori calcolati al passo k-1. Il risultati di EVAL_DIFF sono memorizzati nella tabella $\Delta\mathrm{P_i^k}$. Chiaramente, anche se tale approccio esegue un minor numero di operazioni del Semi-Naïve classico, non è possibile dimostrare che EVAL_DIFF non ricalcola alcuni dei valori di verità già ottenuti in precedenza; pertanto, $\Delta\mathrm{P_i^k}$ va "ripulita" di tali valori prima di passare all'iterazione successiva; ciò è esattamente quello che fa l'istruzione (5) dell'algoritmo.E' opportuno sottolineare che l'ultimo *for* dell'algoritmo (istruzione (8)) è mostrato solo per chiarezza. Infatti, nell'implementazione reale, non è necessario effettuare tale copia, dal momento che la tabella indicata con il nome $\mathrm{P_i^{k-2}}$ è esattamente $\mathrm{P_i}$.

3.2 DLV Wrapper

La programmazione logica disgiuntiva (DLP) è un formalismo, ormai affermato, per la rappresentazione, in maniera semplice e naturale, di forme di ragionamento non monotono, conoscenza incompleta, problemi diagnostici, di planning e, più in generale, problemi di elevata complessità computazionale [14]. Consiste nella evoluzione del paradigma della programmazione logica, con in più la possibilità di considerare la disgiunzione tra predicati nella testa di una regola e la negazione di predicati nel corpo.

E' stato dimostrato che la DLP ha un elevato potere espressivo, e permette di rappresentare anche problemi $\sum_2^P$–completi (NP^{NP}). E l'alta espressività dei linguaggi logici disgiuntivi ha importanti implicazioni pratiche, non solo formali: i programmi logici disgiuntivi, infatti, possono rappresentare situazioni reali che non possono essere espresse da programmi logici normali (senza disgiunzione).

Il sistema DLV, realizzato nell'ambito di una collaborazione internazionale tra Università della Calabria e Politecnico di Vienna, è tra le prime implementazioni che ne supporta pienamente la semantica e la estende attraverso vari

costrutti [42, 43]. Esso supporta sofisticate forme di ragionamento e consente di risolvere problemi complessi attraverso una specifica puramente dichiarativa delle soluzioni desiderate.

Oggi, il sistema DLV è riconosciuto come lo stato dell'arte nell'implementazione della DLP.

Da un punto di vista tecnico, DLV è un programma altamente portabile, scritto in ISO C++, disponibile in formato binario per diverse piattaforme [96].

La portabilità del sistema a supporto di tale linguaggio altamente espressivo è un forte stimolo ad utilizzare sistemi basati sulla logica per lo sviluppo delle applicazioni.

D'altra parte, oggi, un gran numero di applicazioni software sono sviluppate utilizzando linguaggi orientati agli oggetti come C++ e Java e la necessità di integrare tale tipo di applicazioni con sistemi basati sulla logica è in costante aumento. Tuttavia, i sistemi DLP non supportano alcun tipo di integrazione con gli strumenti di sviluppo software attuali. In particolare, il sistema DLV non può essere integrato facilmente in un'applicazione esterna.

Al fine di superare tale difficoltà, è stata sviluppata dall'Università della Calabria una API in Java, denominata DLV Wrapper [76] che consente di incorporare programmi logici disgiuntivi all'interno del codice sorgente object-oriented.

Il DLV Wrapper è una libreria orientata agli oggetti che "avvolge" il sistema DLV in un programma Java. In altre parole, il DLV Wrapper funge da interfaccia tra i programmi Java ed il sistema DLV. Utilizzando un'adeguata gerarchia di classi Java, DLV Wrapper ci permette di combinare codice Java con programmi logici disgiuntivi.

Una invocazione DLV può essere schematizzata nelle seguenti fasi:

1. Configurare i parametri di input e di invocazione.
2. Eseguire DLV.
3. Gestire i risultati di DLV.

Il DLV Wrapper ci dà pieno controllo sull'esecuzione di DLV. Si noti che DLV potrebbe impiegare molto tempo nel calcolo degli answer set, poiché i programmi logici disgiuntivi possono trattare problemi difficili (ci permettono di esprimere tutte le proprietà che sono decidibili in tempo polinomiale deterministico con un oracolo in NP). Ma, non appena un nuovo modello è calcolato, DLV lo restituisce. Per gestire questa situazione, vengono fornite due modalità di invocazione: *sincrona* e *asincrona*.

Se eseguiamo DLV in modalità *sincrona*, il thread Java che richiama DLV è bloccato fino a che DLV non completa la computazione. Il thread Java che richiama DLV può gestire solo l'output di DLV quando l'esecuzione di DLV è terminata. Se eseguiamo DLV in modalità *asincrona*, il thread Java che richiama DLV può accedere ai modelli non appena sono calcolati. Il DLV Wrapper fornisce un metodo che verifica se un nuovo modello è disponibile.

Il DLV Wrapper fornisce una flessibile interfaccia per l'input e l'output. La gestione dell'input e dell'output di DLV avviene attraverso oggetti Java. Questa caratteristica ci permette di incorporare pienamente programmi logici disgiuntivi all'interno di codice sorgente Object-Oriented. Inoltre, i programmi in input possono essere costituiti da diversi file di testo ovvero da oggetti Java in memoria e l'output può essere rediretto specificando il dispositivo di memorizzazione (memoria, hard disk, ecc), per ciascun predicato ground in un modello.

3.2.1 Panoramica sul DLV Wrapper Package

Il DLV Wrapper package è composto da diverse classi Java:

DLV_Manager: Questa è la "core class" che gestisce le chiamate a DLV e il suo output.

Program: Questa classe consente di gestire il programma in input di DLV.

Model: Questa classe rappresenta i modelli.

Model.Predicate: Questa classe rappresenta un predicato contenuto in un modello.

Queste sono le classi più importanti, ma esiste anche un'utile gerarchia di classi per la gestione delle eccezioni:

DLV_Exception: è la classe progenitore di tutte le classi che gestiscono le eccezioni (checked) contenute nel pacchetto DLV.

DLV_ExceptionUnchecked: è la classe progenitore di tutte le classi che gestiscono le eccezioni (unchecked) contenute nel pacchetto DLV.

DLV_InvocationException: viene generata se si è verificato un errore durante l'invocazione di DLV.

NoModelsComputed: viene generata se si desidera gestire l'output DLV prima dell'esecuzione di DLV.

IsNoModelException: viene generata se si chiama un metodo e questo oggetto rappresenta nessun modello.

InvalidParameterException: viene generata se si imposta un parametro non valido in una chiamata DLV.

NoModelsComputed: viene generata quando si tenta di gestire l'output di DLV prima di richiamarlo.

NoSuchModelException: viene generata se viene chiesto a un'istanza *DLV_Manager* di istanziare un modello che non esiste.

ParserException: viene generata se si trova un errore di sintassi.

3.2.2 Architettura interna.

Nel seguito descriviamo brevemente il funzionamento del DLV Wrapper. L'immagine 3.5 mostra una pipeline informale di esecuzione.

Attraverso un'istanza della classe *Program* il programmatore deve predisporre l'input di DLV. E' possibile impostare due tipi diversi di programma in input: *(i)* uno o più file di testo ovvero *(ii)* un programma contenuto in un oggetto *StringBuffer*. Utilizzando un'istanza della classe DLV_*Manager* è possibile impostare i parametri di invocazione di DLV, registrare l'oggetto *Program* precedentemente preparato, quindi è possibile richiamare una funzione su questo oggetto (come computeModels()). Quando viene richiamato uno di questi metodi il DLV_Manager effettua una chiamata ad un eseguibile DLV passandogli parametri ed input. Il programma contenuto nello *StringBuffer* viene passato in un pipe ed altri programmi sono passati attraverso il parametro. DLV_*Manager* raccogliere l'output da un pipe di sistema, lo analizza e crea una tabella di oggetti *Model* che il programmatore può gestire utilizzando una serie di metodi forniti dalla classe DLV_*Manager*.

Esempio di utilizzo. Nel seguito vengono descritte le principali operazioni necessarie al corretto uso del DLV Wrapper.

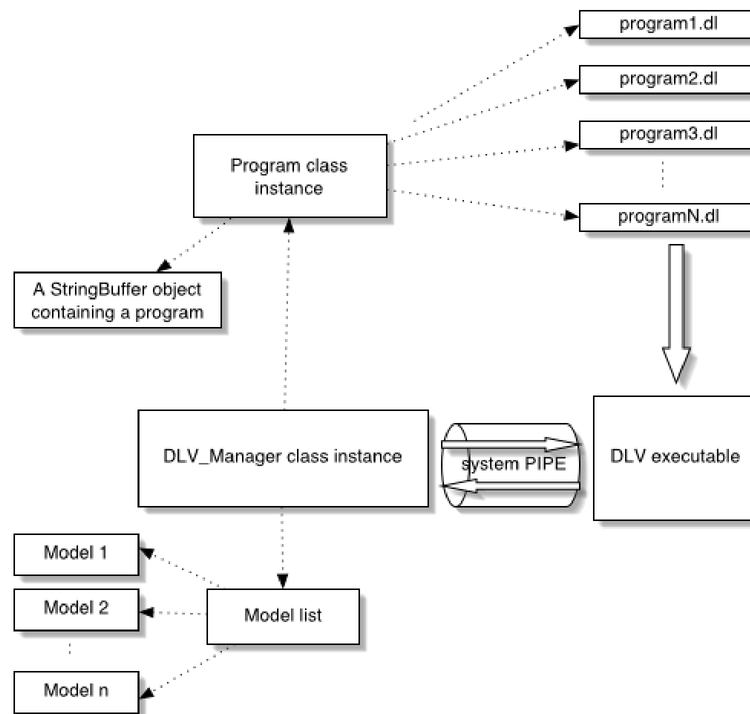


Figura 3.5: DLV Wrapper: Pipeline di esecuzione

Vengono a tal fine utilizzati due famosi esempi di programmi logici per calcolare *(i)* la serie di Fibonacci[1] e *(ii)* un'istanza del problema di colorazione di un grafo[2].

L'algoritmo per chiamare DLV è molto semplice, è sufficiente:

1. Costruire un oggetto *Program* e impostare l'input di DLV.
2. Costruire un oggetto *DLV_Manager*, impostare i parametri di input di DLV ed il *Program* di input.
3. Richiamare una funzione di calcolo di un modello.
4. Gestire l'output di DLV attraverso la classe *Model*

La classe *Program* è una metafora del programma di input di DLV. Questa classe permette di utilizzare due tipi di input. Il primo è lo standard input file, e il secondo è un programma memorizzato in un oggetto *StringBuffer* all'interno del programma Java.

È possibile utilizzare diversi file e un singolo oggetto StringBuffer. È possibile specificare le sorgenti in fase di costruzione o dopo la costruzione.

[1]La funzione di Fibonacci è definita nel modo seguente: Fibonacci (0) = 1, Fibonacci (1) = 1, Fibonacci (x) = Fibonacci (x-1) + Fibonacci (x-2). Si inizia con i seguenti valori: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ... A parte i primi due valori, ciascun valore è definito come somma dei due precedent.

[2]3-colorabilità di un grafo è un problema hard (NP-completo). E' il problema di decidere se esiste una colorazione di una mappa di paesi corrispondenti alla curva data con non più di tre colori in cui non ci sono due paesi vicini (nodi collegati da un arco) hanno lo stesso colore.

Esempio 3.2.1.
$StringBuffermyProgram = \ldots$
$\ldots$
$Programp = newProgram(myProgram, false);$
$p.addProgramFile(''./myProgramFile.dl'');$
$\ldots$
oppure

$StringBufferpath = newStringBuffer(''./myProgramFile.dl'');$
$StringBuffefmyProgram = \ldots$
$Programp = newProgram(path, true);$
$p.setProgram(myProgram);$
$\ldots$

□

La classe Program è estremamente potente poiché è possibile memorizzare la query in uno *StringBuffer* e modificarlo nel programma Java senza doverlo analizzare, modificare un file di testo ed è anche possibile memorizzare un grande DB in più file di testo senza neanche leggerlo nel codice Java!

Per mostrare le potenzialità di Program abbiamo suddiviso il programma di colorazione del grafo in più sorgenti di input.
nodeEDB.dl contiene:

$node(minnesota).$
$node(wisconsin).$
$node(illinois).$
$node(iowa).$
$node(indiana).$
$node(michigan).$
$node(ohio).$

arcEDB.dl contiene:

$arc(minnesota, wisconsin).$
$arc(illinois, iowa).$
$arc(illinois, michigan).$
$arc(illinois, wisconsin).$
$arc(illinois, indiana).$
$arc(indiana, ohio).$
$arc(michigan, indiana).$
$arc(michigan, ohio).$
$arc(michigan, wisconsin).$
$arc(minnesota, iowa).$
$arc(wisconsin, iowa).$
$arc(minnesota, michigan).$

guessEDB.dl contiene:

$\%guesscoloring$
$col(Country, red) \vee col(Country, green) \vee col(Country, blue) : \text{-}node(Country).$

Un oggetto *StringBuffer* contiene:

$\%checkcoloring$
$:\text{-}arc(Country1, Country2),$
$col(Country1, CommonColor), col(Country2, CommonColor).$

L'unione di queste sorgenti di input è il programma in input di DLV. Per impostare questo input è necessario scrivere:

$//prepara\ l'input$
$//prepara\ un\ programma\ StringBuffer\ che\ contiene\ un\ EDB:\ check\ constraint$
$StringBuffer\ check = new\ StringBuffer(":\text{-}arc(Country1, Country2),$
$col(Country1, CommonColor), col(Country2, CommonColor).");$
$Program\ p = new\ Program(check, false);$
$//imposta\ le\ altre\ sorgenti\ di\ input$
$p.addProgramFile("./nodeEDB.dl"); //un\ EDB\ file$
$p.addProgramFile("./arcEDB.dl"); //un\ EDB\ file$
$p.addProgramFile("./guessIDB.dl"); //un\ IDB\ file$

Ora è necessario creare un'istanza della classe *DLV_Manager*, impostare i parametri di invocazione ed il Program di input.

$//creazione\ DLV_Manager\ e\ impostazione\ input\ ed\ opzioni$
$DLV_Manager\ dm = newDLV_Manager(dlv);$
$dm.setNumberOfModels(4);\ //imposta\ i\ parametri\ di\ invocazione$
$dm.setProgram(p);\ //imposta\ il\ programma\ di\ input$

DLV nel codice è un oggetto String che contiene in percorso dell'eseguibile DLV. Alla fine è possible richiamare DLV:

$dm.computeModels();$

ed è possibile gestire il suo output.

Come già evidenziato DLV Wrapper offre due tipi di invocazione: *sincrona* e *asincrona*.

Se si utilizza (come in questo esempio) *computeModels()* viene eseguita la chiamata sincrona, ma se si richiama *computeModelsAsynchronously()* viene eseguita una chiamata asincrona.

I modelli sono memorizzati in un elenco, che è possibile gestire utilizzando una serie di metodi della classe *DLV_Manager*. Per accedere alla lista si può scegliere un'enumerazione Java utilizzando i metodi *hasMoreModels()* e *nextModel()*, o un *Vector* utilizzando i metodi *getModel()* e *goTo()*.

Ogni modello è memorizzato in un oggetto *Model*. È possibile accedere ai predicati all'interno di questo modello utilizzando la funzione *getPredicate()* che restituisce un oggetto *Model.Predicate*. È possibile gestire ogni oggetto *Model.Predicate* come una tabella utilizzando il metodo *getTermAt(x, y)*.

In questo esempio stampiamo sul device standard di output una versione mal formattata dell'output di DLV. Il modello scelto è:

$——— Model\ N:\ XXX\ ———-$
$———- predicate\ name\ ———-$
$arity:\ Y$
$size:\ Z$

term1, 1 ....term1, Y
...
termZ, 1 ....termZ, Y

Il codice Java è:

```
int num = 0;
while(dm.hasMoreModels())
{
    num++;
    System.out.println("------ Model N : " + num + " ------");
    Model m = dm.nextModel();
    String[] names = m.getPredicateNames();
    for(int y = 0; y < names.length; y++)
    {
        System.out.println("------- " + names[y] + " -------");
        Model.Predicate pr = m.getPredicate(names[y]);
        int arity = pr.arity();
        int size = pr.size();
        System.out.println("arity : " + arity);
        System.out.println("size : " + size);
        System.out.println(" ");
        for(int k = 0; k < size; k++)
        {
            for(int g = 0; g < arity; g++)
                System.out.print(pr.getTermAt(k, g) + " ");
            System.out.println(" ");
        }
    }
}
```

Un frammento dell'output di DLV è:

```
------ ModelN : 4 -------
------- arc -------
arity : 2
size : 12
minnesota wisconsin
minnesota iowa
minnesota michigan
wisconsin iowa
illinois wisconsin
illinois iowa
illinois indiana
illinois michigan
indiana ohio
michigan wisconsin
michigan indiana
michigan ohio
------- node -------
arity : 1
```

```
size : 7
minnesota
wisconsin
illinois
iowa
indiana
michigan
ohio
--------- col ---------
arity : 2
size : 7
minnesota red
wisconsin blue
illinois red
iowa green
indiana blue
michigan green
ohio red
```

Nel secondo esempio mostriamo come richiamare DLV come nella shell utilizzando la funzione *callDLVAndRedirectOutputInAFile().*

Il codice Java è:

```
Program p = new Program(new StringBuffer("./fibo.dl"), true);
//prepara DLV_Manager
DLV_Manager dm = new DLV_Manager(dlv);
dm.setProgram(p);
dm.setMaxint(20);
//call DLV
dm.callDLVAndRedirectOutputInAFile("out.txt");
```

La funzione *callDLVAndRedirectOutputInAFile()* richiama DLV e reindirizza l'output nel file out.txt. Questa funzione agisce come una invocazione di DLV dalla riga di comando e questo *DLV_Manager* non gestisce questo output. È necessario gestire l'output manipolando direttamente il file out.txt. Il thread, che chiama questo metodo, viene bloccato fino a quando l'operazione viene completata.

Mentre Java richiama DLV qualcosa potrebbe andare storto. *DLV_Manager* potrebbe generare un'eccezione o semplicemente un "warning", utilizzando il metodo *getWarning()* è possibile ottenere il messaggio di errore stampato da DLV nell'errore standard.

```
//print warnings
System.out.println(" ");
System.out.println("DLV Call Report :");
System.out.println(" ");
String warning = dm.getWarnings();
if(warning == null || warning.length() == 0)
System.out.println("Non warning found");
else
System.out.println("DLV has reported this warning : " + warning);
System.out.println(" ");
```

Capitolo 4

Descrizione della tecnica

Il capitolo illustra la progettazione e l'implementazione di una tecnica per la valutazione distribuita di programmi logici stratificati, senza disgiunzione. Il prototipo sfrutta un'estensione pesata di Hypertree Decomposition che considera anche informazioni quantitative sui dati per calcolare un piano di esecuzione ottimo. Al fine di implementare un modello di costo che contempli correttamente la trasmissione dei dati tra siti diversi, abbiamo dovuto apportare modifiche al tool di decomposizione *cost-k-decomp* affinchè valutasse, nel calcolo della decomposizione, anche i costi di trasmissione dei dati sulla rete.

Diverse tipologie di ottimizzazione caratterizzano il nostro approccio: *(i)* unfolding delle regole, *(ii)* ottimizzazione tra componenti, *(iii)* ottimizzazione all'interno di ciascuna componente. Nel seguito descriviamo ciascuna di esse.

Unfolding delle regole. Poiché la strategia di ottimizzazione strutturale è particolarmente adatta a regole aventi molti atomi nel corpo, l'unfolding procede, partendo dai predicati di output ovvero da quelli specificati nella query, e, seguendo le dipendenze, sostituisce le occorrenze degli atomi nel corpo con la loro definizione. Ciò ha l'effetto di ridurre il numero di regole che, per contro, avranno atomi nel corpo in numero maggiore. Inoltre, eventuali costanti specificate nel programma logico vengono "spinte" fino ai letterali terminali (per letterali terminali si intendono tutti quei letterali che non compaiono in testa a nessuna regola). In questo modo è possibile escludere dalla valutazione tutte le tuple del database che non hanno corrispondenza con i valori delle costanti. L'esecuzione, quindi, è resa più efficiente dall'eliminazione di regole intermedie che hanno il solo scopo di passare dalla query ai letterali terminali.

Ottimizzazione inter-componenti. Ciascun programma può essere associato ad un grafo, denominato grafo delle dipendenze (DG) che, intuitivamente, descrive quali predicati dipendono dagli altri. Il DG induce una suddivisione del programma in moduli in base alle componenti fortemente connesse di DG. Un ordinamento parziale delle Componenti Fortemente Connesse (SCC) consente di valutare i moduli in parallelo, infatti due componenti che non siano dipendenti l'una dall'altra possono essere eseguite simultaneamente poiché la valutazione dell'una non richiede i risultati prodotti dall'altra e viceversa.

Ottimizzazione intra-componente All'interno di ciascun modulo le regole vengono valutate parallelamente. Ogni singola regola subisce, inoltre, un ulteriore processo di ottimizzazione che inizia con la decomposizione strutturale dell'hypergraph corrispondente alla query. L'output è un hypertree che, in accordo alle modifiche apportate a *cost-k-decomp*, è stato esteso con la scelta del sito su cui effettuare la valutazione, con il costo di trasferimento delle tuple da un sito all'altro e con la valutazione di atomi negati. A partire dall'hypertree viene quindi generato un albero corrispondente, nel quale ciascun nodo è un sotto-programma. I sotto-programmi vengono eseguiti bottom-up a partire dalle foglie ed inseriti all'interno di thread che, adeguatamente sincronizzati, consentono un'esecuzione parallela dei sotto-programmi indipendenti.

4.1 Unfolding delle regole

In molti casi, durante la valutazione di un programma, siamo interessati al risultato di un sottoinsieme dei predicati. Infatti, molti valutatori consentono di specificare dei "filtri" (o predicati di output) per il programma. Nel caso di programmi stratificati, specificare un filtro corrisponde sostanzialmente ad individuare una porzione significativa del programma. Del resto, specificare dei filtri su un programma logico può essere visto come effettuare delle query sul programma stesso, ciascuna delle quali richiede tutti i dati specificati dal corrispondente predicato. Pertanto è utile sfruttare in questo conteso le tecniche di ottimizzazione orientate alle query.

Inoltre la strategia di ottimizzazione strutturale utilizzata è in special modo conveniente nel caso di regole aventi molti atomi nel corpo. In presenza di filtri (o query) nel programma adottiamo una strategia di unfolding standard che ha l'effetto di produrre regole aventi molti atomi nel corpo e, se possibile, in numero ridotto.

Per comprendere meglio le potenzialità dell'unfolding consideriamo il programma logico riportato nell'esempio che segue:

Esempio 4.1.1.
$a(X, c), b(X)?$
$a(X, Y) :\text{-} r(X), f(Y).$
$f(T) :\text{-} p_db(T).$

□

Supponiamo che il letterale *p_db* mappi una tabella di un database. La query potrebbe essere eseguita anche senza procedere con l'unfolding del programma, ma questo ci consente di ottimizzare la valutazione. Infatti, se il programma non viene modificato le tuple della tabella *p_db* verranno considerate tutte nella valutazione della query.

Invece, analizzando più a fondo il programma, ci rendiamo conto che le tuple della tabella che ci interessano sono solo quelle che hanno l'attributo che corrisponde alla variabile T, valorizzato a c.

Effettuando l'unfolding noi otteniamo proprio questo risultato, infatti la costante c viene spinta fino al letterale mappato sulla tabella. In questo caso, il risultato dell'algoritmo sarebbe quello mostrato di seguito.

Esempio 4.1.2.
$answer(X) :\text{-} r(X), p_db(c), b(X).$
$answer(X)?$

□

Come si può osservare dall'esempio, la regola intermedia che consente di passare dai letterali nella query ai letterali mappati sul database, viene eliminata. Il risultato è un'unica regola che include i risultati dell'unfolding di tutti i letterali della query.

Il programma che implementa l'algoritmo di unfolding è un modulo indipendente, pertanto, utilizzabile anche fuori dal contesto nel quale è stato presentato. L'algoritmo riceve in input un programma logico (che può anche contenere una query) e restituisce un separato programma equivalente in cui è stato effettuato l'unfolding delle regole.

Ogni processo consta di tre distinti passi: *PreProcessing*, *Unfolding* e *PostProcessing*.

PreProcessing. In questa prima fase, che agisce sull'intero programma, ci occupiamo di processare l'input al fine *(i)* di evitare ambiguità sui nomi durante il processamento e *(ii)* di identificare eventuali letterali che devono essere esclusi dall'elaborazione successiva.

Innanzitutto aggiorniamo il nome delle variabili, affinché siano univocamente riconducibili alla regola nella quale si trovano, ciò allo scopo di evitare ambiguità nel caso in cui più variabili aventi lo stesso nome e provenienti da regole diverse, nel processo di unificazione vengano portate nel corpo della stessa regola.

Il passo successivo riguarda l'individuazione e la selezione di quei letterali contenuti nella testa/corpo di regole che non vogliamo siano sottoposti al processo di unfolding. In particolare, i letterali che vengono etichettati "notUnfoldable" sono:

- I letterali in testa/corpo a regole ricorsive ed exit rule;
- I letterali negati.

Unfolding. Questa fase, particolarmente interessante, contiene l'implementazione dell'algoritmo *Unfold.* In linea generale, questo algoritmo riceve in input il programma logico sottoposto a *PreProcessing.* Il caso più semplice è quello in cui ciascun predicato sia definito da una sola regola, il cui corpo è costituito esclusivamente da letterali terminali.

In questo caso, l'algoritmo che esegue l'unfolding di una regola è il seguente:

Algoritmo 4.1.3. [Algoritmo Unfold con singola regola]

> ***FUNCTION*** *Unfold*(*Query*, *MappingRules*)
>   *List of Conjunction UnfoldedQuery* = *EMPTY*;
>   *UnfoldedQuery* = *null*;
> $n = size(Query)$
>   **for** $i = 1$ to n
>   *1.* A = *Query.get*(i);
>   *2.* R = *l'unica regola di mapping per* A;
>   *3.* *BodyMapping* = *literalUnfold*(A, R);
>   *4.* **add** *BodyMapping* **to** *UnfoldedQuery*;
>   **endfor**

return *UnfoldedQuery*

FUNCTION *literalUnfold*(*Lit*, *RuleMapping*)
1. Unify(*Args*(*Lit*), *Args*(*RuleMapping.Head*), &*RuleMapping.Body*)
return *RuleMapping.Body*

□

L'algoritmo 4.1.3 riceve in input una query (ovvero una regola logica) ed un insieme di regole definite di mapping, esattamente una per ogni letterale nel corpo della query. Il risultato sarà una congiunzione di letterali terminali che costituiranno la query unfoldata.

La funzione analizza singolarmente ogni letterale della query in input (1), individua la regola di mapping in cui questo figura in testa (2) e chiama la funzione literalUnfold (3) che effettua l'unfolding di un singolo letterale. Infine, aggiunge il corpo della regola di mapping, aggiornato correttamente, alla query da restituire come risultato (4).

La funzione literalUnfold ha l'unico scopo, in questo caso semplice, di richiamare la procedura Unify che si occupa di "unificare" gli argomenti del letterale della query con i corrispondenti argomenti dell'unico atomo (nel nostro approccio non gestiamo le regole disgiuntive) che sta in testa alla regola di mapping.

Di seguito, riportiamo la definizione della procedura di unificazione.

Algoritmo 4.1.4. [Procedura Unify]

```
PROCEDURE Unify(Args, Args1, &BodyMapping).

{
   n = length(Args); n1 = length(Args1)
   if n! = n1 then exit
   for i = 1 to i = n {
       T = Args.get(i); T1 = Args1.get(i)
       switch(typeOf(T1)) {
          1. CASE Variable :
                 replace T1 by T in Args1 and BodyMapping
                 break;
          2. CASE Constant :
                 if T is a Variable
                     then replace T by T1 in Args and BodyMapping
                 if T is a Constant then
                     if T = T1 then continue
                     else exit
               break;
     }
}
return DBQuery
```

□

Questa procedura si occupa di unificare gli argomenti di un letterale, del quale si sta facendo l'unfonding, con gli argomenti dell'atomo che sta in testa

ad una regola di mapping del letterale stesso. Il primo controllo riguarda l'arità dei due letterali. Infatti, se i due letterali hanno un numero di argomenti diverso, sicuramente non si potrà procedere con l'unificazione. Se i due letterali hanno la stessa arità, inizia la procedura di unificazione. Se sia l'argomento dell'atomo in testa alla regola di mapping che l'argomento corrispondente nel letterale sono variabili (1), allora si sostituisce nella regola di mapping la variabile della regola con la variabile del letterale. Se invece nella regola di mapping troviamo una costante (2), ma nel letterale c'è una variabile, allora sostituiamo la costante alla variabile. Ma se anche nel letterale c'è una costante, consentiamo il proseguimento dell'unificazione solo se le due costanti sono uguali. Altrimenti interrompiamo la procedura, infatti due costanti diverse non possono corrispondere.

L'algoritmo tralascia però altri aspetti come i mapping multipli e la presenza nel corpo delle regole di mapping di letterali che si trovano in testa ad altre regole (quindi a loro volta unfoldabili) che vengono correttamente gestiti nella versione completa dell'algoritmo.

Algoritmo 4.1.5. [Algoritmo Unfold completo]

```
FUNCTION Unfold(Query, MappingRules)
    List of Conjunction UnfoldedQuery = EMPTY;
    List of List of Conjunction BodyMappings = EMPTY;
    n = size(Query)
    for i = 1 to n
        A = Query.get(i);
        Let mapping(A) be the List of MappingRules with A in head;
        BodyMapping(i) = literalUnfold(A, mapping(A));
    endfor
    Combine BodyMappings to UnfoldedQuery;
return UnfoldedQuery

FUNCTION literalUnfold(Lit, RuleMappings)
    List of List of Conjunction Temp = EMPTY;
    if RuleMappings.isEmpty() then
        ManageLiteralWithNoMapping
    else if Lit is NotUnfoldable or RuleMappings.size() > MultiHead then
        ManageLiteralNotUnfoldable
    else
    for i = 1 to RuleMappings.size()
        if RuleMappings.get(i).isFact() then
            ManageFact
        else
            RuleMapping = RuleMappings.get(i)
            for j = 1 to RuleMapping.Body.size()
                Lit1 = RuleMapping.Body.get(j)
                Unify(Args(Lit), Args(RuleMapping.Head), &RuleMapping.Body)
                add to Temp(i) res of literalUnfold(Lit1, RuleMapping)
            endfor
    endfor
    Combine Temp to BodyMappings;
return BodyMappings
```

L'input che viene dato alla funzione consiste, anche nella versione dell'algoritmo completo, in una query (una congiunzione di atomi) ed un insieme di regole logiche; l'output è sempre la query unfoldata. In questo caso cambia però la funzione di unfolding del singolo letterale. Nel caso dei mapping singoli, essa restituiva una sola congiunzione di letterali. In questo caso l'output restituito sarà un insieme di liste di query congiuntive.

L'algoritmo inizia, come nel caso semplice, analizzando tutti i letterali della query. Per ognuno di questi individua le regole di mapping che passa alla funzione *literalUnfold*. Il caso più semplice da gestire è quando troviamo un letterale che non ha nessuna regola di mapping corrispondente. Questa situazione è quella che si verifica quando l'algoritmo invoca la procedura chiamata *manageLiteralWithNoMapping* e corrisponde anche al passo base della ricorsione. In questo caso, non bisogna fare niente di complicato. Per gestire adeguatamente la situazione è sufficiente (indipendentemente dal fatto che il letterale in questione un letterale mappato su una tabella di un database o un letterale interno al programma che non ha regole di mapping) aggiungere questo letterale ai risultati già ottenuti.

Altrettanto semplice risulta il caso in cui la regola di mapping in esame è un fatto, cioè una regola senza corpo. Questa situazione è gestita dalla procedura che nell'algoritmo in pseudo-codice viene invocata con il nome di *manageFacts*. In questo caso non è necessario aggiungere niente ai risultati già ottenuti.

Se nel processo di unfolding viene trovato un letterale etichettato durante la fase di PreProcessing come "notUnfoldable", il letterale stesso viene aggiunto alla query e vengono mantenute le regole di mapping corrispondenti. La stessa procedura viene richiamata quando il numero delle regole di mapping è superiore ad una costante fissata. Nel caso di mapping multipli, infatti ad ogni letterale da unfoldare corrispondono più liste di congiunzioni, una per ogni regola di mapping. La query risultante dovrà contenere più regole ottenute dalla combinazione di tutte le liste di congiunzione prodotte per ciascun letterale. Per questo motivo, poiché il numero di regole potrebbe crescere esponenzialmente abbiamo introdotto una costante intera *multiHead* che permette di intervenire limitando tale processo, nel senso che vengono unfoldati i letterali i cui mapping sono in numero inferiore a *multiHead*.

Le combinazioni delle liste di congiunzioni generate da Unfolding per ciascun letterale vengono calcolate dalla procedura *Combine* che le aggiunge all'output.

Un po' più complessa è la gestione della chiamata ricorsiva della funzione *literalUnfold*. Rientriamo in questo caso quando le regole di mapping hanno nel corpo dei letterali che sono presenti anche in testa ad altre regole logiche del programma in input.

La prima parte consiste semplicemente nell'unificazione del letterale corrente e del letterale presente in testa alla regola di mapping che richiede la ricorsione. In seguito, bisogna procedere con l'unfolding del corpo della regola di mapping in esame. Come ultimo passo, bisogna analizzare i risultati restituiti dalla chiamata ricorsiva e creare un nuovo risultato per ognuno di quelli restituiti, partendo dai risultati già ottenuti.

PostProcessing. Nell'ultimo modulo che completa l'unfolding ci occupiamo di aggiornare correttamente le variabili, i cui valori sono contenuti nelle varie map-

pe create nel corso dell'esecuzione. Pertanto procediamo nel recuperare le informazioni dalle varie mappe e aggiornare le variabili dei risultati ripristinando, per quanto possibile, i nomi delle variabili. Non in tutti i casi, infatti, possiamo riportare il nome originario delle variabile. Si consideri l'Esempio 4.1.6. In questo caso se mantenessimo il valore del secondo termine del letterale f avremmo il seguente risultato: *answer(A, B) :- b(A), f(B, A).*, che è errato poiché non è detto che il secondo termine di f debba essere uguale all'unico termine di b. Per risolvere questo problema, manteniamo un contatore che distingue le variabili nuove. In questo caso la risposta data dal sistema sarà: *answer(A, B) :- b(A), f(B, A1).*, che è corretta perché mantiene la differenza tra i due termini.

Esempio 4.1.6.
$a(A, B)$?
$a(A, B) : -b(A), c(B, C).$
$c(S, R) : -f(S, A).$

□

4.2 Ottimizzazione inter-componenti

L'ottimizzazione inter-componenti consiste nel suddividere il programma P in input (eventualmente unfoldato) in sottoprogrammi, secondo le dipendenze tra i predicati presenti in esso, identificando quali di essi possono essere valutati in parallelo.

Più nel dettaglio, ogni programma P può essere associato ad grafo, chiamato *Dependency Graph* di P che, intuitivamente, descrive come i predicati di P dipendano l'uno dall'altro.

Definizione 9 (Dependency Graph)**.** Sia P un programma. Il *Dependency Graph* di P è un grafo diretto $Gp = \langle N,E \rangle$, dove N è l'insieme di nodi ed E è l'insieme degli archi. N contiene un nodo per ciascun predicato IDB[1] di P, ed E contiene un arco $e = (p,q)$ se c'è una regola r in P tale che q è presente nella testa di r e p è presente in un letterale positivo del corpo di r.

Il grafo Gp induce una suddivisione di P in sottoprogrammi (chiamati anche moduli) consentendo una valutazione modulare. Diciamo che una regola $r \in P$ *definisce* un predicato p se p compare nella testa di r.

Definizione 10 (Strongly Connected Component)**.** Dato un grafo diretto $G = (V,E)$, una componente fortemente connessa (*SCC*, *Strongly Connected Component*) è un insieme massimale di vertici U sottoinsieme di V tale che per ogni coppia di vertici $(u, v) \in U$, u è raggiungibile da v e viceversa.

Per ciascuna *SCC* C di Gp, l'insieme delle regole che definiscono tutti i predicati in C è detto modulo di C e viene indicato con P_c. Partendo dal grafo delle dipendenze del programma logico costruiamo un grafo orientato in cui ciascun nodo, corrispondente ad una *SCC* del *DependencyGraph*, contiene il programma logico (modulo) associato. Ogni nodo avrà archi uscenti uno per ogni modulo che 'dipenda' dalla sua esecuzione, per ogni modulo, cioè, che necessita dei risultati prodotti dal modulo che lo precede. Chiaramente da

[1] un predicato *IntensionalDataBase* è un predicato che non ricorre soltanto in facts

ciascun nodo potranno uscire più archi, se più moduli dovranno attendere la sua esecuzione, ma anche più di uno potranno essere gli archi entranti, se il nodo dovrà attendere l'esecuzione di più moduli precedenti.

Questa struttura consente l'esecuzione dei moduli partendo parallelamente da quelli 'indipendenti' da quelli cioè che non hanno archi entranti (che corrispondono ai predicati di output ovvero a quelli specificati nella query). La terminazione di ciascun modulo svincolerà i moduli successivi che dipendono da quest'ultimo.

Consideriamo il seguente programma logico P, in cui a è un predicato EDB ed in cui p ed s siano i predicati di output.

Esempio 4.2.1.

$p(X,Y) :\text{-} q(X), q(Y), not\ t(X,Y).$
$t(X,Y) :\text{-} p(X,Y), s(Y).$
$q(X) :\text{-} a(X).$
$p(X,Y) :\text{-} q(X), t(X,Y).$
$s(Y) :\text{-} q(Y), v(X,Y).$
$v(X,Y) :\text{-} q(X), q(Y).$

□

Il *DependencyGraph* di P è mostrato in figura 4.1.

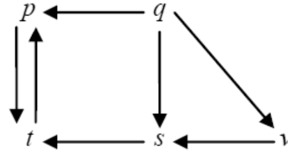


Figura 4.1: Dependency Graph del programma P

Le *Strongly Connected Components* del Dependency Graph sono : {s}, {v}, {q} e {p,t} cui corrispondono rispettivamente i seguenti moduli:

- {*s(Y) :- q(Y), v(X,Y).*}
- {*v(X,Y) :- q(X), q(Y).*}
- {*q(X) :- a(X).*}
- {*p(X,Y) :- q(X), q(Y), not t(X,Y). p(X,Y) :- q(X), t(X,Y). t(X,Y) :- p(X,Y), s(Y).*}

Pertanto, partendo dalle SCC del DG, definiamo un grafo orientato, i cui nodi contenenti i moduli corrispondenti sono connessi dalle dipendenze tra le SCC, che consente l'esecuzione parallela dei moduli. Il grafo di esecuzione dei moduli del programma P è mostrato in figura 4.2.

L'esecuzione inizia parallelamente dai nodi {t,p} ed {s} i quali istanziano ricorsivamente altri thread, corrispondenti ai nodi figli. E' interessante osservare, nell'esempio, che il nodo {q} potrebbe essere istanziato da {t,p} da {s} oppure da {v}. Ovviamente il thread viene istanziato ed eseguito un sola volta, da uno soltanto dei nodi di livello superiore mentre gli altri attendono comunque il termine della sua esecuzione.

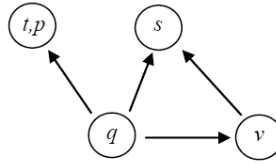


Figura 4.2: Grafo di esecuzione parallela dei moduli del programma P

4.3 Ottimizzazione intra-componente

L'esecuzione di ogni modulo procede attraverso due ulteriori passi di ottimizzazione. Il primo consiste nell'esecuzione parallela, ove possibile, delle regole all'interno di ciascun modulo; il secondo sfrutta un'estensione pesata di *Hypertree Decomposition*, che considera anche informazioni quantitative sui dati, per calcolare un piano di esecuzione ottimo.

Esecuzione parallela delle regole. Ciascun modulo può contenere una o più regole. Ciò è solo in parte direttamente collegato alla cardinalità della SCC corrispondente. Infatti, se da una lato è vero che ad una *SCC* avente cardinalità maggiore di uno, corrisponde un modulo (ricorsivo) costituito da più di una regola, anche da una *SCC* contenente un solo letterale può derivare un modulo costituito da più regole.

Se un modulo contiene più di una regola significa che *(i)* si tratta di un modulo ricorsivo ovvero, *(ii)* esistono più regole aventi lo stesso atomo in testa.

Una regola r che ricorre nel programma logico P_c è detta *ricorsiva* se c'è un predicato $p \in C$ che ricorre nel corpo di r, altrimenti è detta *exit rule*. Le *exit rules* in un modulo ricorsivo vengono sottoposte ad una ulteriore ottimizzazione attraverso il successivo processo di decomposizione, le regole ricorsive vengono, invece, eseguite senza subire ulteriori trasformazioni ma sfruttando i processi di ottimizzazione propri di DLV^{DB}.

Nel caso invece di più regole aventi la stessa testa, esse vengono eseguite parallelamente e sottoposte ciascuna ai successivi processi di ottimizzazione. L'output di ciascuna regola viene però memorizzato in una tabella temporanea. Al termine dell'esecuzione di tutte le regole che compongono il modulo viene generato ed eseguito un programma logico che legge tutte le tabelle temporanee e le accoda nella tabella risultante.

Esempio 4.3.1.
a :-b, c.
a :-d, e
a :-f, g.

□

Il programma logico nell'Esempio 4.3.1 viene modificato come nell'Esempio 4.3.2.

Esempio 4.3.2.
a1 :-b, c.

$a2$:-d, e
$a3$:-f, g.

□

Le regole del programma vengono eseguite parallelamente dopo essere state sottoposte ad ulteriori passi di ottimizzazione e, al termine, viene generato ed eseguito il seguente programma logico:

Esempio 4.3.3.
a :-$a1$.
a :-$a2$.
a :-$a3$.

□

Decomposizione di ogni regola. Al fine di ottimizzare la valutazione di ogni singola regola logica, siamo partiti dall'osservazione che quest'ultima può essere vista come una query congiuntiva (eventualmente con negazione) il cui risultato deve essere memorizzato nel predicato della testa. Abbiamo considerato, quindi, le tecniche di ottimizzazione di query congiuntive, cercando di estenderne i migliori risultati alla valutazione di programmi logici. Abbiamo utilizzato, a questo scopo, un'estensione pesata di *Hypertree Decomposition* che, all'analisi prettamente strutturale, aggiunge la valutazione di rilevanti informazioni quantitative sui dati, come dimensione delle relazioni e selettività degli attributi, e calcola decomposizioni minime rispetto ad una funzione di costo. Tale decomposizione è stata opportunamente modificata per stimare il costo di trasmissione dei dati fra siti diversi derivante dalla distribuzione delle sorgenti e per la corretta valutazione di atomi negati. In termini estremamente generali, il processo svolge le seguenti funzioni:

- Analizza sintatticamente il programma logico costituito da una singola regola ottenuto dai passi precedenti;
- Calcola i dati quantitativi associati alle relazioni ed ai termini della query;
- Calcola una k-hypertree decomposition ottimale per la query, con k specificato dall'utente;
- Esegue, a partire dalla decomposizione effettuata, una suddivisione del programma in sotto-programmi;
- Processa in parallelo i sotto-programmi secondo il piano di esecuzione individuato dall'ottimizzazione.

4.3.1 Generazione statistiche

Il primo passo consiste nella creazione dei files contenenti le informazioni statistiche sui dati. Tali statistiche, richieste dal tool di decomposizione, vengono calcolate attraverso apposite interrogazioni ai cataloghi di sistema dei vari DBMS.

I tre files prodotti in questa fase, richiesti dal tool di decomposizione, contengono informazioni sulla cardinalità delle relazioni, sul DBMS che ospita la

relazione, sulla selettività degli attributi di ciascuna relazione (il numero di valori distinti di un attributo), nonchè informazioni relative ai siti su cui le relazioni sono memorizzate ed ai costi di trasferimento tra essi, e sono generati dal sistema soltanto se non sono già presenti.

I files sono tabelle in formato CSV, in cui ogni riga è rappresentata da una linea di testo, che a sua volta è divisa in campi separati da un carattere separatore ";".

Il primo dei tre file, denominato *nomefile.dlv$card.csv*, dove *nomefile* è dato dall'atomo in testa alla regola e dal numero di riga della regola all'interno del modulo, contiene, per ogni atomo del corpo della regola logica, la cardinalità della relativa tabella ed il sito ove la tabella è memorizzata. Più precisamente, il file, contiene i seguenti attributi: *<ATOM_NUMBER>* è il numero, da 1 ad n, di posizione dell'atomo nel corpo della regola (dove n è il numero di atomi presenti);
<ATOM> è esattamente il predicato cui si riferiscono le informazioni quantitative sulla stessa riga;
<CARDINALITY> è il numero di tuple della tabella associata ad *<ATOM>*;
*<SITE>*è il sito ove si trova memorizzata la tabella associata ad *<ATOM>*. Questo valore, da 0 ad m (dove m è il numero di database sorgenti − 1), corrisponde all'identificativo del database contenente la tabella stessa. Si presuppone, infatti, che ciascun database sorgente sia collocato su un sito diverso. L'identificativo associato (da 0 a m) corrisponde all'ordine con cui il database viene riportato nei mapping;
<LIKE> è la descrizione del DBMS che ospita la tabella. Corrisponde a quanto richiesto da DLV^{DB} nella clausola $LIKE$ del $WorkingDB$.
Il file *nomefile.dlv$card.csv* conterrà esattamente n record, uno per ogni predicato nel corpo della regola, ordinati in base alla posizione che assumono nella regola stessa.

Il secondo dei tre file, denominato *nomefile.dlv$sel.csv*, dove $nomefile$ è dato dall'atomo in testa alla regola e dal numero di riga della regola all'interno del modulo, contiene, per ogni atomo del corpo del programma logico e per ogni variabile presente nell'atomo, la selettività dell'attributo corrispondente nella relativa tabella. Più nel dettaglio, gli attributi sono i seguenti:
<ATOM_NUMBER> è il numero, da 1 ad n, di posizione dell'atomo nel corpo della regola (dove n è il numero di atomi presenti);
<ATOM> è esattamente il predicato cui appartengono le variabili definite nell'attributo seguente;
<VARIABLE> è la variabile dell'atomo $< ATOM >$ corrispondente all'attributo della tabella di cui viene calcolata la selettività;
<SELECTIVITY> è la selettività dell'attributo corrispondente a *<VARIA BLE>*. Tale informazione, definita come il numero di valori distinti di un attributo, viene calcolata attraverso interrogazioni sul database.
Quindi, le informazioni relative ad *<ATOM_NUMBER>* e *<ATOM>*, vengono ripetute per ogni variabile presente nel predicato corrispondente.

L'ultimo dei tre files necessari al processo di decomposizione, è denominato *nomefile.dlv$cost.csv*, dove $nomefile$ è dato dall'atomo in testa alla regola e dal numero di riga della regola all'interno del modulo. Consiste in una matrice quadrata contenente i costi di trasmissione da un sito all'altro, intendendo l'elemento $x_{i,j}$ il costo di trasferimento per tupla dal sito i al sito j, e la cui diagonale è 0. Anche in questo caso la definizione di sito può essere assimilata a

quella di database. Nella semplificazione adottata, infatti, è corretto supporre ciascun database sorgente su un diverso sito.

Il file dovrebbe essere correttamente predisposto dall'utente sulla base di conoscenze specifiche dei database coinvolti nell'interrogazione e della rete ma, nel caso in cui non esista, viene generato dal sistema in una versione di default.

4.3.2 Tecnica di decomposizione

Nel nostro approccio abbiamo esteso un metodo di decomposizione strutturale, originariamente introdotto per ottimizzare la valutazione di query congiuntive. In particolare, la nostra strategia produce una *weighted hypertree decomposition* (vedi [7]) del corpo della regola, il cui risultato è interpretato come un *query plan* distribuito. Il modulo *d-cost-k-decomp* è frutto della revisione del tool di decomposizione *cost-k-decomp*, scritto in linguaggio C++, ed illustrato nella Sezione 2.2.

Pertanto abbiamo effettuato uno studio approfondito del tool, che ha subito modifiche sostanziali per gestire: *(i)* gli atomi negati (eventualmente) presenti nel corpo della regola; *(ii)* le costanti (eventualmente) specificate nella regola; e *(iii)* la distribuzione della valutazione della regola su siti diversi.

Gestione della negazione. Al fine di gestire la presenza di atomi negati in r, la costruzione di hypertrees validi è stata modificata in modo tale che ogni nodo contenente un atomo negato deve essere un nodo foglia e non contenere altri atomi. Questi vincoli sono stati opportunamente posti per isolare gli atomi negati al fine di gestirli in modo specifico nella computazione complessiva.

Tuttavia, si può osservare che la *safety* delle regole implica che questi vincoli non limitano il calcolo di hypertrees validi.

Più formalmente, se esiste una decomposizione valida della versione positiva di una regola r, allora esisterà certamente una valida decomposizione di r.

Gestione delle costanti. La presenza di costanti nella query potrebbe influire significativamente sulle prestazioni, in quanto potrebbe portare a selezioni sui dati originali (dopo l'applicazione dell'unfolding), che potrebbero ridurre significativamente la quantità di dati effettivamente necessari per rispondere alla query. Di conseguenza, abbiamo esteso il metodo di decomposizione affinchè consideri ciò (e ne tragga vantaggio). La modifica parte dalla constatazione che (nella solita ottica di una regola come una query congiuntiva) la presenza di una costante nel corpo di una regola corrisponde ad una variabile (istanziata) che non è coinvolta né in un join, né nel risultato; pertanto possiamo tranquillamente applicare una selezione e una proiezione sui dati originali prima della valutazione. In pratica, ignoriamo le colonne corrispondenti alle costanti ed utilizziamo (per il calcolo della decomposizione) una stima della dimensione dei dati filtrati (cioè, rilevanti per la query) in base alle statistiche disponibili su selettività e cardinalità.

Distribuzione della decomposizione. Al fine di tener conto della distribuzione dei dati nel calcolo della decomposizione, estendiamo la definizione standard di hypertree come segue. Sia $Sites = \{S_1, \ldots, S_n\}$ l'insieme dei siti su cui risiedono i dati coinvolti nella regola. Un hypertree H è una tupla $\langle T, \chi, \lambda, \sigma \rangle$, in cui T, χ, e λ hanno il significato originario (vedi Sezione 2.2), e σ è una

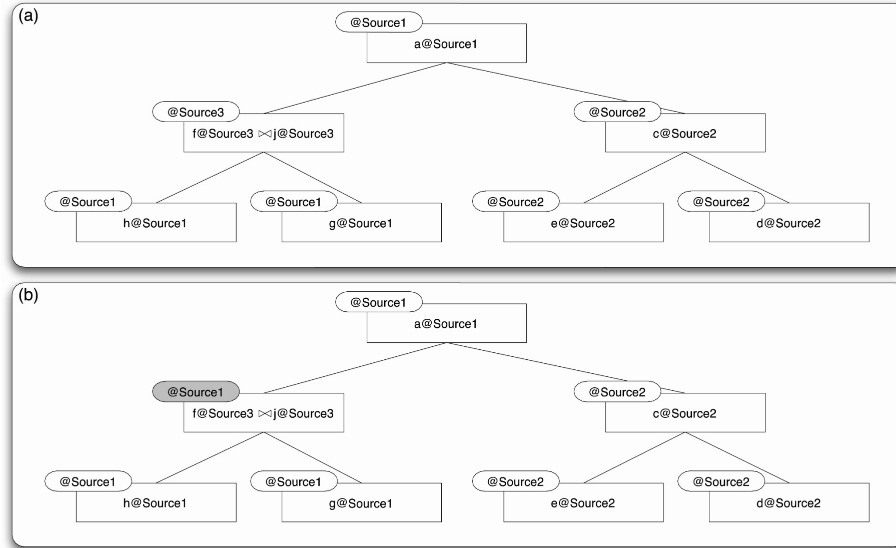


Figura 4.3: Hypertree Decompositions: (a) etichettatura statica delle sorgenti (b) etichettatura alternativa delle sorgenti.

funzione di etichettatura che associa ciascun vertice $p \in N$ con un sito in $Sites$ (formalmente, $\sigma(p) \in Sites$).

$\sigma(p)$ indica il database dove i dati (parziali) associati a p ed il risultato della valutazione di T_p si suppone risiedano. Se p è la root dell'hypertree, $\sigma(p)$ indica il sito dove ci sia aspetta risieda il risultato dell'intera computazione. Chiaramente l'etichettatura $\sigma(p)$ ha un impatto sul costo di valutazione, dal momento che condiziona il trasferimento dei dati.

In questa nuova impostazione, il costo di un hypertree dipende non solo dalla sua struttura, ma anche dalla sua etichettatura. Ciò pone nuove problematiche nell'algoritmo decomposizione.

Esitono diversi metodi per etichettare i nodi dell'hypertree. Ad esempio, una possibilità è quella di calcolare la decomposizione minima senza considerare prima l'etichettatura sito, e in seguito scegliere la migliore etichettatura sito per la decomposizione; un'alternativa è quella di associare un'etichetta "statica" ad ogni nodo dell'albero, in base alle relazioni che coinvolge (ad esempio scegliendo il sito che richiede il minimo trasferimento di dati per combinare insieme tutte le relazioni sul nodo).

Entrambe queste soluzioni presentano alcuni problemi. Infatti, la prima non considera la distribuzione nella ottimizzazione della funzione di costo ma minimizza il trasferimento dei dati soltanto *dopo* che la decomposizione è stata determinata; di conseguenza, essa può produrre query plan non ottimali rispetto ai costi di trasferimento.

Il secondo approccio può perdere ottimalità per una funzione di costo che comprende il trasferimenti di dati. Ciò può essere facilmente compreso attraverso il seguente esempio. La Figura 4.3 riproduce due hypertree (semplificati), ottenuti dalla stessa regola, che condividono la stessa struttura ma hanno etichettatura sito diverse (i nomi dei siti sono preceduti dal simbolo @). In particolare, l'hypertree (a) potrebbe essere ottenuto dall'approccio con etichettatura statica di cui sopra; si noti che l'hypertree (b) potrebbe essere migliore

(il che si traduce in una più rapida valutazione) se le relazioni f e j hanno una dimensione inferiore alle relazioni h e g.

Infatti, i costi di trasferimento sostenuti per la copia di f e j (come richiesto dall'hypertree (b)) da *Source3* a *Source1* sarebbero inferiori a quelli sostenuti per la copia (come richiesto dall'hypertree (a)) di h e g su *Source3*. Intuitivamente, l'etichettatura statica effettua una scelta locale che non guarda la disposizione globale della decomposizione.

Nel nostro approccio, in considerazione delle problematiche esposte, estendiamo l'algoritmo di decomposizione di base nel modo seguente:

sia p un nodo generato dall'algoritmo standard durante il processo di decomposizione; sostituiamo p con un insieme di n nodi $p_1 \dots p_n$ tale che $\sigma(p_i) = S_i$ (cioè un nodo per ciascun sito). Successivamente viene eseguita la decomposizione standard su questa nuova serie di nodi. Intuitivamente, se la funzione di costo viene modificata (come specificato nel seguito) al fine di tener conto dei costi di trasferimento dati, e gli hypertree validi sono condizionati ad avere almeno un p_i per ciascun nodo originale p, allora il risultato è una soluzione globalmente migliore rispetto alla possibile l'etichettatura ed alla struttura dell'hypertree. Si osservi che, poiché il numero di nodi generati dall'algoritmo standard (con massima *tree-width* k fissata) è polinomiale, e di fatto il numero di siti è fissato, la nostra estensione mantiene la complessità dell'algoritmo polinomiale nel numero dei nodi candidati e nel numero delle sorgenti.

Peso dell'hypertree. La funzione di costo adottata nel nostro metodo di decomposizione ponderata deve tener conto sia dei costi di trasferimento dei dati che dell'impatto dei letterali negati sulla computazione complessiva (le selezioni sono trattate attraverso il calcolo delle statistiche). Con un piccolo abuso di notazione, indichiamo con $\sigma(h)$ il sito dove una relazione h risiede in modo nativo. Sia che p, p' definiscano un nodo dell'hypertree o una relazione, $net(\sigma(p), \sigma(p'))$ indica il costo di trasferimento unitario dei dati da $\sigma(p)$ a $\sigma(p')$ (chiaramente, $net(\sigma(p), \sigma(p)) = 0$).

La funzione di costo adottata nel nostro approccio è la seguente:

$$\begin{aligned} &\omega^S_{\mathcal{H}}(HD) = \\ &\Sigma_{p \in N} \big(est(E(p)) + \Sigma_{h_j \in \lambda(p)} |rel(h_j)| \times net(\sigma(h_j), \sigma(p)) \\ &+ \Sigma_{(p,p') \in E} \left(est^*(p, p') + est(E(p')) \times net(\sigma(p'), \sigma(p)) \right) \big) \end{aligned}$$

dove

$$est^*(p, p') = \begin{cases} est(E(p)) - est(E(p) \bowtie E(p')) & \text{se } p' \text{ è negato in } r \\ est(E(p) \bowtie E(p')) & \text{altrimenti} \end{cases}$$

In questo caso, se $\lambda(p)$ contiene soltanto una relazione h e p è una foglia in HD, $est(E(p))$ è esattamente il numero di tuple di h; altrimenti, è pari alla cardinalità dell'espressione associata a p, definita $E(p) = \bowtie_{h \in \lambda(p)} \Pi_{\chi(p)} rel(h)$. Siano R e S due relazioni, $est(R \bowtie S)$ viene calcolato come:

$$est(R \bowtie S) = \frac{est(R) \times est(S)}{\Pi_{A \in attr(R) \cap attr(S)} max\{V(A, R), V(A, S)\}}$$

dove $V(A, R)$ è la selettività dell'attributo A in R. Per join con più di una relazione si può ripetutamente applicare questa formula a coppie di relazioni secondo un dato di ordine di valutazione. Per esprimere un ordinamento nell'esecuzione di tali operazioni, utilizziamo un'euristica molto ben conosciuta e sfruttata: il criterio della selettività della join. Tale criterio, dettagliatamente descritto nel nella sezione 2.2, si basa sulla cardinalità dell'output delle operazioni di join intermedie, le cui espressioni impiegate nel modello di costo adottato sono mostrate nella Figura 2.2.7.

Valutazione della regola. Siamo ora in condizione di descrivere come avviene la valutazione di una regola r con il nostro approccio.
Dopo aver creato l'ipergrafo $\mathcal{H}_r$ per r, richiamiamo il tool *d-cost-k-decomp*, sviluppato come modulo stand-alone, che implementa il metodo di decomposizione pesata esteso come descritto sopra su $\mathcal{H}_r$, ed usando $\omega_{\mathcal{H}}^S(HD)$.

D-cost-k-decomp legge i dati quantitativi direttamente dai file, e restituisce la decomposizione, se esiste, in un file con estensione GML.

Il parametro k, relativo alla massima hypertree-width, specificato in input consente di definire le modalità di decomposizione, indicando per ogni vertice p, la cardinalità massima dell'insieme $\lambda(p)$. E' interessante notare come, avviando decomposizioni con parametri k diversi, possano scaturire differenze anche sostanziali nei costi associati alla decomposizione. Questo risultato in realtà non è sorprendente, poiché fissando valori più grandi per k si ha la possibilità di esplorare molte più decomposizioni che possono contenere anche vertici con più operazioni di join associate e quindi suscettibili anch'essi di investigazioni più ampie.

Nel caso in cui non sia possibile individuare una decomposizione che soddisfi i criteri impostati, viene eseguito il programma logico originario, senza alcuna ottimizzazione. L'esecuzione avviene per mezzo del *DLV Wrapper*, descritto dettagliatamente nella Sezione 3.2, che consente di invocare DLV (nel caso specifico DLV^{DB}) da Java.

Qualora, invece, il processo si completi con successo, viene letto il file *GML* frutto della decomposizione, in base al quale viene istanziato un oggetto Hypertree corrispondente all'albero di costo minimo risultante dal processo. Infatti, in accordo alle modifiche apportate al tool di decomposizione, l'output di *d-cost-k-decomp* è un insieme di n hypertree - dove n è il numero di siti coinvolti nella query - avente ciascuno la radice in un sito diverso, fra i quali viene quindi selezionato l'albero di costo minore.

La decomposizione può essere interpretata come un vero e proprio piano di esecuzione per la query, e consentirà di definire la struttura dei sotto-programmi e la loro sincronizzazione.

Un ulteriore raffinamento nel processo di ottimizzazione, viene effettuato dal metodo *optimize* che, attraversando ricorsivamente l'hypertree, "taglia" dall'insieme χ le variabili di ogni nodo non richieste al passo successivo. In particolare, risalendo l'albero dalle foglie alla radice si analizzano tutte le variabili contenute nell'etichetta χ e si verifica se siano presenti nell'insieme χ del nodo padre. Le variabili non richieste vengono eliminate. Le variabili restanti in χ corrisponderanno alle variabili nella testa del sotto-programma innestato nel nodo.

4.3.3 Creazione ed esecuzione del SubPrograms Tree

Il processo di riscrittura del programma logico in termini di sotto-programmi da eseguire su ciascuna sorgente, è sicuramente il passo più delicato dell'intera operazione di ottimizzazione. Tale operazione, descritta dall'algoritmo 4.3.4 viene effettuata attraverso un procedimento di visita ricorsiva dell'hypertree ed effettua la creazione di un albero di sotto-programmi.

Algoritmo 4.3.4. [Creazione SubPrograms Tree]

Function *Create SubProgramsTree*(Hypertree Node p)
begin
if p è una foglia e $\lambda(p)$ contiene soltanto una relazione h **then**
  if p ha un nodo padre p'**then** effettua la proiezione di h su $\chi(p')$
  Crea un nodo n nel *SubProgramsTree* contenente h_p in $\sigma(p)$
  return n
else if p è una foglia e $\lambda(p)$ contiene relazioni $b_1, \cdots, b_k$ **then**
  if p ha un nodo padre p'**then** effettua la proiezione di h su $\chi(p')$
  Indica di trasferire ciascuna b_i non in $\sigma(p)$ con la clausola USE di DLV^{DB}
  Crea un nodo n nel *SubProgramsTree* contenente la regola $r_p = h_p$:$-b_1, \cdots, b_k$
  return n
else
  if p ha un nodo padre p'**then** effettua la proiezione di h su $\chi(p')$
  Indica di trasferire ciascuna $b_i \subseteq \lambda(p)$ non in $\sigma(p)$ con la clausola USE di DLV^{DB}
  Crea un nodo n nel *SubProgramsTree* contenente la regola $r_p = h_p$:$-b_1, \cdots, b_k$
  for each $p'_i \in \{p'_1 \cdots, p'_m\}$ che è un nodo figlio di p **do**
    Crea nel *SubProgramsTree* un nodo figlio $n' =$ *CreateSubProgramsTree(p'_i)*
    Indica di trasferire ciascuna h_p di n' non in $\sigma(p)$ con la clausola USE di DLV^{DB}
    Aggiunge h_p di n' al body di r_p ;
  endfor
  return n
end else
end Function

□

Partendo dalla radice dell'hyprtree, valutiamo, per ogni nodo, i sotto-programmi da costruire in base alle seguenti condizioni:

- il nodo corrente è un nodo foglia: è necessario valutare la cardinalità dell'insieme λ associato al nodo; nel caso di un unico atomo contenuto, esso dovrà essere processato nel programma del nodo padre, in caso contrario, sarà trasferito il risultato ottenuto proiettando la join degli atomi contenuti in λ sulle variabili χ, attraverso la creazione e l'esecuzione di un ulteriore sotto-programma;

- il nodo corrente è un nodo interno: è necessario costruire un nuovo sotto-programma dagli atomi contenuti nel nodo corrente (o più semplicemente dall'atomo associato al nodo se il nodo contiene esattamente un atomo) e dai risultati dei figli (come specificato al passo precedente).

La creazione dei sotto-programmi sui figli del nodo è ottenuta tramite invocazione ricorsiva della procedura.

A partire dall'hypertree generiamo quindi un albero corrispondente, un *SubPrograms Tree*, nel quale ciascun nodo è un sotto-programma. Più precisamente, per ogni nodo non foglia dell'hypertree, creiamo un vertice nell'albero dei programmi, vale a dire un sotto-programma che risolve il relativo nodo dell'hypertree, compresi i figli diretti.

Ogni sotto-programma è composto da una regola logica, avente un predicato nella testa di nome *prefix*, dove *prefix* è l'identificativo del nodo corrispondente dell'hypertree preceduto dal carattere "V", le cui variabili coincidono con quelle contenute nell'insieme χ dello stesso nodo. Il corpo della regola è dato dalla congiunzione degli atomi presenti nel nodo corrispondente, degli atomi presenti nei nodi figli che siano foglie aventi un solo atomo ed infine dei predicati risultanti dai sotto-programmi di tutti gli altri figli.

La valutazione dei join associati a ciascun nodo è intenzionalmente demandata a DLV^{DB} che, interagendo direttamente con i DBMS, può sfruttarne i processi di ottimizzazione proprietari.

I sotto-programmi vengono processati ricorsivamente partendo dalla root del *SubPrograms Tree*, ed eseguiti da DLV^{DB} all'interno di thread che, adeguatamente sincronizzati, consentono un'esecuzione parallela dei sotto-programmi "risolti".

Nello specifico, ciascun thread associato ad un nodo del *SubPrograms Tree* ed al corrispondente sotto-programma, istanzia altri thread, uno per ogni figlio del *SubPrograms Tree*, per l'esecuzione di DLV^{DB}. Ogni thread, prima di poter eseguire il proprio sotto-programma, attende la terminazione del processo nei thread figli.

L'algoritmo ricorsivo garantisce l'effettiva esecuzione bottom-up dell'albero dei programmi, partendo dai sotto-programmi le cui relazioni sono già note.

4.3.4 Ottimizzazione "bulk copy"

La nostra strategia di ottimizzazione è ulteriormente potenziata dall'utilizzo di tecniche efficienti di trasferimento dei dati. L'osservazione, infatti, che le operazioni di copia tra diverse sorgenti rappresenta un fattore cruciale nella valutazione di query distribuite, ci ha condotto allo sviluppo di un modulo di copia delle tabelle che utilizza funzionalità "bulk". La copia "bulk" consiste in una fase di esportazione dei dati da una tabella in un file ed una fase di caricamento dei dati dal file a una tabella (importazione).

Ciascun DBMS possiede le proprie funzionalità "bulk" che realizza attraverso specifiche operazioni, per cui è stato necessario analizzare i diversi sistemi e le peculiarità proprie di ciascuno.

Il modulo, sviluppato come componente stand-alone e pertanto utilizzabile anche al di fuori del contesto per cui è stato progettato, consente di trasferire intere tabelle nonché proiezioni e/o selezioni delle stesse tra server locali e remoti e DBMS eterogenei. A questo scopo grande attenzione ha richiesto la gestione dei tipi di dato che in sorgenti eterogenee è il presupposto di una corretta definizione della struttura. Durante la fase di esportazione, la tabella (o una sua proiezione/selezione) viene salvata sotto forma di file di testo attraverso la particolare funzionalità messa a disposizione dal DBMS coinvolto.

Qualora il DBMS si trovi su un diverso server viene stabilita una sessione remota attraverso il protocollo SSH. Il file creato viene compresso e trasferito sul server di destinazione, sul quale avviene l'importazione. Allo stesso modo

queste funzioni, qualora coinvolgano server remoti, vengono effettuate attraverso sessioni SSH.

L'utilizzo della funzionalità "bulk" è opzionale e selezionabile attraverso l'esistenza di un file denominato *bulk.properties* nel quale devono essere specificate informazioni essenziali alla regolare esecuzione. Per ogni database coinvolto nel processo devono essere specificate un certo numero di chiavi, di seguito descritte.
$< serverName >$ è il nome ovvero l'indirizzo ip del server su cui si trova il database;
$< userServer >$ è l'account utente su serverName;
$< pswServer >$ è la password dell'utente userServer;
$< port >$ è la porta utilizzata dal protocollo SSH;
$< dir >$ è la directory su serverName dove vengono scritti i file temporanei e su cui l'utente userServer deve avere i diritti di lettura/scrittura;
$< localHost >$ è un booleano che indica se serverName è localhost.
La possibilità di gestire il trasferimento delle tabelle senza far ricorso alle operazioni di copia proprie del sistema DLV^{DB} consente di effettuare trasferimenti più veloci, particolarmente significativi nel caso di trasferimenti tra server remoti.

4.3.5 Ottimizzazione "caching"

Il meccanismo di valutazione *regola per regola* può avere la negativa conseguenza di effettuare più volte il trasferimento degli stessi dati. Ciò avviene in particolare quando esistono più regole aventi gli stessi atomi nel corpo ed è intensificata dalle preliminari operazioni di unfolding.

Per ovviare a tale svantaggio, abbiamo implementato un gestore delle copie che mantiene in una struttura tabellare informazioni su ogni operazione di copia effettuata e che prima di trasferire una tabella (ovvero una sua selezione e/o proiezione) verifica che non esista già sul sito di destinazione, ovvero in altro sito avente costi di trasferimento inferiori. La tabella di origine viene selezionata quindi dal sito avente costo di trasferimento minimo.

Tale tecnica di "caching" delle tabelle, attiva a prescindere dalla "bulk copy", mantiene copia delle tabelle per eventuali successivi utilizzi in altre parti del programma fino al termine dell'esecuzione.

Capitolo 5

Letteratura correlata

Nella prima parte del capitolo illustriamo i concetti fondamentali relativi all'ottimizzazione di query distribuite analizzando nel dettaglio le problematiche relative ai costi di esecuzione del processo locale e della trasmissione dei dati. Descriviamo inoltre i più diffusi algoritmi di ottimizzazione di query distribuite implementati nei DDBMS. Di seguito descriviamo il contesto in cui si colloca il lavoro di tesi. In particolare delineamo lo scenario, fortemente investigato e che coinvolge diverse aree di ricerca, dell'integrazione dati, in cui si inseriscono strumenti e sistemi in grado di combinare tra di loro dati provenienti da sorgenti eterogenee. In particolare discutiamo delle diverse soluzioni e delle caratteristiche architetturali dei diversi approcci. Infine presentiamo alcuni approcci alla parallelizzazione di programmi logici, discutendone limiti e potenzialità. Lo studio di tali metodologie ha contribuito all'implementazione di tecniche di valutazione parallela nel nostro sistema.

5.1 Ottimizzazione di query distribuite

In generale le interrogazioni su database distribuiti richiedono uno scambio di dati tra i siti.

L'ottimizzazione di query distribuite punta a minimizzare il costo del processo locale e della trasmissione dei dati. Un importante criterio di questa ottimizzazione consiste nell'effettuare una grande quantità di computazione su un sito prima di trasferire i dati tra i siti, con un approccio set-oriented. Un altro importante criterio afferma che le condizioni di selezione devono essere valutate prima possibile per ridurre la dimensione delle relazioni intermedie prodotte dalla computazione.

Come risultato dell'ottimizzazione di una query distribuita, il sistema produce un *query execution plan*, che consiste in più programmi locali (i programmi che devono essere eseguiti su ciascun sito), nella loro sincronizzazione attraverso messaggi e nella trasmissione dei dati richiesti tra i siti.

E' importante per gli ottimizzatori avere un piano di costo effettivo dell'esecuzione di una query. A causa degli alti costi di valutazione le operazioni di join sono il principale obiettivo degli ottimizzatori di query. Nel caso di query introdotte dall'utente interattivamente, esse coinvolgono generalmente poche

relazioni. In questo caso può essere eseguita una ricerca esaustiva, possibilmente migliorata con tecniche di pruning che escludono candidati non buoni (con meno di cinque o sei relazioni questa tecnica è plausibile).

L'input del problema è un grafo di query, che consta di nodi (tutte le relazioni che devono essere in join) e di archi (tutti i join). Gli archi sono etichettati con il predicato di join e la selettività del join. Il predicato di join mappa le tuple dal prodotto cartesiano dei nodi adiacenti a [vero, falso], in base a quali tuple sono incluse nel risultato. La selettività del join è il rapporto tuple incluse/totali. Il prodotto cartesiano può essere considerato una operazione di join con predicato uguale a vero e selettività uguale a 1. Lo spazio di ricerca (o spazio delle soluzioni) è l'insieme di tutti i piani di valutazione che calcolano lo stesso risultato.

Un punto nello spazio delle soluzioni è un particolare piano, soluzione del problema. Una soluzione è descritta da un *processing tree* per valutare l'espressione di join. Ogni punto dello spazio delle soluzioni ha un costo associato, una funzione di costo mappa gli alberi ad i rispettivi costi.

Il *processing tree* è esso stesso un albero binario che ha le relazioni base come sue foglie e le operazioni di join come suoi nodi; gli archi denotano il flusso dei dati che va dalle foglie alla radice dell'albero.

L'obiettivo dell'ottimizzazione è trovare il punto nello spazio delle soluzioni con il costo più basso possibile (minimo globale). La combinatoria esplosione rende ineffettuabile l'esaustiva enumerazione di tutte le possibili soluzioni e la caratteristica NP-hard del problema implica che presumibilmente non esiste un algoritmo veloce per cui si utilizzano delle euristiche che calcolano approssimativamente il risultato.

Lo spazio delle soluzioni è quindi l'insieme di tutti gli alberi di processo. Poiché l'operazione di join è commutativa ed associativa, il numero dei processing tree aumenta velocemente con l'aumentare delle relazioni coinvolte nelle espressioni di join. Un sottoinsieme dello spazio delle soluzioni è il cosiddetto left-deep tree dove la relazione interna di ciascun join è una relazione di base. Per un fissato numero di relazioni di base, la forma dell'albero non può essere libera, ma ci sono n! modi di allocare n relazioni di base nelle foglie dell'albero.

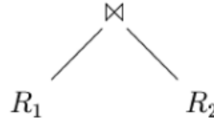
Nei general (bushy) tree non ci sono restrizioni alla forma dell'albero, di conseguenza lo spazio delle soluzioni con left-dept è in sottoinsieme di questo spazio delle soluzioni. La cardinalità può essere molto più grande: per n relazioni di base ci sono $\left(\frac{2(n-1)}{n-1}\right)(n-1)!$ soluzioni possibili.

Il principale fattore che influenza le performance è la dimensione delle relazioni intermedie che vengono prodotte durante l'esecuzione.

Quando un'operazione seguente si trova su un sito differente, la relazione intermedia deve essere trasferita attraverso la rete. Risulta quindi di primaria importanza valutare la dimensione dei risultati intermedi per minimizzare la dimensione dei dati da trasferire.

La stima si basa su informazioni statistiche sulle relazioni di base e formule che predicono la cardinalità dei risultati delle operazioni.

Esistono diversi modelli di costo: ciascuno misura il costo come numero di tuple che devono essere processate. Essi devono includere funzioni di costo che predìcano i costi degli operatori nonché statistiche sul database e formule per calcolare la dimensione dei risultati intermedi anche se l'assoluta accuratezza del calcolo del costo può non essere raggiunta.



La funzione di costo associa il costo dei join alla radice dell'albero e calcola ricorsivamente il costo come la somma dei costi dei nodi figli:

$$C_{total} = C\left(R_1\right) + C\left(R_2\right) + C(R_1 \bowtie R_2)$$

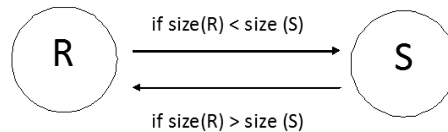
Dove C(R$_k$) è ottenuto

$$C\left(R_k\right) := \begin{cases} 0 & se\ R_k \grave{e}\ una\ relazione\ di\ base \\ C(R_1 \bowtie R_2) & se\ R_k = R_i \bowtie R_j\ \grave{e}\ un\ risultato\ intermedio \end{cases}$$

Il più semplice modello di costo calcola $C\left(R_1 \bowtie R_2\right) = |R_1 \bowtie R_2|$
Se l'ordinamento dei join è un importante aspetto nell'ottimizzazioni di query centralizzate, diventa fondamentale in un contesto distribuito, a causa dei costi di comunicazione.
Esistono due approcci di base per ordinare i join nelle query distribuite:

- Sostituzione dei join con combinazioni di semijoin per minimizzare i costi di comunicazione.
- Ottimizzazione diretta dell'ordinamento dei join (come nell'algoritmo Distributed INGRES [97, 98]).

Sia una query R ⋈ S, dove R ed S sono relazioni memorizzate in differenti siti e ⋈ denota l'operatore di join. L'ovvia scelta è quella di inviare la più piccola relazione al sito dove è memorizzata la relazione più grande.



Più interessante è il caso in cui ci sono più di due relazioni in join. L'obiettivo dell'algoritmo di ordinamento dei join è trasmettere i più piccoli operandi.
La difficoltà maggiore consiste nel fatto che le operazioni di join possono ridurre o incrementare la dimensione dei risultati intermedi: stimare la dimensione dei risultati dell'operazione di join, per quanto difficile, risulta fondamentale.
Trasferire l'intera relazione può significare trasferire anche tuple non necessarie. Il semijoin riduce la dimensione delle relazioni che devono essere trasferite.

$$\text{R} \bowtie_A \text{S} \Leftrightarrow (\text{R} \ltimes_A \text{S}) \bowtie_A \text{S} \Leftrightarrow \text{R} \bowtie_A (\text{S} \ltimes_A \text{R}) \Leftrightarrow (\text{R} \ltimes_A \text{S}) \bowtie_A (\text{S} \ltimes_A \text{R})$$

Il semijoin [77] è vantaggioso se il costo di produrlo ed inviarlo ad un altro sito è inferiore al costo di inviare l'intera relazione.
La sequenza di semijoin è chiamata *semijoin program.*

Per una data relazione, il numero di semi-join programs è esponenziale al numero delle relazioni, anche se soltanto uno è ottimale, il *full reducer*.

L'ottimizzatore globale deve trovare la strategia migliore per l'esecuzione della query, un insieme di operazioni relazionali e di primitive di comunicazione "send" e "receive" tra i siti. Permutando l'ordine delle operazoni su un frammento è possibile individuare delle query equivalenti, ciascuna con un proprio costo, tra le quali l'ottimizzatore deve scegliere quella a costo minimo.

Per semplificare, molti DDBMS trascurano i diversi costi legati all'elaborazione locale per evidenziare solo i costi di comunicazione, questo può essere corretto solo nel caso di reti metropolitane e geografiche con banda limitata.

La strategia viene scelta utilizzando informazioni di tipo statistico sui frammenti e sulla cardinalità delle operazioni relazionali, da queste informazioni dipende la "bontà" dell'ottimizzazione.

Una query può essere ottimizzata staticamente, cioè prima dell'esecuzione, oppure dinamicamente durante l'esecuzione.

I due algoritmi più diffusi di **ottimizzazione centralizzata** sono implementati in due DBMS:

- INGRES: utilizza un algoritmo di dynamic optimization (denominato INGRES-QOA) che ricorsivamente divide le query in pezzi di dimensione minore (utilizzando selezioni e proiezioni) fino a che tutte le query diventano monovariabile. Il risultato della query monovariabile è memorizzata in una struttura, per essere poi utilizzata nell'ottimizzazione successiva.

- System R: basato su un algoritmo static optimization (R-QOA), in input riceve il relational algebra tree, l'output è una strategia di esecuzione che implementa l'albero "ottimo". Per prima cosa si determina il metodo di accesso migliore per ogni relazione basata sul select, poi si calcola il miglior ordinamento per i join.

Vengono di seguito descritti i più importanti algoritmi di **ottimizzazione distribuita**, alcuni dei quali sono ricavati da estensioni dei precedenti.

5.1.1 Distributed INGRES Algorithm

L'algoritmo D-INGRES [97, 98] deriva direttamente dalla versione non distribuita INGRES. Ottimizza dinamicamente le query, cercando di minimizzare i costi di comunicazione e il tempo di risposta; poichè questi due obiettivi sono talvolta in contrasto tra loro occorre dare un "peso" all'obiettivo ritenuto prioritario. Sono considerate anche le reti broadcast per cui l'algoritmo deve poter distinguere la topologia della rete in cui viene eseguito per effettuare correttamente il calcolo dei costi di comunicazione; nel caso broadcast un sito invia i dati a tutti gli altri in un singolo trasferimento, per massimizzare il grado di parallelismo.

L'algoritmo descritto viene eseguito dal sito, chiamato *master site*, in cui la query viene avviata.

- Tutte le query monorelazione (selezione e proiezione) che possono essere isolate vengono prima processate localmente [Step (1)]

- L'algoritmo di riduzione (REDUCE) è applicato alla query originaria [Step (2)]. (La riduzione è la tecnica che isola tutte le sottoquery irriducibili

e le sottoquery monorelazione). Le query monorelazioni sono ignorate poiché sono state già processate allo step (1). Così la REDUCE produce una sequenza di sottoquery irriducibili $q_1 \rightarrow q_2 \rightarrow ... \rightarrow q_n$, con al massimo una relazione in comune tra due sottoquery consecutive.

- Allo step (3.1), basandosi sulla lista delle query irriducibili isolate nello step (2) e sulla dimensione di ciascun frammento, viene scelta la successiva sottoquery MRQ', che ha al massimo 2 variabili, e vengono applicati gli steps (3.2), (3.3), e (3.4).
- (Step 3.2) Si seleziona la migliore strategia per processare la query MRQ'. Questa strategia è descritta da una lista di coppie (F, S), nelle quali F è un frammento da trasferire al processing site S.
- (Step 3.3) Si trasferiscono tutti i frammenti su i loro processing sites.
- (Step 3.4) Viene eseguita MRQ'.
- Se ci sono rimanenti sottoquery, l'algoritmo esegue di nuovo lo step (3) ed esegue la successiva iterazione. Altrimenti, l'algoritmo termina. L'ottimizzazione viene effettuata negli steps (3.1) e (3.2). L'algoritmo ha prodotto sottoquery con diversi componenti ed il loro ordine di dipendenza (simile ad un albero di algebra relazionale).
- Allo step (3.1) una semplice scelta per la successiva sottoquery è data dallo scegliere il successivo non predecessore e che coinvolge i più piccoli frammenti. Questo minimizza la dimensione dei risultati intermedi.

  Es., se una query q ha le sottoquery q_1, q_2, e q_3, con le dipendenze $q_1 \rightarrow q_3$, $q_2 \rightarrow q_3$, e se i frammenti riferiti a q_1 sono più piccoli di quelli riferiti a q_2, allora è selezionata q_1.

  La sottoquery selezionata deve essere eseguita. Poichè la relazione coinvolta in una sottoquery potrebbe essere memorizzata in un sito differente ed anche frammentata, la sottoquery potrebbe essere ulteriormente suddivisa.
- Allo step (3.2), il successivo problema di ottimizzazione è determinare come eseguire la sottoquery selezionando i frammenti che devono essere trasferiti ed i siti dove devono essere processate. Per una sottoquery che coinvolge n relazioni, i frammenti di n-1 relazioni devono essere spostati al sito/i della relazione rimanente, e quindi replicati lì. Anche le rimanenti relazioni possono essere ulteriormente partizionate in k frammenti "equalizzati" per incrementare il parallelismo.

La selezione della relazione rimanente e del numero dei processing sites k su cui dovrebbe essere partizionata è basata su una funzione obiettivo e sulla topologia della rete utilizzando un metodo denominato fragment-and-replicate. (La replica è più vantaggiosa in reti broadcast che in reti punto-a-punto).

Si considerino le regole per minimizzare il costo di comunicazione, dati

- R_1, R_2,..., R_n: *n relations (variabili)*
- R_i^j il frammento di R_i *memorizzato al sito j*
- CC_k(#bytes) il costo di comunicazione per trasferire *#bytes a k siti, $1 \leq k \leq m$, dove m è il numero di siti coinvolti nella query.*

In una rete broadcasting, dal momento che $CC_k(\#bytes)=CC_1(\#bytes)$, la regola per selezionare R_p e k è:

Se $max_{j-1,m}(\sum_{i-1}^{n} size(R_i^j)) > max_{j-1,n}(size(R_i))$

- Allora il processing site è j con il maggior quantitativo di dati
- Altrimenti R_p è la relazione più grande ed i sti di R_p sono i processing sites.

In una rete punto a punto, dal momento che $CC_k(\#bytes)=k*CC_1(\#bytes)$

data R_p la più grande tra le $R_{i,}$la scelta di k (il numero di processing sites) è data da (assumento che i siti siano ordinati in ordine decrescente per quantità di dati per la query):

Se $\sum_{i \neq p} \left(size\left(R_i\right) - size\left(R_i^1\right)\right) > size(R_p^1)$

- Allora k=1 (è il migliore che si possa trovare)
- Altrimenti k è il massimo j tale che $\sum_{i \neq p} \left(size\left(R_i\right) - size\left(R_i^j\right)\right) \leq size(R_p^j)$

In sostanza, un sito è selezionato come processing site solo se la quantità di dati che deve ricevere è inferiore a quella che dovrebbe inviare se non fosse il processing site.

5.1.2 R* algorithm

Questo algoritmo [99] è statico, per cui la ricerca viene effettuata su tutte le possibili soluzioni, non essendo il tempo di ottimizzazione critico come nel caso dinamico. L'overhead introdotto in questa fase viene rapidamente ammortizzato se la query è eseguita frequentemente. La compilazione delle query è coordinata da un master che è il sito in cui la query inizia. L'ottimizzatore sul master prende tutte le decisioni riguardanti la query, quali l'algoritmo di join (nested loop o merge join), il tipo di accesso a ciascuna tabella (se con indici o no), l'ordine in cui i join devono essere eseguiti, i siti su cui ciascun join deve essere eseguito ed il metodo di trasferimento dei dati; i siti che hanno relazioni coinvolte nella query prendono unicamente le decisioni locali. E'basato su statistiche del database, informazioni sul tipo di accesso, e stima delle relazioni intermedie.

Di seguito descriviamo brevemente R*-QOA:

- Per ogni relazione in ingresso si determina il metodo di accesso a costo minimo;
- Quindi si esaminano tutti le possibili permutazioni dei join (per n relazioni ci sono n! permutazioni), e si sceglie la miglior strategia di accesso della query, questo passo viene effettuato costruendo dinamicamente un albero contenente le permutazioni che poi si percorre fino a trovarne una di costo minimo;
- Si sceglie il sito che dovrà contenere il risultato ed il metodo per trasferirvi i dati;
- Infine, si inviano i dati al sito remoto, il quale utilizza una strategia calcolata localmente.

I primi due passi vengono eseguiti utilizzando i dati statistici sul database.

Un algoritmo di questo tipo coinvolge ricerche su permutazioni, ed e un problema di ottimizzazione combinatoria che coinvolge un elevato numero di relazioni. Per ridurre il numero di alternative R* utilizza tecniche di programmazione dinamica che permettono di costruire l' albero delle alternative eliminando le scelte inefficienti (in modo simile agli algoritmi branch-and-bound).

5.1.3 SDD-1 algorithm

Questo algoritmo [100], nato esclusivamente per l'ottimizzazione delle query distribuite, è stato implementato nel primo DDBMS operativo (SDD-1) sviluppato dalla Computer Corporation of America (CCA) intorno al 1979-80 su macchine DEC, connesse in rete ARPANET (WAN 10kbit/sec), e operanti localmente su un DBMS preesistente. La limitata banda della rete considerata porta a considerare unicamente i costi di trasmissione, e conseguentemente a operare con semijoin al fine di ridurre la quantità di dati da trasmettere.

E' basato su Hill-Climbing Algorithms, algoritmi di tipo "greedy" che, data una soluzione ammissibile, tentano di migliorarla.

L'idea che sta alla base dell'algoritmo di ottimizzazione di SDD-1 è considerare i *costi-benefici* di ciascun possibile semijoin, e quindi di ordinare gli stessi sulla base di tali quantità definiti come segue:

Si consideri il join R_1 (@s_1) $\bowtie$ R_2 (@s_2) sul campo J, da produrre al sito s_2, il cui costo di trasmissione è pari a:

$$T_{MSG} + len(R_1)*card(R_1)*T_{byte}$$

dove T_{MSG} è il costo di trasmissione del messaggio e T_{byte} è il costo di trasmissione di un byte.

Il *beneficio* del semijoin (indicato con $\ltimes$) $R_1 \ltimes R_2$è dato dal (mancato) costo di trasferimento delle tuple di R_1 che non contribuiscono al risultato del join, ovvero:

$$benefit(R_1 \ltimes R_2) = (1 - f_{\ltimes(R1,R2,J)})*len(R_1)*card(R_1)*T_{byte}$$

in cui si fa uso del *fattore di selettività del semijoin*, definito come:

$$f_{\ltimes(R1,R2,J)} = card(R_1 \ltimes R_2) \ / \ card(R_1)$$

Il *costo* del semijoin $R_1 \ltimes R_2$ è pari a:

$$cost(R_1 \ltimes R_2) = T_{MSG} + len(R_2.J)*card(R_2.J)*T_{byte}$$

in quanto è necessario trasmettere a R_1 i valori di R_2.J.

La soluzione è uno schema di esecuzione globale, comprendente le comunicazioni tra siti. E' ottenuta calcolando i costi di tutte le strategie di esecuzione che trasferiscono tutte le relazioni richieste verso i siti candidati, tra le quali viene scelta quella a costo minimo. Il difetto è quello di trovare soluzioni ottime locali che non lo sono globalmente.

Nel SDD-1 si usa ampiamente il semijoin, volendo minimizzare i costi di comunicazione, e non si considerano i costi locali e i tempi di risposta.

L' algoritmo è statico e richiede statistiche sul database ("database profiles"), ciascun profile è associato ad una relazione.

L' algoritmo si compone dei seguenti passi:

1. Inserisce tutti i processi locali nel ES (“excution strategy”).
2. Riporta gli effetti dei processi locali nel database profile.
3. Costruisce uninsieme BS (“beneficial semijoin”) dei semijoin più vantaggiosi (con i costi minori dei benefici) nel modo seguente:

$$BS = \varnothing$$

$$Per\ ciascun\ semijoin\ SJ_i$$

$$BS\ \longleftarrow\ BS\ \cup\ SJ_i\ \ se\ cost\,(SJ_i) < benefit\,(SJ_i)$$

4. Rimuove il più vantaggioso semijoin da BS e lo aggiunge ad ES.
5. Modifica il database profile sulla base di 4.
6. Modifica appropriatamente BS
   - Ricalcola i nuovi valori costi/benefici
   - Verifica se qualche semijoin debba essere incluso in BS
7. Se BS≠∅ torna al passo 4.
8. Trova il sito ove è memorizzata la maggior quantità di dati e lo seleziona quale “assembly site”
9. Per ciascuna R_i all'assembly site, trova i semijoin di tipo (R_i semijoin R_j) dove il costo totale di ES senza questo semijoin è inferiore al costo con questo, e rimuove il semijoin da ES
10. Permuta l'ordine dei semijoin se questo migliora il costo totale.

5.1.4 AHY algorithm

Apers, Hevner e Yao hanno proposto una famiglia di algoritmi che utilizzano i semijoins per minimizzare il tempo di risposta o il tempo totale [101]. Il metodo è basato su un insieme di algoritmi applicabile solo a “query semplici”, ossia query che interessano (dopo una elaborazione locale) solo relazioni con un attributo di join in comune che è l'output della query. Sono ignorati i costi di elaborazione locale, in quanto si assume la rete point-to-point WAN.

L'AHY processa una query applicando prima delle operazioni locali, quindi decompone la query in query semplici per le quali viene pianificata una strategia ottima utilizzando il semijoin, infine vengono integrate le diverse strategie in una comune.

Gli algoritmi proposti sono due: SERIAL per minimizzare il tempo totale e PARALLEL per minimizzare il tempo di risposta.

Il SERIAL basa il suo funzionamento sulla minimizzazione della quantità globale di dati trasmessa, cerca di spostare le piccole relazioni verso i siti in cui risiedono quelle di dimensione maggiore, e in questi siti effettuare il semijoin, questo viene fatto preparando una lista contenente le relazioni ordinate per dimensione crescente, quindi si trasferiscono le relazioni più piccole nel sito contenente la relazione successiva (di dimensione maggiore) fino a quando è

possibile (cioè finchè ci sono relazioni piccole dirette verso un sito contenente una relazione maggiore).

Il PARALLEL cerca di minimizzare la quantita di dati da trasferire serialmente per favorire la trasmissione in parallelo, le relazioni sono ordinate per dimensione crescente, viene scelta come soluzione possibile (iniziale) quella che consente di portare il maggior numero di relazioni verso un unico sito. A questo punto l' ottimizzatore sceglie tra le alternative anche le relazioni da inviare attraverso siti intermedi.

Negli ottimizzatori di solito è implementato un algoritmo che include entrambi i precedenti, poi l'ottimizzatore sceglie a quale obiettivo dare la priorità. L' algoritmo comune, denominato GENERAL procede nel modo seguente:

1. In ogni sito periferico viene processata localmente la query per ricavare la query semplice che corrisponde ad ogni attributo di join della query.

2. Ogni query semplice viene isolata ed in funzione dell'obiettivo da minimizzare si esegue l'algoritmo SERIAL o PARALLEL, si produce un insieme di strategie candidate (una per ogni query semplice).

3. Le strategie per le query semplici vengono integrate in una strategia globale, questo avviene introducendo per prime le strategie che minimizzano il parametro che si vuole ottimizzare.

4. Infine vengono eliminate le ridondanze, cioe le relazioni che sono già state trasmesse come parte di altre.

In sintesi:

- L'ottimizzazione è dinamica per il D-INGRES-QOA, statica per gli altri.

- L'obiettivo del SDD-1 e del R*-QOA è minimizzare il costo totale, mentre D-INGRES-QOA e AHY minimizzano la combinazione dei costi totali e dei tempi di risposta.

- I fattori di ottimizzazione sono:
  - le dimensioni dei messaggi per SDD-1,
  - le dimensioni e il numero dei messaggi per AHY,
  - le dimensioni e il numero dei messaggi, costi di CPU locali nel sito di esecuzione e i costi di I/O per R*-QOA,
  - la combinazione di tutti i fattori precedenti per D-INGRES-QOA.

- La topologia è assunta come point-to-point WAN da SDD-1 e AHY, mentre D-INGRES-QOA e R*-QOA lavorano sia con LAN che WAN.

- L'uso dei semijoins come tecnica di ottimizzazione è caratteristica di SDD-1 e AHY. D-INGRES-QOA e R*-QOA eseguono i join in modo simile alla versione non distribuita dei rispettivi algoritmi.

- Tutti gli algoritmi prelevano le statistiche dai dati, SDD-1 e AHY che utilizzano i semijoin richiedono un numero maggiore di informazioni. Le informazioni utilizzate sono: la cardinalità delle relazioni, il numero di valori unici per attributo, il fattore di selettività dei join, la dimensione della proiezione su ogni attributo dei join e infine la dimensione delle tuple e degli attributi.

- Solo INGRES può gestire la frammentazione, ma solo in partizioni orizzontali.

5.2 Data Integration

Negli ultimi anni, la ricerca ha dedicato molta attenzione al problema di fornire accesso trasparente a sorgenti di dati eterogenee. In questo contesto, l'esigenza principale consiste nell'integrare i dati operazionali di un'organizzazione, tipicamente distribuiti su più di una sorgente informativa e di fornire una visione omogenea degli stessi. Sebbene l'integrazione e la possibilità di accedere in maniera trasparente ai dati sembrerebbero implicare a prima vista la necessità di una centralizzazione dei dati e delle applicazioni, spesso questo approccio non è desiderabile o fattibile. Al contrario, decentrare le sorgenti informative, appare una metodologia di lavoro più naturale ed idonea: infatti, questo approccio rispecchia la struttura organizzativa di molte aziende, che sono logicamente suddivise in settori, dipartimenti, che lavorano su progetti distinti, e sono fisicamente distribuite in uffici, unità, stabilimenti ognuno dei quali gestisce i propri dati operativi. Oggi, lo sviluppo delle reti di computer consente di preservare questo approccio favorendo le metodologie di integrazione di sorgenti informative senza necessitare di una loro centralizzazione. In questo ambiente distribuito, la condivisione dei dati e l'efficienza nell'accesso dovrebbe essere migliorata dallo sviluppo di un sistema distribuito in grado di ricalcare questa struttura organizzativa, rendendo i dati di tutte le unità facilmente accessibili, e memorizzandoli in prossimità delle locazioni dove sono più frequentemente utilizzati.

L'integrazione dati dovrebbe aiutare a risolvere il problema delle cosiddette *isole di informazione*. I database sono spesso considerati come isole elettroniche remote, vale a dire posti distinti e generalmente inaccessibili. Ciò può essere il risultato di una separazione geografica, di architetture di computer incompatibili, di protocolli di comunicazione differenti, e così via: integrare i database in un framework logico unitario può risolvere questo problema.

La ricerca scientifica in ambito *data integration* ha reso disponibili numerose tecniche e sistemi noti con il generico nome di *wrapping systems*. ll termine *wrapper* indica, in questo ambito, il software che si occupa della acquisizione e trasformazione dei dati, ovvero del livello di interconnessione tra i sistemi di integrazione e le sorgenti informative eterogenee esterne. In particolare, negli ultimi anni il termine *wrapper* è stato usato per indicare software che si occupa di trasformare dati semi- o non-strutturati in dati strutturati, per estrarre dati da pagine web in formato HTML. La maggior parte dei Relational Data Base Management System (RDBMS) fornisce primitive di wrapping, e sono molti

i sistemi che permettono il collegamento a RDBMS fornendo funzionalità di estrazione e trasformazione più potenti di quelle da essi forniti.

Una delle *architetture* principali per l'integrazione dei dati prevede la figura di un *mediatore* tra le basi di dati sorgente (dette anche *sorgenti operazionali*) e l'utente finale del sistema integrato. Lo scopo del mediatore è quello di fornire all'utente finale una visione globale ed uniforme di tutte le informazioni disponibili nelle sorgenti operazionali. Generalmente, ciò viene realizzato definendo uno schema globale di una base di dati virtuale, messa a disposizione dell'utente, ed alcune regole di interdipendenza tra lo schema globale e le sorgenti operazionali (*mapping*). Un mapping ha lo scopo di definire il modo in cui i dati delle sorgenti contribuiscono a creare l'istanza dello schema globale, tenendo anche conto delle eventuali correlazioni tra i dati presenti in sorgenti diverse.

In questo contesto di sorgenti operative distribuite ma uniformemente accessibili, la risposta ad una *query* Q posta dall'utente sullo schema globale è ottenuta attraverso un processo detto di *unfolding* che ha l'obiettivo di esprimere Q solo in termini di relazioni delle sorgenti operative; in tal modo, Q può essere scomposta in tante sotto-query (una per ciascuna sorgente coinvolta) per ricomporre successivamente i risultati in accordo con lo schema globale. Tale approccio consente, in genere, di distribuire parzialmente la valutazione di Q direttamente sulle sorgenti operazionali.

Tuttavia, uno dei principali problemi nella definizione di tecniche di integrazione consiste nel fatto che i dati risultanti dall'integrazione, tipicamente, non soddisfano i vincoli definiti per esprimere importanti proprietà semantiche dell'applicazione. Ad esempio, un vincolo potrebbe essere definito dall'unicità del codice fiscale: tuttavia potrebbero esistere dati contraddittori associati allo stesso codice fiscale, provenienti da sorgenti differenti. Questo è dovuto al fatto che le sorgenti da integrare sono autonome. Pertanto, quando un utente esegue una interrogazione Q su una base di dati inconsistente, non è banale capire quale debba essere la risposta, dato che il database complessivo è in uno stato incoerente.

Per risolvere questo problema, in letteratura è stata proposta la nozione di *repair* (riparazione) di database inconsistenti. Intuitivamente, un repair è un altro database che soddisfa tutti i vincoli e differisce il meno possibile da quello originale. Dato che potrebbero esistere più repair, l'approccio tipico nella risposta alle interrogazioni è quello di calcolare quelle risposte che sono vere in tutte i possibili repair. Tali risposte sono definite in letteratura come risposte consistenti, ed il processo di calcolo è chiamato *consistent query answering.* Si noti che l'adeguatezza di un repair dipende dalle assunzioni semantiche adottate dai mapping, che hanno il compito di recuperare i dati, e dal tipo di vincoli sullo schema generale.

Anche nel contesto delle tecniche di riparazione di basi di dati le soluzioni basate sulla logica computazionale, e in particolare sull'*Answer Set Programming (o ASP)*, giocano un ruolo cruciale. Infatti, in accordo con diverse proposte nella letteratura, la riparazione di un database inconsistente può essere effettuata codificando i vincoli nello schema attraverso un opportuno programma logico, tale che il ragionamento "cauto" (cioè le conclusioni logiche che sono vere in tutti i possibili modi di riparare un database) su questo programma porti ad una risposta consistente alle interrogazioni.

In letteratura sono state discusse diverse semantiche di riparazione di basi di dati inconsistenti; tuttavia, al momento, non c'è unanimità nel considerare

una semantica migliore dell'altra ed, inoltre, le stesse semantiche proposte non hanno ancora raggiunto un grado di maturità ed un'integrazione con sistemi software per la loro valutazione tali da poter essere effettivamente utilizzate in contesti reali. E' particolarmente rilevante notare che l'applicazione di qualunque semantica di repair trasforma la query utente in un complesso programma logico (spesso disgiuntivo), il che rende estremamente più complesso il processo di *unfolding* della query utente e, pertanto, anche la sua distribuzione sulle sorgenti operazionali.

Sebbene le tecniche basate sulla logica computazionale forniscano un'elevata espressività semantica che consente di esprimere in modo piuttosto semplice e compatto le regole di repair (ed eventualmente di cambiarne la semantica senza modificare il framework di integrazione), i sistemi per la valutazione di programmi logici al momento esistenti presentano diverse carenze. Ad esempio, prevedono la valutazione dei programmi logici (e quindi delle regole di repair) in memoria centrale. Ciò costituisce una forte limitazione in un contesto di integrazione in cui le basi di dati sorgente sono usualmente molto grandi. Inoltre, i sistemi esistenti generalmente prevedono che i dati in input siano forniti in formato testuale su file, mentre i dati reali risiedono normalmente nei database delle sorgenti operazionali che godono di una propria autonomia nella definizione del formato di rappresentazione.

Il termine *data integration* individua un ambito di ricerca che ha raccolto un vasto interesse negli ultimi anni. Obiettivi rilevanti in questo settore sono la realizzazione di sistemi per l'integrazione di dati, nonché sistemi di querying integrati, ma anche di wrapping di sorgenti operazionali. Il problema dell'integrazione di sorgenti informative ha ricevuto grande attenzione dalla comunità scientifica nell'ultimo ventennio. Infatti, il lavoro di ricerca ha prodotto un gran numero di approcci volti all'integrazione di sorgenti informative caratterizzate da una grande varietà di formati e dimensioni. Tali approcci presentano una notevole eterogeneità sia nel loro grado di automatismo che nella tecnica utilizzata per realizzare l'integrazione.

Innumerevoli sono gli ostacoli ed i problemi che possono sorgere in questo ambito: la capacità di un sistema di integrazione dati nel riuscire a fornire risposte adeguate alle interrogazioni utente anche in situazioni in cui i dati integrati potrebbero risultare tra di loro inconsistenti; la diversità delle architetture su cui costruire un sistema per l'integrazione dati; la reale applicabilità di un tale sistema a casi reali.

Il modo più naturale di pensare ad un sistema di data integration consiste di una architettura *multistrato* in cui lo schema globale, il livello accessibile agli utenti, è collegato allo schema sorgente per mezzo di mapping. Il criterio con cui tali mapping sono definiti e la loro forma influiscono sulle tecniche adottate dal sistema per fornire le risposte alle query utente. Il risultato dell'analisi delle sorgenti operazionali finalizzata all'integrazione è composto tipicamente da due elementi: lo schema riconciliato e il mapping, ossia l'insieme di corrispondenze tra gli elementi presenti negli schemi sorgenti e quelli dello schema destinazione. Le funzioni di mapping vengono utilizzate ogni qualvolta si debba eseguire sui database locali una interrogazione espressa sullo schema riconciliato. Normalmente, il termine mapping è usato in maniera del tutto generica, ovvero denota sia semplici corrispondenze, a livello intensionale, tra gli elementi dello schema globale e degli schemi locali, ma anche corrispondenze complesse, a livello esten-

sionale, tra lo schema globale e gli schemi locali che stabiliscono come le istanze dello schema globale corrispondano alle istanze di quelli locali .

Il problema dell'integrazione di sorgenti informative eterogenee può essere affrontato classificando gli approcci in due grandi famiglie. Nella prima ricadono gli approcci basati *sull'integrazione di schemi semi-strutturati*, nella seconda ricadono le famiglie di approcci per *l'integrazione di schemi strutturati*.

Una tipica architettura per l'integrazione dei dati prevede la figura di un *mediatore* tra le sorgenti operazionali e l'utente finale del sistema integrato. Lo scopo del mediatore è quello di fornire all'utente finale una visione globale ed uniforme di tutte le informazioni disponibili nelle sorgenti operazionali. Tale visione unificata costituisce una vista riconciliata delle informazioni e può essere interrogata dall'utente. In altre parole, una vista riconciliata, può essere considerata come un insieme di relazioni *virtuali*, la cui estensione, cioè, non è memorizzata in nessun posto. Un sistema di interrogazione dei dati rende l'utente capace di effettuare query al fine di identificare le sorgenti informative di interesse, operare con queste in maniera esclusiva, combinare manualmente i dati provenienti da diverse sorgenti.

Il processo di integrazione riceve in input gli schemi delle sorgenti informative locali, le interrogazioni e le transazioni e produce uno schema globale, un mapping tra lo schema globale e gli schemi delle sorgenti informative locali ed, infine, un mapping delle query e delle transazioni tra la sorgente informativa globale e quella locale. In tal senso, i mapping altro non sono che regole di interdipendenza tra lo schema globale e le sorgenti operazionali, ed hanno lo scopo di definire il modo in cui i dati delle sorgenti contribuiscono a creare l'istanza dello schema globale, tenendo anche conto delle eventuali correlazioni tra i dati presenti in sorgenti diverse. Generalizzando, si può dire che un sistema di integrazione di dati eterogenei è costituito da tre componenti principali:

- uno schema globale
- uno schema delle sorgenti, comprendente lo schema di tutte le sorgenti esterne
- uno schema per il mapping tra i due schemi precedenti

Lo schema di mapping può essere specificato utilizzando diversi approcci di seguito presentati. In particolare, la maggior parte dei sistemi di integrazione di dati eterogenei è basato su una vista unificata chiamata *mediated* o *global schema* e su diversi moduli software. Molto spesso, un componente software, detto *wrapper*, si occupa di prelevare i dati dalle sorgenti e di presentarli nella forma adottata nel sistema di integrazione, può inoltre gestire l'accesso ai dati tenendo conto delle capacità delle sorgenti, delle modalità di querying, del formato di output delle sorgenti. Un componente *mediator* è quello che stabilisce come raccogliere i dati provenienti dai wrapper e come fonderli, allinearli, riconciliarli in accordo con la struttura dello schema globale.

A questo punto, il problema del query processing consiste nel trovare metodi efficienti di interrogazione dello schema globale sulla base dei dati contenuti nelle sorgenti. Gli approcci al query processing possono essere svariati in dipendenza delle soluzioni scelte per la mediazione.

Nel seguito si discutono i due principali approcci, noti come *Global-As-View* (GAV) e *Local-As-View* (LAV). Nel primo caso gli elementi dello schema globale sono rappresentati per mezzo di query espresse sullo schema sorgente. Al

contrario, nell'approccio LAV gli elementi dello schema sorgente sono definiti mediante query sullo schema globale. Entrambi gli approcci presentano vantaggi e punti di debolezza. Rispondere ad una interrogazione in un contesto GAV significa applicare delle tecniche di *unfolding* relativamente semplici, mentre aggiungere o rimuovere delle sorgenti può diventare un task oneroso per la ridefinizione dei mapping. Analogamente, le operazioni di aggiunta e rimozione di sorgenti operazionali in sistemi di tipo LAV si ottiene semplicemente, fornendo la definizione del nuovo schema o rimuovendo quella di uno schema esistente, mentre il processo di query answering potrebbe essere più complesso in presenza di informazione incompleta.

5.2.1 Mapping Global as View (GAV)

L'approccio all'integrazione dati noto come *query-centric* o *Global - As - View* (GAV) richiede che lo schema globale sia espresso in termini delle sorgenti di dati. Più precisamente, ad ogni concetto dello schema globale è associata una vista delle sorgenti sottostanti, così la semantica del concetto è specificata in termini dei dati che risiedono nelle sorgenti di partenza. La specifica del mapping tra le sorgenti e lo schema globale è focalizzata sugli elementi dello schema globale ed associa ad ogni parte dello schema globale una vista sulle sorgenti esterne.

In questo contesto, la risposta ad una query Q posta dall'utente sullo schema globale è ottenuta attraverso un processo detto di *unfolding* che ha l'obiettivo di esprimere Q solo in termini di relazioni delle sorgenti operative. In tal modo, Q può essere scomposta in tante sotto-query (una per ciascuna sorgente coinvolta) per ricomporre successivamente i risultati in accordo con lo schema globale. In altre parole, il querying viene effettuato interpretando ogni concetto globale in termini della sua definizione nelle sorgenti esterne.

Nello specifico, un algoritmo di *unfolding* funziona in modo tale da sostituire atomi che si riferiscono alle relazioni globali che appaiono nel corpo di una query q con la corrispondente vista contenuta nel mapping. Questa implementazione richiede una profonda conoscenza delle relazioni esistenti tra le sorgenti operazionali, infatti, una grossa parte del lavoro è dato dalla riformulazione della query (che è tipicamente effettuata in fase di design del sistema, quando si definiscono i mapping). Un mediatore in questo caso realizza un livello di integrazione tra l'utente finale e le sorgenti.

Per processare una query utente è sufficiente tener conto del database globale che si può ottenere valutando le viste nel mapping sul database sorgente (detto anche *retrieved global database*). Quindi, per ottenere dei risultati da una query dell'utente il sistema deve valutare la query sull'intero *retrieved global database*.

Tale approccio consente di *distribuire parzialmente* la valutazione di Q direttamente sulle sorgenti operazionali. Questo approccio, riduce l'estensibilità dello schema riconciliato poiché l'aggiunta di una nuova sorgente richiederà la modifica di tutti i concetti dello schema globale che la utilizzano. Al contrario, un beneficio è rappresentato dalla semplificazione nelle definizioni delle interrogazioni poiché per capire quali concetti degli schemi sorgente sono coinvolti sarà sufficiente sostituire ad ogni concetto dello schema globale la definizione della vista che lo definisce rispetto ai concetti sugli schemi locali. Estensioni di questo modello prevedono la possibilità di estendere il framework GAV mediante la possibilità di definire vicoli di integrità sulle relazioni globali.

Sistemi GAV basati su *unfolding* *TSIMMIS* (*Stanford-IBM Manager of Multiple Information Sources*) [102] è un progetto congiunto che fa capo alla Stanford University ed al gruppo di ricerca *Almaden IBM database research group*. La sua architettura è basata su una gerarchia di wrapper e mediatori: i *wrapper* estraggono e convertono i dati da una sorgente nel modello dati chiamato OEM (*Object Exchange Model*), mentre i *mediator* combinano ed integrano i dati esportati dai wrapper o da altri mediatori. Non è data una chiara definizione di schema globale, che è sostanzialmente costituito dall'insieme di oggetti esportati dai wrapper e dai mediatori. Per la definizione dei mediatori si utilizza un linguaggio logico chiamato MSL (*Mediator Specification Language*): si tratta di Datalog esteso per il supporto a oggetti OEM. OEM è un modello dati semi-strutturato e auto-descrittivo in cui ogni oggetto ha associata una etichetta (o *label*), un tipo per il valore dell'oggetto ed un valore (o un insieme di valori).

Le interrogazioni utente vengono poste in termini di oggetti sintetizzati dal mediatore o direttamente esportati da un wrapper. Una query si esprime in MSL o in uno specifico linguaggio di query chiamato LOREL (*Lightweight Object Repository Language*), una estensione object-oriented di SQL. Un oggetto *Mediator Specification Interpreter* (MSI) processa le query ed è costituito da diversi componenti, tra cui: il *View Expander*, che produce una pianificazione logica per l'esecuzione della query, il *Plan Generator* che converte il planning logico in una pianificazione fisica espressa rispetto alle sorgenti, e l'*Execution Engine* che si occupa dell'esecuzione della query e produce le risposte.

Il progetto *Garlic* [103], sviluppato presso l'*IBM Almaden Research Center*, è costituito da due moduli principali: un livello intermedio per il query processing ed un software per l'accesso ai dati, *Query Services & RunTime System*. Il middleware adotta un modello dei dati object-oriented basato sullo standard ODMG che consente di ottenere dati da svariate sorgenti in maniera uniforme. Lo schema globale è costituito dall'unione degli schemi locali, senza i vincoli di integrità. Il *Garlic Data Language* (GDL), che è basato sullo standard *Object Definition Language* (ODL), è usato per esportare gli oggetti globali. Ogni wrapper descrive dati presso una certa sorgente nel formato OMDG e fornisce delle descrizioni circa la capacità delle sorgenti di rispondere alle query in termini del query plan.

Non è esplicitamente definita una figura di mediatore i cui task sono implementati dai wrapper. Il *Query Services & RunTime System* produce un piano di esecuzione in cui una query utente è decomposta in un insieme di query inviate alle sorgenti, estendendo ciascun oggetto coinvolto con la relativa descrizione data dallo specifico wrapper. Ciascun wrapper trasforma le sotto-query nel linguaggio nativo della sorgente inviando le risposte all'utente.

Il sistema *Squirrel* [104], sviluppato all'università del Colorado, offre un framework GAV per l'integrazione di dati basato sulla nozione di mediatore. L'approccio iniziale di *Squirrel* all'integrazione dati non considera le viste virtuali ed adotta alcune tecniche che sfruttano la materializzazione dei dati. Una caratteristica interessante dei mediatori usati da Squirrel è la loro abilità di mantenere in maniera incrementale delle viste integrate facendo leva sulle specifiche capacità delle sorgenti operative. Più precisamente, all'avvio, il mediatore invia ai database sorgente una specifica delle informazioni aggiuntive necessarie per mantenere consistente le proprie viste ed attende dalle sorgenti questi aggiornamenti.

MOMIS [105, 106] è un sistema sviluppato presso l'Università di Milano congiuntamente con le Università di Modena e Reggio Emilia. Una delle sue caratteristiche peculiari è la tecnica semi-automatica per l'estrazione e la rappresentazione di proprietà valide in un singolo schema sorgente (relazioni intra-schema), tra schemi sorgente differenti (relazioni inter-schema), e per operazioni di *schema clustering and integration.* Tali relazioni possono essere intensionali o estensionali, entrambe definite da un designer o inferite in maniera automatica dal sistema. Il processo di integrazione è basato su un modello indipendente dalla sorgente ed object-oriented chiamato ODM_I, ed usato per modellare sorgenti dati (strutturate o semi-strutturate) in maniera uniforme. Ciascun mediatore è costituito da due moduli: il *Global Schema Builder* ed il *Query Manager.* Il primo costruisce lo schema globale, mentre il secondo è responsabile del query processing e delle funzioni di ottimizzazione.

Gestione dei vincoli di integrità. Una delle funzionalità mancanti nei sistemi appena descritti è la gestione dei vincoli di integrità espressi sullo schema globale e la capacità di lavorare in presenza di sorgenti dati incomplete o inconsistenti. Risolvere il problema del query processing mediante tecniche di *unfolding* limita l'abilità di questi sistemi per ciò che riguarda l'integrazione in scenari complessi, dove si registra la presenza di sorgenti dati incomplete e vincoli di integrità.

Il sistema *IBIS* (*Internet-Based Information System*) [107] è un tool di integrazione dati per l'integrazione semantica di dati eterogenei sviluppato dall'Università "La Sapienza" di Roma in collaborazione con CM Sistemi. IBIS adotta delle soluzioni innovative per gestire tutti gli aspetti di un ambiente di integrazione dati (source wrapping, limitazione sull'accesso alle sorgenti, query answering in presenza di vincoli di integrità). In IBIS lo schema globale è rappresentato in linguaggio relazionale e può essere combinato con una varietà di sorgenti di dati (sul Web, relational databases, legacy sources). Opportuni wrappers offrono accesso relazionale a sorgenti non relazionali. Il sistema rappresenta uno dei primi tentativi nella direzione di consentire la specifica di vincoli di integrità sullo schema sorgente. Sullo schema globale si possono specificare chiavi e chiavi esterne, sullo schema sorgente si possono specificare dipendenze funzionali e dipendenze di *full-width inclusion.*

Il query processing in IBIS è suddiviso in tre passi: *(i) query expansion* che tiene conto dei vincoli di integrità nello schema globale; *(ii) unfolding* per ottenere una query espressa sulle sorgenti; *(iii) query execution* per la valutazione dei *retrieved source databases*, e per produrre la risposta alla query originaria. Da notare che, mentre i processi di query unfolding ed execution sono gli step standard del query processing secondo l'approccio GAV, la fase di expansion fa uso di un apposito algoritmo.

Il sistema *DIS@DIS System* (*Data Integration System ad Dipartimento di Informatica e Sistemistica*) [108] è un prototipo sviluppato presso l'Università "La Sapienza" di Roma che implementa una varietà di algoritmi per l'integrazione dati in presenza di vincoli di integrità. Sia lo schema globale che lo schema sorgente sono rappresentati per mezzo del linguaggio relazionale e il mapping è espresso dall'unione di query congiuntive. Per quanto riguarda il trattamento di vincoli di integrità il sistema è capace di gestire dati incompleti e anche inconsistenti. A tal proposito il sistema implementa una semantica di tipo sound,

in cui cioè, i database globali costruiti in accordo al mapping sono interpretati come sottoinsiemi di database che specificano lo schema globale. Il query processing secondo tale semantica fornisce risposte che sono conseguenza logica dello schema globale e dell'informazione disponibile presso le sorgenti. Il query processing sotto tale semantica consente la computazione di risposte consistenti per dati incompleti ed inconsistenti.

5.2.2 Mapping Local as View (LAV)

L'approccio *source-centric* o *Local-As-View* (LAV) prevede che lo schema globale sia specificato indipendentemente dalle sorgenti sottostanti. Le relazioni tra lo schema globale e le sorgenti sono espresse associando ogni elemento delle sorgenti ad una vista sullo schema globale. In questo modo, nell'approccio LAV la semantica delle sorgenti viene specificata in termini di viste sullo schema globale. Tale approccio favorisce l'estensibilità del sistema di integrazione e fornisce un insieme più appropriato di funzionalità per la sua manutenzione. Per esempio, l'aggiunta di una nuova sorgente al sistema non implica necessariamente cambiamenti nello schema globale. La specifica del mapping tra le sorgenti e lo schema globale è focalizzata sulle sorgenti esterne, nel senso che una vista dello schema globale è associata ad ogni sorgente esterna.

Tale approccio offre due diverse soluzioni al query processing: *(i) query rewriting* che consiste nel calcolare una riscrittura delle query in base alle viste e nel valutare il risultato della riscrittura in base alle viste stesse; *(ii) query answering* nel quale si punta ad ottenere una risposta diretta alle query basandosi sulle estensioni delle *viste*.

L'approccio LAV richiede trasformazioni complesse per capire quali elementi degli schemi sorgente devono essere presi in considerazione per ricreare il concetto espresso nello schema globale. Uno dei modi più utilizzati per gestire il query processing in approcci di tipo LAV è per mezzo del cosiddetto *query rewriting.* Data in input una query q ed un insieme di viste definite su uno schema di database, il processo di riscrittura si occupa di produrre una nuova query, detta *rewriting* di q, la cui valutazione sulle viste fornisce le risposte della query originaria. Infatti, il processo è suddiviso in due fasi: uno di *riformulazione* della query ed uno di *valutazione.* Da notare che tale processo di riscrittura non tiene conto delle estensioni delle viste, ma solo del linguaggio utilizzato per esprimere la query e le viste. Un approccio più generale chiamato *query answering using views* tiene invece conto della query ma anche delle viste e dei dati presso le sorgenti.

Di seguito si riassume il comportamento di alcuni noti algoritmi usati nei sistemi di integrazione di tipo LAV: il *bucket algorithm* ed il *MiniCon algorithm.* Un ultimo esempio è costituito dall'*inverse rules algorithm*, che è una delle tecniche più adottate in sistemi di integrazione LAV.

L'algoritmo noto come *bucket algorithm* è un algoritmo di riscrittura che si applica nel caso in cui la query è una unione di query congiuntive e le viste sono query congiuntive. Per calcolare le riscritture contenute nella query originaria, l'algoritmo sfrutta una opportuna euristica per tagliare lo spazio delle possibili riscritture candidate. In particolare, gestisce la presenza di dipendenze di inclusione e funzionali sullo schema globale e le limitazioni nell'accesso alle sorgenti, ed usa query congiuntive come linguaggio per la descrizione delle sorgenti e interrogazione del sistema.

Per calcolare le risposte di una query q:

1. per ogni atomo g in q, crea un *bucket* contenente le viste da cui le tuple di g possono essere rintracciate;
2. considera come candidati ciascuna query congiuntiva ottenuta combinando una vista da ciascun *bucket* e verifica per mezzo di un algoritmo di contenimento se tale query è contenuta in q. Se questo è il caso, il *rewriting* è aggiunto alla risposta. Se, invece, la riscrittura candidata non è contenuta in q, prima di scartarla, si verifica se è possibile modificarla aggiungendo dei predicati di confronto tali che essa risulti contenuta in q. L'algoritmo è in grado di generare il *maximal contained rewriting* quando il linguaggio di query è dato dall'unione di query congiuntive.

Esistono delle ottimizzazioni dell'algoritmo che vanno sotto il nome di *MiniCon algorithm* e *shared-variable bucket algorithm*. In entrambi i casi, l'idea di base è quella di esaminare l'interazione tra le variabili della query originaria e le variabili nelle viste, con l'intento di ridurre il numero di viste inserite nei buckets e quindi il numero di rewriting candidati.

Infine, l'algoritmo noto come *inverse rules algorithm* si pone l'obiettivo di produrre un rewriting (*query plan*) in tempo polinomiale rispetto alla dimensione dell'input. Esso costruisce un insieme di regole che invertono le definizioni delle viste e consentono al sistema di effettuare un *mapping inverso* che stabilisce come ottenere i dati delle relazioni globali dai dati presenti nelle sorgenti. L'idea è quella di rimpiazzare le variabili esistenziali nel corpo di ciascuna vista per mezzo di una funzione di *skolemizzazione*. Data una query Datalog q non ricorsiva ed un insieme di viste V, la riscrittura è il programma Datalog costituito dalla query e dalle regole inverse ottenute da V. Si dimostra che tale algoritmo restituisce un *maximally contained rewriting* rispetto all'unione di query congiuntive, ed opera in tempo polinomiale rispetto alla dimensione della query.

Sebbene il costo computazionale della ricostruzione del query plan sia polinomiale, la riscrittura ottenuta potrebbe contenere regole che causano l'accesso a viste irrilevanti per la query, ed il problema di eliminare tali regole irrilevanti ha una complessità temporale esponenziale. Infine, l'algoritmo è capace di gestire query Datalog ricorsive, la presenza di dipendenze funzionali sullo schema globale, o la presenza di limitazioni nell'accesso alle sorgenti, estendendo il query plan ottenuto per mezzo di specifiche regole.

5.2.3 Nuove architetture per sistemi di integrazione dati

I recenti sviluppi nell'ambito di infrastrutture e algoritmi per sistemi distribuiti, unitamente alla rapida crescita del numero di sorgenti informative eterogenee sul Web, hanno ampiamente influenzato la ricerca nel campo della integrazione dati ed incoraggiato lo sviluppo di sistemi idonei. Oltre a sottolineare l'inadeguatezza delle soluzioni architetturali di tipo centralizzato, tali studi hanno evidenziato come in uno scenario dinamico, in cui sorgenti dati possono essere distaccate o agganciate ad un contesto pre-esistente, il design ed il mantenimento di uno schema globale e l'aggiornamento dei mapping diventano dei task particolarmente onerosi dal punto di vista computazionale. In questa sezione si presenta

una nuova direzione di sviluppo nel contesto dei sistemi per l'integrazione dati dove gli obiettivo sono il dinamismo e la flessibilità.

Sistemi di Data Management di tipo peer-to-peer (*PDMS*) Il paradigma P2P (peer-to-peer) [106, 109, 110, 111] riflette uno scenario in cui i diversi nodi di una rete sono risorse computazionali "alla pari", o nodi informativi, che cooperano tra di loro scambiandosi servizi e/o informazioni. I vantaggi di questa architettura logica sono certamente la scalabilità, la robustezza e il fatto che non necessita di una gestione amministrativa centralizzata. La bontà di questi sistemi tipicamente cresce col crescere del numero di nodi partecipanti, giacché con essi aumentano la memoria a disposizione e la potenza computazionale dell'intero sistema. Tuttavia, come noto, spesso i sistemi P2P non si preoccupano di preservare la semantica dei dati scambiati. Ciò potrebbe costituire un reale problema quando, al crescere delle dimensioni della rete, diventa sempre più difficile poter predire la locazione e la qualità dei dati forniti dal sistema.

Un sistema cosiddetto P*2P Data Management System* si pone l'obiettivo di superare proprio tali limitazioni, offrendo una semantica di interoperabilità in assenza di uno schema globale dell'informazione. Tutta la conoscenza circa la topologia e l'estensione della rete risiede presso i peer stessi. Infatti, ciascun peer è interconnesso con gli altri peer mediante formule di coordinazione che consentono a ciascun singolo nodo di sfruttare le conoscenze (*acquaintances*) che provengono dagli altri peer.

Un sistema P2P decompone una query utente applicando ricorsivamente le formule di coordinazione, che possono agire come una sorta di vincolo per la propagazione degli aggiornamenti sulla rete.

Un esempio di modello dati per questi sistemi, è quello conosciuto come *Local Relational Model* (LRM) [112, 113] che è specificamente pensato per sistemi di gestione dati P2P. Ciascun nodo ha uno schema di database locale la cui semantica è definita su un dominio locale. L'informazione si propaga tra i peer grazie alle regole di coordinazione e alle relazioni tra i domini dei diversi peer.

Tali relazioni sono in grado di esprimere *overlapping* tra i database locali di diversi peer, ad esempio, costanti differenti che rappresentano in realtà lo stesso oggetto. Le formule di coordinazione, servono perciò per fornire in maniera dichiarativa le inter-dipendenze semantiche tra diversi database locali. Non è presente una idea di consistenza globale, ma piuttosto di consistenza locale, a livello di peer.

Diversi lavori di ricerca sono stati condotti in merito alla applicabilità dell'approccio P2P in ambienti distribuiti [114]. Un interessante argomento riguarda come esprimere le interconnessioni logiche tra i peer. Ad esempio, molte risorse Web possono essere descritte come un grafo diretto in cui i nodi sono i dati del Web ed i link tra di essi sono gli archi. Una possibile formalizzazione dell'architettura dello schema dati del Web (a supporto di tecniche di query answering) prevede di poter interrogare dati Web nel momento in cui esiste un entry point per essi e se i link tra i dati possono essere usati per navigare tramite lo schema. A partire dalla query utente, la tecnica proposta produce un piano di navigazione che viene trasformato in una estensione dell'algebra relazionale tramite l'operatore *traverse* che consente di attraversare i link tra differenti dati Web. Questa soluzione fa uso del paradigma GLAV, una generalizzazione di GAV e LAV in cui i mapping sono costituiti da una coppia di query, una espressa sullo

schema globale ed una espressa sullo schema sorgente. Questo tipo di mapping GLAV definisce il limite di *tradeoff* tra la potenza espressiva e la trattabilità nel query answering.

Integrazione dati nel modello peer-to-peer In generale gli approcci descritti non si adattano a qualsivoglia topologia di interconnessione dei peer, e ciò è dovuto alla particolare semantica adottata nella definizione delle tecniche di query answering. Tuttavia, imporre delle limitazioni sulle interconnessioni dei peer non è un approccio praticabile, dal momento che in un ambiente dinamico la topologia della rete potrebbe essere essa stessa fuori dal controllo del sistema stesso.

È noto che il tentativo di poter assegnare una semantica globale ad un ambiente distribuito può condurre a indecidibilità del query answering. La soluzione in tal caso potrebbe essere quella di adottare una semantica epistemica per il sistema invece della semantica basata sulla logica del primo ordine. Sotto questa assunzione le asserzioni di mapping espresse in GLAV sono interpretate in modo che solo la conoscenza del peer è trasferita ad altri peer. In questo approccio, è possibile utilizzare algoritmi distribuiti per il query answering basati sul meccanismo delle transazioni per garantire la correttezza semantica dell'intero processo rispetto alla semantica epistemica.

Questa visione è stata estesa con l'intento di sviluppare una infrastruttura P2P di data integration, implementata come *Data Grids*. Questo nuovo sviluppo si propone di modellare un sistema P2P come un set di *data peer* e *hyper peer*. I primi sono sistemi che offrono dati in termini di uno schema esportato, mentre i secondi non esportano dati ma sono interconnessi sia con *data* che *hyper peer*, costruendo la topologia della rete. Un *hyper peer* connesso ad altri *data peer* corrisponde al modello GAV classico, e la sua semantica è definita mediante la logica epistemica. Il query answering nel *hyper framework* è effettuato suddividendo ogni mapping GLAV in due parti: una asserzione LAV ed una GAV, collegate per mezzo di un nuovo simbolo relazionale. Le asserzioni LAV sono usate per produrre un programma logico che sfrutta algoritmi per il query answering tramite l'utilizzo di viste, mentre le asserzioni GAV sono usate dal sistema per generare le estensioni su cui il programma logico può essere valutato.

Altri approcci di Data Integration Il problema noto come *what-to-ask* (*WTA*) [115] è quello di rispondere a query poste su un sistema P2P facendo affidamento solo sui servizi di query answering disponibili presso i diversi peer. In particolare, uno scenario tipico prende in considerazione due peer: un peer remoto e un peer locale. Una possibile soluzione al problema WTA (quando si utilizza un linguaggio ontologico per esprimere la base di conoscenza dei due peer) consiste nel calcolare le query che il peer locale deve sottoporre al peer remoto per poter rispondere alla interrogazione posta sul peer locale. È interessante sottolineare che, quando si arricchisce la potenza espressiva del linguaggio ontologico utilizzato per descrivere la base di conoscenza dei peer, il problema WTA potrebbe essere non più risolubile.

In generale, gli approcci fin qui discussi alla integrazione dati mirano a rispondere alle interrogazioni sfruttando tecniche di riscrittura: ovvero, i dati sono

disponibili solo presso le sorgenti ed il sistema non è in grado di materializzarli localmente (*virtual data integration*).

Un approccio diverso è fornito dalle soluzioni di *data exchange* che adottano una strategia di materializzazione dei dati remoti. In questo scenario esiste uno *schema sorgente remoto*, ad esempio, un set di sorgenti informative, ed uno *schema target*, analogo allo schema globale dei sistemi di integrazione dati, che deve essere materializzato. Il problema di data exchange consiste nel trovare una istanza dello schema target a partire da una istanza dello schema sorgente remoto. Tale problema che in generale ammette più soluzioni è stato analizzato e formalizzato anche dal punto di vista semantico e rispetto alle problematiche del query answering. In particolare, è stato definito il concetto di *soluzione universale*, ad esempio una soluzione che è omomorfa alle altre soluzioni, e quindi è la più generale. Tuttavia, dato che possono esistere più soluzioni universali, è necessario un criterio di minimalità per scegliere la migliore. In tal senso si vuole individuare una sotto-struttura comune alle soluzioni universali che può essere isomorfa a quelle. Tale struttura detta "core" si può calcolare per mezzo del noto algoritmo della teoria dei grafi ed eseguito, nello scenario del data exchange, in tempo polinomiale.

5.3 Valutazione parallela di programmi logici

Negli ultimi anni abbiamo assistito ad un progressivo sviluppo nella produzione di hardware che ha determinato l'abbandono del modello a processore singolo a favore di quello multi-processore.

Ciò ha permesso l'adozione di sistemi paralleli di tipo Symmetric Multi Processing, da sempre area di studio di accademici e grossi centri di ricerca, anche su computer desktop e portatili, determinando la diffusione su larga scala dei benefici del calcolo parallelo.

In particolare, la valutazione di programmi Datalog può essere effettuata in modo efficiente attraverso strategie di valutazione parallela in grado di trarre vantaggio da tali architetture.

I programmi logici possiedono, infatti, un intrinseco parallelismo: l'esistenza di non-determinismo, l'alto livello dei linguaggi, oltre alla loro trasparenza referenziale li rendono candidati interessanti per ottenere incrementi nella velocità attraverso l'esecuzione parallela.

Esistono diversi approcci al parallelismo dei programmi logici, caratterizzati in base al punto in cui il parallelismo viene esplicitato.

L'*OR-parallelism* deriva dall'osservazione che un obiettivo può unificare con più clausole diverse della base di conoscenza; in questo caso i corpi delle clausole possono essere eseguiti da processi differenti. Questo approccio è senza dubbio quello più utilizzato poiché più semplice dal punto di vista implementativo.

L'*AND-parallelism* consiste invece nella possibilità di risolvere parallelamente le congiunzioni nel corpo di una regola. L'AND-parallelism è quindi più difficile da ottenere a causa della consistenza dei legami tra le variabili.

L'*Unification Parallelism* scaturisce durante il processo di unificazione tra un predicato e le teste delle clausole che lo definiscono. Si tratta di un approccio di granularità molto fine, che richiede architetture specializzate per conseguire risultati di interesse e non è obiettivo di ricerca.

Sono stati proposti numerosi studi per la valutazione di programmi ASP attraverso tecniche di esecuzione parallela, alcuni dei quali pongono l'attenzione alla fase di ricerca del modello mentre altri alla fase di istanziazione del programma.

Alcuni studi preliminari rappresentano un tentativo di introdurre il parallelismo nei sistemi di ragionamento non-monotono. In particolare, in [116] viene effettuata una panoramica delle principali problematiche nell'utilizzo del parallelismo in un modello di esecuzione basato su ASP e vengono identificate due principali forme: il parallelismo orizzontale ed il parallelismo verticale, che, rispettivamente, corrispondono ai due casi di non-determinismo presenti nelle semantiche operazionali comunemente utilizzate in ASP. In [116] viene utilizzata una strategia di istanziazione parallela delle regole che assegna staticamente una regola ad ogni unità di processo e che quindi risulta poco efficace nel caso di programmi con poche regole.

La maggior parte degli algoritmi per l'istanziazione parallela dei programmi logici utilizza la tecnica *copy and constrain* per parallelizzare la valutazione delle basi di dati deduttive [117, 118, 119, 120, 121].

In molte delle opere citate (che risalgono agli anni '90), sono parallelizzabili soltanto classi limitate di programmi Datalog; mentre le più generali [118, 120] sono applicabili a programmi Datalog normali, nessuno di essi considera le peculiarità di programmi disgiuntivi e la negazione non stratificata. In particolare [118] fornisce le basi teoriche per la tecnica *copy and constrain* mentre [120] apporta dei miglioramenti in modo da minimizzare l'overhead dovuto alla comunicazione di rete nei sistemi distribuiti.

La tecnica *copy and constrain* lavora nel modo seguente: le regole vengono replicate aggiungendo a ciascuna copia dei vincoli; tali vincoli sono generati da una funzione hash e consentono la selezione di un sottoinsieme delle tuple. Le regole ottenute sono valutate in parallelo. Le funzioni di hash utilizzate per associare le regole ai processi di istanziazione sono generalmente indipendenti dall'istanza di input e potrebbero non bilanciare correttamente il carico non considerando la reale dimensione dei predicati.

Riguardo le euristiche utilizzate su database paralleli, appare interessante quanto proposto in [121] e [122]. In [121] è descritta un'euristica per bilanciare la distribuzione del carico nella valutazione parallela di PARULEL, un linguaggio simile a Datalog. Qui, il bilanciamento del carico è fatto da un server di gestione che registra i tempi di esecuzione in ogni sito, e sfrutta queste informazioni per la distribuzione del carico in base al protocollo PDLB (predictive dynamic load balancing).

In [122] le euristiche proposte sono state progettate per ridurre al minimo i costi di comunicazione e per la scelta del sito ove effettuare l'elaborazione di sub-query tra vari DBMS collegati in rete.

La tecnica proposta in [123] è stata diffusamente analizzata ed ha rappresentato l'ispirazione da cui sono stati sviluppati gli algoritmi di parallelizzazione del nostro sistema. In [123] viene descritto un algoritmo per l'istanziazione parallela di programmi logici. Esso consente di suddividere il programma in moduli, in base alle dipendenze tra i predicati, che possono essere valutati in parallelo senza l'uso di mutex-lock (minimizzando così i processi di controllo della concorrenza). La tecnica permette inoltre la valutazione parallela delle regole all'interno di ogni modulo e, attraverso una strategia semi-naïve, il parallelismo è sfruttato anche per la valutazione delle regole ricorsive.

Capitolo 6

Applicazioni ed Esperimenti

In questo capitolo presentiamo l'applicazione della nostra tecnica in diversi scenari. Il prototipo implementato è stato, infatti, adattato a contesti eterogenei, nei quali sono stati condotti gli esperimenti. Nel seguito descriviamo quindi diffusamente ciascun contesto, discutendo i risultati dell'analisi sperimentale.

6.1 Data integration

Il primo contesto cui rivolgiamo l'attenzione è quello in cui sia necessario combinare tra loro dati residenti in modo nativo su diverse sorgenti autonome e distribuite ed in cui sia opportuno implementare meccanismi di ragionamento per estrarre conoscenza dai dati e realizzare processi di inferenza, così come avviene nei sistemi di basi di dati deduttive.

In tale contesto, l'integrazione dei dati [125] gioca un ruolo sempre più rilevante. Intuitivamente, un sistema di integrazione offre un accesso trasparente alle fonti di dati esistenti e fornisce all'utente una vista unificata di dati disponibili attraverso uno schema globale. Generalmente sullo schema globale sono posti alcuni vincoli (ad esempio la chiave o vincoli di inclusione) per garantire la qualità dei dati integrati.

Quindi, il sistema di integrazione dovrebbe essere in grado di fornire risposte coerenti [126] per le query su dati eventualmente incoerenti o incompleti. Il calcolo di risposte coerenti (conosciuto anche come CQA) è a carico del motore di ragionamento, ed è stato effettivamente risolto attraverso approcci all'Answer Set Programming (ASP) [127, 126]. CQA può essere, nei casi trattabili, ridotto ad una semplice valutazione di programmi logici stratificati [8] su dati distribuiti.

Tuttavia, nonostante la dichiarativa e naturale codifica ASP di CQA, i sistemi disponibili non consentono di gestire efficientemente la valutazione su sistemi distribuiti. E' frequente, infatti, per i sistemi per CQA spostare tutti i dati sul sito su cui il motore logico risiede, prima di iniziare la valutazione. E' facilmente comprensibile come questa strategia naïve possa spesso compromettere le prestazioni. Infatti, è noto che azioni come il trasferimento di dati e/o di regole tra sorgenti diverse possano influenzare significativamente le prestazioni. Inoltre, la distribuzione naturale dei dati, tipico scenario di integrazione dei dati, potrebbe

(probabilmente) consentire valutazioni parallele di regole del programma (o loro frammenti) nei siti di origine.

Gli aspetti alla base delle interrogazioni su sistemi distribuiti di basi dati deduttive è stato ampiamente affrontato in letteratura [128, 129]; anche la valutazione distribuita di Answer Set Programming ha ricevuto una certa attenzione [130]; ed i sistemi di integrazione dati, senza CQA, usano per lo più database basati su mediatori [131].

Le tecniche proposte per la valutazione dei programmi prevedono dati memorizzati localmente ed opportunamente ripartiti tra le sorgenti (ad esempio, secondo la tecnica copy-and-constraint [128, 132]) per distribuire la valutazione, al fine di bilanciare euristicamente il carico di lavoro sulle macchine disponibili [133, 4].

Tuttavia, questi approcci di solito non considerano l'ipotesi che i dati siano distribuiti in modo nativo. Inoltre, un ruolo importante per la valutazione efficiente di una regola è svolto anche dalla analisi delle proprietà strutturali delle interrogazioni; infatti, nel caso di regole logiche esistono spesso molteplici interazioni tra le variabili dei predicati in esse contenute che potrebbero influenzare i costi di valutazione [2].

L'analisi del problema nasce dall'osservazione che una singola regola logica può essere vista come una query congiuntiva (eventualmente con negazione) il cui risultato deve essere memorizzato nel predicato della testa (della regola). I metodi strutturali [3, 134, 5, 6] permettono la trasformazione di una query congiuntiva in un albero di sub-query, consentendo valutazioni efficienti, ad esempio con il noto algoritmo Yannakakis [79].

Nonostante l'attenzione ricevuta dai metodi strutturali per l'ottimizzazione di query, l'applicazione di tali tecniche ad interrogazioni su dati nativamente distribuiti, non è stata sufficientemente studiata.

6.1.1 Esperimenti

In questa sezione presentiamo i risultati preliminari degli esperimenti effettuati con un'implementazione prototipale del nostro approccio. Abbiamo condotto gli esperimenti su dati provenienti da contesti reali ed abbiamo testato alcuni programmi logici presenti nel benchmark OpenRuleBench [135].

Abbiamo confrontato il nostro approccio con due ben noti DBMS che consentono di gestire ed effettuare interrogazioni su dati distribuiti, ovvero Oracle e SQLServer. Dal momento che siamo interessati a confrontare le prestazioni del nostro approccio con DBMS commerciali, abbiamo valutato i programmi con il nostro sistema ed il corrispondente insieme di query SQL con il DBMS.

SQLServer consente di interrogare i dati distribuiti attraverso *linked server*, mentre Oracle mette a disposizione *database link*. Nel nostro approccio DLV^{DB} è stato accoppiato sia con SQLServer che con Oracle.

Gli esperimenti sono stati eseguiti su un sistema operativo Windows 2003 Server installato su rack HP ProLiant DL120 G6 dotato di processore Intel Xeon X3430, 2.4 GHz, con 4 GB di RAM. Abbiamo impostato un limite di tempo di 120 minuti dopo i quali l'esecuzione dell'interrogazione è stata arrestata. Per ciascun test, abbiamo calcolato la media dei risultati di tre esecuzioni consecutive dopo la prima (che non è stata considerata al fine di escludere eventuali ottimizzazioni dei DBMS, come la memorizzazione nella cache). Nei grafici (vedi

P1:
`exam_record(X1,X2,Z,W,X4,X5,Y) :- dati_esami(X1,A1,X2,X5,X4,A2,Y),`
`affidamenti_ing_informatica(X2,X3,Y), dati_professori(X3,Z,W).`
`course(X1,X2) :- esame(A1,X1,X2,A2).`
`active_courses(CD):-course(C,CD), exam_record(X0,C,X1,X2,X3,X4).`

P2:
`student(X1,X2,X3,X4,X5,X6,X7) :-`
`studenteS1(X1,X3,X2,A1,A2,A3,A4,A5,A6,A7,A8,A9,X6,X5,A10,A11,`
`X4,A12,A13,A14,A15,Y,A16), diploma_maturitaS1(Y,X7).`
`exams(X1,X2) :- dati_esami(X1,A1,X2,X5,X4,A2,Y).`
`hasCommon(X1,X3) :- student(X1,X2,X3,X4,X5,X6,X7), exams(X1,C),`
`student(Y1,Y2,Y3,Y4,Y5,Y6,X7), exams(Y1,C).`

P3:
`teaching(X1,Z,W,X3) :- affidamenti_ing_informatica(X1,X2,X3),`
`dati_professori(X2,Z,W).`
`exam_record(X1,X2,Z,W,X4,X5,Y) :- dati_esami(X1,A1,X2,X5,X4,A2,Y),`
`affidamenti_ing_informatica(X2,X3,Y), dati_professori(X3,Z,W).`
`course(X1,X2) :- esame(A1,X1,X2,A2).`
`student(X1,X2,X3,X4,X5,X6,X7) :-`
`studenteS1(X1,X3,X2,A1,A2,A3,A4,A5,A6,A7,A8,A9,X6,X5,A10,A11,`
`X4,A12,A13,A14,A15,Y,A16), diploma_maturitaS1(Y,X7).`
`commonProf(M1,CCODE1,M2,CCODE2):- student(M1,A2,A3,A4,A5,A6,A7),`
`exam_record(M1,CCODE1,B3,B4,B5,B6,B7), course(CCODE1,E2),`
`exam_record(M2,CCODE2,C3,C4,C5,C6,C7), course(CCODE2,F3),`
`student(M2,D2,D3,D4,D5,D6,D7),`
`teaching(CCODE1,PFN,PLN,G4), teaching(CCODE2,PFN,PLN,H5).`

Tabella 6.1: Test su programmi logici.

figura 6.1), riportiamo i tempi di esecuzione di DLV^{DB} associato ad SQLServer (`D+S`), SQLServer (`S`), DLV^{DB} associato ad Oracle (`D+O`), e Oracle (`O`).
Test su dati provenienti dal mondo reale. Abbiamo utilizzato il framework di integrazione dati sviluppato nell'ambito del progetto Infomix (IST-2001-33570) [56], che integra i dati di un reale contesto universitario. In particolare, sono stati considerati i dati resi disponibili dall'Università degli Studi di Roma "La Sapienza". Questi comprendono informazioni su studenti, professori, programmi di studio ed esami in diverse facoltà dell'università. Nello scenario applicativo sono presenti circa 35 sorgenti, mappate in 12 schemi globali con circa 20 mapping GAV. Nel seguito definiremo **Infomix** questo insieme di dati. Inoltre, abbiamo considerato due ulteriori insiemi di dati, vale a dire **Infomix-x-10** e **Infomix-x-50**, in cui sono memorizzate, rispettivamente, 10 copie (per un totale di 160 MB di dati) e 50 copie (800Mb) del database originale.

Definiamo **Infomix**$\subset$**Infomix-x-10**$\subset$**Infomix-x-50**. Abbiamo poi distribuito i set di dati **Infomix** su 5 siti ed abbiamo confrontato i tempi di esecuzione del nostro prototipo con i due DBMS. I programmi che abbiamo testato sono riportati in tabella 6.1.
L'output del programma **P1** è `active_courses`, e definisce le descrizioni dei corsi per i quali esiste almeno un esame. Il programma **P2** ha l'obiettivo di trovare studenti aventi lo stesso diploma ed almeno un esame comune; qui il predicato di output è `hasCommon`. Il programma **P3** identifica gli studenti che hanno avuto un professore comune, quindi il focus è su `commonProf`. Si noti che questi tre programmi sono stati sottoposti ad unfolding rispetto ai predicati di output e le corrispondenti regole sono state riscritte come query SQL per essere eseguite da Oracle e da SQLServer. I risultati dei nostri esperimenti sono presentati in Figura 6.1.

Dall'analisi di tale grafico è possibile osservare come il nostro approccio con-

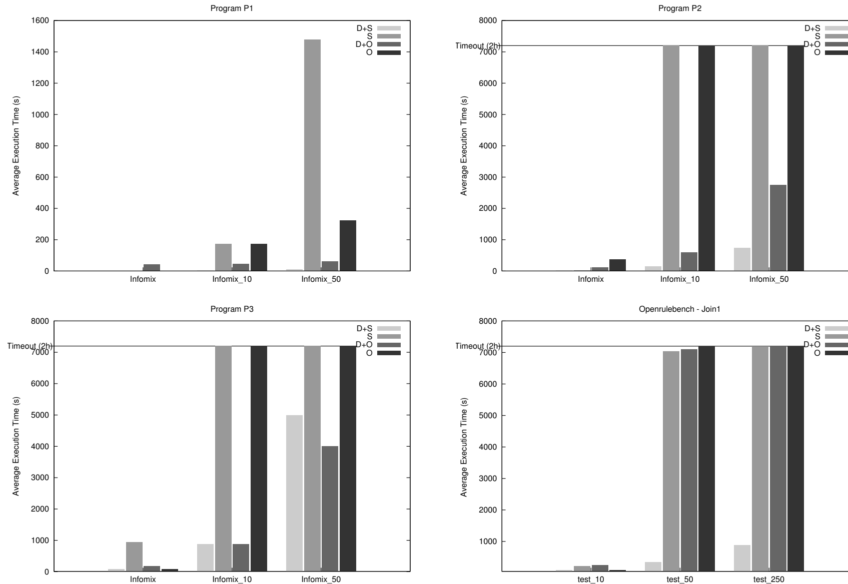


Figura 6.1: Risultati dei test su programmi logici.

senta di ottenere significativi miglioramenti delle prestazioni. Infatti, mentre per dataset di non elevate dimensioni migliora l'esecuzione di pochi secondi (**P1**) o centinaia di secondi (**P2** e **P3**), consente di ottenere prestazioni significativamente superiori nel caso di dataset molto grandi (i DBMS superano il timeout sia con **P2** che con **P3** già per **Infomix-x-10**. Inoltre, quando i DBMS non superano il timeout, `D+S` permette di ottenere un guadagno fino al 99% rispetto a S e `D+O` fino al 80% rispetto ad `O`.

Test da OpenRuleBench. In [135] è stato presentato un benchmark per testare sistemi rule based. Anche se non è stato progettato per testare i programmi su dati distribuiti, sosteniamo che potrebbe essere interessante adattare alcuni di questi programmi al nostro contesto.

Abbiamo concentrato l'attenzione sul programma chiamato **join1** in [135] che è costituito dalle seguenti regole:

```
a(X, Y) :- b1(X, Z), b2(Z, Y). b1(X, Y) :- c1(X, Z), c2(Z, Y).
b2(X, Y) :- c3(X, Z), c4(Z, Y). c1(X, Y) :- d1(X, Z), d2(Z, Y).
```

dove `a` è il predicato di output.

Le relazioni di base `c2,c3,c4,d1` e `d2` sono state generate in modo random in [135] in tre insiemi di dati di dimensione crescente aventi, rispettivamente, 10000 (`test_10`), 50000 (`test_50`) e 250000 (`test_250`) tuple per ogni relazione, che abbiamo distribuito in tre siti. I risultati per i tre data set sono mostrati in figura 6.1; in questo caso, la scalabilità di `D+S` è straordinaria rispetto agli altri sistemi, mentre è parso abbastanza sorprendente il comportamento di `D+O`. Abbiamo, pertanto, ulteriormente investigato questo aspetto rilevando che: *(i)* il tempo necessario per il trasferimento dei dati tra i *database links* in Oracle è doppio rispetto a quello richieste dai *linked server* in SQLServer; *(ii)* mentre `D+O`ha richiesto quasi 2 ore per `test_50`, `O` non ha terminato il test in 5 ore; *(iii)* abbiamo anche provato ad eseguire sia `D+O` che `O` su una singola macchina, ma abbiamo interrotto dopo 3 ore. Quindi, abbiamo dovuto concludere che questo test risulta particolarmente problematico per Oracle, indipendentemente dalla nostra ottimizzazione.

6.2 Ontology based Data Access distribuito

Nell'ambito della gestione dei dati e della conoscenza, le interrogazioni basate su ontologie (OBQA) stanno assumendo un ruolo sempre più rilevante [137, 138, 139, 140]. Di fatto numerose organizzazioni oltreché collaboratori autonomi stanno dando origine al cosiddetto "Web of Data", rendendo pubblicamente disponibili repository di web semantico realizzati da zero o da traduzioni in forma ontologica di dati esistenti. Allo stesso tempo alcuni fornitori di database - come ORACLE[1], Ontotext[2] e Ontoprise[3] - hanno iniziato a costruire moduli di ragionamento ontologico sui loro software. Inoltre, il ragionamento ontologico è l'obiettivo di diversi sistemi (basati sulla ricerca o sul ragionamento), come ad esempio Quest [141], Owlgres [142], Owlim [143], e QuOnto [144], solo per citarne alcuni. Un importante settore di ricerca relativo a OBQA mira ad estendere l'espressività di teorie ontologiche trattabili (si veda, ad esempio, [145, 146]). I risultati ottenuti, in termini di complessità dei dati da interrogazioni sui frammenti più significativi di Ontology Web Language (OWL) [147] hanno rivelato l'impossibilità pratica di gestire grandi quantità di dati. Per superare questa limitazione, sono state imposte numerose restrizioni sui linguaggi di ontologie, dando vita alle cosiddette ontologie leggere, incluse, ad esempio, la famiglia *DL-Lite* [139] e la famiglia $\mathcal{EL}$ [148], che garantiscono limiti di complessità ottimale (cioè, da LOGSPACE fino a polinomiale) per le interrogazioni. (L'ultima specifica di OWL include esplicitamente tali ontologie nei profili EL e QL, vedi [147]).

Alcuni approcci al OBQA si basano su una ri-formulazione della query, in cui la query originale posta sull'ontologia è riscritta in una serie equivalente di query "semplici" che possono essere valutate direttamente sulle istanze dell'ontologia. Esistono numerosi riscrittori di query [149, 144, 150, 151, 142], tuttavia spesso privi del livello di valutazione. Nello specifico abbiamo focalizzato l'attenzione al caso di query su ontologie che possono essere riscritte in linguaggio Datalog [152]. Questo contesto risulta particolarmente interessante in quanto la valutazione delle query corrispondenti può essere affidata a valutatori Datalog, che eventualmente possono avvalersi dei vari DBMS [11]. Un ulteriore problema in questo contesto è che, nonostante sembri naturale gestire a livello globale dati distribuiti, gli approcci esistenti non tengono conto esplicitamente della distribuzione dei dati su siti diversi. In realtà, il problema di collegare dati provenienti da diverse sorgenti ad una concettualizzazione formale del dominio di interesse, è stato affrontato in [153], in cui i dati memorizzati nei sistemi di gestione come database relazionali sono collegati alle teorie dell'ontologia dal concetto di Abox virtuale. Tuttavia, [153] non tratta specificamente della valutazione efficiente di query su dati distribuiti.

Una soluzione naïve al problema della distribuzione consiste nel copiare tutti i dati distribuiti su un'unica ontologia centralizzata; tuttavia questa soluzione non è chiaramente adatta quando le ontologie sono numerose o di grandi dimensioni e subiscono frequenti modifiche, o quando le applicazioni richiedono di operare su dati "freschi". Dalle considerazioni espresse appare chiaro che la capacità di trattare efficacemente OBQA in ambiente distribuito è un'attività cruciale.

[1] Vedi: `http://www.oracle.com/`
[2] Vedi: `http://www.ontotext.com/`
[3] See: `http://www.ontoprise.de/`

Nel seguito riportiamo alcune nozioni preliminari su ontologie e su query answering su ontologie.

Ontologie. Un'ontologia [54] $\mathcal{O}$ è costituita da due componenti: la *TBox* e la *ABox*. La *TBox* definisce la *terminologia*, cioè un vocabolario di *concetti* (che denotano insiemi di individui) e *ruoli* (che denotano relazioni binarie tra concetti), nonché le loro associazioni (ad esempio, relazioni subconcetti/superconcetti). Più in dettaglio, la *TBox* è un insieme finito di *assiomi terminologici* che esprimono inclusioni/equivalenze tra concetti e ruoli che sono definiti per mezzo di specifici costruttori di concetti/ruoli; linguaggi diversi possono essere ottenuti attivando o meno i costruttori. L'*ABox* è un insieme di asserzioni sugli individui che possono essere espresse in termini di fatti logici sul vocabolario definito dalla *TBox*. Ad esempio, l'assioma terminologico $Person \subseteq \exists hasFather.Person$ significa che la classe $Person$ è contenuta nella classe di persone aventi come padre una persona, e l'asserzione $Person(bob)$ stabilisce che *bob* è un'istanza di $Person$. Le nozioni di "satisfation" e di "model" di una ontologia sono quelle standard, di cui in [54] esiste una descrizione formale.

Query. Una *query congiuntiva* Q su un'ontologia $\mathcal{O}$ può essere vista come una regola Datalog $q(\overline{x})\text{:-}conj(\overline{x}, \overline{y})$ dove $\overline{x}$ ed $\overline{y}$ sono tuple di variabili distinte (rispettivamente chiamate *distinguished* e *non-distinguished*), e $conj$ è una congiunzione di atomi costruiti dai concetti e dai predicati di ruolo su $\overline{x}$ e $\overline{y}$. La risposta a una query Q è una tupla $\bar{t}$ che soddisfa Q e tale che $conj(\bar{t}, \overline{t'})$, per qualche $\overline{t'}$, è soddisfatto per ogni modello di $\mathcal{O}$. Una unione di query congiuntive (UCQ) è un insieme di query aventi la stessa testa; una tupla $\bar{t}$ è una risposta ad una UCQ U se si tratta di una risposta ad alcune delle query di U.

Mapping. Soprattutto nel caso di grandi quantità di dati, la *Abox* può essere memorizzata sfruttando efficienti sistemi di gestione dei dati come i database relazionali. Questa impostazione (originariamente introdotta in [153]) può essere modellata mediante asserzioni di mapping che definiscono una *ABox virtuale* in termini dei dati memorizzati in un database relazionale DB.

Data un'ontologia $\mathcal{O}$, un'asserzione di *mapping* è un'espressione della forma $\phi(t) \rightsquigarrow \psi(t')$ dove $\phi(t)$ è una query su un database relazionale DB e $\psi(t')$ è una query congiuntiva su $\mathcal{O}$.

Ontology-Based Data Access . La nozione di *Ontology-Based Data Access* (OBDA) fu originariamente introdotta in [153], partiamo da questi concetti per effettuare l'esecuzione di query su dati distribuiti in ambiente Datalog. Nel framework originario un insieme di sorgenti pre-esistenti si assume costituiscano il Data Layer di un sistema informativo (Figura 6.2). La vista concettuale dei dati contenuti nel sistema è espressa in termini di un'ontologia che è collegata al Data Layer da una serie di mapping di asserzioni (ABox virtuali) (Figura 6.2a). In scenari complessi, è naturale assumere che questi dati siano nativamente distribuiti tra un insieme di sorgenti autonome e indipendenti, e che questi non necessariamente risiedano sullo stesso host.

Nel framework [153] ciascun mapping di asserzione è inteso essere una query SQL su un database DB. Qui estendiamo il setting ammettendo che ciascun mapping di asserzione sia espresso come un generico insieme di regole Datalog

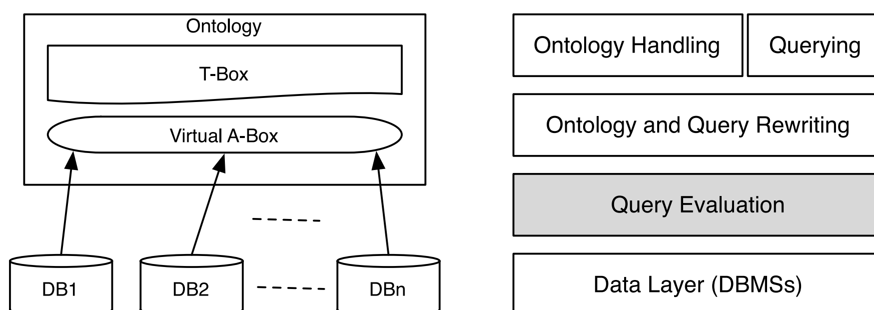


Figura 6.2: Ontology-based Data Access distribuito: (a) Framework e (b) Architettura del sistema.

(eventualmente con negazione stratificata e ricorsione) in cui ciascuna regola può fare riferimento a relazioni eventualmente distribuite su un insieme di database relazionali $DB_1, \ldots, DB_n$ $(n > 0)$ (Vedere Figura 6.2a). In questo framework, l'ontologia non soltanto modella il dominio di interesse, ma rappresenta anche un punto di accesso globale per i client per interagire con il sistema distribuito. L'*Ontology Handling* ed i *Query layers* dell'architettura del sistema possono essere usati dall'utente a questo scopo.

In molti approcci ad OBQA, le query sono gestite sfruttando riscritture sia dell'ontologia che della query, il livello *Ontology and Query Rewriting* dovrebbe essere in grado di trasformare la query di input e l'ontologia così da delegare l'esecuzione al *Query Evaluation Layer* che a sua volta interagisce direttamente con il *Data Layer*. Sono stati sviluppati diverse tecniche e sistemi che possono soddisfare i requisiti del livello *Ontology and Query Rewriting* (ricordiamo Presto [151], Requiem [150], QuOnto [144], Owlgres [142], e Rapid [149]). Gli output prodotti sono spesso query SQL o regole Datalog stratificate; tuttavia, questi sistemi *(i)* sono del tutto privi di un livello di valutazione, o *(ii)* sfruttano un livello di valutazione non concepito per trattare dati distribuiti in modo nativo, o *(iii)* non sono concepiti per gestire generici mapping Datalog di ABox su dati distribuiti.

Quindi, un'implementazione del livello *Ontology and Query Rewriting* nel nostro framework deve: *(i)* produrre riscritture Datalog standard; *(ii)* comprendere mapping (eventualmente complessi) per ABox virtuali nel processo di riscrittura, *(iii)* essere a conoscenza della distribuzione dei dati.

Per quanto riguarda la valutazione della query, la nostra idea è quella di utilizzare tecniche e sistemi concepite per la valutazione distribuita di programmi Datalog stratificati per attuare un efficiente Query Evaluation layer (lo strato grigio scuro in Figura 6.2b) in grado di trattare i dati ontologici (eventualmente) distribuiti.

In questo contesto il nostro sistema ben si presta ad una efficiente gestione di query (Datalog riscrivibili) poste su ontologie "light" possibilmente distribuite su siti fisicamente diversi. Ciò è confermato dai risultati di alcuni esperimenti preliminari nel contesto del OBQA, su un ben noto benchmark di ontologie, che dimostra la validità dell'approccio.

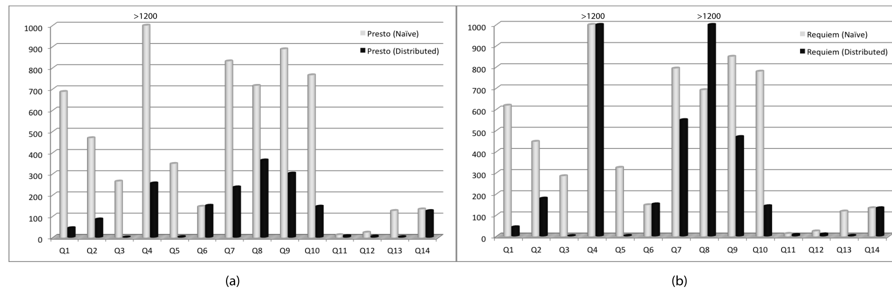


Figura 6.3: Esecuzione delle query Ontology-based: riscritture di (a) Presto e (b) Requiem.

6.2.1 Esperimenti

Al fine di verificare la validità dell'approccio proposto, abbiamo generato un'istanza proof-of-concept del framework presentato nel Capitolo 4. Per quanto concerne il livello dei dati, abbiamo considerato una variante del benchmark di ontologie ampiamente studiato LUBM (vedi `http://swat.cse.lehigh.edu/projects/lubm/`), determinata nel modo seguente: Una prima copia, ottenuta con il generatore di dati Univ-Bench (UBA), è stata replicata due volte modificando in modo casuale una percentuale fissa di simboli (30% nei nostri test); l'insieme risultante, costituito da tre ontologie, ha in comune la stessa TBox ed ha ABox parzialmente sovrapponibili. Le tre ontologie (per un totale di circa 20Gb), sono state distribuite su tre diversi server della nostra rete e sono state collegate attraverso un insieme di ABoxes virtuali, uno per ogni *concept* e per ogni *role* nella Tbox. Più in dettaglio, dato un *role* $r(X, Y)$ e le relative istanze $r@s1(X, Y)$, $r@s2(X, Y)$, $r@s3(X, Y)$ sui tre server, il mapping virtuale adottato aveva la forma $r(X, Y) : -r@s1(X, Y), r@s2(K, Y), r@s3(Z, Y)$ (regole analoghe sono state formulate per le entità). Per quanto riguarda il livello *Ontology and Query Rewriting*, abbiamo considerato la riscrittura di 14 query LUBM generate da [151] e Requiem [150] sull'ontologia standard; i programmi Datalog risultanti sono stati estesi con le regole che implementano la ABox virtuale. Il Query Evaluation Layer è stato realizzato con il nostro approccio di valutazione distribuita, che ha ricevuto in input la query riscritta e la specificazione della distribuzione dei dati, ed ha prodotto i risultati dell'interrogazione sul sito desiderato. Per verificare l'efficacia del metodo, abbiamo confrontato i tempi di esecuzione del setting proposto con una versione "naïve", in cui i dati coinvolti dalla query vengono prima spostati sul sito di destinazione; in questo modo, il processo di valutazione diventa completamente locale ed è eseguito dalla versione standard di DLV^{DB}. Per gli esperimenti sono stati stati utilizzati tre rack HP ProLiant DL120 G6 dotati di processori Intel Xeon X3430, 2.4 GHz, con 4 Gb Ram, due dei quali con sistema operativo Windows 2003 Server e DBMS MS SQL Server, ed uno con sistema operativo Linux Debian Lenny e DBMS PostgreSQL. Abbiamo fissato un limite di tempo di 1200 secondi (20 minuti) oltre i quali l'esecuzione del sistema è stata fermata.

I risultati dei nostri esperimenti sono rappresentati nella Figura 6.3; l'istogramma a sinistra mostra i tempi di esecuzione ottenuti eseguendo le query

riscritte con Presto, sulla destra sono riportati i tempi ottenuti eseguendo le query riscritte con Requiem; per ciascuna query abbiamo riportato accanto (affinchè possa essere facilmente confrontato) le prestazioni della valutazione "naïve" (in grigio) e quelle ottenute con il metodo "distributed" (in nero). La figura mostra le migliori performance ottenute dal valutatore distribuito che è quasi sempre più veloce della strategia "naïve", spesso di un ordine di grandezza. Il buon funzionamento è confermato in entrambe le riscritture della query. L'unica eccezione è per la query 8 riscritta con Requiem, in cui un problema tecnico determina un degradamento della prestazione della versione "distributed". Più in dettaglio, la riscrittura prodotta da Requiem per la query 8 contiene diverse regole aventi gli stessi predicati nel corpo; dal momento che la nostra implementazione procede regola per regola e, al momento in cui sono stati effettuati i test, non prevedeva alcuna tecnica di caching, le stesse tabelle venivano trasferite più volte provocando il degradamento della prestazione. Questo problema è stato facilmente risolto evitando trasferimenti multipli degli stessi dati, attraverso l'adozione di una semplice tecnica di caching.

Questi risultati, anche se preliminari, mostrano chiaramente come le nostre tecniche rappresentino interessanti candidati per lo sviluppo di un efficiente *query evaluation layer* per la valutazione di query poste su ontologie, in particolare quando i dati sono nativamente distribuiti su diversi siti.

6.3 Multi-Context Systems relazionali

I Multi-Context Systems (MCS) [155] rappresentano un potente framework per interconnettere conoscenza da contesti diversi attraverso delle cosiddette "bridge rules". I contesti possono essere definiti in formalismi eterogenei di rappresentazione della conoscenza e possono essere distribuiti in rete. Recenti risultati di ricerca forniscono algoritmi efficienti (distribuiti) per implementare semantiche "well-established model-based" espresse in termini dei cosiddetti "equilibria". Tuttavia, questi algoritmi si sono rivelati inefficienti nel supporto al query answering, in particolare nel caso rilevante di MCS relazionali [156] (per la presenza di *bridge rules* contenenti variabili, che, in linea di principio, possono comportare lo scambio di grandi quantità di dati). In tale contesto, considerando quale task prioritario il query answering, adattiamo la nostra tecnica basata su unfolding ed hypertree decomposition descritta nel Capitolo 4. Lo scopo dell'unfolding è quello di esprimere una query direttamente in termini di dati coinvolti nella query, mentre l'obiettivo dell'hypertree decomposition è quello di calcolare una decomposizione della query originale organizzata come una join-tree, che può essere valutata in modo efficiente seguendo una strategia bottom up.

Nel seguito definiamo formalmente il problema e caratterizziamo una classe di MCS che può essere adattata. Riportiamo, inoltre, i risultati di un esperimento preliminare effettuato su dati reali considerando un modello MCS in un dominio biomedico [157] mediante contesti che adottano ASP come linguaggio di rappresentazione della conoscenza.

MCS relazionali. I MCS relazionali [156] generalizzano MCS nonmonotoni, come definiti in [155], come descriveremo nel seguito. I MCS rappresentano la conoscenza contestuale attraverso una nozione astratta di una cosiddetta *logica*

L che è definita nei termini di una firma Σ_L, un insieme di basi di conoscenza well-formed KB_L, un insieme di possibili belief sets BS_L, ed una funzione $ACC_L : KB_L \to 2^{BS_L}$ (che rappresenta intuitivamente la semantica di L) che assegna ad ogni base di conoscenza un insieme di accettabili belief sets. Mentre le logiche (relazionali) hanno la finalità di rappresentare la conoscenza contestuale in termini di basi di conoscenza, le cosiddette *bridge rules* modellano la loro interconnessione. Intuitivamente, una logica relazionale L (talora definita semplicemente "logica" nel seguito) ulteriormente calcola elementi relazionali in KB_L e BS_L [156]. Dato un insieme di logiche relazionali $\{L_1, \dots, L_n\}$, sia V un insieme numerabile di nomi di variabili distinte. Noi diciamo che un (non-ground) elemento relazionale di L_i è nella forma $p(t_1, \dots, t_k)$, dove t_j è un termine su Σ_{L_i} e V. Allora, una *relational bridge rule* è nella forma

$$(k\colon s) \leftarrow (c_1\colon p_1), \dots, (c_j\colon p_j), not(c_{j+1}\colon p_{j+1}), \dots, not(c_m\colon p_m), \qquad (6.1)$$

dove $1 \leq k \leq n$, e s è un elemento ordinario o relazionale della base di conoscenza di L_k, così come $1 \leq c_\ell \leq n$, e p_ℓ è un belief ordinario o relazionale di L_ℓ, per $1 \leq \ell \leq m$. Il belief della testa s è denotato da $hd(r)$, mentre $head(r) = (k\colon s)$. Inoltre, $pos(r) = \{(c_{\ell_1}\colon p_{\ell_1}) \mid 1 \leq \ell_1 \leq j\}$, $neg(r) = \{(c_{\ell_2}\colon p_{\ell_2}) \mid j < \ell_2 \leq m\}$, e $body(r) = pos(r) \cup \{not(c_{\ell_2}\colon p_{\ell_2}) \mid j < \ell_2 \leq m\}$. Un MCS relazionale è costituito da un insieme di contesti ciascuno dei quali composto da una base di conoscenza di un'associata logica relazionale e un insieme di *bridge rules* relazionali. Più precisamente:

Definizione 11 (Relational MCS)**.** Un *relational MCS* $M = (C_1, \dots, C_n)$ è una collezione di contesti $C_i = (L_i, kb_i, br_i, D_i)$, dove L_i è una logica relazionale, kb_i è una base di conoscenza, br_i è un insieme di *bridge rules* relazionali, e D_i è una collezione di domini $D_{i,\ell}$, $1 \leq \ell \leq n$, tale che $D_{i,\ell}$ è un sottoinsieme dell'universo di Σ_{L_i}.

Noi assumiamo che $D_{i,\ell} = D^A_\ell$, cioè il dominio attivo delle costanti che compaiono in kb_ℓ oppure in $hd(r)$, per qualche $r \in br_\ell$ tale che $hd(r)$ è relazionale.

Usiamo br_M per identificare $\bigcup_{i=1}^n br_i$. L'*import neighborhood* di un contesto C_k è l'insieme $In(k) = \{c_i \mid (c_i : p_i) \in pos(r) \cup neg(r), r \in br_k\}$. L'*import closure* di C_k è $IC(k) = \bigcup_{j \geq 0} IC^j(k)$, dove $IC^0(k) = In(k)$, e $IC^{j+1}(k) = \bigcup_{i \in IC^j(k)} In(i)$. Il *dependency graph* di un MCS M è il digrafo $G = (\{C_1, \cdots, C_n\}, \to)$, dove $C_i \to C_j$ se e solo se $j \in In(i)$.

La semantica di un MCS relazionale è definita da *bridge rules grounding.* Il grounding di un belief relazionale p di L_ℓ è rispetto a D^A_ℓ e indicato con $Ground(p)$. L'insieme di *ground instances* $Ground(r)$ di una *bridge rule* relazionale $r \in br_i$ è limitata dalla ammissibile sostituzione delle variabili in r, cioè, tale che le costanti che sostituiscono la variabile X sono contenute nell'intersezione dei domini associati con i beliefs relazionali di tutte le occorrenze di X in r (cf. anche [156]). Il *grounding* di un MCS relazionale M, indicato con $Ground(M)$, consiste nella collezione di contesti ottenuta sostituendo br_i con $Ground(br_i) = \bigcup_{r \in br_i} Ground(r)$.

Uno stato belief $S = (S_1, \dots, S_n)$ è dato da $S_i \in BS_i$. Una *bridge rule ground* r nella forma (6.1) è applicabile rispetto a S, denotato da $S \models body(r)$, se e solo se $p_\ell \in S_{c_\ell}$ per $1 \leq \ell \leq j$ e $p_\ell \notin S_{c_\ell}$ for $j \leq \ell \leq m$. Da $app_i(S)$ indichiamo l'insieme $\{hd(r) \mid r \in Ground(br_i) \wedge S \models r\}$; e $S = (S_1, \dots, S_n)$ è un equilibrium di MCS M se e solo se $S_i \in ACC_i(kb_i \cup app_i(S))$.

MCS relazionali. Mentre gli *equilibria* forniscono una semantica dichiarativa model-based per MCS, e quindi anche una base per query answering su MCS, il calcolo degli equilibria e il query answering è, per molte impostazioni pratiche, una soluzione non praticabile. Di conseguenza consideriamo il query answering su MCS relazionali un task primario. Consideriamo *conjunctive queries* $Q_{C_\ell}(\mathbf{t})$ poste su un *query context* C_ℓ della forma:

$$q(\mathbf{t}) \leftarrow (c_1 \colon p_1), \ldots, (c_j \colon p_j), not(c_{j+1} \colon p_{j+1}), \ldots, not(c_m \colon p_m),$$

dove q è un predicato della query, $\mathbf{t} = (t_1, \ldots, t_k)$ è una k-tuple di termini e ogni p_i, $1 \leq i \leq m$, è un belief ordinario o relazionale di L_{c_i}.

Limitiamo anche le interrogazioni alle query *conformi alla topologia*, cioè, contesti nel corpo provenienti dall'*import closure* di C_ℓ; formalmente, $(c_i \colon p_i) \in pos(Q_{C_\ell}(\mathbf{t})) \cup neg(Q_{C_\ell}(\mathbf{t}))$ implica $c_i \in IC(\ell)$.

Come sempre, il query answer è definito in termini di sostituzioni delle variabili in $\mathbf{t}$ (eventuali) che rende vera la query.

Più nel dettaglio, dato uno stato belief S scriviamo $S \models Q_{C_\ell}(\mathbf{t})$ se esiste $q_g \in Ground(Q_{C_\ell}(\mathbf{t}))$ tale che q_g è applicabile rispetto a S.

Definizione 12. Dato un MCS relazionale M ed una query congiuntiva $Q_{C_\ell}(\mathbf{t})$, una k-tupla di termini ground $\mathbf{t}' = (t'_1, \ldots, t'_k)$ è definita una (*possibile*) *query answer*, rispettivamente *certain query answer*, su $Q_{C_\ell}(\mathbf{t})$, se

(i) $\theta\mathbf{t} = \mathbf{t}'$ per qualche sostituzione ammissibile θ, e

(ii) $S \models \theta Q_{C_\ell}(\mathbf{t})$ vale per alcuni (rispettivamente tutti) gli equilibria S di M.

L'insieme $Ans(M, Q_{C_\ell}(\mathbf{t}))$ (risp. $Cert(M, Q_{C_\ell}(\mathbf{t}))$) indica tutti i (certain) query answers.

Un primo presupposto verso il calcolo efficace di certain answer è la consistenza: un MCS è caratterizzato dal fatto di avere equilibria. Ciò piò essere spesso garantito senza computazione ovvero conoscendo particolari equilibria e produce una importante proprietà. Per MCS consistenti gli answers possono in modo equivalente essere ottenuti considerando la restrizione di M ai contesti nell'*import closure* $IC(\ell)$ di C_ℓ (denotato da $M|_{IC(\ell)}$).

Teorema 6.3.1 (Relevance). *Sia M un MCS consistente, allora, $Ans(M, Q_{C_\ell}(\mathbf{t})) = Ans(M|_{IC(\ell)}, Q_{C_\ell}(\mathbf{t}))$ and $Cert(M, Q_{C_\ell}(\mathbf{t})) = Cert(M|_{IC(\ell)}, Q_{C_\ell}(\mathbf{t}))$, per ogni $Q_{C_\ell}(\mathbf{t})$ conformi alla topologia.*

Un'altra naturale limitazione che spesso si applica, riguarda la topologia del sistema. Diciamo che un MCS relazionale è *gerarchico* se il suo grafo delle dipendenze è aciclico, cioè è una foresta.

In termini di programmazione logica sono caratterizzati da *bridge rules* stratificate e non-ricorsive.

In molti scenari rilevanti anche il non-determinismo è limitato a particolari contesti che implicano informazioni deterministiche e forniscono capacità di ragionamento. Di conseguenza, dato un MCS gerarchico M ed un (query) context C_ℓ, diciamo che M è *query-deterministico* per C_ℓ se ACC_{L_j} è deterministico per ogni $j \in IC(\ell)$ tale che $j \neq \ell$. Cioè, ogni contesto $C_j \neq C_\ell$ nell'import closure di C_ℓ è deterministico. Così, un potenziale non-determinismo è limitato al query context:

Procedure EvaluateMCSQuery(Query q, Context c, MCS M, Semantics $\mathcal{S}$)

```
    Output: Set of k-tuple of ground terms Res
1   begin
2       Neighbor := {q} ∪ ⋃_{c_i ∈ In(c)} br_{c_i}; Ext_Know := ∅;
3       foreach BridgeRule br ∈ Neighbor do
4           Unfold := BridgeUnfold(br,M);
5           foreach Query ubr ∈ Unfold do
6               foreach (c_o : p_o) ∈ pos(ubr) ∪ neg(ubr) s.t. c_o is opaque do
7                   Ext_Know := Ext_Know ∪
                    EvaluateMCSQuery(q_o(t) ← (c_o : p_o(t)),c,M);
8           Ext_Know := Ext_Know ∪ HT_Evaluation(hd(br),c, Unfold);
9       Res := evaluate(q,c,Ext_Know,S);
```

Proposizione 6.3.2. *Sia M un MCS gerarchico e consistente che è query-deterministico per C_ℓ. Se $i \neq \ell$, allora $S_i^1 = S_i^2$, per ogni due equilibria S^1 e S^2 di $M|_{IC(\ell)}$.*
Pertanto, se C_ℓ è anche deterministico, allora $M|_{IC(\ell)}$ ha un solo equilibrium e tutti i rispettivi query answers sono certain.

Il query answering su un MCS, rispettando le condizioni descritte nella sezione precedente, può essere effettuato con la procedura EvaluateMCSQuery, che prende in input un MCS M ed una query q posta sul contesto c. Inoltre, esso è parametrizzato a seconda se si è interessati ad answer *possible* o *certain* (ad esempio, il parametro $\mathcal{S}$ può essere "Certain" o "Possible"). Il primo passo individua le *bridge rules* degli import neighbors di c. Tutte le istanze ground dei corrispondenti elementi relazionali presenti nell'unico equilibrium di $M|_{IC(c)}$ sono collegati nell'insieme *Ext_Know*. Per calcolarlo, ogni *bridge rule* in *Neighbor* viene prima sottoposta ad unfolding con una chiamata alla funzione BridgeUnfold; questa operazione produce una unione di query (memorizzate in $Unfold$) la cui valutazione viene effettuata applicando un adattamento (HT_Evaluation) dell'algoritmo di valutazione di query distribuite proposto in [158] alle query MCS, come brevemente descritto in seguito.

L'unfolding è effettuato adattando alle *bridge rules* l'usuale strategia di unfolding per programmi Datalog. In particolare, ogni regola bridge è considerata una query separata e l'unfolding è effettuato seguendo le dipendenze testa-corpo tra l'insieme delle *bridge rules* del MCS: ove possibile, gli elementi sono ricorsivamente sostituiti nel corpo della regola con la loro "definizione", come specificato nel MCS. Diciamo che un contesto c è *opaco* se per c non è definita una procedura di unfolding. Intuitivamente, se un contesto è opaco, non è possibile "guardare dentro", cioè, accedere alla definizione dei suoi elementi (o per il modo in cui la logica è definita o per problemi di privacy), e di conseguenza, non è possibile effettuare l'unfolding attraverso esso. Data una coppia $(c : e)$ presente in una query, diciamo che e non è unfoldabile se (i) $(c : e)$ appare nella parte negativa della query, oppure (ii) $(c : e)$ è testa di più di una *bridge rule* nel MCS, o (iii) il suo contesto c è opaco. Se è soddisfatta la condizione (i) o la condizione (ii), allora l'elemento $(c : e)$ rimane invariato nella query, ma l'algoritmo cerca di unfoldare in modo ricorsivo le *bridge rules* nel MCS, vale a dire, quelle con in testa $(c : e)$. Se si verifica la condizione (iii), allora $(c : e)$ viene interpretato come una query che deve essere posta sul contesto c, gestito da una chiamata ricorsiva alla procedura EvaluateMCSQuery. Infine, la funzione *ContextUnfold* applica una

Function BridgeUnfold(BridgeRule br, MCS M)

```
   Output: Set of Bridge rules Unfold
1  begin
2  |  Unfold := ∅;
3  |  foreach (c : p) ∈ pos(br) ∪ neg(br) do
4  |  |  if p is unfoldable then
5  |  |  |  Let r_p ∈ br_M s.t. head(r_p) = (c : p);
6  |  |  |  br := Replace( br, (c : p), body(r_p) ) ;
7  |  |  if c is not opaque then
8  |  |  |  foreach r ∈ br_M s.t. (c : p) ∈ head(r) do
9  |  |  |  |  R := ContextUnfold(r, c);
10 |  |  |  |  Unfold := Unfold ∪ ⋃_{r'∈R} BridgeUnfold(r',M);
11 |  Unfold := Unfold ∪ br;
```

procedura di unfolding specifica al contesto all'interno di contesti che non sono opachi, restituendo un insieme di query. È quindi applicabile al contesto logico ammettendo procedure di unfolding per query answering (ad esempio, questo vale per ASP, DL-Lite, Datalog, ecc).

L'effetto dell'unfolding è duplice: (i) limita il calcolo ai dati rilevanti per rispondere alla query considerando catene di dipendenze tra *bridge rules*; e (ii) riorganizza le query originali in una serie equivalente di query con corpi più lunghi, che quindi sono più adatte ad essere scomposti mediante la successiva hypertree decomposition.

Infatti, la funzione HT_Evaluation prima applica una hypertree decomposition pesata [7] alle query in *Unfold*, e poi le valuta bottom-up seguendo le dipendenze corpo-testa. Nella tecnica di decomposizione menzionata, una query q è associata ad un hypergraph H dove gli hyper-edges rappresentano join tra le variabili. Un hypertree decomposition di H decompone q in sotto-query valutabili efficientemente in modo bottom-up.

Nella nostra impostazione, gli atomi del corpo sono elementi relazionali definiti in un certo contesto, e una query deve essere eseguita in un contesto per calcolare le relative istanze ground. Inoltre, i contesti possono essere distribuiti su diverse macchine in rete, implicando il trasferimento dei dati sulla rete. Pertanto, utilizziamo la nostra versione modificata di hypertree decomposition definita in [158], che valuta anche i costi di trasferimento in un ambiente distribuito. Infine, una volta che la conoscenza esterna è disponibile al contesto c, la funzione calcola i possible/certain answer di q nel contesto c.

Proposizione 6.3.3. *Dato un MCS relazionale M ed una query congiuntiva $Q_{C_\ell}(\mathbf{t})$ posta sul contesto C_ℓ, allora si assume che $Ans(M, Q_{C_\ell}(\mathbf{t})) =$ Evaluate MCSQuery($Q_{C_\ell}(\mathbf{t})$, C_ℓ,M,Possible) e $Cert(M, Q_{C_\ell}(\mathbf{t})) =$ EvaluateMCSQuery ($Q_{C_\ell}(\mathbf{t})$, C_ℓ,M,Certain).*

6.3.1 Esperimenti

Presentiamo in questa sezione una serie esperimenti condotti con la nostra implementazione dell'approccio descritto nel Capitolo 4. Il nostro prototipo è limitato a MCS le cui logiche di contesto sono definite da programmi logici nell'ambito delle semantiche ASP, che non sono opachi. Gli elementi relazionali ground si assume siano residenti su DBMS, localmente in ogni contesto; essi possono essere eventualmente acceduti da altri contesti, attraverso connessioni ODBC. In questo approccio l'implementazione della funzione HT_Evaluation è

Query	Distributed (sec)	Naïve (sec)	Contesti coinvolti	Tuple coinvolte	Tuple risult.
$q1$	38,5	534,3	PharmGKB,ctd	399.713	41
$q2$	61,1	607,4	PharmGKB,ctd,Sider	400.207	535
$q3$	38,9	572,1	PharmGKB,ctd,BioGrid	431.528	1
$q4$	381,0	1.847,6	PharmGKB,ctd	1.375.819	33
$q5$	36,9	345,4	PharmGKB,ctd,DrugBank	261.315	1.726
$q6$	145,7	1.836,3	PharmGKB,ctd,DrugBank,BioGrid	1.380.398	11

Tabella 6.2: Risultati dei test su MCS.

un adattamento del sistema [158], avente DLV^{DB} come il motore di base per la valutazione di programmi logici. Trattare con contesti ASP ci ha consentito anche di sfruttare più ottimizzazioni nella valutazione. Infatti, la nostra implementazione spinge verso il basso, ove possibile, le costanti presenti nella query attraverso il processo di unfolding. Ciò consente di ridurre significativamente i trasferimenti di dati tra contesti diversi.

Abbiamo valutato il nostro approccio attraverso l'esecuzione di un esperimento preliminare in un'applicazione sul mondo reale presentata in [157]. Il dominio applicativo contempla risorse di conoscenza biomedica su geni, farmaci e malattie, provenienti dalle banche dati on line PHARMGKB, DRUGBANK, BIOGRID, CTD, e SIDER. Questo scenario è stato modellato come un MCS assumendo ciascuna sorgente di conoscenza come un contesto; abbiamo distribuito questi contesti, e i dati corrispondenti, su alcuni server collegati attraverso una rete Ethernet standard. Un certo numero di *bridge rules* hanno anche definito correttamente le interconnessioni [157] tra contesti ed abbiamo preso in considerazione sei query (deterministiche) da [157]. Quindi, data una query q posta su un contesto c, abbiamo effettuato due tipi di esecuzioni (Tabella 6.2): La prima richiamando la procedura EvaluateMCSQuery, e l'altra eseguendo l'algoritmo naïve. L'algoritmo naïve consiste in una esecuzione "normale" di DLV^{DB}, che prima trasferisce verso c tutti i dati ottenuti attraverso le *bridge rules*, e quindi valuta localmente q su c. La Tabella 6.2 mostra i risultati ottenuti che appaiono molto promettenti. Si osservi che il numero di tuple nel risultato delle query considerate è molto più piccolo rispetto alle tuple partenza; ciò è dovuto principalmente alle selezioni presenti nella maggior parte delle query che spingono verso il basso i dati originali grazie all'unfolding.

Capitolo 7

Conclusioni

Il lavoro di tesi ha riguardato la progettazione e l'implementazione di una tecnica per la valutazione distribuita di programmi logici, nel contesto dell'integrazione di informazioni attraverso strumenti che supportino meccanismi di ragionamento adeguati per estrarre conoscenza complessa dai dati disponibili.

Abbiamo condotto uno studio approfondito sulle problematiche connesse all'ottimizzazione di query distribuite con particolare riferimento alla modellazione delle funzioni di costo ed all'utilizzo del semijoin per ridurre la dimensione delle relazioni da trasferire sulla rete.

Al fine di generare un piano di esecuzione abbiamo utilizzato un'estensione pesata di Hypertree Decomposition in grado di sfruttare pienamente le tecniche di decomposizione strutturale integrandole con informazioni quantitative sui dati per il calcolo di decomposizioni minime rispetto ad una funzione di costo. Il modello di costo implementato considera informazioni sulla cardinalità delle tabelle, sulla selettività degli attributi di ciascuna tabella, nonché sui costi di trasmissione dei dati da un sito all'altro.

Il metodo di decomposizione utilizzato rappresenta l'aspetto maggiormente significativo del nostro approccio che, tuttavia, contempla anche tecniche di esecuzione parallela dei programmi, in grado di ridurre significativamente i tempi di risposta.

Abbiamo adattato tecniche di riscrittura di query basate su "unfolding" allo scopo di individuare l'insieme minimale di dati sorgente effettivamente necessari al calcolo della risposta alla query e di ristrutturare la query utente per generare sotto-query su cui la tecnica sviluppata risulta essere particolarmente efficace.

Il nuovo approccio è stato comparato con DBMS commerciali su dati del mondo reale e benchmark, in contesti eterogenei quali "Ontology-Based Query Answering" (OBQA) e "nonmonotonic Multi-Context Systems" dimostrando la validità della soluzione proposta.

Bibliografia

[1] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[2] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[3] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. pages 56–70. Springer-Verlag, 1998.

[4] Hasanat M. Dewan, Salvatore J. Stolfo, Mauricio Hernández, and Jae-Jun Hwang. Predictive dynamic load balancing of parallel and distributed rule and query processing. In *Proc. of ACM SIGMOD 1994*, pages 277–288, New York, NY, USA, 1994. ACM.

[5] Neil Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309 – 322, 1986.

[6] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '99, pages 21–32, New York, NY, USA, 1999. ACM.

[7] F. Scarcello, G. Greco, and N. Leone. Weighted hypertree decompositions and optimal query plans. *Journal of Computer and System Sciences*, 73(3):475–506, 2007.

[8] Nicola Leone, Francesco Ricca, Luca Agostino Rubino, and Giorgio Terracina. Efficient application of answer set programming for advanced data integration. In *PADL*, volume 5937 of *LNCS*, pages 10–24, 2010.

[9] Giorgio Terracina, Erika Francesco, Claudio Panetta, and Nicola Leone. Enhancing a dlp system for advanced database applications. In *Proceedings of the 2Nd International Conference on Web Reasoning and Rule Systems*, RR '08, pages 119–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] G. Terracina, E. De Francesco, C. Panetta, and N. Leone. N.: Experiencing asp with real world applications. In *In: Proceedings of the Fifteenth RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2008.

[11] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming (TPLP)*, 8(2):129–165, 2008.

[12] Patrik Simons, Ilkka Niemelá, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, June 2002.

[13] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135:69–112, 1997.

[14] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, Cambridge, MA, USA, 1992.

[15] N. Leone. *Disjunctive Logic Programming: Knowledge Representation Techniques, System, and Applications.* Department of Mathematics.

[16] John McCarthy. Programs with common sense. In *Semantic Information Processing*, pages 403–418. MIT Press, 1968.

[17] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[18] Alain Colmerauer and Philippe Roussel. History of programming languages—ii. chapter The Birth of Prolog, pages 331–367. ACM, New York, NY, USA, 1996.

[19] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[20] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[21] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming, 1993.

[22] John Mccarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

[23] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[24] Robert A. Kowalski. Predicate logic as programming language. 1974.

[25] Robert Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, July 1979.

[26] International Organization for Standardization. ISO/IEC 13211-1:1995:. *Information technology- Programming languages - Prolog - Part 1: General core.* International Organization for Standardization, Geneva, Switzerland, 1995.

[27] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[28] Keith L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322, New York, 1978. Plenum Press.

[29] R. Reiter. Readings in nonmonotonic reasoning. chapter On Closed World Data Bases, pages 300–310. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[30] K. R. Apt, H. A. Blair, and A. Walker. Foundations of deductive databases and logic programming. chapter Towards a Theory of Declarative Knowledge, pages 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[31] Allen Van Gelder. Negation as failure using tight derivations for general logic programs. In *Foundations of Deductive Databases and Logic Programming.*, pages 149–176. Morgan Kaufmann, 1988.

[32] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '88, pages 221–230, New York, NY, USA, 1988. ACM.

[33] Victor W. Marek and Miroslaw Truszczynski. *Nonmonotonic logic - context-dependent reasoning.* Artificial intelligence. Springer, 1993.

[34] Michael Gelfond. On stratified autoepistemic theories. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 207–211. AAAI Press, 1987.

[35] Nicole Bidoit and Christine Froidevaux. *Minimalism subsumes Default Logic and Circumscription in Stratified Logic Programming.* In Proc. Symposium on Logic in Computer Science (LICS 87), pages 89-97. IEEE, 1987.

[36] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988.

[37] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, pages 620–650, 1991.

[38] Jack Minker. On indefinite databases and the closed world assumption. In *Proceedings of the 6th Conference on Automated Deduction*, pages 292–308, London, UK, UK, 1982. Springer-Verlag.

[39] Adnan Yahya and LawrenceJ. Henschen. Deduction in non-horn databases. *Journal of Automated Reasoning*, 1(2):141–160, 1985.

[40] Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

[41] Kewen Wang and Lizhu Zhou. Comparisons and computation of well-founded semantics for disjunctive logic programs. *ACM Trans. Comput. Logic*, 6(2):295–327, April 2005.

[42] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the dlv system.

[43] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[44] SP Radziszowski. Small ramsey numbers. *ElJC 1994-2014*, pages 1–94, 1994.

[45] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[46] M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. *Knowledge and Data Engineering, IEEE Transactions on*, 9(3):448–463, May 1997.

[47] Francesco Calimeri, Wolfgang Fabery, Nicola Leone, and Simona Perri. Declarative and computational properties of logic programs with aggregates. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 406–411, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[48] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Logic*, 5(2):206–263, April 2004.

[49] Yannis Dimopoulos, Antonis C. Kakas, and Loizos Michael. Reasoning about actions and change in answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings*, pages 61–73, 2004.

[50] Maarten Marien, David Gilis, and Marc Denecker. On the relation between id-logic and answer set programming. In *In Logics in Artificial Intelligence, 9th European Conference (JELIA), volume 3229 of Lecture Notes in Computer Science*, pages 108–120. Springer, 2004.

[51] Francesco Ricca and Nicola Leone. Disjunctive logic programming with types and objects: The dlv+ system. *Journal of Applied Logic, Elsevier, Volume 5, Issue*, 3:545–573, 2007.

[52] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Semantic web reasoning with conceptual logic programs. In Grigoris Antoniou and Harold Boley, editors, *Proc. of 3th International Workshop on Rules and Rule Markup Languages for the Semantic Web*, volume 3323 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2004.

[53] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05*, pages 90–96. Professional Book, 2005.

[54] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[55] Francesco Calimeri, Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and MariaCarmela Santoro. A system with template answer set programs. In JoseJulio Alferes and Joao Leite, editors, *Logics in Artificial Intelligence*, volume 3229 of *Lecture Notes in Computer Science*, pages 693–697. Springer Berlin Heidelberg, 2004.

[56] N. Leone et al. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proc. of SIGMOD'05*, pages 915–917, New York, NY, USA, 2005. ACM.

[57] Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Source inconsistency and incompleteness in data integration. In *KRDB-02*. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-54`, 2002.

[58] Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Integrating inconsistent and incomplete data sources. In *SEBD*, pages 299–306, 2002.

[59] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 305–319, London, UK, UK, 1998. Springer-Verlag.

[60] Massimo Ruffolo, Nicola Leone, Marco Manna, Domenico Saccà, Amedeo Zavatto, and Exeura S. R. L. Exploiting asp for semantic information extraction. In *In Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation*, 2005.

[61] Chiara Cumbo, Salvatore Iiritano, and Pasquale Rullo. Reasoning-based knowledge extraction for text classification. In Einoshin Suzuki and Setsuo Arikawa, editors, *Discovery Science*, volume 3245 of *Lecture Notes in Computer Science*, pages 380–387. Springer Berlin Heidelberg, 2004.

[62] Rosario Curia, Mario Ettorre, L Gallucci, S Iiritano, and P Rullo. Textual document pre-processing and feature extraction in olex. In *Proceedings of Data Mining 2005, Skiathos, Greece*, 2005.

[63] Luigia Carlucci Aiello and Fabio Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Logic*, 2(4):542–580, October 2001.

[64] Chitta Baral and Cenk Uyan. Declarative specification and solution of combinatorial auctions using logic programming. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Logic Programming*

and Nonmotonic Reasoning, volume 2173 of *Lecture Notes in Computer Science*, pages 186–199. Springer Berlin Heidelberg, 2001.

[65] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Representing school timetabling in a disjunctive logic programming language, October 1998.

[66] Elisa Bertino, Alessandra Mileo, and Alessandro Provetti. User preferences vs minimality in ppdl. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 110–122, 2003.

[67] Gianluigi Greco, Antonella Guzzo, and Domenico Saccà. A logic programming approach for planning workflows evolutions. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 75–85, 2003.

[68] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. on Knowl. and Data Eng.*, 15(6):1389–1408, November 2003.

[69] Esra Erdem, Vladimir Lifschitz, Luay Nakhleh, and Donald Ringe. Reconstructing the evolutionary history of indo-european languages using answer set programming. In *In Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages(PADL 2003*, pages 160–176. Springer, 2003.

[70] Francesco Buccafurri and Gianluca Caminiti. A social semantics for multiagent systems. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 3662 of *Lecture Notes in Computer Science*, pages 317–329. Springer Berlin Heidelberg, 2005.

[71] Stefania Costantini and Arianna Tocchio. The dali logic programming agent-oriented language. In JoseJulio Alferes and Joao Leite, editors, *Logics in Artificial Intelligence*, volume 3229 of *Lecture Notes in Computer Science*, pages 685–688. Springer Berlin Heidelberg, 2004.

[72] Alfredo Garro, Luigi Palopoli, and Francesco Ricca. Exploiting agents in e-learning and skills management context. *AI Commun.*, 19(2):137–154, January 2006.

[73] Sea09 : Software engineering for answer set programming. Other, Bath, U. K., November 2009. ID number: CSBU-2009-20.

[74] Martin Brain and Marina De Vos. M.: Debugging logic programs under the answer set semantics. In *Proceedings of the 3rd International Workshop on Answer Set Programming (ASP'05). CEUR Workshop Proceedings (2005) 141-152*, 2005.

[75] Omar El-Khatib, Enrico Pontelli, and Tran Cao Son. Justification and debugging of answer set programs in asp. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 49–58, New York, NY, USA, 2005. ACM.

[76] Francesco Ricca. The dlv java wrapper. In Marina de Vos and Alessandro Provetti, editors, Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation, pages 305-316, Messina, Italy, 2003.

[77] P. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[78] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, July 1983.

[79] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. of VLDB 81*, pages 82–94, Cannes, France, 1981.

[80] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: A survey. In Jiri Sgall, Ales Pultr, and Petr Kolman, editors, *MFCS*, volume 2136 of *Lecture Notes in Computer Science*, pages 37–57. Springer, 2001.

[81] G. Gottlob, N. Leone, and F. Scarcello. Advanced Parallel Algorithms for Processing Acyclic Conjunctive Queries, Rules, and Constraints. In *Software Engineering and Knowledge Engineering*, 2000.

[82] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 115–126, New York, NY, USA, 1995. ACM.

[83] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, July 1984.

[84] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM.

[85] Xiaolei Qian. Query folding. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 48–55, Washington, DC, USA, 1996. IEEE Computer Society.

[86] T. Korimort. *Constraints satisfaction Problems- heuristic Decompositions.* PhD thesis, Vienna University of Technology, 2003.

[87] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation.* Prentice-Hall, 2000.

[88] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, May 2001.

[89] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Computing {LOGCFL} certificates. *Theoretical Computer Science*, 270(1-2):761 – 777, 2002.

[90] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *Journal of Computer and System Sciences*, 66(4):775 – 808, 2003. Special Issue on {PODS} 2001.

[91] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: Np-hardness and tractable variants. *J. ACM*, 56(6):30:1–30:32, September 2009.

[92] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007.

[93] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124:243–282, 2000.

[94] B. McMahan. Bucket elimination and hypertree decompositions. 2004.

[95] F. Scarcello and A. Mazzitelli. *The hypertree decomposition home page.* http://wwwinfo.deis.unical.it/ frank/Hypertrees/.

[96] Wolfgang Faber and Gerald Pfeifer since 1996. *DLV homepage.* http://www.dlvsystem.com/.

[97] Michael Stonebraker. The ingres papers: Anatomy of a relational database system. chapter The Design and Implementation of Distributed INGRES, pages 187–196. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[98] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *Software Engineering, IEEE Transactions on*, SE-5(3):188–194, May 1979.

[99] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. *SIGMOD Rec.*, 15(2):84–95, June 1986.

[100] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. Query processing in a system for distributed databases (sdd-1. *ACM Transactions on Database Systems*, 6:602–625, 1981.

[101] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, 1983.

[102] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The tsimmis project: Integration of heterogeneous information sources, 1994.

[103] M.J. Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, AW. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. Towards heterogeneous multimedia information systems: the garlic approach. In *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM '95. Fifth International Workshop on*, pages 124–131, Mar 1995.

[104] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, New York, NY, USA, 1996. ACM.

[105] Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Alberto Corni, R. Guidetti, G. Malvezzi, Michele Melchiori, and Maurizio Vincini. Information integration: The momis project demonstration. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, 2000.

[106] Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Alberto Corni, R. Guidetti, G. Malvezzi, Michele Melchiori, and Maurizio Vincini. Information integration: The momis project demonstration. pages 611–614. Morgan Kaufmann, 2000.

[107] Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Paolo Naggar, and Fabio Vernacotola. Ibis: Data integration at work. In *SEBD*, 2002.

[108] Andrea Calì, Domenico Lembo, Riccardo Rosati, and Marco Ruzzi. Experimenting data integration with dis@dis. In Anne Persson and Janis Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2004.

[109] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 241–251, New York, NY, USA, 2004. ACM.

[110] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What can databases do for peer-to-peer? In *IN WEBDB*, 2001.

[111] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Logical foundations of peer-to-peer data integration. pages 241–251. ACM, 2004.

[112] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data management for peer-to-peer computing : A vision. In *WebDB*, pages 89–94, 2002.

[113] Luciano Serafini, Fausto Giunchiglia, John Mylopoulos, and Philip A. Bernstein. The local relational model: Model and proof theory. Technical Report IRST Technical Report 0112-23, December 2001. Submitted to ICDT 2003 The 9th International Conference on Database Theory Siena, Italy, 8-10 January 2003.

[114] Marc Friedman, Alon Levy, and Todd Millstein. Navigational plans for data integration. In *In Proceedings of the National Conference on Artificial Intelligence (AAAI*, pages 67–73. AAAI Press/The MIT Press, 1999.

[115] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. What to ask to a peer: Ontology-based query reformulation. In *Proc. of the 9th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2004)*, 2004.

[116] Marcello Balduccini, Enrico Pontelli, Omar Elkhatib, and Hung Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.

[117] Ouri Wolfson and Abraham Silberschatz. Distributed Processing of Logic Programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 329–336, Chicago, Illinois, USA, June 1988.

[118] Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 133–142, New York, NY, USA, 1990.

[119] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. A Framework for the Parallel Processing of Datalog Queries. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 143–152, 1990.

[120] Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data Partition and Parallel Evaluation of Datalog Programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):163–176, 1995.

[121] Hasanat M. Dewan, Salvatore J. Stolfo, Mauricio Hernández, and Jae-Jun Hwang. Predictive dynamic load balancing of parallel and distributed rule and query processing. In Pascal Van Hentenryck, editor, *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 277–288, New York, NY, USA, 1994. ACM.

[122] Michael J. Carey and Hongjun Lu. Load balancing in a locally distributed db system. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 108–119, New York, NY, USA, 1986. ACM.

[123] Francesco Calimeri, Simona Perri, and Francesco Ricca. Experimenting with Parallelism for the Instantiation of ASP Programs. *Journal of Algorithms in Cognition, Informatics and Logics*, 63(1–3):34–54, 2008.

[124] Rosamaria Barilaro, Nicola Leone, Francesco Ricca, and Giorgio Terracina. Optimizing the distributed evaluation of stratified datalog programs via structural analysis. In *Sistemi Evoluti per Basi di Dati - SEBD 2011, Proceedings of the Nineteenth Italian Symposium on Advanced Database Systems, Maratea, Italy, June 26-29, 2011*, pages 295–302, 2011.

[125] Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proceedings of PODS'02*, pages 233–246, New York, NY, USA, 2002. ACM.

[126] Leopoldo E. Bertossi, Anthony Hunter, and Torsten Schaub, editors. *Inconsistency Tolerance*, volume 3300 of *LNCS*, Berlin / Heidelberg, 2005. Springer.

[127] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *TPLP*, 3(4):393–424, 2003.

[128] Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 133–142, New York, NY, USA, 1990.

[129] Ouri Wolfson and Abraham Silberschatz. Distributed Processing of Logic Programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 329–336, Chicago, Illinois, USA, June 1988.

[130] Marcello Balduccini, Enrico Pontelli, Omar Elkhatib, and Hung Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.

[131] L.M. Haas, E.T. Lin, and M.A. Roth. Data Integration through database federation. *IBM System Journal*, 41(4):578–596, 2002.

[132] Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data Partition and Parallel Evaluation of Datalog Programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):163–176, 1995.

[133] Michael J. Carey and Hongjun Lu. Load balancing in a locally distributed db system. *SIGMOD Rec.*, 15(2):108–119, 1986.

[134] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49:716–752, November 2002.

[135] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proc. of WWW'09*, pages 601–610, 2009.

[136] Rosamaria Barilaro, Nicola Leone, Francesco Ricca, and Giorgio Terracina. Distributed ontology based data access via logic programming. In *Web Reasoning and Rule Systems - 6th International Conference, RR 2012, Vienna, Austria, September 10-12, 2012. Proceedings*, pages 205–208, 2012.

[137] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 77–86, New York, NY, USA, 2009. ACM.

[138] Andrea Calì, Georg Gottlob, and Andreas Pieris. New Expressive Languages for Ontological Query Answering. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, pages 1541–1546, 2011.

[139] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[140] Ilianna Kollia, Birte Glimm, and Ian Horrocks. Sparql query answering over owl ontologies. In *Proceedings of the 24th International Workshop on Description Logics*, volume 6643 of *Lecture Notes in Computer Science*, pages 382–396. Springer Berlin / Heidelberg, 2011.

[141] Mariano Rodriguez-Muro and Diego Calvanese. High performance query answering over dl-lite ontologies. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2012)*, 2012. To appear.

[142] Markus Stocker and Michael Smith. Owlgres: A scalable owl reasoner. In Catherine Dolbear, Alan Ruttenberg, and Ulrike Sattler, editors, *5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008)*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[143] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semant. web*, 2:33–42, January 2011.

[144] Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QUONTO: querying ontologies. In *Proc. of the 20th national conference on Artificial intelligence*, volume 4, pages 1670–1671. AAAI Press, 2005.

[145] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In *Proc. of the 11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 70–80. AAAI Press, 2008. Revised version: http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf.

[146] Marie-Laure Mugnier. Ontological query answering with existential rules. In *Proceedings of the 5th international conference on Web reasoning and rule systems*, RR'11, pages 2–23, Berlin, Heidelberg, 2011. Springer-Verlag.

[147] W3C. OWL 2 web ontology language guide. W3C Recommendation, 2003. http://www.w3.org/TR/owl2-guide/.

[148] Franz Baader, Sebastian Brand, and Carsten Lutz. Pushing the el envelope. In *In Proc. of IJCAI 2005*, pages 364–369. Morgan-Kaufmann Publishers, 2005.

[149] Alexandros Chortaras, Despoina Trivela, and Giorgos B. Stamou. Optimized query rewriting for owl 2 ql. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803, pages 192–206. Springer, 2011.

[150] H. Pérez-Urbina, B. Motik, and I. Horrocks. A comparison of query rewriting techniques for dl-lite. In *Proceedings of the 22st International Workshop on Description Logics*, volume 477 of *DL '09*. CEUR-WS.org, 2009.

[151] R. Rosati and A. Almatelli. Improving Query Answering over DL-Lite Ontologies. In *Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR 2010)*, KR '10, pages 290–300, Toronto, Ontario, Canada, 2010. AAAI Press.

[152] Stijn Heymans, Thomas Eiter, and Guohui Xiao. Tractable reasoning with dl-programs over datalog-rewritable description logics. In *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 35–40, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

[153] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking Data to Ontologies. (10):133–173, 2008.

[154] Rosamaria Barilaro, Michael Fink, Francesco Ricca, and Giorgio Terracina. Towards query answering in relational multi-context systems. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, pages 168–173, 2013.

[155] Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI*, pages 385–390. AAAI Press, 2007.

[156] Michael Fink, Lucantonio Ghionna, and Antonius Weinzierl. Relational information exchange and aggregation in multi-context systems. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 120–133. Springer, 2011.

[157] Esra Erdem, Yelda Erdem, Halit Erdogan, and Umut Öztok. Finding answers and generating explanations for complex biomedical queries. In Wolfram Burgard and Dan Roth, editors, *AAAI*. AAAI Press, 2011.

[158] R. Barilaro, F. Ricca, and G. Terracina. Optimizing the distributed evaluation of stratified programs via structural analysis. In *Proc. of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 217–222, Vancouver, Canada, 2011. Lecture Notes in Computer Science, Springer, Heidelberg.