UNIVERSITÀ DEGLI STUDI DI MODENA E
REGGIO EMILIA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Il componente Query Manager di MOMIS: utilizzo della Conoscenza Estensionale

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Massimiliano Franceschi

Correlatore
Dott. Ing. Domenico Beneventano

Controrelatore
Chiar.mo Prof. Michele Colajanni

Anno Accademico 1998 - 99

Parole chiave:

Conoscenza Estensionale
Query Manager
Intelligent Information Integration
Database eterogenei
Query Processing

RINGRAZIAMENTI

Un sentito ringraziamento va alla Professoressa Sonia Bergamaschi per l'aiuto che mi ha fornito durante la realizzazione della presente tesi e per la costante disponibilità dimostrata.
Vorrei inoltre ringraziare tutti i componenti del team MOMIS, in particolare l'Ing. Alberto Corni, per i consigli ed i chiarimenti di ordine pratico ed implementativo.

Indice

Elenco delle figure

Introduzione

In questi ultimi tempi abbiamo assistito ad una crescita esponenziale del numero e della varietà di sorgenti di informazioni disponibili e del quantitativo di dati reperibili da parte di ognuno di noi. Questo grazie alla disponibilità crescente di tecnologie sempre più avanzate nell'ambito delle comunicazioni, delle reti di calcolatori, dei sistemi di elaborazione.
L'esempio più eclatante è quello della rete Internet che oggi va sempre più affermandosi come fenomeno di massa grazie anche al diminuire dei costi relativi e all'aumentare della facilità d'uso degli strumenti necessari, dando la possibilità ad un numero crescente di utenti di accedere ad una quantità enorme di dati.
Questa vasta disponibilità di informazioni per un'utenza anch'essa allargata rispetto al passato, costituisce sicuramente una conquista ed un bene prezioso, se gestita correttamente. Le difficoltà maggiori nascono proprio nel riuscire a trasformare in un effettivo vantaggio per il singolo utente, per l'azienda, o per la comunità, questa accresciuta mole di dati.

L'aumento della disponibiltà di conoscenza è avvenuto in modo disordinato creando numerose sorgenti eterogenee, di conseguenza i dati che ne derivano risultano difficilmente aggregabili. Se a questo si aggiunge l'inevitabile duplicazione e ridondanza di contenuti, si capisce come sia difficile ottenere una informazione significativa. Integrare dati consente di aumentare il valore dell'informazione ad essi collegata purché prima si proceda a selezionarli, filtrarli, collegarli e sintetizzarli [1].
L'eterogeneità dei dati si manifesta a diversi livelli: differenti piattaforme hardware e software (DBMS, linguaggi di interrogazione, ...) sulle quali sono implementate le sorgenti, differenti modelli dei dati (relazionale, ad oggetti, ...), differenti schemi di rappresentazione della struttura logica. Per poter gestire correttamente tutte le informazioni disponibili l'utente dovrebbe conoscere le strutture ed i linguaggi di interrogazione delle varie sorgenti, non solo: dovrebbe anche essere in grado di analizzare la propria richiesta e, dopo averla opportunamente scomposta, indirizzarla alle varie sorgenti in parti comprensibili da ognuna di loro. Ma tutto ciò non basterebbe ancora: i risultati ottenuti andrebbero opportunamente uniti

in modo da ottenere la risposta cercata. Ora, vista la rapida moltiplicazione delle sorgenti di informazioni e dei dati in esse contenute, si evince come possedere uno strumento che effettui queste operazioni in tutto o in parte in maniera automatica, permettendo all'utente nel formulare la propria richiesta di esulare dall'effettiva natura e organizzazione delle sorgenti, costituisca una importantissima risorsa.
Lo studio di applicazioni in grado di combinare e gestire dati provenienti da più sorgenti è di grande attualità e di interesse anche commerciale, come dimostra lo sviluppo di sistemi quali *Sistemi di Workflow, Datawarehouse, Dataminer, Federazioni di Database*, etc.
Dal punto di vista della ricerca, nonostante il campo dell'Integrazione Intelligente di Informazioni sia relativamente nuovo, esistono già in letteratura numerosi sistemi che intendono realizzare, in modo più o meno flessibile, un modulo di ricerca informazioni oppure un più sofisticato modulo integratore [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12].

Questa tesi si inserisce in un progetto più ampio denominato **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources) [13, 14, 15, 16, 17, 18, 19, 20] che si pone l'obiettivo di integrare sorgenti eterogenee e distribuite al fine di permettere all'utente di interrogare la vista aggregata ottenuta senza dover conoscere la struttura delle singole sorgenti e di ottenere da essa un'unica risposta esaustiva.
Più precisamente l'obiettivo del presente lavoro è consistito nella progettazione e implementazione delle strutture dati e procedure atte a sfruttare la Conoscenza Estensionale[1] in fase di query processing. Gli sforzi sono stati volti quindi all'ampliamento del modulo Query Manager del *Mediatore* di MOMIS, già progettato in [13, 14], fornendolo dei mezzi necessari per la generazione di una risposta corretta e, quanto più possibile, completa e non ridondante.

Nel Capitolo 1 si introduce il concetto di Integrazione Intelligente di Informazioni, trattando anche il programma di ricerca I^3 e la struttura di un mediatore. Si prenderà poi in esame il sistema MOMIS descrivendolo brevemente e soffermandosi più in dettaglio sulla sua architettura.

Il Capitolo 2 sarà dedicato alla Conoscenza Estensionale: verrà spiegato che cosa è ed a che cosa serve, saranno descritte le operazioni per generare una Gerarchia Estensionale e soprattutto come utilizzare queste informazioni in fase di query processing.

Nel Capitolo 3 sarà introdotto il modulo Query Manager di MOMIS, unitamente ad una completa descrizione dei passi che deve compiere per elaborare correttamente una query.

[1] Il concetto di Conoscenza Estensionale unitamente al ruolo che ricopre nell'Integrazione Intelligente di Informazioni saranno trattati nel capitolo 2

Il Capitolo 4 presenterà in dettaglio lo studio e la realizzazione del software necessario alla gestione della Conoscenza Estensionale.

Il Capitolo 5 infine si occuperà di una tipologia di sorgenti (quelle semistrutturate) che stanno assumendo una importanza crescente. Ci si soffermerà sull'XML, linguaggio per la descrizione di sorgenti semistrutturate sul web, ed in particolar modo su di un suo linguaggio di interrogazione: l'XML-QL.

In appendice si avrà: nella A il glossario dei termini utilizzati in ambito I^3; nella B un esempio in ODL_{I^3} che sarà utilizzato come riferimento; nella C il codice di una classe Java particolarmente significativa (BaseExtension.java) e la documentazione relativa; nella D la rappresentazione in BNF della grammatica XML-QL.

Capitolo 1

Un Sistema Intelligente di Integrazione

Oggi è possibile accedere ad una quantità di informazioni decisamente molto ampia, questo grazie alla crescita delle sorgenti di dati tanto all'interno della azienda quanto sulla rete. La probabilità quindi di reperire un dato che interessa è aumentata vertiginosamente, ma allo stesso tempo vediamo diminuire quella di giungerne in possesso nei tempi, ma soprattutto nei modi, desiderati. Questo è dovuto soprattutto alla grande eterogeneità dei dati disponibili, sia per quanto riguarda la natura (testi, immagini, etc.), sia il modo in cui vengono descritti.
A questi problemi va aggiungendosi anche quello del cosiddetto sovraccarico di informazioni, o *information overload*, vale a dire della difficoltà da parte dell'utente di scernere ed isolare i dati veramente interessanti ai fini delle decisioni da prendere dalla enorme mole disponibile che aumenta in continuazione, purtroppo disordinatamente.

Gli standard esistenti (TPC/IP, ODBC, OLE, CORBA, SQL, etc.) risolvono parzialmente i problemi relativi alle diversità hardware e software, dei protocolli di rete e di comunicazione tra i moduli; rimangono però irrisolti quelli relativi alla modellazione delle informazioni. Difatti i modelli e gli schemi dei dati sono differenti e questo crea una eterogeneità semantica (o logica) non risolvibile da questi standard.

Un'importante area, sia di ricerca che di applicazione, riguarda l'integrazione di DataBase eterogenei ed anche i *datawarehouse* (magazzino di dati). Questi lavori studiano la possibilità di materializzare presso l'utente finale delle viste, ovvero porzioni di sorgenti, replicando fisicamente i dati e quindi si devono poi affidare a complicati algoritmi di mantenimento ai fini di garantirne la consistenza.

Con il temine Integrazione di Informazioni (I^2) [1] si indicano invece in letteratura quei sistemi che, basandosi sulle descrizioni dei dati, combinano tra loro

informazioni provenienti da diverse sorgenti (o parti selezionate di esse) senza quindi dover ricorrere alla duplicazione fisica.

1.1 Integrazione Intelligente di Informazioni

L'integrazione delle Informazioni va dunque distinta da quella dei dati (e dei DataBase); per ottenere risultati essa richiede *conoscenza* ed *intelligenza* volte all'individuazione delle sorgenti e dei dati, nonché alla loro fusione e sintesi.
Quando l'Integrazione di Informazioni fa uso di tecniche di Intelligenza Artificiale si parla allora di Integrazione Intelligente di Informazioni (*Intelligent Integration of Information*, I^3). Questa forma di integrazione si prefigge lo scopo di accrescere il valore delle informazioni gestite anche ottenendone di nuove dai dati utilizzati, si distingue quindi dalle tecniche tradizionali che si limitavano ad una semplice aggregazione.

1.1.1 Il programma I^3

Dal 1992 è operativo il Programma I^3 , un progetto di ricerca fondato e sponsorizzato dall'ARPA (agenzia facente capo al Dipartimento della Difesa degli Stati Uniti), che si prefigge di individuare un'architettura di riferimento che realizzi in maniera automatica l'integrazione di sorgenti di dati eterogenee [21].
In questo ambito le tecniche di Intelligenza Artificiale, che possono dedurre informazioni dagli schemi delle sorgenti, risultano uno strumento utile e prezioso per la costruzione automatica di soluzioni integrate flessibili e riusabili.

I^3 propone l'introduzione di architetture modulari sviluppabili secondo i principi proposti da uno standard che ponga le basi dei servizi da soddisfare dall'integrazione ed abbassi i costi di sviluppo e mantenimento. Questo renderebbe possibile ovviare ai problemi di realizzazione, manutenzione, adattabilità che sorgerebbero nella costruzione premeditata di supersistemi comprendenti una notevole quantità di sorgenti non correlate semanticamente; tali supersistemi risulterebbero inoltre strettamente legati alla risoluzione del problema specifico per cui sono stati implementati.
Se si riesce a riutilizzare la tecnologia già sviluppata, la costruzione di nuovi sistemi risulta più veloce e meno difficoltosa, con conseguente abbassamento dei costi. Per poter sfruttare un'elevata riusabilità bisogna disporre di interfacce ed architetture standard. Il paradigma suggerito per la suddivisione dei servizi e delle risorse nei diversi moduli si articola su due dimensioni:

- l'orizzontale, divisa in tre livelli: livello utente, moduli intermedi che fanno uso di tecniche di IA, risorse di dati;

- la verticale: molti domini, con un numero limitato (e minore di 10) di sorgenti.

I domini nei vari livelli non sono strettamente connessi, ma si scambiano dati ed informazioni la cui combinazione avviene a livello dell'utilizzatore, riducendo la complessità totale del sistema e permettendo lo sviluppo di applicazioni con finalià diverse.

I^3 si concentra sul livello intermedio della partizione, quello che media tra gli utenti e le sorgenti. In questo livello sono presenti vari moduli, tra i quali è giusto evidenziare:

- **Facilitator** e **Mediator** (le differenze tra i due sono sottili ed ancora ambigue in letteratura): ricercano le fonti *interessanti* e combinano i dati da esse ricevuti;

- **Query Processor**: riformula le query aumentando le loro probabilità di successo;

- **Data Miner**: analizza i dati per estrarre informazioni intensionali implicite.

Nell'Appendice A è presente un glossario di termini comunemente usato in ambito I^3, questo ha lo scopo di spiegare quei termini che dovessero risultare ambigui o poco chiari, visto il campo recente ed in evoluzione in cui si muove il progetto.

1.1.2 Architettura di riferimento per sistemi I^3

L'obiettivo del programma I^3 è di ridurre il tempo necessario per la realizzazione di un integratore di informazioni, raccogliendo e strutturando le soluzioni prevalenti finora nel campo della ricerca. Esso individua cinque famiglie di attività omogenee, illustrate in Figura 1.1 unitamente ai loro legami. La reciproca iterazione tra queste attività consente di eseguire le operazioni di comunicazione, traduzione ed integrazione dei dati nelle sorgenti. Sono inoltre evidenti i due assi, orizzontale e verticale, che permettono di intuire i diversi compiti dei vari servizi.

Più in dettaglio, percorrendo l'asse orizzontale, si nota il rapporto tra i servizi di Coordinamento ed Amministrazione, che hanno il compito di mantenere informazioni sulle capacità delle sorgenti, vale a dire che tipo di dati sono in grado di fornire e come vanno interrogate; mentre i servizi Ausiliari, responsabili dei servizi di arricchimento semantico delle sorgenti, forniscono anche funzionalità di supporto.
Lungo l'asse verticale invece viene messo in evidenza come avviene lo scambio di informazioni nel sistema: i servizi di Wrapping provvedono ad estrarre le informazioni dalle sorgenti, queste saranno poi impacchettate ed integrate dai servizi di

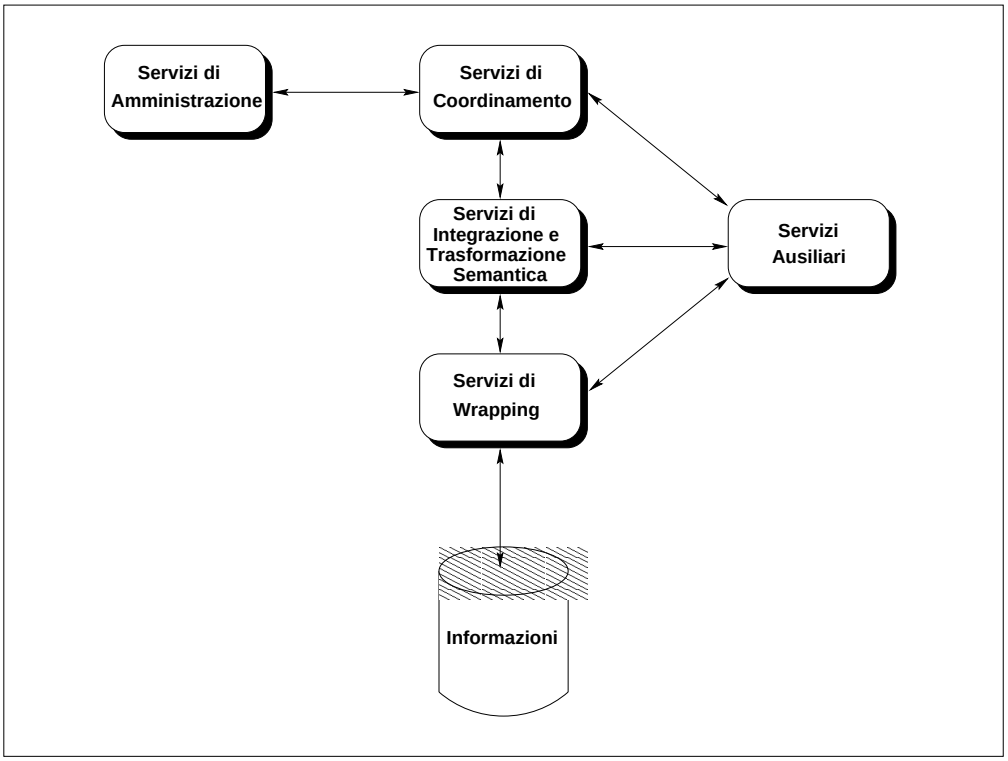


Figura 1.1: Diagramma dei servizi I^3

Integrazione e Trasformazione semantica, per poi essere infine passate ai servizi di Coordinamento che ne avevano fatto richiesta.

Da quanto appena esposto si evince l'importanza centrale dei **Servizi di Coordinamento** che, grazie alle funzionalità messe a loro disposizione dalle altre famiglie, riescono a coordinare le operazioni da eseguire tanto in fase di progettazione dei *link* fra livelli e domini informativi quanto in fase di esecuzione in tempo reale di una richiesta dell'utente. Essi sono servizi ad alto livello, costituiscono l'interfaccia con l'utente e devono dargli l'impressione di trattare con un sistema omogeneo. A seconda della complessità dell'integratore che si vuole realizzare, i moduli di coordinamento possono spaziare dalla selezione dinamica delle sorgenti (*Facilitator e Broker*) al semplice *Matchmaker*, nel quale il mappaggio tra le informazioni locali da integrare è svolto una volta per tutte.

I **Servizi di Amministrazione** sono utilizzati da quelli di Coordinamento per individuare le sorgenti, determinarne le capacità, creare ed interpretare *template* (strutture dati che descrivono i servizi, le sorgenti ed i moduli da utilizzare). I

template servono per ridurre al minimo le possibilità di decisione del sistema, consentendo di definire a priori le azioni da eseguire a fronte di una determinata richiesta. In alternativa ad essi si possono utilizzare le *yellow pages*: servizi di directory che mantengono le informazioni sul contenuto delle sorgenti e sul loro stato (attiva, inattiva, occupata), consentendo al Mediatore di inviare la richiesta di informazioni alla sorgente giusta o, se non fosse disponibile, ad una equivalente. Fa parte di questa famiglia di servizi il modulo denominato *Browser* che permette appunto di navigare tra le descrizioni degli schemi delle sorgenti, recuperando informazioni.
Altri moduli interessanti sono: gli *Iterative Query Formulation*: aiutano l'utente a rilassare o specificare meglio alcuni vincoli dell'interrogazione al fine di ottenere risposte più precise; i *Resource Discovery*: riconoscono e ritrovano gli strumenti che gestiscono una richiesta a run-time; le *Primitive di costruzione delle configurazioni*: scelgono le sorgenti, i servizi e i tools più appropriati e anche come collegarli tra loro.

I **Servizi di Integrazione e Trasformazione Semantica** hanno come input una o più sorgenti dati, tradotte dai servizi di Wrapping, e generano una vista integrata o trasformata di queste informazioni. Essendo tipici dei moduli mediatori vengono indicati spesso come servizi di mediazione. I principali sono:

- *Servizi di Integrazione di Schemi*: creano il vocabolario e le ontologie condivise dalle sorgenti, integrano gli schemi in una vista globale, mantengono il mapping tra schemi globali e sorgenti;

- *Servizi di Integrazione di Informazioni*: aggregano, riassumono ed astraggono i dati per fornire presentazioni analitiche significative;

- *Servizi di Supporto al processo di integrazione*: sono utilizzati quando una query deve essere scomposta in più sottoquery da inviare a fonti differenti, con la necessità di integrare poi i loro risultati.

I **Servizi di Wrapping** realizzano il primo passo verso l'integrazione rendendo le informazioni provenienti dalle sorgenti omogenee grazie alla loro traduzione in un linguaggio standard. Un obiettivo è anche quello di standardizzare il processo di traduzione delle sorgenti ai fini di creare una libreria di fonti accessibili; anzi il processo stesso di realizzazione di un wrapper dovrebbe essere standardizzato per poter essere riutilizzato per altre fonti.

I **Servizi Ausiliari** aumentano le funzionalità degli altri servizi; vanno dai semplici servizi di monitoraggio del sistema ai servizi di propagazione degli

aggiornamenti (mantenimento della consistenza dei dati), dai servizi di arricchimento semantico (già accennati in precedenza) a quelli di ottimizzazione.

Concludendo questa rapida descrizione dell'architettura di riferimento per sistemi I^3 è opportuno sottolineare che non sarà necessario che ogni sistema integratore di informazioni comprenda l'intero insieme di funzionalità descritte; questo sarà anzi molto improbabile che avvenga, l'importante è che utilizzi quelle necessarie allo svolgimento del compito che si prefigge.

1.1.3 Il mediatore

Questa tesi fa parte di un progetto di ricerca più ampio che si prefigge di progettare e realizzare un **Mediatore** (come esaurientemente descritto in [15, 16, 17]), cioè il modulo intermedio nell'architettura precedentemente descritta che si pone tra l'utente e le sorgenti di informazioni. Secondo la definizione proposta in [22] "un mediatore è un modulo software che sfrutta la conoscenza su un certo insieme di dati per creare informazioni per una applicazione di livello superiore . . . Dovrebbe essere piccolo e semplice, così da poter essere amministrato da uno, o al più pochi, esperti."
Un mediatore presenta allora i seguenti compiti:

- assicurare un servizio stabile, anche quando cambiano le risorse;
- amministrare e risolvere le eterogeneità delle diverse fonti;
- integrare le informazioni ricavate da più risorse;
- presentare all'utente le informazioni attraverso un modello scelto dall'utente stesso.

La prima ipotesi che è stata formulata per restringere il campo applicativo del sistema da progettare (e di conseguenza per restringere il campo dei problemi a cui dare risposta) è di avere a che fare, per il momento, esclusivamente con sorgenti di dati testuali strutturati, come possono essere basi di dati relazionali, ad oggetti e file di testo. L'approccio architetturale scelto è stato quello *classico*, che consta principalmente di 3 livelli:

1. utente: attraverso un'interfaccia grafica l'utente pone delle query su uno schema globale e riceve un'unica risposta, come se stesse interrogando un'unica sorgente di informazioni;
2. mediatore: il mediatore gestisce l'interrogazione dell'utente, combinando, integrando ed eventualmente arricchendo i dati ricevuti dai wrapper, ma

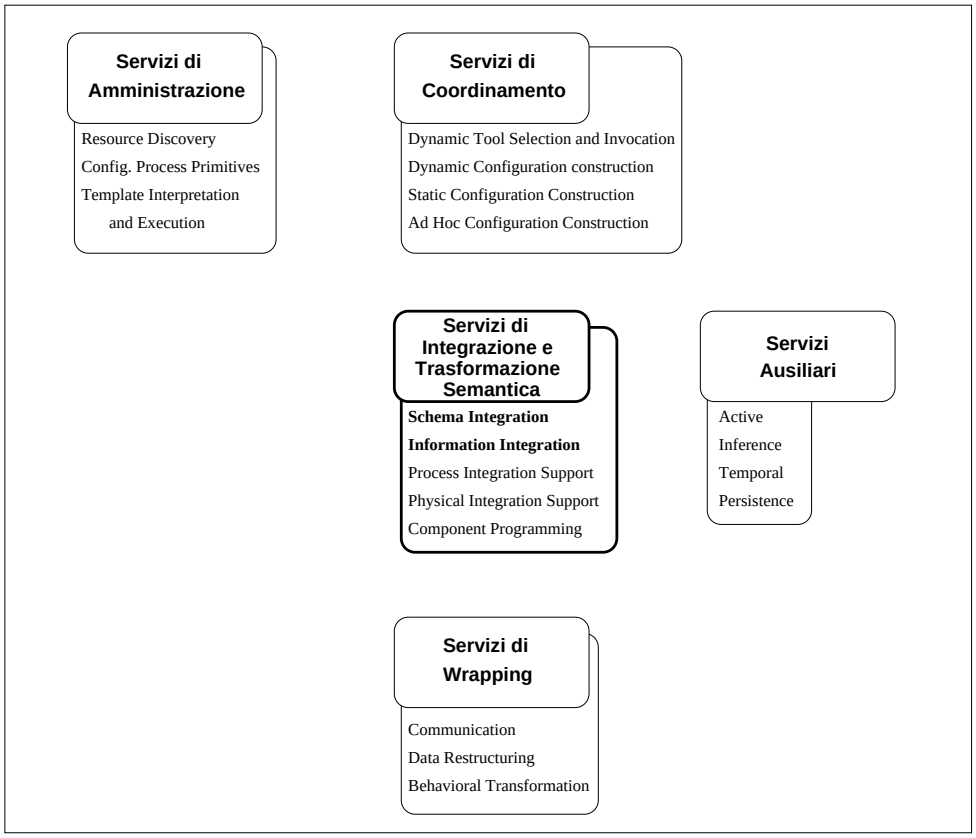


Figura 1.2: Servizi I^3 presenti nel mediatore

usando un modello (e quindi un linguaggio di interrogazione) comune a tutte le fonti;

3. wrapper: ogni wrapper gestisce una singola sorgente, convertendo le richieste del mediatore in una forma comprensibile dalla sorgente, e le informazioni da essa estratte nel modello usato dal mediatore.

L'architettura del mediatore che si è progettato è riportata in Figura 1.2. In particolare sono stati esaminati i servizi di Integrazione e Trasformazione Semantica, saranno cioè forniti dal mediatore servizi che facilitino l'integrazione sia degli schemi che delle informazioni.

Parallelamente a questa impostazione architetturale il progetto si vuole distaccare dall'approccio *strutturale*, cioé sintattico, tuttora dominante tra i sistemi presenti sul mercato. Questo è caratterizzato dal fatto di usare un self-describing model per rappresentare gli oggetti da integrare, limitando l'uso delle informazioni semantiche a delle regole predefinite dall'operatore. In pratica, il sistema

non conosce a priori la semantica di un oggetto che va a recuperare da una sorgente (e dunque di questa non possiede alcuno schema descrittivo) ma è l'oggetto stesso che, attraverso delle etichette, si autodescrive specificando tutte le volte, per ogni suo singolo campo, il significato che ad esso è associato. Le caratteristiche di questo approccio sono quindi le seguenti:

- la possibilità di integrare in modo completamente trasparente al mediatore basi di dati fortemente eterogenee e magari mutevoli nel tempo: il mediatore non si basa infatti su una descrizione predefinita degli schemi delle sorgenti, ma sulla descrizione che ogni singolo oggetto fa di sè. Oggetti simili provenienti dalla stessa sorgente possono quindi avere strutture differenti, cosa invece non ammessa in un ambiente tradizionale object-oriented;

- per trattare in modo omogeneo dati che descrivono lo stesso concetto, o che hanno concetti in comune, ci si basa sulla definizione manuale di rule, che permettono di identificare i termini (e dunque i concetti) che devono essere condivisi da più oggetti.

Altri progetti, e tra questi quello a cui si fa qui riferimento, seguono invece un approccio definito *semantico*, che è caratterizzato dai seguenti punti:

- il mediatore deve conoscere, per ogni sorgente, lo schema concettuale (metadati);

- le informazioni semantiche sono codificate in questi schemi;

- deve essere disponibile un modello comune per descrivere le informazioni da condividere (e dunque per descrivere anche i metadati);

- deve essere possibile una integrazione (parziale o totale) delle sorgenti di dati.

In questo modo, sfruttando le informazioni semantiche che necessariamente ogni schema sottintende, il mediatore può individuare concetti comuni a più sorgenti e relazioni che li legano.

Problematiche da affrontare. Pur avendo a disposizione gli schemi concettuali delle varie sorgenti, non è certamente un compito banale individuare i concetti comuni ad esse, le relazioni che possono legarli, né tantomeno è banale realizzare una loro coerente integrazione. Tralasciando le differenze tra i sistemi fisici (alle quali dovrebbero pensare i moduli wrapper) i problemi che si è dovuto risolvere, o con i quali si è dovuto giungere a compromessi, sono (a livello di mediazione, ovvero di integrazione delle informazioni) essenzialmente di due tipi:

1. problemi ontologici;

2. problemi semantici.

Vediamoli più in dettaglio.

Problemi Ontologici. Nell'Appendice A si può trovare una definizione di ontologia, qui ci si limita solamente a riportare una semplice classificazione delle ontologie (mutuata da [23, 24]) per inquadrare l'ambiente in cui ci si muove. I livelli di ontologia (e dunque le problematiche ad essi associate) sono essenzialmente:

1. *top-level ontology*: descrive concetti molto generali come spazio, tempo, evento, azione . . . , che sono quindi indipendenti da un particolare problema o dominio: si considera ragionevole, almeno in teoria, che anche comunità separate di utenti condividano la stessa top-level ontology;

2. *domain e task ontology*: descrivono rispettivamente il vocabolario relativo a un generico dominio (come può essere un dominio medico, o automobilistico) o a un generico obiettivo (come la diagnostica, o le vendite), dando una specializzazione dei termini introdotti nelle top-level ontology;

3. *application ontology*: descrive concetti che dipendono sia da un particolare dominio che da un particolare obiettivo.

Come ipotesi semplificativa si è assunto di muoversi all'interno delle domain ontology, ipotizzando quindi che tutte le fonti informative condividano almeno i concetti fondamentali (ed i termini con cui identificarli).

Problemi Semantici. Pur ipotizzando che anche sorgenti diverse condividano una visione similare del problema da modellare, e quindi un insieme di concetti comuni, niente ci assicura che i diversi sistemi usino esattamente gli stessi vocaboli per rappresentare questi concetti, né tantomeno le stesse strutture dati. Visto che le diverse sorgenti sono state progettate e modellate da persone differenti, è molto improbabile che queste persone condividano la stessa concettualizzazione del mondo esterno, ovvero non esiste nella realtà una semantica univoca a cui chiunque possa riferirsi.
Ad esempio se la persona P1 disegna una fonte di informazioni (DB1) e un'altra persona P2 disegna la stessa fonte DB2, le due basi di dati avranno sicuramente differenze semantiche: le coppie sposate potranno essere rappresentate in DB1 usando degli oggetti della classe COPPIE, con attributi MARITO e MOGLIE, mentre in DB2 potrebbe esserci una classe PERSONA con un attributo SPOSATO.
Come riportato in [25] la causa principale delle differenze semantiche si può

identificare nelle diverse concettualizzazioni del mondo esterno che persone distinte possono avere, ma non è l'unica. Le differenze nei sistemi di DBMS possono portare all'uso di differenti modelli per la rappresentazione della porzione di mondo in questione: partendo così dalla stessa concettualizzazione, determinate relazioni tra concetti avranno strutture diverse a seconda che siano realizzate attraverso un modello relazionale o ad oggetti.

L'obiettivo dell'integratore, che è fornire un accesso integrato ad un insieme di sorgenti, si traduce allora nel non facile compito di identificare i concetti comuni all'interno di queste sorgenti e risolvere le differenze semantiche che possono essere presenti tra di loro. Possiamo classificare queste contraddizioni semantiche in tre gruppi principali:

1. *eterogeneità tra le classi di oggetti*: benché due classi in due differenti sorgenti rappresentino lo stesso concetto nello stesso contesto, possono usare nomi diversi per gli stessi attributi, per i metodi, oppure avere gli stessi attributi con domini di valori diversi o ancora (dove questo è permesso) avere regole differenti su questi valori;

2. *eterogeneità tra le strutture delle classi*: comprendono le differenze nei criteri di specializzazione, nelle strutture per realizzare una aggregazione, ed anche le discrepanze schematiche, quando cioè valori di attributi sono invece parte dei metadati in un altro schema (come può essere l'attributo SESSO in uno schema, presente invece nell'altro implicitamente attraverso la divisione della classe PERSONE in MASCHI e FEMMINE);

3. *eterogeneità nelle istanze delle classi*: ad esempio, l'uso di diverse unità di misura per i domini di un attributo, o la presenza/assenza di valori nulli.

È però il caso di sottolineare la possibilità di sfruttare adeguatamente queste differenze semantiche per arricchire il nostro sistema: analizzando a fondo queste differenze e le loro motivazioni si può arrivare al cosiddetto *arricchimento semantico*, ovvero all'aggiungere esplicitamente ai dati tutte quelle informazioni che erano originariamente presenti solo come metadati negli schemi, dunque in un formato non interrogabile.

1.2 Il sistema MOMIS

Considerando le problematiche descritte nel paragrafo precedente, nonché alcuni sistemi preesistenti [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], si è giunti alla progettazione di un sistema intelligente di integrazione di informazioni da sorgenti di dati strutturati e semistrutturati denominato **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple

Information **S**ources). Il contributo innovativo di questo progetto, rispetto ad altri similari, risiede nella fase di analisi ed integrazione degli schemi sorgenti, realizzata in modo semi-automatico [15, 16, 17]. Un lavoro approfondito è stato svolto anche riguardo alla fase di *query processing* ([13, 14], nonché la presente tesi), cioè per il processo che dalla query posta sullo schema unificato provvede a generare automaticamente le sottoquery da inviare alle sorgenti ed ad integrare i risultati. MOMIS nasce all'interno del progetto MURST 40% INTERDATA dalla collaborazione tra i gruppi operativi dell'Università di Modena e Reggio Emilia e di quella di Milano.

1.2.1 L'approccio adottato

MOMIS adotta un approccio di integrazione delle sorgenti *semantico* e *virtuale* [13]. Il concetto di semantico è stato illustrato nella sezione 1.1.3. Con virtuale [26] si intende invece che la vista integrata delle sorgenti, rappresentata dallo schema globale, non viene *materializzata*, ma il sistema si basa sulla decomposizione delle query e sull'individuazione delle sorgenti da interrogare per generare delle subquery eseguibili localmente; lo schema globale dovrà inoltre disporre di tutte le informazioni atte alla fusione dei risultati ottenuti localmente per poter ottenere una risposta significativa.

Le motivazioni che hanno portato all'adozione di un approccio come quello descritto sono varie:

- la presenza di uno schema globale permette all'utente di formulare qualsiasi interrogazione che sia con esso consistente;
- le informazioni semantiche che comprende possono contribuire ad una eventuale ottimizzazione delle interrogazioni;
- l'adozione di una semantica *type as a set* per gli schemi permette di controllarne la consistenza facendo riferimento alle loro descrizioni;
- la vista virtuale rende il sistema estremamente flessibile, in grado cioè di sopportare frequenti cambiamenti sia nel numero che nel tipo delle sorgenti, ed anche nei loro contenuti (non occorre prevedere onerose politiche di allineamento);

Inoltre si è deciso di adottare un unico modello dei dati basato sul paradigma ad oggetti, sia per la rappresentazione degli schemi che per la formulazione delle interrogazioni. Il modello comune dei dati utilizzato nel sistema (ODM_{I^3}) è di alto livello e facilita la comunicazione tra il mediatore ed i wrapper. Per definire questo modello si è cercato di seguire le raccomandazioni relative alla proposta di

standardizzazione per i linguaggi di mediazione, nata in ambito I^3: un mediatore deve poter essere in grado di gestire sorgenti dotate di formalismi complessi (ad es. quello ad oggetti) ed altre decisamente più semplici (come i file di strutture), è quindi preferibile l'adozione di un formalismo il più completo possibile.

Per la descrizione degli schemi si è arrivati a definire il linguaggio ODL_{I^3} che si presenta come estensione del linguaggio standard ODL proposto dal gruppo di standardizzazione ODMG-93. Questo permette di cogliere le indicazioni emerse in ambito I^3 ed al contempo di discostarsi il meno possibile dalle proposte del gruppo appena citato. Un'applicazione di ODL_{I^3} viene proposta in Appendice B nella stesura dell'esempio di riferimento. ODL_{I^3} sarà spesso utilizzato in questa tesi ma, vista anche la sua natura intuitiva, non sarà ulteriormente descritto in questa sede; una trattazione esauriente può comunque essere trovata in [14, 15, 16, 17].

Per quanto riguarda il linguaggio di interrogazione si è adottato OQL_{I^3} che adotta la sintassi OQL senza discostarsi dallo standard. Questo linguaggio richiede un maggiore sforzo dal punto di vista dello sviluppo di moduli per l'interpretazione e la gestione delle interrogazioni (implementando le funzionalità tipiche di un ODBMS), ma risulta altresì estremamente versatile ed espressivo fornendo la possibiltà di sfruttare le informazioni rappresentate nello schema globale. Le maggiori difficoltà implementative sono quindi ampliamente giustificate da una maggiore versatilità e da una migliore facilità d'uso per l'utente finale.

Riassumendo: in MOMIS i wrapper si incaricano di tradurre in ODL_{I^3} le descrizioni delle sorgenti, i moduli del mediatore di costruire lo schema globale, mentre tutte le interrogazioni sono poste dall'utente su questo schema utilizzando il linguaggio OQL, o meglio OQL_{I^3}, disinteressandosi dell'effettiva natura ed organizzazione delle sorgenti; sarà il sistema stesso che si incaricherà di tradurre le interrogazioni in un linguaggio comprensibile dalle singole sorgenti e di riorganizzare le risposte ottenute in una unica corretta e completa da fornire all'utilizzatore. Questi concetti saranno ripresi nella Sezione seguente e nella 5.1.

1.2.2 L'architettura generale di MOMIS

In Figura 1.3 è illustrata dettagliatamente l'architettura generale di MOMIS. Lo schema evidenzia l'organizzazione a tre livelli utilizzata.

Livello Mediatore. Il nucleo centrale del sistema è costituito dal **Mediatore** (o *Mediator*) che presiede all'esecuzione di diverse operazioni. Per meglio comprendere i suoi compiti, è opportuno a questo punto illustrare le due fasi ben distinte in cui si articola la sua attività.

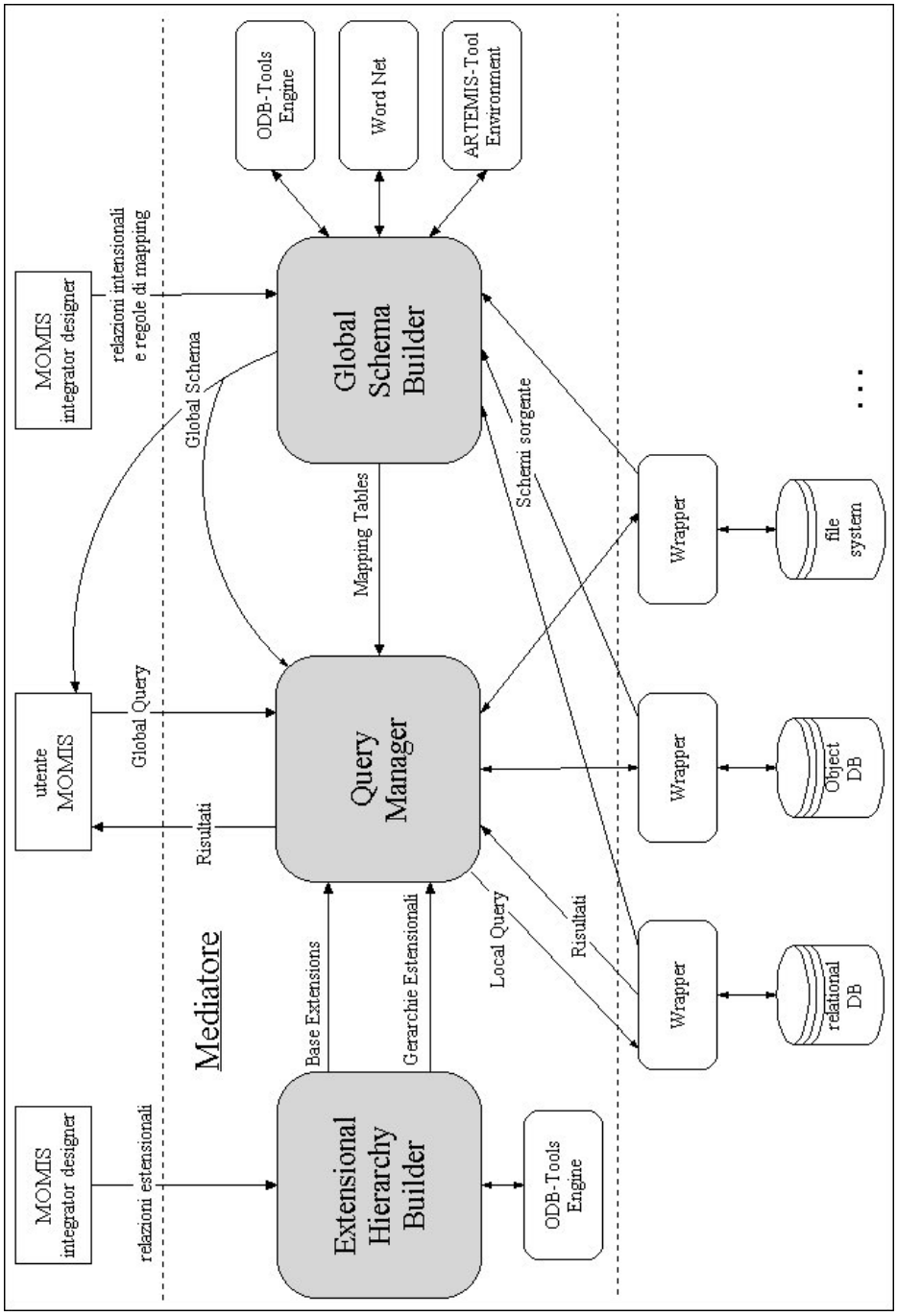


Figura 1.3: Architettura generale di MOMIS

La prima funzionalià del Mediatore è quella di generazione dello Schema Globale. In questa fase il modulo del Mediatore denominato **Global Schema Builder** riceve in input le descrizioni degli schemi locali delle sorgenti espressi in ODL_{I^3} e forniti ognuno dal relativo wrapper. A questo punto (poggiandosi su strumenti di ausilio quali ODB-Tools Engine, WordNet, ARTEMIS) il Global Schema Builder è in grado di costruire la vista virtuale integrata (**Global Schema**) utilizzando tecniche di clustering e di Intelligenza Artificiale. In questa fase è prevista anche l'interazione con il progettista il quale, oltre ad inserire le regole di mapping, interviene nei processi che non possono essere svolti automaticamente dal sistema (come ad es. l'assegnamento dei nomi alle classi globali, la modifica di relazioni lessicali, ...). Oltre allo Schema Globale, altri "prodotti" di questa fase sono le **Mapping Table**, tabelle che descrivono il modo in cui gli attributi globali presenti nello schema generato hanno corrispondenza (*mappano*) nei vari attributi locali presenti negli schemi delle sorgenti.

L'altro importante modulo del mediatore è il **Query Manager** che presiede alla fase di query processing. In questa fase la singola query posta in OQL_{I^3} dall'utente sullo Schema Globale (che chiameremo *Global Query*) sarà rielaborata in più *Local Query* (anche'esse espresse in OQL_{I^3}) da inviare alle varie sorgenti, o meglio ai wrapper predisposti alla loro traduzione, come vedremo più dettagliatamente in seguito. Questa traduzione avviene in maniera automatica da parte del Query Manager utilizzando tecniche di logica descrittiva. Le operazioni svolte dal Query Manager saranno più approfonditamente illustrate nel Capitolo 3.

L'ultimo modulo del Mediatore è rappresentato dall'**Extensional Hierarchy Builder** il quale si occupa della generazione della Conoscenza Estensionale (Gerarchie Estensionali e Base Extension) necessaria per ottimizzare le interrogazioni. Questi aspetti saranno invece trattati nel Capitolo 2.

Livello Wrapper. I **Wrapper** costituiscono l'interfaccia tra il mediatore e le sorgenti; ad ogni sorgente corrisponde un determinato wrapper ed ogni wrapper deve essere disegnato esclusivamente per la sorgente (o la tipologia di sorgenti) che servirà. Ogni wrapper ha due compiti ben precisi:

- in fase di integrazione deve fornire al Global Schema Builder la descrizione della sorgente su cui si poggia in formato ODL_{I^3};

- in fase di query processing deve tradurre la local query (rivolta alla "sua" sorgente) che gli è stata indirizzata dal Query Manager (e che è espressa in OQL_{I^3}) in un linguaggio comprensibile dalla sorgente stessa. Poi dovrà anche fornire al Mediatore i dati ottenuti come risposta nel formalismo comune (ODL_{I^3}).

Collegate ai wrapper sono quindi le **Sorgenti**, per questo a volte si parla anche di quattro livelli. Esse sono le fonti da integrare, possono essere DataBase (ad oggetti o relazionali) o parti di essi, file system ed anche sorgenti semistrutturate.

Livello Utente L'utilizzatore del sistema potrà interrogare lo schema globale, per lui sarà come interrogare un DataBase tradizionale: le sorgenti ed il modo in cui i dati saranno recuperati da esse risultano all'utente del tutto trasparenti, sarà il sistema ad occuparsi di tutte le operazioni necessarie per reperire le informazioni e combinare le risposte in un'unica risposta corretta, completa e non ridondante.

In precedenza si sono citati alcuni tool di ausilio per il mediatore, vediamo una loro rapida descrizione:

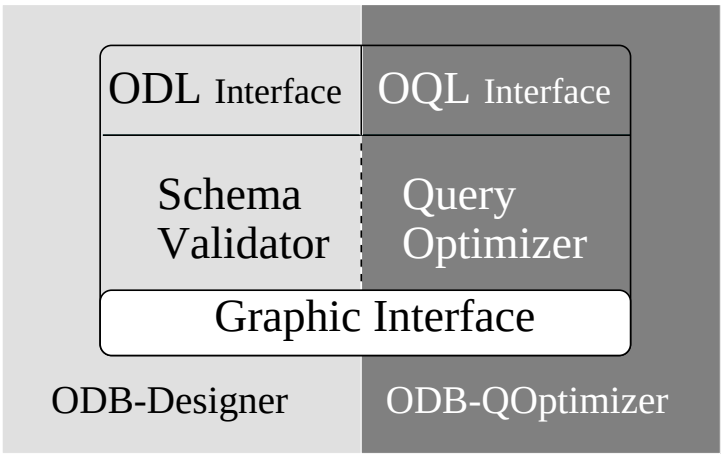


Figura 1.4: Architettura ODB-Tools

- *ODB-Tools* è uno strumento software sviluppato presso il dipartimento di Ingegneria dell'Università di Modena e Reggio Emilia [27, 28, 29]. Esso si occupa della validazione di schemi e dell'ottimizzazione semantica di interrogazioni rivolte a Basi di Dati orientate agli Oggetti (OODB). Facciamo riferimento alla figura 1.4 per una breve spiegazione. ODB-Designer

si occupa della validazione di schemi: si può inserire la descrizione di uno schema di DataBase (in ODL) ed il sistema realizzerà automaticamente la sua validazione e la sua riclassificazione; vale a dire che si occuperà di verificare che non esistano classi incoerenti (cioè che non possano essere popolate da nessun oggetto) e di determinare eventuali relazioni di specializzazione non esplicitate dallo schema stesso. ODB-Qoptimizer si occupa invece dell'ottimizzazione semantica delle interrogazioni: se si inserisce una query (in OQL) posta su di un determinato schema, questa viene automaticamente riformulata in una equivalente, ma più efficiente, sfruttando l'espansione semantica ed i vincoli di integrità.

- *WordNet* [30] è un DataBase lessicale on-line in lingua inglese. Esso è capace di individuare relazioni semantiche fra termini; cioè, dato un insieme di termini, WordNet è in grado di identificare l'insieme di relazioni lessicali che li legano.

- *ARTEMIS* [31] riceve in ingresso il *thesaurus*, cioè l'insieme delle relazioni terminologiche (lessicali e strutturali) precedentemente generate, e sulla base di queste assegna ad ogni classe coinvolta nelle relazioni un coefficiente numerico indicante il suo grado di affinità. Questi coefficienti serviranno per raggruppare le classi locali in modo tale che ogni gruppo (*cluster*) comprenda solo classi con coefficienti di affinità simili.

Capitolo 2

La Conoscenza Estensionale in MOMIS

Uno dei principali obiettivi che ci si è preposti con MOMIS è la realizzazione di un sistema di mediazione versatile ed efficiente, capace di assistere l'utente nel reperimento di informazioni su di un insieme estremamente diversificato di sorgenti. Per riuscire in questo intento si è agito in due direzioni. Da un lato sono stati progettati strumenti in grado di assistere il progettista nella complessa fase di integrazione degli schemi e dall'altro si è realizzato un Query Manager che, posta un'interrogazione sulla vista globale, automatizza il processo di reperimento ed integrazione delle informazioni.
Per garantire che la risposta fornita sia al contempo corretta e, quanto più possibile, completa e minima, entrambe queste fasi sfruttano un insieme di conoscenze relative alle sovrapposizioni delle estensioni delle classi, alle relazioni tra le intensioni e alla semantica degli schemi. Tale conoscenza è in parte fornita dal progettista ed in parte ricavata dalle informazioni implicitamente rappresentate negli schemi locali.

In questo capitolo si intende studiare l'integrazione tra gli schemi, ponendo particolare attenzione alle fasi nelle quali viene trattata la *conoscenza estensionale*.

2.1 Integrazione intensionale ed estensionale

Gli schemi locali vengono integrati in MOMIS secondo criteri sia *intensionali* che *estensionali*. Altri termini usati per indicare intensione ed estensione sono *connotazione* e *denotazione*. Per meglio comprendere la distinzione tra di essi possiamo appoggiarci alla logica: secondo la definizione contenuta in [32] un termine (elemento di una proposizione) può essere considerato in due modi, sia

come una classe di oggetti (che al limite può avere un solo membro), sia come insieme di attributi o caratteristiche che determinano l'oggetto. Il primo aspetto è appunto chiamato denotazione (o estensione), mentre il secondo si indica con connotazione (o intensione). Ad esempio l'estensione del termine "filosofo" è: "Socrate, Platone, . . . "; la sua intensione invece può essere: "amante della conoscenza, saggio, . . . ". Questa definizione può ritenersi valida anche in ambito di integrazione di schemi se si sostituisce il concetto di "elemento di un contesto applicativo" a quello di "elemento di una proposizione".

Schemi locali parzialmente sovrapposti possono rappresentare uno stesso concetto adottando strutture, descrizioni e definizioni differenti, pertanto il presupposto fondamentale per la realizzazione di un Mediatore consiste nella capacità di individuare e risolvere questi conflitti, che possono essere quindi definiti intensionali. Tali incompatibilità derivano dall'aver porzioni di schemi sovrapposte, cioè che descrivono gli stessi aspetti del dominio applicativo ma in modo diverso. L'integrazione intensionale quindi consiste nell'integrazione di schemi e fornisce una rappresentazione unificata ed omogenea dei medesimi concetti descritti nelle varie sorgenti.
Per ottenere però un'effettiva integrazione di informazioni non basta l'integrazione degli schemi, ma è necessario risolvere anche i conflitti derivanti dalle sovrapposizioni delle estensioni, cioè dalla presenza, in sorgenti diverse, di informazioni relative alla stessa entità del *mondo reale*.

Il processo di integrazione viene quindi svolto in momenti distinti:

1. **Unificazione degli schemi** [19]: in questa fase l'uso della logica descrittiva OCDL e di tecniche di *clustering* [33] permette di realizzare un processo semi-automatico di integrazione di schemi, fino a pervenire alla definizione dello Schema Globale, direttamente interrogabile dall'utente, che rappresenti l'unione di tutti gli schemi locali e dal quale saranno rimosse incongruenze e ridondanze;

2. **Fusione delle istanze**: qui l'originalità del lavoro [14] si manifesta nell'introduzione dei cosiddetti *assiomi estensionali* e nel loro trattamento al fine di generare gli oggetti *virtuali* fusi dalla sintesi di quelli *reali* provenienti dalle sorgenti. In questa fase il progettista può introdurre conoscenza sulle relazioni fra le estensioni di classi simili in sorgenti distinte, questa sarà utilizzata successivamente (in fase di esecuzione delle interrogazioni) dal sistema stesso. La buona realizzazione di questa fase si manifesterà a run time in quanto influirà sulla correttezza e completezza delle risposte ottenute, ma anche sulla velocità con cui queste saranno generate.

Insieme a questi due momenti, strettamente di integrazione, si parla a volte

anche della fase di **Query processing** [13], quella cioè nella quale, partendo da una query posta dall'utente sullo schema globale, si arriva alla scelta ed alla definizione dell'insieme ottimo di interrogazioni da inviare alle singole sorgenti potenzialmente interessate ed alla presentazione di un'unica risposta. In realtà questa fase ha una frequenza di esecuzione diversa rispetto alle altre due: essa sarà avviata ogni volta che un utente pone un'interrogazione. Questo va di pari passo con il diverso *utilizzo temporale* dei moduli di MOMIS preposti all'esecuzione di ognuna di esse: il Global Schema Builder e l'Extensional Hierarchy Builder difatti vengono utilizzati una volta per tutte per creare la vista virtuale integrata, o al limite richiamati quando si decide di aggiungere o modificare alcune sorgenti o parti di esse, mentre il Query Manager al contrario sarà utilizzato *giornalmente* dagli utenti del sistema.
Quest'ultima fase, e specialmente i suoi aspetti che sfruttano la conoscenza estensionale generata in 2, sono stati oggetto della progettazione del software realizzato per questa tesi e saranno ampliamente trattati nei Capitoli 3 e 4.

Per quanto riguarda invece strettamente il processo di integrazione, si è adottato un approccio semantico che si articola nei punti seguenti:

1. *Estrazione di relazioni terminologiche*. Durante questa fase, anche attraverso l'iterazione col progettista, viene generato un *Thesaurus Comune* di *relazioni terminologiche*; con questo termine si intendono quelle relazioni che esprimono la conoscenza inter-schema su sorgenti diverse, possono essere relazioni di sinonimia (SYN), di corrispondenza (RT), di maggiore (BT) o minore (NT) ampiezza (generalità) del significato tra termini (nomi di classi, di attributi, ...).
   Partendo dalle descrizioni delle sorgenti in ODL_{I^3} queste relazioni vengono derivate in maniera semi-automatica attraverso l'analisi strutturale e di contesto delle classi coinvolte, grazie anche all'ausilio di ODB-Tools. Questo passo è realizzato dal modulo $\textbf{SIM}_1$ di MOMIS [16];

2. *Analisi delle affinità intensionali fra classi*. Questa fase è realizzata considerando le relazioni terminologiche memorizzate nel Thesaurus ed i coefficienti di affinità;

3. *Creazione dei cluster*. I *cluster* sono raggruppamenti di classi affini: si tratta di classi intensionalmente affini per le quali si presume esista anche una qualche sovrapposizione tra le estensioni. I passi 2 e 3 sono realizzati mediante ARTEMIS [31];

4. *Generazione dello schema globale del mediatore*. Da ogni cluster si definisce una *Classe Globale* che ha un'estensione, formata dall'unione

delle estensioni delle classi sorgenti che costituiscono il cluster, ed un'intensione, ricavata dall'unione ragionata degli attributi delle stesse. Questa fase ha come risultato la definizione in ODL_{I^3} delle Classi Globali con le relative regole di mapping fra i loro attributi *globali* e quelli *locali* delle classi delle sorgenti.

Il processo fin qui descritto è quello presentato in [16, 19, 20], successivamente in [14] si sono studiate altre fasi relative alla gestione della conoscenza estensionale, che saranno trattate nel prosieguo di questo capitolo, e che estendono il precedente punto 4 attraverso:

- *Arricchimento dei cluster*. Il progettista arricchisce la descrizione dei vari cluster ponendo nuovi assiomi estensionali intra ed inter classi sorgente. Può succedere di doverne modificare alcuni qualora siano riconosciuti particolari legami estensionali fra classi in cluster diversi. Questo passo viene ripetuto finché il progettista non ottiene un insieme di cluster soddisfacente dal quale generare lo schema globale;

- *Fusione della Gerarchia Virtuale*. Una volta individuate le classi globali definitivamente, ODB-Tools ne verifica la congruenza e porta alla deduzione degli assiomi impliciti. Ad ogni classe globale si applica un algoritmo di *individuazione delle Base Extension* e di *fusione delle gerarchie* [34]. Viene costruita una gerarchia composta da nuove classi, l'intensione delle quali è un sottoinsieme delle proprietà del cluster e la cui estensione è costituita da tutti gli oggetti reali e/o fusi provenienti dalle sorgenti che posseggono almeno le proprietà specificate nell'intensione della classe.

2.2 Esempio di riferimento

Il seguente esempio verrà utilizzato per illustrare le fasi di integrazione e query processing, fa riferimento alle definizioni degli schemi delle sorgenti espresse in ODL_{I^3} e riportate in Appendice B. In Figura 2.1 viene invece presentato in modo schematico per maggiore semplicità.
Esso si riferisce ad una realtà universitaria: le sorgenti da integrare sono tre.

La prima sorgente, `University` (S_1), è un DataBase di tipo relazionale, che contiene informazioni sullo staff e sugli studenti di una determinata università. È composta da sei tabelle: `University_Worker`, `Research_Staff`, `School_Member`, `Department`, `Section` e `Room`. Per ogni professore (presente nella tabella `Research_Staff`), sono memorizzate informazioni sul suo dipartimento (attraverso la foreign key `dept_code`), sul suo indirizzo di posta elettronica (`email`), e sul corso da lui tenuto (`section_code`). Per il corso

Sorgente University (S_1)

```
University_Worker(first_name,last_name,dept_code,pay)
Research_Staff(first_name,last_name,relation,email,
            dept_code,section_code)
School_Member(first_name,last_name,faculty,year)
Department(dept_name,dept_code,budget,dept_area)
Section(section_name,section_code,length,room_code)
Room(room_code,seats_number,notes)
```

Sorgente Computer_Science (S_2)

```
CS_Person(name)
Professor:CS_Person(title,belongs_to:Division,rank)
Student:CS_Person(year,takes:set⟨Course⟩,rank)
Division(description,address:Location,fund,sector,employee_nr)
Location(city,street,number,county)
Course(course_name,taught_by:Professor)
```

Sorgente Tax_Position (S_3)

```
University_Student(name,student_code,faculty_name,tax_fee)
```

Figura 2.1: Esempio di riferimento

inoltre viene memorizzata l'aula (`Room`) dove questo si svolge, mentre del dipartimento sono descritti, oltre al nome (`dept_name`) ed al codice (`dept_code`), il budget (`budget`) che ha a disposizione e l'area (`dept_area`) a cui appartiene, sia essa Scientifica, Economica, ... Per gli studenti presenti nella tabella `School_Member` sono invece mantenuti il nome (nella coppia `first_name` e `last_name`), la facoltà di appartenenza (`faculty`) e l'anno di corso (`year`).

La sorgente `Computer_Science` (S_2) contiene invece informazioni sulle persone afferenti a questa facoltà, è un DataBase ad oggetti. Sono presenti sei classi: `CS_Person`, `Professor`, `Student`, `Division`, `Location` e `Course`. I dati mantenuti sono comunque abbastanza simili a quelli della sorgente S_1: per quanto riguarda i professori, sono memorizzati il titolo (`title`), e la divisione di appartenenza (`belongs_to`), che a sua volta fa parte di un dipartimento (e ne può quindi essere considerata una specializzazione); per gli studenti sono memorizzati i corsi seguiti (`takes`) e l'anno di corso (`year`). Il corso ha poi un attributo complesso che lo lega al professore che ne è titolare (`taught_by`), mentre per la divisione si tiene l'indirizzo (`address`), i fondi (`fund`) e il numero di impiegati (`employee_nr`).

È presente infine una terza sorgente, `Tax_Position` (S_3), facente capo alla segreteria studenti, che mantiene i dati relativi alle tasse da pagare (`tax_fee`). In

University_ Person	name	dept *(Complex Map)*	pay	faculty	year	tax	..
University_ Worker	first_name **and** last_name	dept_code	pay	null	null	null	...
Research_ Staff	first_name **and** last_name	dept_code	pay	null	null	null	...
School_ Member	first_name **and** last_name	null	null	faculty	year	null	...
CS_ Person	name	null	null	'CS'	null	null	...
Professor	name	belongs_to	null	'CS'	null	null	...
Student	name	null	null	'CS'	year	null	...
University_ Student	name	null	null	faculty_ name	null	tax_fee	...

...	rank	email	relation	takes *(Complex Map)*	title	section *(Complex Map)*	studcode
...	null	null	null	null	null	null	null
...	'Professor'	e_mail	relation	null	null	section_ code	null
...	'Student'	null	null	null	null	null	null
...	null	null	null	null	null	null	null
...	rank	null	null	null	title	null	null
...	rank	null	null	takes	null	null	null
...	'Student'	null	null	null	null	null	student_ code

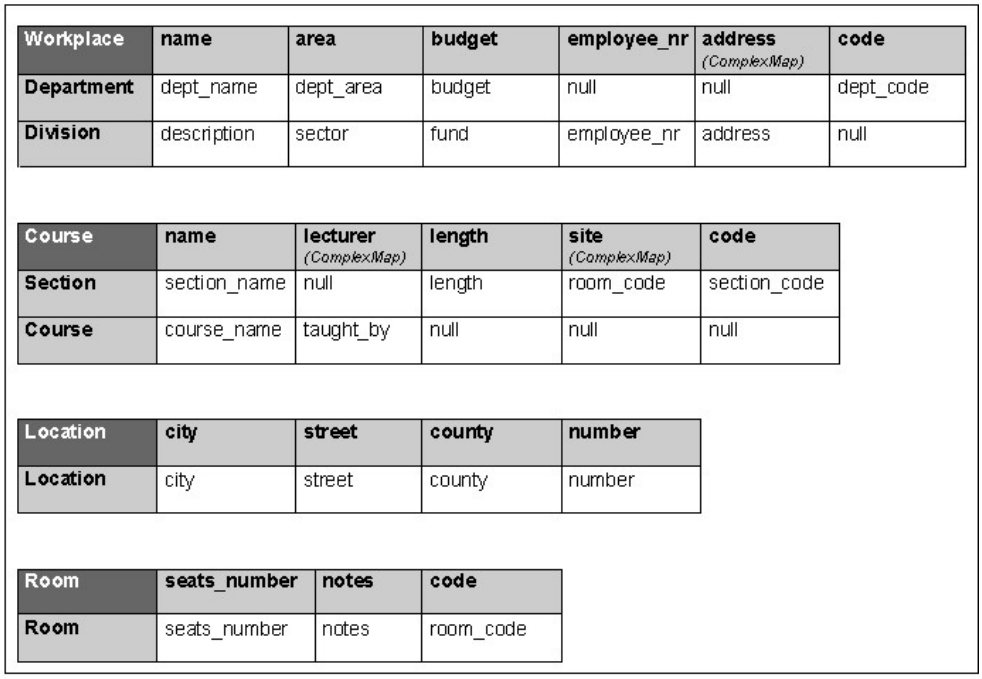
Figura 2.2: Mapping Table della classe globale University_Person

quest'ultimo caso non si tratta di un DataBase ma di un file system, che contiene quindi semplici tracciati record.

Il *processo di unificazione degli schemi* ed i passi da 1 a 4 descritti nella precedente Sezione 2.1, portano all'individuazione di cinque classi globali:

Global Schema: **University_Schema**

- Univerity_Person (name, pay, faculty, year, tax, rank, email, relation, takes, title, section, studcode)
- Workplace (name, area, budget, employee_nr, address, code)
- Course (name, lecturer, length, site, code)
- Location (city, street, country, number)



Workplace	name	area	budget	employee_nr	address *(ComplexMap)*	code
Department	dept_name	dept_area	budget	null	null	dept_code
Division	description	sector	fund	employee_nr	address	null

Course	name	lecturer *(ComplexMap)*	length	site *(ComplexMap)*	code
Section	section_name	null	length	room_code	section_code
Course	course_name	taught_by	null	null	null

Location	city	street	county	number
Location	city	street	county	number

Room	seats_number	notes	code
Room	seats_number	notes	room_code

Figura 2.3: Mapping Table delle altre classi globali

- Room (seats_number, notes, code)

Ognuna di queste classi globali ha una *Mapping Table* nella quale è indicato come i vari attributi globali mappano su quelli delle classi locali; anche le Mapping Table, come lo Schema Globale, sono prodotti del Global Schema Builder, ottenuti nelle fasi a cui si è fatto riferimento poco sopra. In Figura 2.2 è riportata la mapping table relativa alla global class University_Person, in Figura 2.3 quelle delle altre classi globali. Da queste due figure si evince che i comportamenti di mapping tra attributo globale ed attributi locali possono essere di tipo diverso:

- *mapping semplice*: l'attributo globale corrisponde ad un attributo locale;
- *null mapping*: l'attributo globale non ha corrispondenza tra gli attributi locali;
- *default mapping*: l'attributo globale corrisponde ad un valore ben preciso dell'attributo locale;
- *and mapping*: l'attributo globale corrisponde all'unione, in uno specificato ordine, di più attributi locali;

- *mapping complesso*: l'attributo globale mappa su altre classi.

Nel generare la mapping table si possono anche inserire *nuovi attributi* a livello globale (che dovranno essere manualmente correlati ad attributi locali o settati a valori di default) o anche prevedere *or mapping*: l'attributo globale è equivalente ad ogni singolo attributo locale con cui è messo in corrispondenza (deve quindi originare più interrogazioni contemporaneamente per quella classe).

Passiamo ora a prendere in considerazione le fasi dell'integrazione che comprendono la conoscenza estensionale e come, partendo dai risultati della fase di unificazione degli schemi, si ottenga l'effettiva unificazione delle informazioni.

2.3 Relazioni Estensionali

Per comprendere le problematiche connesse alla presenza di *sovrapposizioni estensionali* occorre chiarire la distinzione tra *Oggetti* di un DataBase ed *Entità* di un certo contesto applicativo [13].
Studiando un determinato contesto possono essere infatti individuate entità caratterizzate da un certo insieme di comportamenti e proprietà, esse esprimono concetti ben definiti del *mondo reale* ai quali però possono essere associate molte rappresentazioni alternative. In particolare DataBase indipendenti forniranno modellazioni differenti, non solo descrivendo con strutture diverse le stesse proprietà ma anche andando a cogliere, in funzione delle loro finalità ed obiettivi, aspetti differenti della stessa entità. Pertanto un'entità rappresenta un concetto astratto che prescinde da una particolare rappresentazione, mentre un oggetto è uno strumento di modellazione che, utilizzando una particolare struttura, ne cattura determinati aspetti.
È evidente, a questo punto, che sorgenti autonome possono contenere oggetti corrispondenti alla stessa entità ed ognuno di questi, in parte, replicherà proprietà già presenti in altri oggetti ma potrà anche fornire un proprio contributo descrivendone di nuovi. Pertanto l'obiettivo dell'integrazione deve essere non solo il reperimento dei singoli oggetti ma piuttosto la ricomposizione dell'entità a cui sono associati.
Perché ciò sia possibile è necessario comprendere come le informazioni provenienti dalle varie sorgenti debbano essere combinate e quindi occorre impiegare sia le relazioni tra le estensioni sia quelle sulle intensioni. Le prime permettono di individuare e ricostruire le istanze della classe *astratta* di entità e le seconde specificano quali sono le proprietà effettivamente note di tali istanze.

Al fine di realizzare un'effettiva integrazione di informazioni, bisogna quindi predisporre strumenti che consentano la fusione degli oggetti recuperati dalle sor-

genti in modo di poter ricostruire le estensioni delle classi di entità del *dominio applicativo*.
L'approccio teorico adottato in MOMIS [14] si basa sulla teoria della *Formal Context Analysis* [35] la quale ha lo scopo di generare una gerarchia di ereditarietà in cui viene rappresentata la conoscenza disponibile, nell'insieme di schemi locali, riguardo ad un determinato aspetto della realtà.
Gli elementi ed i passi che caratterizzano questo approccio sono i seguenti:

1. *Assiomi Estensionali*: definizione e traduzione in proprietà intensionali;
2. *Base Extension*: verifica di congruenza degli assiomi e individuazione dell'insieme di base extension;
3. *Gerarchia Estensionale*: generazione del concept lattice.

Vediamoli in dettaglio.

2.4 Assiomi Estensionali

Gli assiomi estensionali descrivono le relazioni insiemistiche esistenti tra le estensioni delle sorgenti: date due classi (A e B) sono individuabili quattro possibili tipologie di relazioni tra esse[1]:

- *sovrapposizione*: $\forall t : S_A^t \cap S_B^t \neq \emptyset$
- *inclusione*: $\forall t : S_A^t \subseteq S_B^t$
- *equivalenza*: $\forall t : S_A^t = S_B^t$
- *disgiunzione*: $\forall t : S_A^t \cap S_B^t = \emptyset$

2.4.1 Definizione

L'analisi estensionale fatta in MOMIS si basa su due presupposti:

1. per classi appartenenti ad uno stesso cluster, e per le quali non è specificata nessuna relazione, si assume che le loro estensioni siano sovrapposte;
2. tra classi appartenenti a cluster diversi deve sussistere una relazione di disgiunzione estensionale.

[1] S_A^t e S_B^t indicano gli stati delle classi A e B all'istante t

L'ipotesi 2 è stata introdotta perché le classi globali dovrebbero raccogliere tutte le informazioni relative ad uno stesso concetto del dominio applicativo, pertanto non possono essere presenti entità aventi proprietà partizionate su cluster distinti. Se uno o più assiomi violassero questa condizione allora sarebbe necessario o rimuoverli, perché non corretti, o modificare i cluster, ad esempio fondendoli in uno unico.

In MOMIS le relazioni estensionali vengono espresse come rule nel linguaggio ODL_{I^3} , senza bisogno di estendere ulteriormente la sintassi e senza creare ulteriori complicazioni all'utilizzatore; questo è possibile in quanto assiomi estensionali e rule sono semanticamente equivalenti, infatti queste ultime non sono che un modo diverso per esprimere il fatto che un insieme di istanze di un certo concetto, e che godono di certe proprietà, appartengono ad un altro concetto.

Per quanto appena detto gli assiomi sono quindi rappresentati nel seguente modo:

- relazione di *inclusione*:
  ```
  rule RE1 forall x in B then x in A
  ```
- relazione di *disgiunzione*:
  ```
  rule RE2 forall x in (A and B) then x in bottom
  ```
  [2]
- relazione di *equivalenza*:
  ```
  rule RE3 forall x in A then x in B
  rule RE4 forall x in B then x in A
  ```

Si può notare come le relazioni di sovrapposizione non debbano essere espresse in modo esplicito, questo deriva dal precedente presupposto 1.

La definizione degli assiomi estensionali avviene in buona parte ad opera del progettista. Parte degli assiomi possono essere ricavati direttamente dalle definizioni degli schemi (ad es. una relazione di specializzazione tra classi corrisponde ad un assioma di inclusione), ma per le relazioni inter-schema l'intervento del progettista rimane fondamentale. Risulta quindi molto importante poter disporre di strumenti di ausilio sia nella fase di definizione (ad es. metodi che permettano di individuare l'insieme corretto di assiomi attraverso raffinamenti successivi) che in quella di verifica delle relazioni. Difatti l'individuazione di tutte le relazioni esistenti tra le varie classi non è mai un'operazione semplice, nemmeno per chi sia dotato di una buona conoscenza del dominio applicativo e della semantica degli schemi da integrare. Da notare che l'insieme di risultati ottenuti in queste fasi potrebbe portare ad una revisione parziale dello schema

[2] Nella logica descrittiva OCDL un tipo o classe *bottom* rappresenta un concetto incongruente, cioè che non può essere in nessun caso popolato da dati o istanze

globale fornito dal Global Schema Builder.

Considerando l'esempio di riferimento, la cui descrizione in ODL_{I^3} si può trovare in Appendice B, potremmo immaginare di definire per la classe University_Person gli assiomi estensionali rappresentati dalle seguenti rule:

```
rule RE1a forall x in School_Member
           then x in University_Student;

rule RE1b forall x in University_Student
           then x in School_Member;

rule RE2 forall x in Research_Staff
          then x in University_Worker;

rule RE3 forall x in Student
          then x in School_Member;

rule RE4 forall x in Professor
          then x in Research_Staff;

rule RE5 forall x in (Professor and School_Member)
          then x in bottom;

rule RE6 forall x in (Research_Staff and University_Student)
          then x in bottom;

rule RE7 forall x in (Research_Staff and Student)
          then x in bottom;
```

2.4.2 Traduzione in proprietà intensionali

L'affermazione di disgiunzione, se unita a quella di inclusione o di equivalenza, può generare errori impliciti. Ad esempio le seguenti asserzioni portano ad affermare che C è un concetto incoerente:

1. A e B hanno estensioni disgiunte
2. C è inclusa in A
3. C è inclusa in B

Per individuare queste ed altre possibili incoerenze si fa ricorso ad ODB-Tools, che però non riesce ad interpretare gli assiomi estensionali. Per questo prima si devono tradurre le rule estensionali in proprietà intensionali: l'inclusione si traduce in una relazione ISA, l'equivalenza genera una classe equivalente, la disgiunzione una classe intersezione di tipo bottom, mentre per tutte le coppie di classi per cui non è indicata nessuna relazione estensionale si assume che abbiano intersezione non nulla tra le rispettive estensioni. Tutte queste relazioni implicite vengono tradotte attraverso la definizione di classi virtuali, ognuna rappresentante l'intersezione tra le classi aventi una relazione estensionale. Gli assiomi relazionali vengono quindi tradotti in relazioni di ereditarietà adottando il seguente approccio:

- ogni asserzione di *equivalenza* fra due classi porta alla generazione di una classe con intensione corrispondente all'unione delle due intensioni. Ad esempio date le seguenti definizioni:

```
interface School_Member          interface University_Student
{ attribute string  name;        { attribute string name;
  attribute string  faculty;       attribute integer studcode;
  attribute integer year; } ;      attribute string faculty;
                                   attribute integer tax; } ;

rule RE1a forall x in SU.School_Member
            then x in STP.University_Student;

rule RE1b forall x in STP.University_Student
            then x in SU.School_Member;
```

  si introduce la *classe equivalente* che sostituisce `STP.University_Student` e `SU.School_Member` e la cui intensione è pari all'unione delle rispettive intensioni:

```
interface School_Member_University_Student
{ attribute string name;
  attribute string  faculty;
  attribute integer studcode;
  attribute integer year;
  attribute integer tax; } ;
```

- ogni asserzione di *inclusione* ridefinisce la classe inclusa introducendo l'ereditarietà dalla superclasse:

```
interface Research_Staff          interface CS_Person
```

```
 { attribute string name          { attribute string name; } ;
   attribute string relation;
   attribute string email;        interface Professor : CS_Person
   attribute integer deptcode;    { attribute string title;
   attribute integer sectioncode;   attribute Division belongsto;
                                    attribute string relation; } ;

rule RE4 forall x in SCS.Professor
          then x in SU.Research_Staff;
```

`SCS.Professor` è così ridefinita:

```
interface Professor : CS_Person, Research_Staff
{ attribute string title;
  attribute Division belongsto;
  attribute string relation; } ;
```

- ogni asserzione di *disgiunzione* introduce un tipo *bottom* che eredita dalle classi disgiunte:

```
rule RE5 forall x in (SCS.Professor and
    School_Member_University_Student) then x in bottom;

rule RE6 forall x in (SU.Research_Staff and
    School_Member_University_Student) then x in bottom;
```

generano le nuove classi *bottom*:

```
view Bottom_P_SS : Professor ,
                   School_Member_University_Student
 {};
view Bottom_RS_SS: Research_Staff ,
                   School_Member_University_Student
 {};
```

- ogni asserzione di *sovrapposizione*[3], porta alla definizione di una nuova classe *virtuale* che specializza entrambe le classi di partenza:

```
view Inter_CSP_UW : CS_Person , University_Worker
 {} ;
```

[3]Sono implicite per ogni coppia di classi per cui non sia stata predicata esplicitamente nessun'altra relazione estensionale

Questa classe rappresenta le estensioni comuni a `CS_Person` ed `University_Worker`. Da notare, infine, che oltre alle sovrapposizioni tra le classi inizialmente presenti nel cluster occorrerà prendere in considerazione, e quindi trasformare, anche tutte le possibili sovrapposizioni presenti tra le nuove classi virtuali inserite dalla fase stessa di traduzione[4].

2.5 Creazione della Gerarchia Estensionale

Definire una relazione tra le estensioni di due classi significa affermare l'esistenza, in classi distinte, di istanze corrispondenti alla stessa entità del dominio applicativo. Tale entità può quindi essere immaginata come appartenente ad un classe virtuale avente come schema l'unione delle intensioni delle classi di partenza[5].
In questo modo viene creato uno schema virtuale che costituisce la base per le elaborazioni successive. È importante sottolineare che questo schema, e la gerarchia estensionale da esso ottenuta, non andranno a sostituire l'insieme iniziale di classi ma si affiancheranno ad esso per arricchire la conoscenza disponibile sul cluster.

2.5.1 Verifica di congruenza ed individuazione delle Base Extension

Tramite gli assiomi estensionali si arrivano a ridefinire, o a creare, classi che vengono inserite nello schema. A questo punto ci si basa su di un componente denominato Schema Validator [28, 29] di ODB-Tools che organizza lo schema virtuale precedentemente ottenuto in una gerarchia di ereditarietà. Essa permette di scoprire eventuali classi incoerenti, cioè sussunte da classi bottom, e di individuare le base extension. Se nessuna view incoerente è stata individuata significa che gli assiomi dati sono tra loro congruenti; se, al contrario, si è trovata una vista incoerente, analizzando la gerarchia d'ereditarietà fornita da ODB-Tools si è in grado di recuperare l'assioma di disgiunzione che l'ha generata. A questo punto spetta al progettista la scelta della soluzione da adottare: può eliminare l'assioma oppure specificare un ulteriore assioma di disgiunzione fra le classi che definiscono la vista.
Ad esempio: le rule dichiarate precedentemente per la classe `University_Person` ci dicono che non è stata predicata nessuna re-

[4] L'algoritmo che deve essere impiegato è quindi di tipo iterativo e terminerà solo quando non sarà più possibile aggiungere classi virtuali che non siano specializzazione di classi bottom

[5] In caso di disgiunzione basta adottare il formalismo illustrato per imporre che la nuova classe introdotta abbia estensione sempre vuota

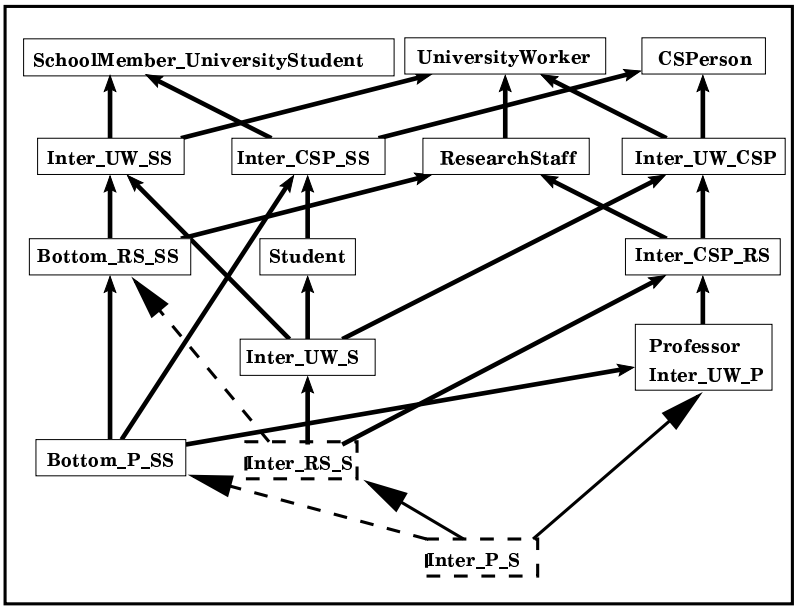


Figura 2.4: Gerarchia di ereditarietà della classe University_Person

lazione tra le classi `Student` e `Professor`, dalle quali ha quindi origine la vista `Inter_P_S`:

```
view Inter_P_S : Professor , Student
 { } ;
```

La Figura 2.4 mette in evidenza che il tipo `Inter_P_S` è sussunto dal tipo bottom `Bottom_P_SS` costituendone una specializzazione: il progettista può a questo punto scegliere se eliminare la rule RE6 che ha generato la definizione `Bottom_P_SS` oppure specificare la rule che esprime la disgiunzione tra le estensioni di `Professor` e `Student`:

```
rule RE8 forall x in (Professor and Student)
          then x in bottom;
```

Un discorso analogo si può fare per il tipo `Inter_RS_SS`.

Definizione 1 (Base Extension) *Una base extension è un sottoinsieme dell'insieme complessivo delle estensioni ed identifica oggetti realmente esistenti in una o più sorgenti.*

Conclusa la fase di revisione e dopo aver eliminato tutti i concetti bottom, la gerarchia illustrata in figura 2.4 permette di ottenere una descrizione completa

Base Extension	1	2	3	4	5	6	7	8	9	10	11
University_Worker **C1**	X	X	X		X	X	X				X
Research_Staff **C2**	X				X						X
School_Member **C3**		X					X	X	X	X	
CS_Person **C4**	X	X		X		X		X	X		X
Professor **C5**	X										
Student **C6**		X						X			
University_Student **C7**		X					X	X	X	X	

Figura 2.5: Tabella delle base extension della classe University_Person

delle base extension. Ad ogni classe presente nello schema corrisponde difatti una base extension composta da tutte le classi di cui è specializzazione. Ad esempio la classe `Inter_CSP_RS` rappresenta la base extension generata dall'intersezione tra le classi `SU.Research_Staff`, `SU.University_Worker` e `SCS.CS_Person`.

La partizione dell'insieme complessivo delle estensioni, costituita da tutte le base extension, si può rappresentare mediante una matrice dove le righe indicano le classi sorgenti e le colonne le base extension. In Figura 2.5[6] è riportata la matrice delle base extension relativa alla classe globale University_Person. Leggendola per righe si evince che l'estensione di una sorgente è data dall'unione delle base extension corrispondenti alle colonne per cui è presente un segno nelle casella. Per colonne invece si ha un segno in corrispondenza di ogni riga la cui classe comprende nella propria estensione la base extension.

2.5.2 Generazione della Gerarchia Estensionale

Partendo dalle base extension individuate al punto precedente, ad ogni classe globale si associa una gerarchia estensionale. In Figura 2.6 è illustrata la rappresentazione di quella relativa alla classe `Univerity_Person`[7].

[6] C1, . . . ,C7 sono gli identificativi di ogni **classe locale** nell'ambito di una classe globale, da non confondere con quelli che saranno introdotti nella prossima sezione 2.5.2 e riguarderanno le classi virtuali

[7] C1, . . . ,C11 sono gli identificativi di ogni **classe virtuale** costituente la gerarchia estensionale associata ad una classe globale

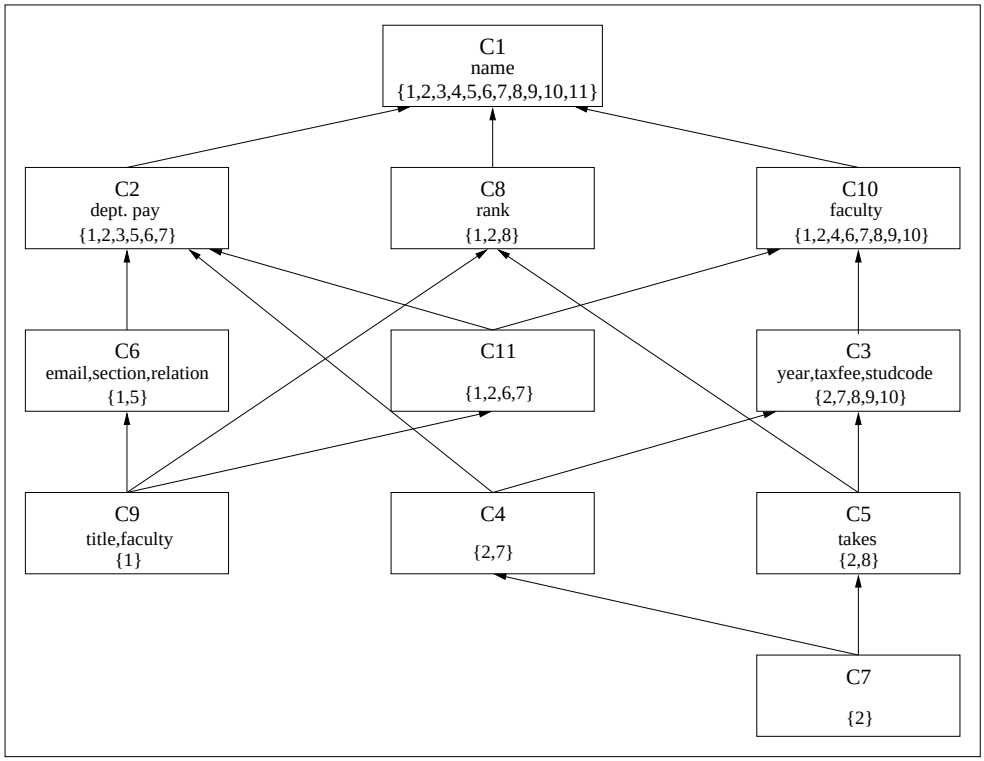


Figura 2.6: Rappresentazione della gerarchia estensionale della classe University_Person

Queste gerarchie sono dunque relazionate alle informazioni contenute nelle base extension, come dimostra il fatto che ognuna delle classi virtuali che le compongono è caratterizzata da:

- una *intensione* corrispondente agli schemi delle base extension presenti nella classe globale;
- una *estensione* data dall'unione di tutte le base extension che hanno almeno tutti gli attributi presenti nell'intensione della classe.

Il modo nel quale sono formate le intensioni delle classi virtuali ci permette di individuare un insieme di intensioni sovrapposte rappresentanti le descrizioni di tutti i tipi di entità[8] raccolte nel cluster. Per ogni intensione valgono due condizioni: esisterà almeno un tipo di entità appartenente alla classe globale

[8]Si parla di "tipi di entità" e non di "entità" in quanto quello che riguarda la descrizione estensionale esprime condizioni di potenziale esistenza: un insieme di entità individuato non è necessariamente popolato

Classe Virtuale	Intensione	Estensione
C1	name	1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11
C2	name; dept; pay	1; 2; 3; 5; 6; 7
C3	name; faculty; year; tax; studcode	2; 7; 8; 9; 10
C4	name; dept; pay; faculty; year; tax; studcode	2; 7
C5	name; faculty; year; tax; rank; takes; studcode	2; 8
C6	name; dept; pay; email; relation; section	1; 5
C7	name; dept; pay; faculty; year; tax; rank; takes; studcode	2
C8	name; rank	1; 2; 8
C9	name; dept; pay; faculty; rank; email; relation; title; section	1
C10	name; faculty	1; 2; 4; 6; 7; 8; 9; 10
C11	name; dept; pay; faculty	1; 2; 6; 7

Figura 2.7: Tabella della gerarchia estensionale della classe University_Person

e caratterizzato da tutti e soli gli attributi dell'intensione stessa ed inoltre non saranno previsti tipi di entità non associati ad almeno una delle intensioni. In Figura 2.7 viene presentata una tabella nella quale ad ogni classe virtuale si associano le rispettive intensione (comprendente attributi globali) ed estensione (comprendente base extension).

Il processo che per ogni classe globale, partendo dalle base extension, arriva a creare il concept lattice rappresentante la relativa gerarchia estensionale è stato codificato in un algoritmo ed esaurientemente descritto in [14].

2.6 Utilizzo della Conoscenza Estensionale nel Query Manager

Le gerarchie estensionali e le base extension, ottenute come descritto nelle precedenti sezioni di questo capitolo, formano quella *Conoscenza Estensionale* che verrà poi utilizzata in fase di query processing. Ancora una volta viene evidenzia-

ta la differente natura dei moduli del MOMIS Mediator, nonché il diverso uso che se ne farà.

Le fasi illustrate in precedenza sono caratteristiche dell'Extensional Hierarchy Builder, i risultati ottenuti (gerarchie estensionali e descrizioni delle base extension) vanno aggiunti a quelli forniti Global Schema Builder (schema globale e mapping table) per riorganizzare la conoscenza e migliorare il processo di interrogazione.

Il Query Manager, preposto alla fase di query processing, sfrutterà tali conoscenze[9] sia nella sottofase di *generazione del piano di accesso* (Query Plan Generation) che in quella di *esecuzione* (Query Execution).

Nella *Query Plan Generation*, la possibilità di disporre delle informazioni estensionali permetterà al Query Manager di individuare la classe virtuale dotata di tutte le proprietà richieste (grazie ad un'analisi tra gli attributi presenti nella query e l'intensione della classe stessa) e poi da questa (sfruttando le informazioni dedotte dalla sua estensione) potrà risalire alle classi locali interessate dalla query in questione. Questo permetterà di migliorare le prestazioni poiché le classi locali alle quali si accederà effettivamente per reperire i dati cercati saranno in numero ridotto, ma allo stesso tempo garantirà che la risposta generata sia corretta e, quanto più possibile, completa e minima. La correttezza si riferisce al fatto di essere in grado di reperire le sole istanze soddisfacenti tutte le condizioni imposte, la completezza è relativa alla capacità di interrogare tutte le sorgenti che possano contribuire alla risposta, la minimalità indica la possibilità di eliminare inutili duplicazioni e ridondanze. Per riuscire in questo intento è indispensabile poter usufruire sia della conoscenza intensionale che di quella estensionale: la prima è preposta alla valutazione della correttezza delle risposte fornite, la seconda ne assicura la tendenza alla completezza ed alla minimalità. Si noti che la correttezza è effettivamente garantita, mentre per quanto riguarda la completezza e la minimalità non si può assicurare il loro assoluto ottenimento. Il grado di completezza e minimalità di una risposta difatti sono legati alla scelta dei cluster e delle classi globali, quello che si può garantire è quindi che una risposta sia completa e minima rispetto alla scelta compiuta per i cluster, ma non in termini assoluti.

Alla fine della *Query Execution* si pone invece il problema della *fusione delle istanze*. Ogni sorgente fornisce una risposta alla subquery posta su di essa ed a

[9] L'utilizzo della conoscenza estensionale ai fini della traduzione da *Basic Query* a *Local Query* è l'argomento principale del software realizzato nell'ambito di questa tesi e quindi gli aspetti qui accennati saranno trattati più approfonditamente nei capitoli 3 e 4

questo punto sorgono due problemi: come identificare istanze equivalenti, cioè relative ad una stessa entità del modo reale, ma provenienti da sorgenti distinte? Ed ancora: come generare un'unica istanza a partire da due equivalenti? La soluzione proposta per MOMIS [14] vuole poter gestire sia la capacità di fondere gli oggetti che quella di crearne di complessi. Per assicurare la fusione delle istanze si definiscono su ogni classe globale una o più chiavi che saranno poi propagate alle classi virtuali della gerarchia estensionale. La fusione è realizzata mediante degli outer-join sui valori delle chiavi stesse, questo avviene al termine della fase di query processing quando le sorgenti rendono disponibili le risposte alle rispettive query locali. Infine ogni oggetto finale viene dotato di un proprio OID (Object IDentifier): questo permette di conservare la capacità di descrivere anche oggetti complessi.

Capitolo 3

Il Query Manager di MOMIS

Il Query Manager è preposto alla fase di query processing nella quale, sfruttando le strutture dati prodotte nella fase di integrazione (svolta dal Global Schema Builder e dall'Extensional Hierarchy Builder), risolve ed esegue le interrogazioni poste dall'utente. Per svolgere il suo compito deve prevedere due funzionalità di base: *Definizione del Query Plan* ed *Esecuzione della Query*.

La prima si riferisce al fatto che la query posta sullo schema globale (che chiameremo *Global Query*) non può essere inviata direttamente ai wrapper, poiché è espressa in termini globali che possono essere diversi (sia sintatticamente che semanticamente) rispetto a quelli locali ed inoltre è posta su di una vista integrata, quindi conterrà e richiederà dati di solito non presenti in tutte le sorgenti. Bisogna quindi generare *un piano di accesso ed esecuzione* (detto anche Query Plan) che ci permetta di individuare le informazioni recuperabili dalle singole sorgenti; questo piano sarà formato da un insieme di query eseguibili localmente (*Local Query*), ma conterrà anche informazioni su come poter ricostruire, da quelle fornite da ogni singola sorgente, la risposta globale che, come già più volte accennato, deve essere corretta e, quanto più possibile, completa e minima.

La seconda funzionalità deve permettere l'esecuzione della query, cioè l'applicazione del piano ad essa associato ed individuato in precedenza. In questa fase i wrapper eseguono le query locali direttamente sulle sorgenti e le risposte ottenute da esse devono ora essere elaborate e combinate al fine di ottenere una risposta globale che possieda le qualità appena ricordate.

I passi che il Query Manager deve compiere per giungere ad una risposta di questo tipo, partendo dalla query posta dall'utente sullo schema globale, sono stati individuati nei seguenti:

1. Ottimizzazione semantica globale
2. Parsing e validazione della query globale
3. Decomposizione della Global Query in Basic Query
4. Individuazione delle Classi Locali coinvolte
5. Traduzione da Basic Query a Local Query
6. Ottimizzazione semantica locale
7. Esecuzione della query

Vediamoli in dettaglio.

3.1 Ottimizzazione semantica globale, parsing e validazione

Come già detto, la query posta dall'utente non è direttamente eseguibile da parte del Query Manager, ma deve subire alcuni processi che la rendano tale. Il primo consiste nella sua acquisizione: la query viene caricata nelle strutture dati che ne rappresentano il contenuto e che sono predisposte dal Query Manager per poterla gestire.

Per acquisire una query bisogna verificarne la correttezza sintattica e semantica e, se possibile, ottimizzarla; questo avviene mediante moduli di *parsing*, *validazione* e *ottimizzazione*. Un modulo di parsing è un modulo di riconoscimento grammaticale ed ha quindi il compito di verificare la correttezza sintattica di un'espressione (nel nostro caso una query) rispetto ad una determinata grammatica (per noi OQL_{I^3}) ed infine di produrre un'immagine dell'espressione stessa in memoria centrale. La struttura generata sarà utilizzata per effettuare il controllo semantico con il quale si accerta la correttezza della query rispetto ad un determinato schema (si verifica cioè se le classi e gli attributi coinvolti appartengono effettivamente allo schema interrogato) e la si valida. La query così validata sarà sottoposta alla fase di ottimizzazione semantica che, in modo automatico, modificherà l'interrogazione e quindi la sua immagine in memoria. In Figura 3.1 vengono mostrati questi processi relativamente a MOMIS. Si può notare che nel

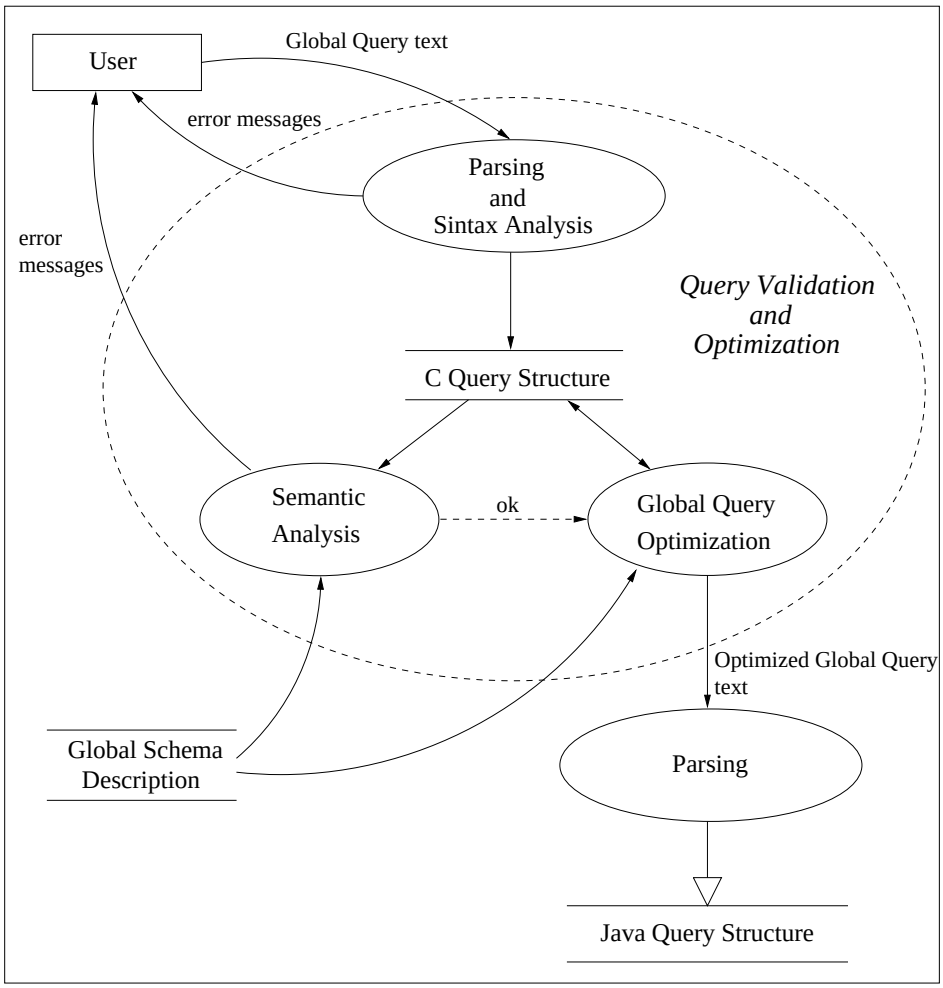


Figura 3.1: Schema di acquisizione di una query

nostro caso le fasi di ottimizzazione e parsing sono invertite, questo deriva dalla necessità di sfruttare il software esistente[1].

L'ottimizzazione di cui si è parlato è realizzabile se sono presenti regole di integrità inter-sorgenti definite sullo schema globale. Tali vincoli sono rappresentati mediante rule ODL_{I^3} sullo schema globale; ODB-Tools li sfrutta per riformulare la query iniziale: ne viene prodotta una semanticamente equivalente ma che risulta eseguibile in modo più efficiente. Ad esempio vengono aggiunte nuove condizioni che potrebbero diminuire i tempi della risposta (nelle sorgenti difatti potrebbero essere presenti indici sui predicati introdotti) oppure vengono eliminate condizioni ridondanti, questo permette di evitare costose navigazioni, necessarie se i predi-

[1]Per una più completa trattazione del software si rimanda al Capitolo 4 ed a [13]

cati scartati dovessero contenere join impliciti. Vediamo un esempio concreto. Supponiamo che nel dominio universitario introdotto nella sezione 2.2 esista una relazione a livello globale tra la facoltà a cui appartiene un docente (attributo *faculty*) e lo stipendio dello stesso (*pay*). Il progettista può quindi definire la seguente regola di integrità sulla classe globale University_Person utilizzando ODL_{I^3}:

```
rule RG1  for all X in University_Person: (X.faculty = 'CS')
             then X.pay > 30000
```

Supponiamo a questo punto di porre la seguente interrogazione, con la quale ricerchiamo i nomi di tutti i corsi di durata superiore ai 3 mesi tenuti da professori della facoltà di Computer Science:

```
Q:  select name
    from Course
    where length > 3
    and lecturer.faculty = 'CS'
```

Il Query Manager attiva ODB-Tools che espande semanticamente la query Q sfruttando la regola RG1 ed ottiene la seguente:

```
Q':  select name
     from Course
     where length > 3
     and lecturer.faculty = 'CS'
     and lecturer.pay > 30000
```

Le query Q e Q' sono semanticamente equivalenti, ma in Q' è stato aggiunto un predicato nelle clausola *where*, che potrebbe velocizzare la fase di query processing delle singole sorgenti se sull'attributo aggiunto *pay* fossero definiti degli indici. Il prezzo da pagare è un'aumento della complessità della fase di query plan definition: deve essere riformulata una query più complessa.

3.2 Decomposizione in Basic Query

A questo punto la query globale è validata ed ottimizzata, inoltre è stata verificata la sua correttezza sintattica. Una query di questo tipo è indicata in Figura 3.2 come *Global Query*, questa figura inoltre riassume tutti i passi che da questo punto portano all'esecuzione della query e all'ottenimento del risultato.

Una Global Query contiene richieste che, in generale, non potranno essere soddisfatte tutte localmente: la query deve quindi essere analizzata per individuare le classi globali coinvolte e le proprietà da recuperare da ognuna di esse (questo

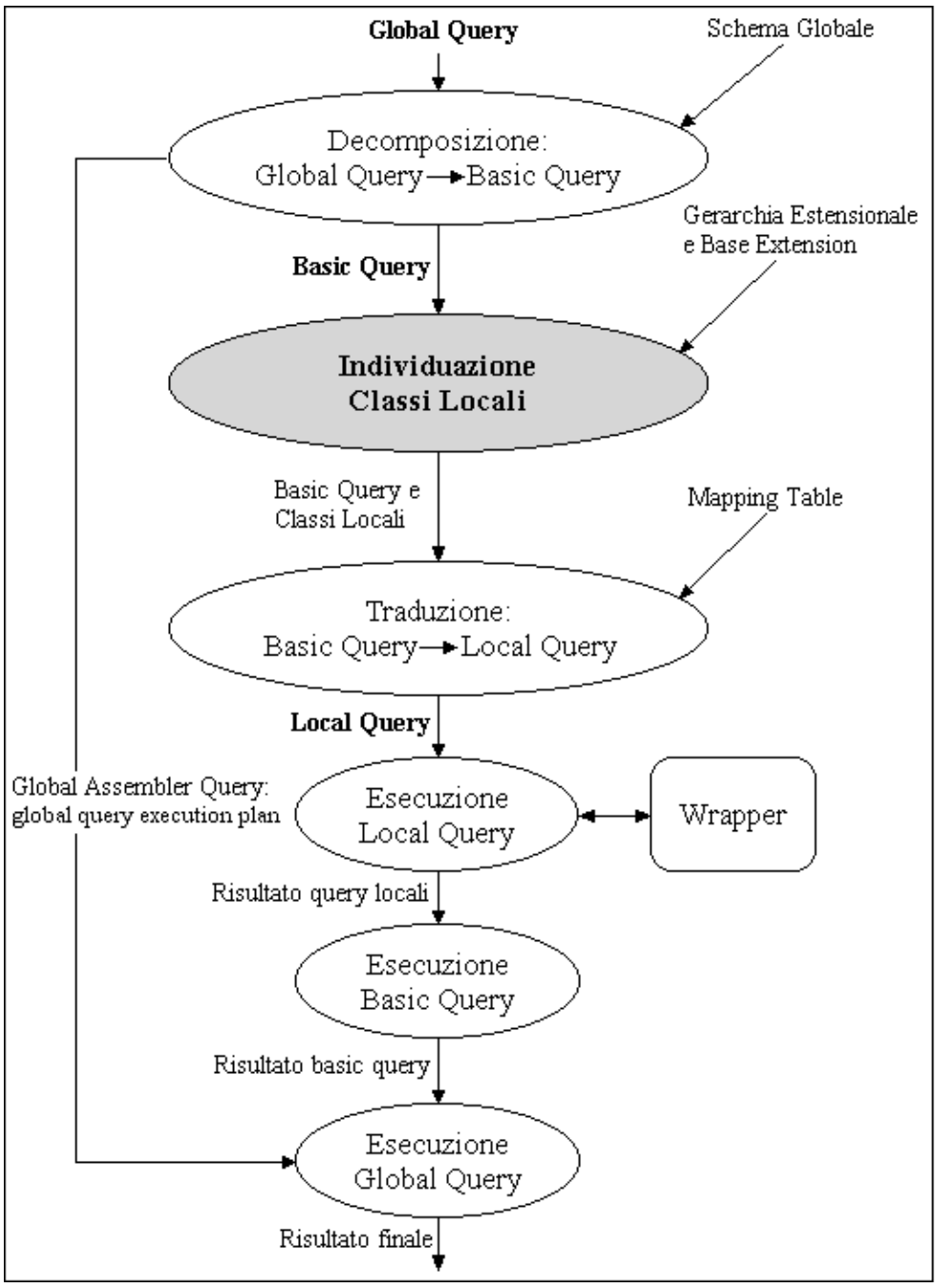


Figura 3.2: Operazioni del Query Manager

è analogo a quello che avviene in un normale DBMS), poi dovrà essere tradotta ed inoltrata alle sorgenti. Bisogna quindi predisporre un *piano di esecuzione* che

individui le query rivolte ad ogni classe globale (*Basic Query*) e come manipolare i risultati restituiti da queste per ottenere la risposta globale. Vale dunque la seguente definizione:

Definizione 2 (Basic Query) *Una Basic Query è una query contenente tutte le richieste rivolte ad una singola classe globale.*

Le Basic Query rappresentano quindi gli elementi di base del piano di esecuzione e, essendo rivolte ad una singola classe globale, non conterranno operatori complessi (in particolare join tra classi), ma questi dovranno essere presenti nel piano di esecuzione generato affinché, attraverso le operazioni da essi svolte, sia possibile ricostruire la risposta globale. Le Basic Query sono espresse mediante un sottoinsieme del linguaggio OQL, in particolare non sono ammessi:

- join espliciti tra classe diverse (è permessa solamente la navigazione tra aggregazioni ed associazioni per recuperare oggetti complessi),
- subquery (query innestate),
- operatori di ordinamento (order by) o di conversione (listtoset, element, flatten),
- restituzione di strutture complesse (list, array, struct).

Gli operatori complessi non gestiti dalle Basic Query dovrebbero essere implementati ad un livello superiore (come ad esempio funzioni invocabili dal piano di esecuzione) perché agiranno su insiemi di informazioni già integrate.

Nella fase di decomposizione da Global Query a Basic Query oltre all'individuazione di queste ultime (per la quale serve conoscere lo schema globale), deve essere anche creata la *Global Assembler Query* (o query assemblatrice) che ha il compito di generare il risultato finale della Global Query partendo dai risultati parziali ottenuti dalle singole Basic Query, come illustrato in Sezione 3.5.

Consideriamo, ad esempio, che in un ipotetico schema globale, derivante dall'aggregazione di DataBase calcistici, esistano, tra le altre, due classi globali: una contenente i dati relativi ai giocatori (`Player(name, born_date, team_name, nationality, ...)`) e l'altra quelli relativi alle squadre (`Team(name, country, address, ...)`). Un'interrogazione che richieda tutti i calciatori italiani militanti in squadre straniere, avrebbe la seguente rappresentazione:

```
Q:   select Player.name, Team.name
     from Player, Team
```

```
      where Player.team_name = Team.name
      and Player.nationality = 'Italian'
      and Team.country <> 'Italy'
```

Nella query Q è presente un join tra classi globali, che può essere trattato nel seguente modo:

```
Qa:   select name, team_name
      from Player
      where nationality = 'Italian'

Qb:   select name
      from Team
      where country <> 'Italy'

Qc:   select Qa.name, Qb.name
      from Qa, Qb
      where Qa.team_name = Qb.name
```

Più precisamente: Qa e Qb sono le Basic Query rivolte alle classi globali, mentre Qc rappresenta l'operazione di join che il Query Manager dovrà compiere invocando la corrispondente funzione.

3.3 Individuazione delle Classi Locali coinvolte

Una volta ottenute le Basic Query bisogna, per ognuna di esse, individuare le sorgenti e le classi locali alle quali effettivamente accedere per reperire i dati richiesti (Figura 3.2).

Ricordiamo ancora una volta che la risposta finale che si desidera ottenere deve essere corretta e, quanto più possibile, completa e minima.
Una Basic Query è posta su di un'unica classe globale, quindi coinvolge un unico cluster; l'utilizzo delle informazioni (intensionali) contenute nella mapping table consente solo in parte di ottenere una riposta con le caratteristiche su esposte: consente di determinare quali tra le proprietà richieste dalla Basic Query siano presenti in ogni classe locale costituente il cluster, ma non permette di ricostruire gli oggetti virtuali rappresentanti le entità descritte in termini globali. La risposta così ottenuta potrà risultare corretta, ma molto probabilmente sarà anche incompleta: le entità del contesto applicativo hanno la caratteristica di possedere proprietà distribuite su più classi locali appartenenti anche a sorgenti diverse, considerando le sorgenti singolarmente si perde la capacità di individuare e ricostruire tali entità.

L'utilizzo di informazioni estensionali (contenute nelle base extension e nella gerarchia estensionale), unitamente a quelle intensionali della mapping table, permettono quindi a livello di Basic Query di individuare l'insieme ottimale di classi locali che devono essere effettivamente interrogate per fornire la risposta globale.

L'importanza dell'utilizzo concertato dei due tipi di conoscenza può essere chiarito meglio da un esempio. Consideriamo sempre il nostro esempio di riferimento (Sezione 2.2 e Appendice B) e poniamo su di esso la seguente interrogazione:

```
select title, email
from University_Person
```

Prendiamo ora in considerazione la mapping table della classe globale sulla quale è posta questa query (`University_Person`): nessuna delle sue classi locali possiede entrambi gli attributi richiesti. Sfruttando solamente la conoscenza intensionale fornitaci dalla mapping table (Figura 2.2) potremmo arrivare a generare una risposta recuperando `title` da `Professor` e `email` da `Research_Staff`, ma tale risposta risulterà inevitabilmente incompleta in quanto ottenuta facendo riferimento a istanze singole che non hanno un criterio di fusione definito. Avendo a disposizione anche la conoscenza estensionale si può pervenire ad una risposta più completa: dalla gerarchia estensionale (Figura 2.7) si evince che la classe virtuale C9 ha nella propria intensione entrambi gli attributi cercati, l'estensione di questa classe è formata dalla base extension 1 e quindi fondendo le classi `University_Worker`, `Research_Staff`, `CS_Person` e `Professor` (che formano la base extension 1) si ottiene tale risposta. La minimalità della risposta è dovuta allo sfruttamento delle informazioni derivanti dalle relazioni estensionali introdotte. Questo sarà approfondito nella Sezione 3.3.4, ci limitiamo qui a considerare che se le duplicazioni non venissero trattate correttamente porterebbero ad un maggior costo di esecuzione ed a un errore nella risposta: dati inerenti ad una determinata entità del contesto applicativo verrebbero presentati più volte figurando di fatto come associati ad entità diverse.

I passi costituenti la fase di individuazione delle classi locali coinvolte, cioè dell'insieme minimo ed ottimale di classi locali a cui accedere per ottenere una risposta corretta, completa e minima, sono schematizzati in Figura 3.3 e spiegati qui di seguito.

3.3.1 Analisi della Basic Query

L'analisi della Basic Query consiste nell'esaminare il testo della query per individuare gli attributi globali che interessano. Sono da prendere in considerazione

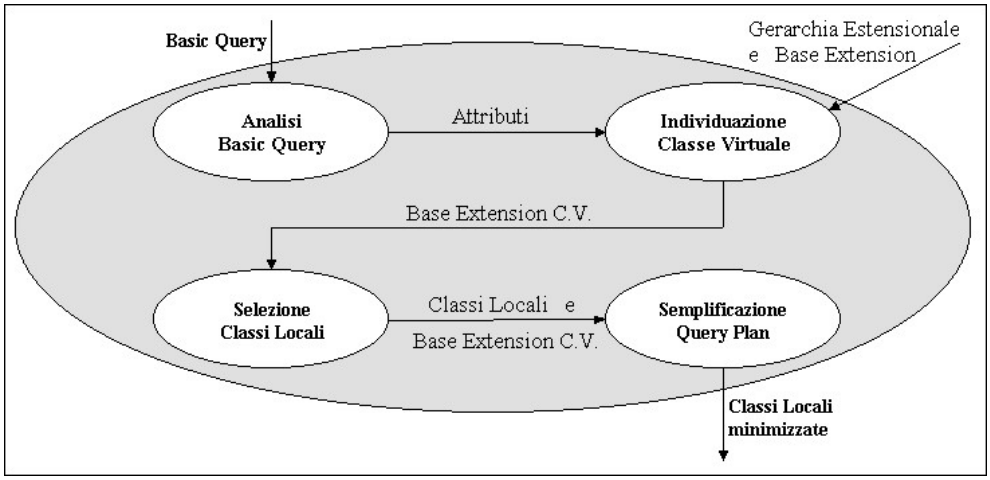


Figura 3.3: Passi della fase di individuazione delle classi locali

sia i predicati di proiezione che di selezione, cioè la query viene scandita per determinare gli attributi utilizzati nelle clausole SELECT e WHERE.

Consideriamo la seguente query posta sulla classe globale University_Person dell'esempio di riferimento:

```
Q1:  select name, rank
     from University_Person
     where faculty = 'CS'
     and year < 1973
```

In Q1 sono presenti gli attributi globali `name`, `rank`, `faculty` e `year`, questi sono da utilizzare nei passi successivi, ma non solo: è rispetto ad essi che deve essere valutata la correttezza della risposta.

3.3.2 Individuazione della Classe Virtuale Target

Bisogna adesso individuare su quale, tra le classi virtuali della gerarchia estensionale, va posta la query, bisogna cioè determinare la *Classe Virtuale Target*.

Per farlo occorre conoscere la gerarchia estensionale e l'insieme di attributi individuati al punto precedente. Poi, analizzando gli schemi delle classi virtuali, si individuano quelle che contengono nella propria intensione tutti gli attributi cercati, quindi che godono di tutte le proprietà richieste. In generale le classi virtuali così ottenute possono essere più di una; se questo avviene, si prende

come classe virtuale target quella più generalizzata, cioè quella con estensione massima, cioè quella composta da più base extension.

Tornando al nostro esempio e facendo riferimento alla gerarchia estensionale di University_Person di Figura 2.7, si vede che le classi virtuali C5 e C7 contengono nella propria intensione tutti gli attributi presenti in Q1. Tra esse quella più generalizzata è C5 che ha un'estensione formata dalle base extension 2 e 8 (mentre quella di C7 conteneva solo la 2). La nostra classe target sarà quindi C5.

Vediamo un altro esempio:

```
Q2:  select pay, rank
     from University_Person
```

L'analisi di questa semplicissima query ci fornisce l'insieme di attributi {`pay`, `rank`}; le classi virtuali che contengono entrambi nella loro intensione sono C7 e C9, la prima ha estensione formata dalla sola base extension 1, la seconda dalla sola 2. Nessuna delle classi virtuali candidate ad essere quella target ha estensione maggiore dell'altra, in questo caso nella fase seguente bisognerà considerare l'insieme delle base extension delle due classi.

3.3.3 Selezione delle Classi Locali

Una volta note le base extension, sono anche automaticamente note le classi locali alle quali rivolgere le richieste di dati. Questo è possibile perché le strutture dati associate ad ogni base extension indicano quali classi locali devono essere combinate per ricostruirne l'estensione.

Nel nostro esempio relativo alla query Q1 adesso disponiamo delle base extension 2 e 8, esaminando la tabella delle base extension di University_Person riportata in Figura 2.5, dalle colonne delle base extension in questione si evince che le classi locali alle quali accedere risultano essere SU.University_Worker, SU.School_Member, SCS.CS_Person, SCS.Student, STP.University_Student.

3.3.4 Semplificazione del Query Plan

Abbiamo visto precedentemente che conoscendo le base extension si conoscono contestualmente anche le classi locali ad esse relazionate; il passo precedente quindi non viene effettivamente eseguito. Esso viene però ugualmente tenuto distinto dagli altri per sottolineare la qualificazione delle informazioni che in esso si rendono disponibili. Ogni passo difatti porta ad un insieme di nozioni più qualificato: si passa dagli attributi alle base extension e dalle base extension alle classi

locali, che costituisce proprio l'output desiderato per questa fase (Figura 3.3). Vedremo ora come l'insieme così ottenuto non sia in realtà ancora quello ottimale.

Una volta note le base extension e le classi locali che le compongono, si può pensare di mettere in atto alcuni passi di semplificazione che, riducendo il numero di classi locali alle quali effettivamente accedere, permettano di diminuire il numero di sottoquery da generare, semplificando di conseguenza il query plan.
L'obiettivo di questa fase è quello di limitare il numero di classi locali alle quali rivolgere le sottoquery; per ottenere questo risultato il processo di semplificazione avviene in due momenti ben distinti:

1. Eliminazione di base extension
2. Eliminazione di classi locali

Eliminazione di base extension. Per prima cosa difatti può succedere che non tutte le base extension recuperate siano effettivamente da ricostruire, cioè alcune base extension potrebbero risultare ridondanti al fine di ottenere una risposta minima. Per meglio comprendere questo aspetto introduciamo la seguente definizione:

Definizione 3 (Dominazione) *Date due Base Extension BE_1, BE_2 ed un insieme di attributi A = $\{a_1, \ldots, a_n\}$, si dice che BE_1* domina *BE_2 rispetto ad A, se A è compreso nelle intensioni di entrambe e se le classi che compongono BE_1 sono un sottoinsieme di quelle che formano BE_2.*

Per ricostruire una base extension si deve effettuare il join tra tutte le classi locali che la compongono, ma le base extension rappresentano insiemi disgiunti di entità, quindi dal join tra le classi si ottiene in realtà un soprainsieme della base extension desiderata. Facciamo riferimento alla Figura 3.4 nelle quale sono rappresentate 3 classi locali (A, B, C) e le base extension che esse vanno a formare (1, . . . , 7). La base extension 2 è composta dalle classi A e B, ma il join (intersezione, evidenziata in grigio) tra queste due classi produce oltre alle entità della BE_2 (in grigio chiaro) anche una rappresentazione parziale delle entità della base extension 5 (in grigio scuro), parziale perché mancano alcuni attributi: quelli di C. La BE_5 è data da: $A \bigcap B \bigcap C$, quindi $BE_2 = (A \bigcap B) - BE_5$. Ricostruendo BE_2 dunque inevitabilmente si ricostruisce anche una parte di BE_5, quella limitata agli attributi di A e B. La definizione su riportata ci dice che, rispetto a questo insieme di attributi, BE_2 domina BE_5.

Quindi si possono, anzi si devono, non prendere in considerazione le base extension dominate, ma si devono ricostruire solo quelle dominanti per evitare duplicazioni.

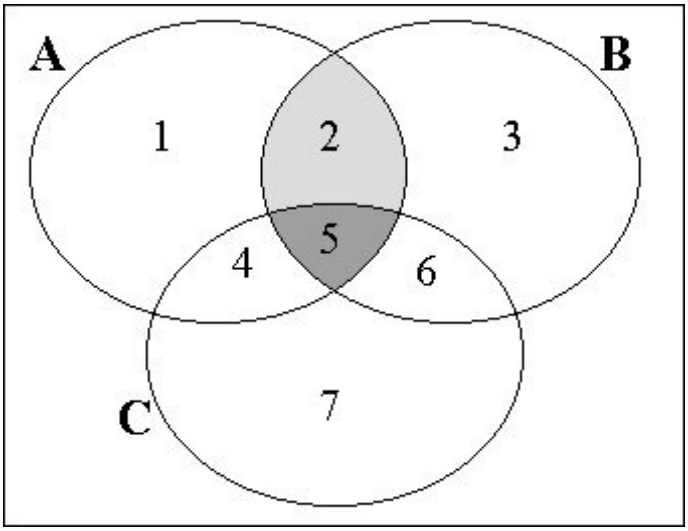


Figura 3.4: Esempio di dominazione tra Base Extension

Tornando alla query di esempio Q1 avevamo individuato l'insieme di base extension {2, 8}: rispetto all'insieme di attributi della query si vede che la 8 domina la 2, difatti la prima è composta dalle classi locali SU.School_Member, SCS.CS_Person, SCS.Student e STP.University_Student, mentre la seconda oltre a queste ha anche SU.University_Worker. Dobbiamo quindi ricostruire solo la base extension 8.

Scartando le base extension dominate da altre abbiamo notevolmente limitato le possibilità di duplicazioni, ma non le abbiamo eliminate del tutto. Nel caso infatti che due base extension ne dominino una terza, questa viene scartata, ma le altre due una volta ricostruite generano insiemi di entità parzialmente sovrapposti: la sovrapposizione è rappresentata proprio dall'estensione della base extension eliminata. Per ovviare a ciò è necessario tenere traccia, ad esempio nel query plan, di come unire le base extension per una determinata Basic Query: di default bisognerà effettuare una unione delle entità, ma se si presenta un caso come quello appena discusso bisognerà effettuare un outer join tra le base extension al fine di presentare una sola volta la porzione comune.

Eliminazione di classi locali. Una volta ottenuto l'insieme minimo di base extension, ognuna di questa viene esaminata separatamente al fine di diminuire il numero di classi locali alle quali effettivamente accedere. Questo permette di semplificare il query plan perché si riduce il numero di subquery da generare, ma

non solo: si potrebbe anche scoprire che eliminando alcune classi locali si può evitare del tutto di accedere ad una o più sorgenti, migliorando le prestazioni. Le classi locali che rimarranno dopo la semplificazione devono in ogni caso consentire la ricostruzione della base extension. Per capire quali classi locali possano essere tralasciate occorre considerare il tipo di relazioni estensionali esistenti ed anche il tipo di DataBase.

In una base extension può accadere che due tra le sue classi siano estensionalmente equivalenti, pertanto, se nessuna delle due aggiunge informazioni rispetto all'altra, è possibile effettuare una semplificazione del piano scartandone una delle due. Se invece una aggiunge informazioni, cioé contiene attributi richiesti dalla query e non presenti nell'altra, quella da scartare sarà l'altra.

Consideriamo di nuovo l'esempio della query Q1: a questo punto è rimasta la sola base extension 8 contenente le classi SU.School_Member, SCS.CS_Person, SCS.Student e STP.University_Student. Dalle relazioni estensionali introdotte in Sezione 2.4.1 si ha che le classi SU.School_Member e STP.University_Student sono estensionalmente equivalenti, ma mentre la seconda presenta solo gli attributi `name`, `faculty` e `rank`, la prima ha oltre a questi anche `year`; si può quindi evitare di accedere a STP.University_Student, e di conseguenza alla sorgente Tax_Position.

Un'altra situazione in cui risulta possibile tralasciare classi locali relativamente ad una stessa base extension, si verifica quando due classi appartenenti allo stesso DataBase sono una la specializzazione dell'altra. In questo caso, se la sorgente in questione è object-oriented o se la classe più specializzata contiene tutte le proprietà richieste, risulta lecito scartare la superclasse, visto che non produrrebbe nessuna conoscenza aggiuntiva.

Torniamo alla query Q1: dall'esame delle sorgenti si ha che SCS.Student è sottoclasse di SCS.CS_Person e che la sorgente di appartenenza (COMPUTER_SCIENCE) è object, quindi possiamo scartare la classe locale SCS.CS_Person.

Alla fine quindi, la nostra query Q1 sarà diretta soltanto alle classi locali SU.School_Member e SCS.Student.

L'insieme minimo di classi ottenuto secondo i due criteri di semplificazione testé illustrati potrebbe non essere però sufficiente per ricostruire tutte le base extension coinvolte.
Può accadere difatti che la chiave primaria non sia la stessa per tutte le classi e che quindi il join diretto tra due di esse non sia consentito; per essere in grado di ricostruire la base extension, in casi come questo, bisogna ricorrere ad una terza classe di collegamento tra le due. Per assicurarsi che questo avvenga non è

sufficiente considerare nei passi precedenti i soli attributi presenti nella query, ma bisogna tener conto, ove questi non siano sufficienti a ricostruire la base extension, anche di quegli attributi necessari per fondere le classi; questi sono determinanti nella scelta se scartare o meno una determinata classe che, se anche dovesse risultare da eliminare secondo i criteri discussi, potrebbe essere decisiva come classe di collegamento per ricostruire la base extension e quindi andrebbe tenuta.

In pratica, una volta ottenuto l'insieme minimo di classi, si deve vedere se con queste risulta possibile ricostruire le base extension coinvolte; una base extension è ricostruibile se è possibile fondere tra loro, mediante gli attributi individuati, tutte le classi che la formano. Se una base extension non dovesse risultare ricostruibile si deve recuperare una classe tra quelle precedentemente scartate che consenta di effettuare tutti i join necessari; se questo è reso possibile mediante attributi non compresi tra quelli della query, anche questi nuovi attributi dovranno essere recuperati.

Concludendo: alla fine della fase di individuazione delle classi locali si avrà a disposizione un insieme minimo di classi alle quali effettivamente accedere. Queste classi, unitamente agli attributi richiesti dalla Basic Query ed a quelli eventualmente aggiunti per poter ricostruire la base extension, costituiscono l'input del passo seguente. Ma da questa fase non si deve ottenere solo un elenco di classi locali: si dovrà memorizzare anche come queste classi vanno fuse per ricostruire le base extension e come le varie base extension vanno combinate tra loro (se semplicemente unite o se invece necessitano degli outer join).

Questi aspetti, ed altri emersi in precedenza, saranno ripresi nel Capitolo 4 nel quale verrà illustrato il software realizzato.

3.4 Traduzione da Basic Query a Local Query ed ottimizzazione locale

A questo punto ogni Basic Query è pronta per essere tradotta in sottoquery (che chiameremo *Local Query*) indirizzabili alle singole sorgenti e classi locali.

Dal punto precedente abbiamo ottenuto gli attributi da recuperare e l'insieme di classi locali alle quali effettivamente accedere. Il Query Manager deve riscrivere ogni Basic Query in funzione degli schemi locali delle singole sorgenti, ma una Basic Query è posta sullo schema globale e quindi gli attributi da essa richiesti sono attributi relativi a classi globali. Durante la fase di integrazione sono stati generati i nomi globali degli attributi partendo da quelli locali, cioè dai nomi degli attributi presenti nelle classi locali. Affinché una query sia comprensibile da una sorgente bisogna recuperare i nomi originali degli attributi, questo avviene

grazie all'utilizzo della mapping table che, contenendo al suo interno tutte le informazioni necessarie per risolvere conflitti intensionali, permette la traduzione della Basic Query in Local Query eseguibili sulle singole sorgenti.

Grazie al modello comune dei dati (ODL_{I^3}) attraverso il quale il Mediatore comunica con i wrapper, il Query Manager non si deve preoccupare di quali linguaggi o formalismi siano utilizzati dalle singole sorgenti. Saranno difatti i wrapper stessi ad incaricarsi di tradurre ogni singola Local Query (espressa in OQL) nel linguaggio adottato in ogni sorgente.

Torniamo all'esempio rappresentato dalla Basic Query Q1: dal passo precedente, dopo tutte le semplificazioni del caso, ottenevamo il seguente insieme di classi da interrogare {`SU.School_Member`, `SCS.Student`} ed il seguente insieme di attributi {`name`, `rank`, `faculty`, `year`}. Da cui si ottengono le seguenti Local Query[2]:

```
Q1.1:  select first_name, last_name
       from School_Member
       where faculty = 'CS'
       and year < 1973

Q1.2:  select name, rank
       from Student
       where year < 1973
```

Un miglioramento del piano si potrebbe ottenere nel caso in cui dovesse essere ricostruita un sola base extension e tutte le classi locali coinvolte appartenessero alla stessa sorgente. In queste condizioni difatti si potrebbero raggruppare le Local Query in una unica ed effettuare localmente il join.

Ad esempio, consideriamo la seguente query:

```
Q3:  select name, year
     from University_Person
     where faculty = 'CS'
     and pay > 30000
```

Per quanto detto nelle sezioni precedenti, la classe virtuale target risulterebbe C4 e quindi dovremmo considerare la sola base extension 7 [3]; le classi locali alle quali rivolgere la query saranno soltanto `SU.University_Worker` e `SU.School_Member`[4] che appartengono alla stessa sorgente (UNIVERSITY).

[2] Q1.1 per la sorgente UNIVERSITY e Q1.2 per la sorgente COMPUTER_SCIENCE

[3] L'estensione di C4 è formata dalle base extension 2 e 7, ma la prima è dominata dalla seconda

[4] Difatti le classi `SU.School_Member` e `STP.University_Student` risultano estensionalmente equivalenti e quindi quest'ultima va scartata

Si può quindi migliorare il plan raggruppando le *Local Query* Q3.1 e Q3.2, in cui viene tradotta meccanicamente Q3, nell'unica query Q3.1_2, come evidenziato qui di seguito:

```
Q3.1:  select firt_name, last_name
       from University_Worker
       where pay > 30000

Q3.2:  select firt_name, last_name, year
       from School_Member
       where faculty = 'CS'


Q3.1_2: select a.first_name, a.last_name, b.year
        from University_Worker as a
             School_Member as b
        where a.first_name = b.first_name
        and a.last_name = b.last_name
        and a.pay > 30000
        and b.faculty = 'CS'
```

Un'ulteriore possibilità di semplificazione ci viene offerta dall'utilizzo dei valori di default. Difatti analizzando i predicati di selezione delle interrogazioni e la mapping table per verificare la presenza di valori di default su alcuni attributi, si può scoprire di poter ridurre il numero di Local Query da generare.

Se ad esempio avessimo avuto in una Basic Query la clausola: `where faculty = 'Economics'`, avremmo potuto scartare subito dalle base extension le classi `SCS.CS_Person`, `SCS.Professor` e `SCS.Student` in quanto aventi l'attributo faculty con valore uguale a "CS" (Computer Science).

Le semplificazioni viste finora vengono effettuate in fase di traduzione da Basic a Local Query; una tipologia diversa di semplificazione è quella che viene chiamata **Ottimizzazione Semantica Locale**. In comune con la sua omonima globale (Sezione 3.1), questa ottimizzazione semantica ha il fatto di essere legata all'esistenza di regole d'integrità, questa volta però definite sulle singole sorgenti, e di sfruttare ODB-Tools [27]. Grazie ad essa si è in grado di generare Local Query meno onerose, difatti l'esecuzione di una query in un DataBase viene migliorata dall'ottimizzazione semantica in quanto quest'ultima:

- aumenta la possibilità di utilizzo di indici: aggiungendo un predicato implicato da una rule si può introdurre nella query un attributo che potrebbe essere indicizzato;

- consente di eliminare o modificare dei join impliciti: una rule permette di riconoscere se due condizioni sono ridondanti e di conseguenza di eliminare quella implicata se comporta l' esecuzione di un join;
- permette di evitare o ridurre l' accesso a dati inutili: questo succede quando una query può essere trasformata in una equivalente su di una sottoclasse.

Se, ad esempio, disponessimo della seguente rule sulla classe locale Division:

```
rule RL1:  forall X in Division: (X.fund > 50000)
             then X.sector = 'Engineering'
```

Allora una delle Local Query, derivanti da una Basic Query posta sulla classe globale Workplace, potrebbe essere:

```
QL1:  select description, employee_nr
      from Division
      where fund > 60000
```

La rule RL1 porterebbe alla seguente espansione semantica:

```
QL1': select description, employee_nr
      from Division
      where fund > 60000
      and sector = 'Engineering'
```

Il vantaggio di QL1' rispetto a QL1 risiede nell'eventuale esistenza di un indice sull'attributo `sector`. È aumentato il costo di analisi dell'interrogazione, ma esistono risultati sperimentali [18] che dimostrano che il costo complessivo di esecuzione di una query ottimizzata decresce rapidamente all'aumentare del numero di istanze nel DataBase ed all'aumentare del numero di query mediamente poste.

La fase di ottimizzazione semantica locale è stata posta a livello di mediatore, e non di DataBase locali, in quanto non è detto che tutte le sorgenti siano in grado di effettuarla, mentre includere anche questa funzionalità nei wrapper appariva troppo oneroso.

3.5 Esecuzione della query

Come già accennato in Sezione 2.6 la fase di *Query Processing* (alla quale, ricordiamo, è preposto il Query Manager) è a sua volta costituita da due sottofasi ben distinte, anche temporalmente.

Prima difatti si svolge la fase di *Query Plan Definition*, quella nella quale si prepara la query per l'esecuzione, individuando il piano da associarvi. La realizzazione di questa sottofase, quindi come individuare le Local Query ed il piano di accesso ed esecuzione partendo da una Global Query, è stato descritto nelle precedenti sezioni di questo capitolo.

La successiva sottofase è quella che viene indicata con *Query Execution* e tratta di come poter presentare all'utente una risposta integrata; per riuscirci il Query Manager deve sfruttare le indicazioni presenti nel piano generato al passo precedente per recuperare i dati dalle sorgenti e fonderli poi insieme ottenendo la risposta desiderata; deve cioè *eseguire* la query. Questa fase non è ancora stata implementata in MOMIS; si presenta allora qui di seguito un'ipotesi di come, basandosi sul piano, si possa pervenire al risultato cercato [37].

Facendo riferimento alle figura 3.2 si vede come la fase di esecuzione, per ogni query posta dall'utente, sia in realtà suddivisa in tre fasi distinte:

1. Esecuzione delle Local Query;
2. Esecuzione delle Basic Query;
3. Esecuzione della Global Query.

In Figura 3.5 vengono di nuovo evidenziati i passi del Query Manager ponendo però l'attenzione, invece che sulla loro sequenzialità, sui diversi stadi che si possono individuare.

Dalla decomposizione della Global Query si generano le Basic Query, la loro esecuzione porta a dei risultati che sono relazioni temporanee direttamente utilizzabili dalla query assemblatrice globale (Global Assembler Query, Figura 3.2) per fornire il risultato finale. A questo livello risulta del tutto trasparente il fatto che le Basic Query saranno ulteriormente tradotte in più Local Query e che la loro esecuzione fornirà altri risultati creando altre relazioni temporanee.

Esaminiamo meglio le varie fasi di esecuzione.

Esecuzione delle Local Query. Il primo passo consiste, per ogni Basic Query, nell'eseguire sulle singole sorgenti le Local Query. Le Local Query generate dal Query Manager sono espresse in OQL, sarà quindi compito di ogni parser trasformarle in un formalismo comprensibile, traducendo la sintassi OQL in quella del linguaggio di interrogazione utilizzato nella relativa sorgente. Le Local Query vengono eseguite parallelamente una all'altra e forniscono dei risultati che vengono memorizzati in relazioni temporanee.

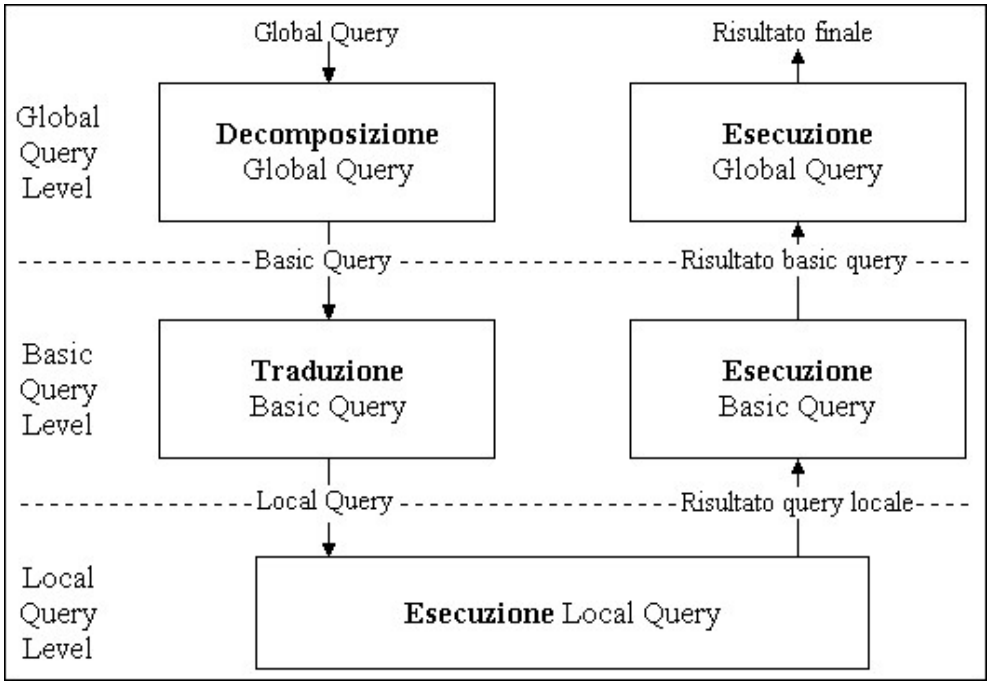


Figura 3.5: I diversi livelli di query processing

Esecuzione delle Basic Query. Più complicato si presenta il secondo passo, nel quale partendo dai risultati delle Local Query, e sfruttando il piano d'accesso, si voglio ottenere delle nuove relazioni temporanee derivanti dall'esecuzione delle Basic Query. Questa fase è divisa in due parti:

1. Ricostruzione di ogni base extension;
2. Unione delle base extension.

Vediamole in dettaglio.

1. Gli attributi che una base extension presenta derivano dall'insieme degli attributi delle classi locali che la costituiscono. Tra gli attributi di una base extension selezionata saranno presenti tutti quelli di interesse, quelli cioè presenti nelle clausole di proiezione e selezione della query[5]; ogni classe locale invece avrà solo alcuni di questi attributi. Bisogna quindi unire gli oggetti, provenienti da diverse classi locali, ma riferiti alla stessa entità del mondo reale, in modo da ottenere la struttura dati prevista per ogni base extension. Questa attività

[5]Se così non fosse difatti la base extension non apparterrebbe all'estensione della classe virtuale target e quindi non sarebbe presa in considerazione

(detta di *ricostruzione* delle base extension) avviene mediante dei join tra i risultati provenienti dalle classi locali appartenenti alla base extension che si sta considerando. Grazie al piano di accesso si può individuare come vadano fuse le classi e quali attributi si debbano usare per i join, i quali devono essere eseguiti su attributi identificanti in modo univoco gli elementi delle classi locali.

2. A questo punto disponiamo di base extension popolate, cioè contenenti oggetti distinti e dotati di uno schema uguale a quello della base extension stessa. Se si è individuata una sola base extension di interesse questi oggetti costituiscono già il risultato di questa fase. Ma in generale il risultato di una Basic Query proviene da più base extension, quelle contenute nell'estensione della classe virtuale target e non eliminate durante la semplificazione; occorre quindi *unire* gli oggetti precedenti in modo da ottenere un unico risultato. I modi in cui eseguire queste unioni ci sono ancora una volta forniti dal piano di accesso; infatti, per quanto già detto in Sezione 3.3.4, può succedere che due base extension ne dominino una terza: questo è il caso di sovrapposizione tra base extension e, per evitare duplicazioni, occorrerà effettuare degli outer join tra gli oggetti. In tutti gli altri casi si può procedere all'unione dei risultati senza rischio di ridondanze, avendo eliminato le base extension dominate.

Esecuzione della Global Query. I risultati temporanei forniti dalle Basic Query vengono utilizzati dalla Global Assembler Query per generare la relazione rappresentate il risultato finale. Si sfrutta il piano di esecuzione che, a questo livello, contiene invocazioni a funzioni complesse; vengono quindi eseguiti tutti i join tra classi globali, le aggregazioni e le altre operazioni non trattabili dalle Basic Query, ma necessarie per ottenere una risposta veramente corrispondente alla Global Query formulata dall'utente.

Capitolo 4

Progetto e realizzazione del software

L'attività principale svolta nell'ambito di questa tesi è stata rivolta alla realizzazione del software necessario per poter gestire la Conoscenza Estensionale in fase di Query Plan. Si è cioè cercato di fornire al Query Manager quegli strumenti che gli permettessero di individuare l'insieme minimo di classi locali a cui indirizzare le Local Query in modo da poter ottenere, alla fine della successiva fase di Query Execution, una risposta corretta e, quanto più possibile, completa e non ridondante.

In questo capitolo vedremo innanzitutto come sia stato modellato il Query Manager a livello software, illustrandone il progetto presentato e realizzato in [13].
Passeremo poi in seguito ad analizzare il software realizzato per questa tesi, descrivendo dettagliatamente le classi Java implementate, le loro caratteristiche e funzioni.
Infine, facendo riferimento ai passi descritti nel capitolo precedente, si spiegherà come *funziona* il software esistente, si giustificheranno le scelte implementative compiute e si incherà quali sviluppi ci si aspetta in futuro.

4.1 Il modello software del Query Manager

Il progetto e la modellazione del componente Query Manager di MOMIS sono stati presentati e sviluppati in [13]. L'obiettivo dichiarato era quello di realizzare un modulo software componibile ed estendibile, che fosse in grado di affrontare un sottoinsieme delle problematiche individuate nella fase di Query Processing e consentisse una facile realizzazione di estensioni future. A tale scopo è stata effettuata una modellazione di tipo object-oriented e si è utilizzato il linguaggio Java per l'implementazione.

Si sono quindi individuati i vari moduli preposti all'esecuzione delle opera-

zioni descritte nel capitolo precedente e si è organizzato il software nei seguenti package che raccolgono le classi Java implementate:

- package *queryman*
- package *globalschema*
- package *oql*
- package *utility*

Le giustificazioni per l'utilizzo di Java e l'abbandono del linguaggio C, utilizzato per gli ODB-Tools, ci sono fornite dall'autore stesso e risiedono principalmente nella natura stessa di Java che, essendo un linguaggio completamente orientato agli oggetti, favorisce la modularità e la componibilità del software, caratteristiche indispensabili nell'ambito di un progetto di ricerca a lungo termine come MOMIS che si avvale del contributo di numerose persone.

4.1.1 I moduli preposti alla definizione del Query Plan

Si è più volte ripetuto come la fase di Query Processing sia formata dalle sottofasi Query Plan Definition e Query Execution, in [13] si sono individuati i moduli che dovranno essere preposti a queste due fasi ed in Figura 4.1 se ne riporta lo schema relativo alla prima di queste.
Si può notare come questi moduli siano collegati alle funzioni tipicamente svolte dal Query Manager e descritte nel Capitolo 3.

Prima di proseguire è d'obbligo precisare meglio la distinzione tra *piano di accesso* e *piano di esecuzione*, termini più volte utilizzati finora ma mai meglio specificati. Quello che si vuole ottenere da questa fase di definizione è un *query plan* che praticamente è formato dai due piani precedenti, difatti è stato anche definito *piano di accesso ed esecuzione* nel Capitolo 3. Si faccia riferimento alla Figura 3.5.
Il piano di esecuzione è definito a livello di Global Query ed è quella parte del query plan dove sono indicate le varie Basic Query in cui è stata decomposta la Global Query con l'aggiunta delle invocazioni alle funzioni complesse (join tra classi globali, aggregazioni, . . .) necessarie per ottenere il risultato finale.
Mentre il piano d'accesso è definito a livello di Basic Query e per ognuna di esse indica le varie Local Query in cui è stata tradotta, nonché tutte le informazioni necessarie per ricostruire ogni base extension e per unirle tra loro al fine di ottenere il risultato della Basic Query.

Quanto appena detto risulta evidente in Figura 4.1 nella quale sono presentati due diversi moduli per la definizione delle due differenti tipologie di piano:

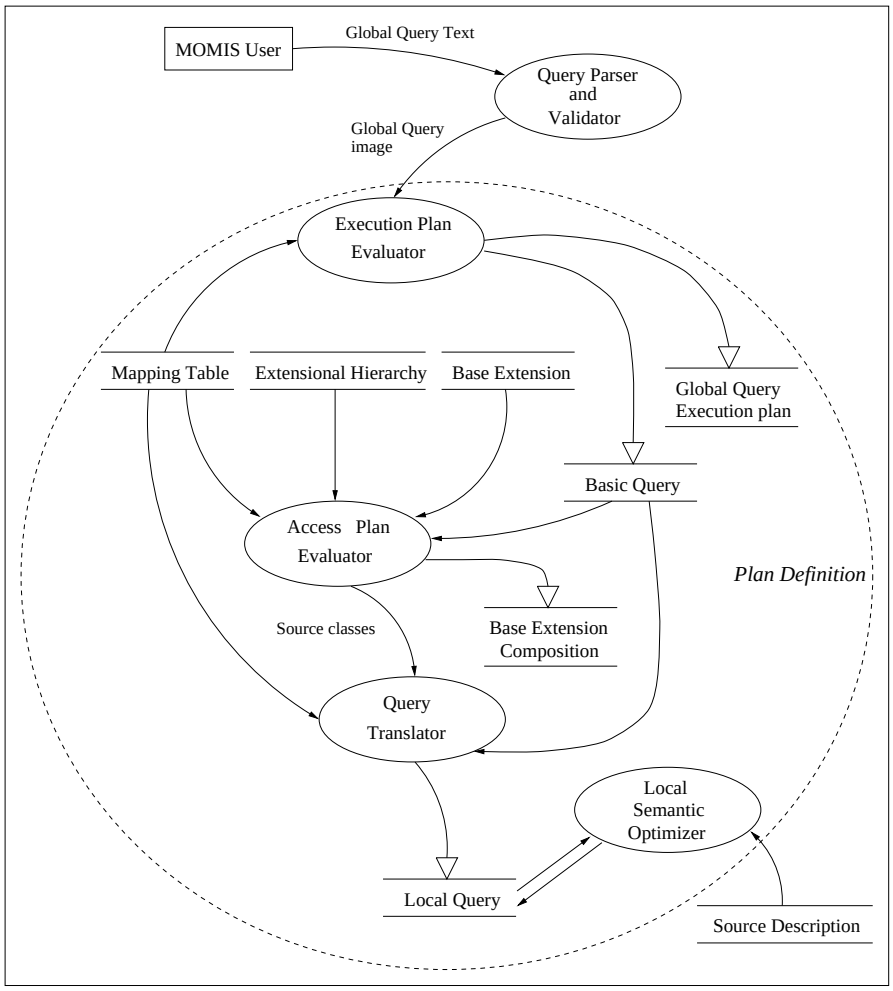


Figura 4.1: Definizione del piano di una Query

- l'*Execution Plan Evaluator* che fornisce come output il piano di esecuzione: le Basic Query e come combinarle (Global Query Execution plan);

- l'*Access Plan Evaluator* che fornisce in output le classi locali da interrogare ed il piano d'accesso: come comporre e combinare le base extension (Base Extension Composition).

I moduli che si sono implementati in [13] sono il *Query Parser and Validator* e il *Query Translator*; grazie a questi due moduli è già possibile porre query sullo schema globale ed ottenerne la traduzione in Local Query. Le query devono essere necessariamente già di tipo Basic in quanto non è stata ancora implementata la fase *Execution Plan Evaluator* che scompone una Global Query in Basic

Query ed inoltre non viene sfruttata la conoscenza estensionale per scegliere le classi locali, compito del modulo *Access Plan Evaluator*, il cui sviluppo è proprio il tema di questa tesi e sarà approfonditamente discusso nelle sezioni seguenti. Questa strategia è frutto della progettazione modulare adottata: si sono difatti implementati solo i moduli più urgenti, quelli che consentono cioè di trattare un sottoinsieme significativo delle interrogazioni, permettendo di implementare gli altri in fasi successive, grazie alla stesura contestuale del progetto modulare ed object-oriented di tutto il Query Manager.

Descriviamo brevemente i due moduli implementati finora.

Il *Query Parser and Validator* ha il compito di eseguire le fasi di analisi sintattica e semantica della Global Query posta dall'utente. Facendo riferimento alla Figura 3.1 quello che è stato implementato è in pratica il modulo in essa denominato *Parsing*. Come già accennato in Sezione 3.1 in MOMIS risultano invertite le fasi di ottimizzazione e parsing rispetto alla teoria dei mediatori e si erano giustificate queste scelte con la possibilità di sfruttare il software esistente; vediamo meglio. In MOMIS difatti la fase di validazione ed ottimizzazione viene svolta dal Query Optimizer di ODB-Tools (scritto in C) il quale fornisce una struttura rappresentante la query ottimizzata, da questa si ottiene un file di testo che rappresenta la query OQL che analizzata un'altra volta (*Parsing*) ci fornisce la struttura Java della query. Quest'ultima fase di parsing viene eseguita mediante classi Java, più precisamente esiste un riconoscitore sintattico realizzato mediante la classe **parser** del package *utility* che individua la correttezza di una query OQL. Questa è stata ottenuta mediante l'utility di pubblico dominio *BYACC* [36] che è in grado di generare la classe Java che implementa il parser della grammatica. In pratica però il controllo sintattico avviene mediante un'altra classe (**OqlAnalizer**) dello stesso package che specializza **parser** e consente di analizzare query OQL fornite in vari formati (stringa, file di testo), inoltre utilizza la gerarchia di classi raccolta nel package *oql* per rappresentare la query.

Il *Query Translator* ha lo scopo di tradurre la Basic Query nelle Local Query da inviare alle sorgenti. Per farlo si deve avvalere delle mapping table nelle quali sono espresse le regole di mapping tra attributi globali e locali. Quello che deve fare consiste proprio nel trasformare gli operatori presenti nella Basic Query in funzione dei costrutti impiegati localmente; visto che si è adottato un linguaggio comune di interrogazione (OQL), il processo di traduzione può avvenire indipendentemente dai linguaggi utilizzati nelle sorgenti, sarà poi il relativo wrapper ad incaricarsi di quest'ulteriore traduzione. L'output di questo modulo consiste quindi in un insieme di Local Query, espresse sempre in OQL, ognuna da

inviare al wrapper relativo alla sorgente contenente la classe locale a cui è rivolta[1].

Un'ultima considerazione può essere fatta sul perché non si sia implementato per primo il modulo *Execution Plan Evaluator* che, stando alla Figura 4.1, sembrerebbe necessario per poter passare poi alle fasi successive. In realtà la percentuale maggiore di interrogazioni significative per un mediatore è rappresentata da Basic Query. Quindi anche non implementando questo modulo si riesce a gestire una quantità significativa di query; bisogna però che l'utente ponga solamente query di tipo basic in modo da non dover ricorrere alla fase di decomposizione. La query viene inviata al modulo di parsing nel quale è stato previsto un controllo per verificarne l'effettiva natura di Basic Query; la classe **BQChecker** del package *oql* esegue tale controllo e se individua una query di tipo non basic sospende l'esecuzione ed invia un messaggio all'utente. Questa verifica verrà ovviamente eliminata una volta che il Query Manager sarà in grado di gestire ogni tipo di interrogazione.

4.1.2 L'organizzazione del software

Un package permette di raggruppare classi Java per scopo o per ereditarietà e consente una loro maggiore protezione. Come già accennato, il software del Query Manager è stato organizzato in quattro package distinti, passiamo ora ad analizzarli brevemente.

Package *queryman*. In questo package sono raggruppate le classi che descrivono gli elementi del piano, esse sono quindi preposte alla rappresentazione ed all'esecuzione della query. Tra le altre sono da segnalare la classe **QueryManager** e la classe **Query**.
Con la prima si crea un'istanza di un oggetto di tipo QueryManager inizializzando tutte le strutture dati che devono essere utilizzate in fase di query processing.
La seconda generalizza le query che compongono il piano; nella sua interfaccia sono presenti l'attributo *plan*, che conterrà le indicazioni rappresentati il piano associato alla query, e l'attributo *data*, che conterrà il risultato ritornato dall'esecuzione del piano. Questa classe viene specializzata dalle seguenti: **GlobalQuery**, **BasicQuery**, **LocalQuery** e **UnionQuery**[2]; le prime tre estendono l'interfaccia con l'attributo *subQueries* in cui vengono memorizzate le query di livello inferiore (le BasicQuery nelle GlobalQuery, le LocalQuery nelle

[1] Prima di essere effettivamente inviata al wrapper, una Local Query potrà subire il processo di ottimizzazione semantica locale

[2] Nel caso di sorgenti semistrutturate (Capitolo 5) può accadere che per una singola classe locale vengano generate più query per poter gestire tutte le possibili rappresentazioni associate ad un unico oggetto; tali query sono qui indicate come *Union Query*

BasicQuery, le UnionQuery nelle LocalQuery).

Package *globalschema*. Questo package riunisce le classi che descrivono lo schema globale[3]. Attraverso la classe **GlobalClass** vengono implementate le classi globali; ognuna di esse, tra gli altri, è dotata di un attributo di tipo **MappingTable**. La classe MappingTable rappresenta le mapping table di ogni classe globale, grazie ad essa è possibile tradurre la Basic Query; presenta tra i suoi attributi anche un vettore contenente oggetti di tipo **GlobalAttribute** che descrivono gli attributi globali da mappare. Infine la classe **MappingElement**, grazie alle sue specializzazioni, descrive i possibili tipi di mapping, un campo di questo tipo sarà presente in ogni oggetto di tipo GlobalAttribute.

Package *oql*. In questo package sono state raccolte le classi che costituiscono gli elementi della struttura dati usata per rappresentare una query OQL. Esse sono state definite per poter generare in memoria centrale un'immagine della query che servirà poi come punto di partenza per le successive elaborazioni. In pratica, quando una query viene ritenuta sintatticamente corretta, ogni suo campo viene memorizzato mediante un corrispondente oggetto Java, specializzazione della classe **Oql_Query**.

Package *utility*. Questo package contiene le classi che svolgono funzioni di utilità generale nell'ambito del Query Manager. Abbiamo già incontrato le più significative (**parser** e **OqlAnalizer**) che svolgono la funzione di parsing.

4.2 Il software per la gestione della Conoscenza Estensionale

Il software descritto finora, partendo da una Basic Query validata ed ottimizzata, permette quindi di:

- verificarne la correttezza sintattica;
- verificare la sua effettiva natura di Basic Query;
- generare le Local Query corrispondenti interrogando TUTTE le classi locali presenti nella mapping table.

Per quanto detto si ha che, rispetto alla Figura 4.1, non viene implementata la fase di *Access Plan Evaluation*, ma si passa direttamente a quella di *Query Translation*.

[3]Ed è qui che si dovranno inserire le classi necessarie per gestire la conoscenza estensionale

Per la traduzione da Basic a Local Query ci si avvale, come è giusto, solamente dell'ausilio della mapping table, ma in precedenza non viene effettuato nessun controllo sulle classi locali e quindi le Local Query vengono generate per ogni classe locale presente nella mapping table della classe globale sulla quale è posta la Basic Query.

In pratica per tradurre la Basic Query non si sfrutta la conoscenza estensionale, ma solo quella intensionale; questo, come già evidenziato in Sezione 3.3, porta ad ottenere una risposta che è sicuramente corretta, ma non necessariamente anche completa e minima.
Per poter giungere ad una risposta che soddisfi quanto più possibile anche questi requisiti si deve implementare il modulo di *Access Plan Evaluation* con il quale, sfruttando le informazioni derivanti da **Gerarchia Estensionale** e **Base Extension**, si arriva ad individuare l'insieme ottimale di classi locali e a definire il **Piano di Accesso**.

I termini evidenziati in grassetto rappresentano i concetti principali nei quali si può scomporre la descrizione di questo modulo. Qui di seguito andremo ad analizzare nel dettaglio le classi Java che li implementano.

4.2.1 La Gerarchia Estensionale

In Figura 4.2 è illustrato il modello ad oggetti della classe **ExtensionalHierarchy** che implementa la gerarchia estensionale associata ad una classe globale. Tutte le classi descritte in questa sezione e nella successiva sono state inserite nel package *globalschema*.

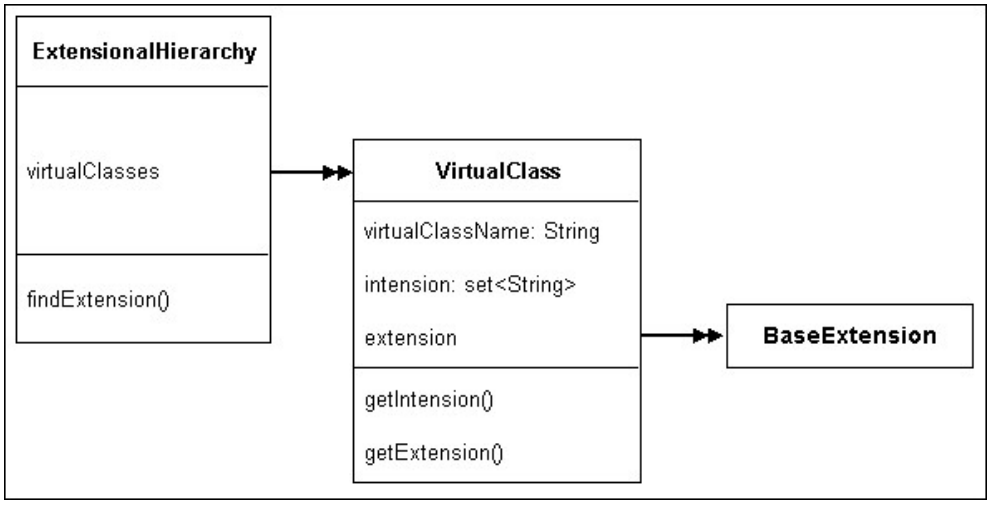


Figura 4.2: Modello ad oggetti della classe ExtensionalHierarchy

Classe ExtensionalHierarchy

PROPRIETÀ

- *virtualClasses*: questo attributo è un vettore di oggetti di tipo **VirtualClass**, ognuno di essi rappresenta una classe virtuale appartenente alla gerarchia estensionale in questione.

METODI

- *findExtension()*: fornisce l'estensione della classe target; più precisamente questo metodo riceve come parametro un vettore di stringhe ognuna contenente un attributo presente nella query, individua la classe virtuale target tra tutte quelle appartenenti alla gerarchia estensionale[4] e restituisce come risultato l'estensione della classe trovata, vale a dire un vettore di oggetti di tipo **BaseExtension**. Nel caso particolare in cui esistano più classi virtuali identificabili come target, il risultato fornito sarà una combinazione delle loro estensioni, cioè un vettore contenente le base extension presenti in entrambe, senza duplicazioni.

Classe VirtualClass

PROPRIETÀ

- *virtualClassName*: è una stringa contenente un identificativo che individua in modo univoco una classe virtuale all'interno di una gerarchia estensionale;
- *intension*: è un vettore di stringhe contenente i nomi degli attributi globali presenti nel cluster di appartenenza;
- *extension*: è un vettore di oggetti di tipo **BaseExtension**;

è presente anche un altro attributo: *allVirtualClasses*. Esso ha carattere globale e contiene i riferimenti a tutti gli oggetti presenti nella classe, cioè a tutte le classi virtuali della gerarchia estensionale cui appartengono.

METODI

[4] Cioè quella con estensione maggiore tra quelle contenenti tutti gli attributi della query (si veda a tal proposito la Sezione 3.3.2)

- *getIntension()*: fornisce l'intensione della classe virtuale; più precisamente riceve come parametro l'identificativo di una classe virtuale e restituisce come risultato un vettore contenente i nomi degli attributi presenti nella sua intensione. Se non dovesse essere presente nella gerarchia estensionale una classe virtuale con nome uguale al parametro passato, verrà restituito un valore null;
- *getExtension()*: fornisce l'estensione della classe virtuale; in realtà esistono due versioni di questo metodo. Una non richiede parametri e restituisce come risultato un vettore di oggetti di tipo **BaseExtension** rappresentante l'estensione della classe virtuale attuale. L'altra invece richiede come parametro il nome di una classe virtuale e ne fornisce l'estensione, sempre sotto forma di vettore di base extension; se la classe richiesta non dovesse esistere, restituirà un valore null.

4.2.2 Le Base Extension

La Figura 4.3 presenta il modello ad oggetti della classe **BaseExtension** che implementa la singola base extension[5].

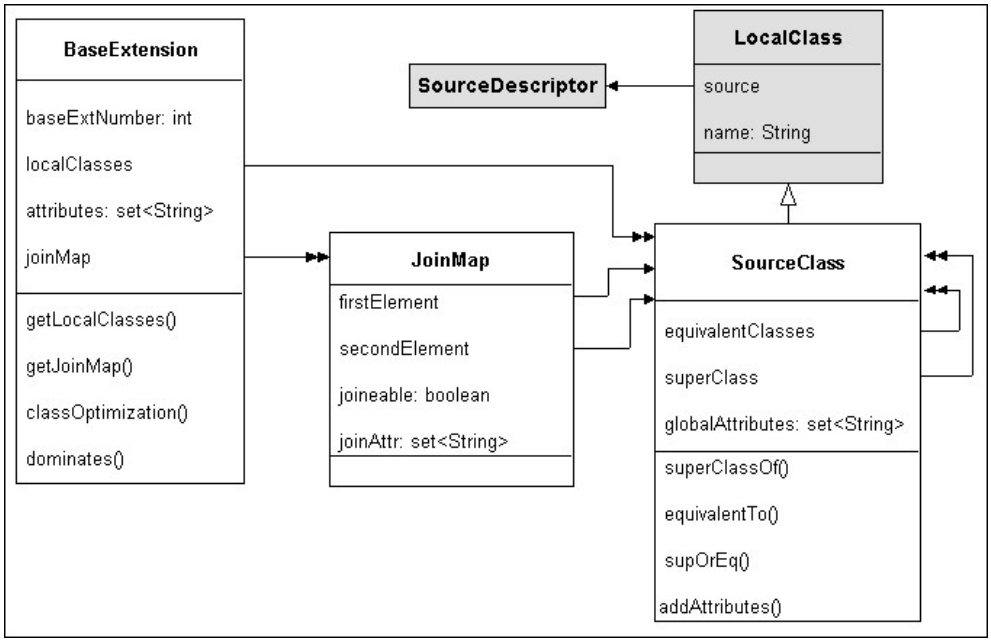


Figura 4.3: Modello ad oggetti della classe BaseExtension

Classe BaseExtension [6]

PROPRIETÀ

- *baseExtNumber*: questo attributo è un semplice intero che rappresenta il numero con il quale identificare univocamente la base extension;
- *localClasses*: si tratta di un vettore di oggetti di tipo **SourceClass**, ognuno dei quali rappresenta una classe locale presente nella base extension;

[5] In grigio sono rappresentate le classi già implementate in [13]; questa notazione sarà adottata anche nelle successive figure presenti in questo capitolo

[6] Il listato del codice di questa classe è riportato in Appendice C unitamente alla sua documentazione HTML prodotta con javadoc

- *attributes*: è un vettore di stringhe contenente i nomi degli attributi che compongono l'intensione della base extension;
- *joinMap*: vettore di oggetti di tipo **JoinMap**; in pratica (come si può evincere dalla struttura della classe **JoinMap** descritta successivamente) è una matrice di **SourceClass** e per ogni coppia di classi locali della base extension ci dice se è possibile effettuare il join tra di esse e, in caso affermativo, attraverso quali attributi.

METODI

- *getLocalClasses()*: fornisce l'insieme delle classi locali; più precisamente restituisce un vettore di oggetti di tipo **SourceClass** (il campo *localClasses* della base extension in questione);
- *getJoinMap()*: permette di sapere se si può effettuare un join su due classi della base extension. Richiede come parametri due **SourceClass** (cioè due classi locali) e restituisce un oggetto di tipo **JoinMap** indicante attraverso quali attributi è possibile effettuare un join tra di esse. Se non risulta possibile effettuare un join (nella base extension che si sta considerando) tra le due classi richieste, allora viene restituito un valore null;
- *dominates()*: permette di scoprire se una base extension ne domina un'altra. Richiede come unico parametro un oggetto di tipo **BaseExtension** e restituisce un valore booleano vero se la base extension attuale domina quella passata, falso se ciò non accade;
- *classOptimization()*: questo metodo è importantissimo in quanto fornisce l'insieme di classi locali da interrogare ottimizzato rispetto ad una determinata base extension ed una determinata Basic Query. La base extension è quella attuale, quella cioè che richiama il metodo, e la Basic Query è rappresentata dagli attributi in essa presenti e passati al metodo come parametro (un vettore di stringhe); il risultato restituito consiste in un vettore di oggetti di tipo **SourceClass** rappresentante l'insieme ottimizzato di classi locali da interrogare. Per pervenire a questo risultato si eliminano le classi locali alle quali non si deve accedere seguendo i dettami della teoria (Sezione 3.3.3) e sfruttando i metodi della classe **SourceClass** per scoprire se un classe locale sia equivalente oppure superclasse di un'altra. Ottenuto un primo insieme di classi locali si controlla se con esse sia effettivamente possibile ricostruire la base extension, se così non è vengono recuperate ed aggiunte al risultato anche le classi che lo rendono possibile.

Classe JoinMap

PROPRIETÀ

- *firstElement*: è un oggetto di tipo **SourceClass** e rappresenta una delle due classi locali delle quali, con un oggetto di tipo **JoinMap**, si descrivono la possibilità e le modalità di effettuare join;
- *secondElement*: è del tutto simile a *firstElement*;
- *joineable*: è un attributo booleano, quando il suo valore è vero significa che tra le due classi indicate in *firstElement* e *secondElement* è possibile effettuare il join;
- *joinAttr*: quando *joineable* è vero è un vettore di stringhe contenente i nomi degli attributi sui quali si può effettuare il join; quando *joineable* è falso è un vettore vuoto.

Classe SourceClass

Questa classe estende la classe **LocalClass** dalla quale eredita due attributi:

- *name*: è una stringa contenente il nome della classe locale;
- *source*: è un oggetto di tipo **SourceDescriptor**[7];

PROPRIETÀ

- *globalAttributes*: è un vettore di stringhe contenenti i nomi degli attributi globali con un corrispettivo nella classe locale, cioè con un mapping di tipo non null;
- *equivalentClasses*: è un vettore di elementi di tipo **SourceClass** che contiene le classi appartenenti alla stessa classe globale e che sono estensionalmente equivalenti a quella attuale;

[7] **SourceDescriptor** è una classe implementata in [13] che serve per descrivere le sorgenti. Presenta, tra gli altri, un attributo *name* che contiene il nome della sorgente ed un attributo *type* che ne indica il tipo (object, relational, file, . . .)

- *superClass*: è un vettore contenente oggetti di tipo **SourceClass** indicanti ognuno una classe appartenente allo stesso database e superclasse di quella attuale.

METODI

- *superClassOf()*: richiede come parametro un oggetto di tipo **SourceClass** e restituisce un valore booleano, vero se la classe attuale (quella cioè che richiama il metodo) è superclasse di quella passata come parametro, falso in caso contrario;
- *equivalentTo()*: questo metodo indica, restituendo un valore booleano, se la classe che lo richiama è o meno estensionalmente equivalente a quella passata come parametro;
- *supOrEq()*: questo metodo restituisce un valore booleano che è vero se la classe attuale è superclasse oppure equivalente ad almeno una di quelle comprese nel vettore di **SourceClass** passato come parametro, falso se non è né equivalente né superclasse di nessuna di esse;
- *addAttributes()*: questo metodo restituisce un vettore di stringhe contenente i nomi degli attributi da aggiungere a quelli presenti nella query per poter permettere alla classe attuale di effettuare join in ognuna delle base extension indicate. Come parametri necessita quindi di un vettore di base extension (che sono quelle nelle quali interessa che sia possibile effettuare i join), un vettore di oggetti di tipo **SourceClass** (che è l'insieme di classi da interrogare), un vettore di stringhe (che sono i nomi degli attributi della query).

4.2.3 Il Piano di Accesso

In figura 4.4 è riportato il modello ad oggetti relativo alla classe **Plan**. Essa è dotata di un attributo *planElements* che è un vettore di oggetti di tipo **PlanElement**, ognuno dei quali rappresenta un elemento del piano.

Tutte le classi descritte in questa sezione sono state inserite nel package *queryman*.

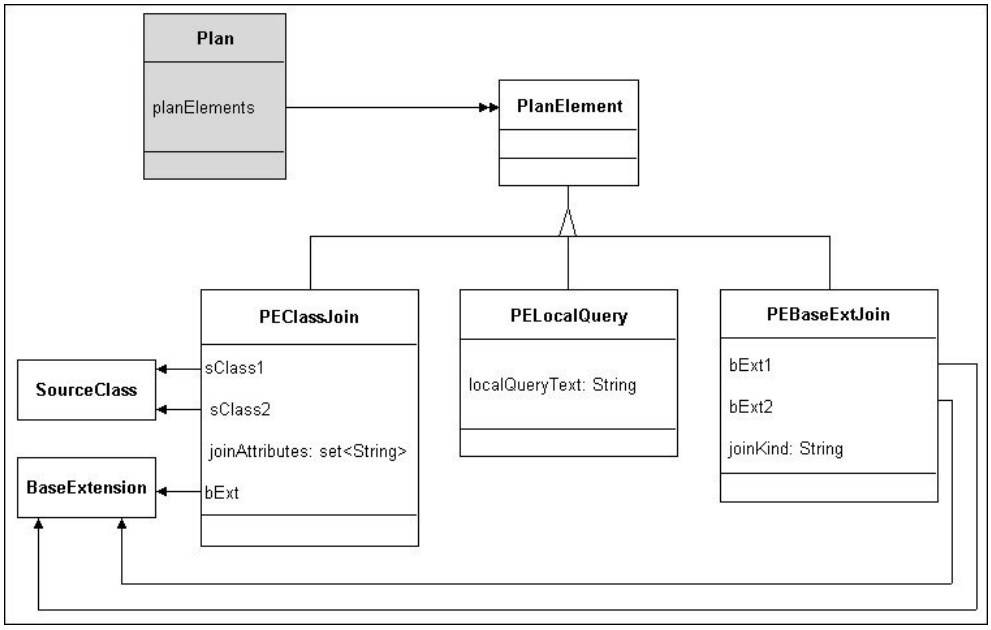


Figura 4.4: Modello ad oggetti della classe Plan

Da notare che la specializzazione della classe **PlanElement** è stata effettuata relativamente alla gestione del piano di accesso da associare ad una Basic Query. Ogni oggetto di tipo **Query** possiede difatti un attributo *plan*[8], ma a seconda della specializzazione esso avrà caratteristiche diverse: se si tratta di un oggetto **GlobalQuery** l'attributo *plan* implementerà il piano di esecuzione e di conseguenza gli oggetti **PlanElement** saranno invocazioni a funzioni complesse (ad es. join tra classi globali), se invece si ha a che fare con oggetti **BasicQuery** l'attributo *plan* implementa il piano di accesso ed i suoi elementi dovranno essere indicazioni su come ricostruire le base extension e come combinarle insieme.

A questo livello sono state individuate tre diverse tipologie di elementi del piano di accesso:

- le Local Query, implementate dalla classe **PELocalQuery**;
- i join tra le classi locali per ricostruire una base extension, implementati dalla classe **PEClassJoin**;
- le unioni tra base extension per ottenere il risultato di una Basic Query, implementate dalla classe **PEBaseExtJoin**.

[8]Come già riportato in Sezione 4.1.2

L'interfaccia di un oggetto di tipo **PELocalQuery** è formata dal solo attributo *localQueryText* che è una stringa contenente il testo della query locale.

Vediamo meglio quelle degli altri due oggetti che specializzano **PlanElement**.

Classe PEClassJoin

Questa classe implementa un passo per ricostruire una determinata base extension, più dettagliatamente implementa un singolo join tra classi locali.

PROPRIETÀ

- *bExt*: è un oggetto di tipo **BaseExtension** che rappresenta la base extension da ricostruire;
- *sClass1*: è un oggetto di tipo **SourceClass** che indica una delle due classi locali tra le quali effettuare il join;
- *sClass2*: è del tutto simile a *sClass1*;
- *joinAttributes*: è un vettore di stringhe contenente i nomi degli attributi su cui effettuare il join.

Classe PEBaseExtJoin

Questa classe implementa un passo per la ricostruzione della risposta ad una Basic Query, più precisamente indica se tra due base extension possa essere effettuata un'unione delle entità, oppure si debba effettuare un outer join.

PROPRIETÀ

- *bExt1*: è un oggetto di tipo **BaseExtension** che rappresenta una delle due base extension da unire;
- *bExt2*: è del tutto simile a *bExt1*;
- *joinKind*: è una stringa contenente il tipo di unione da effettuare (unione vera e propria o outer join).

4.2.4 Gli altri aspetti trattati

Per poter gestire appieno la conoscenza estensionale nel Query Manager, oltre ad implementare le classi descritte nelle precedenti sezioni, si è anche dovuto intervenire su altre classi aggiungendo nuovi metodi e attributi, oppure creando nuove versioni di metodi già esistenti (overloading).

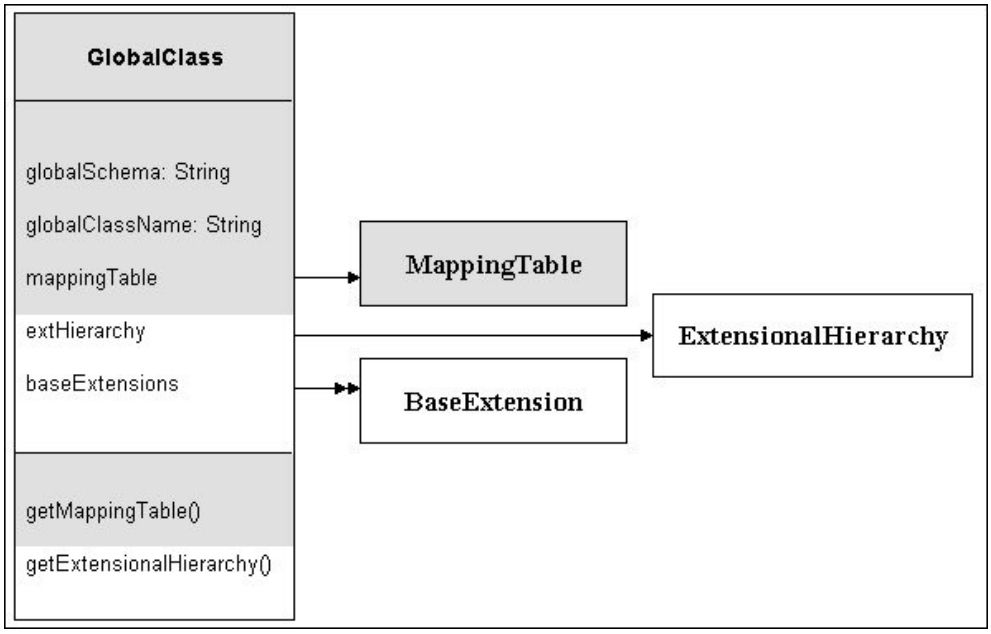


Figura 4.5: Modello ad oggetti della classe GlobalClass

Nel package *globalschema*, ad esempio, si è intervenuto sulla classe **GlobalClass** che implementa la singola classe globale, aggiungendo gli attributi *extHierarchy* (oggetto di tipo **ExtensionalHierarchy** rappresentante la gerarchia estensionale costruita sulla classe globale in questione) e *baseExtensions* (vettore di oggetti di tipo **BaseExtension** ognuno dei quali rappresenta una base extension individuata nella classe globale). In Figura 4.5 si possono notare gli altri attributi della classe che, come si può facilmente evincere dai loro nomi, rappresentano il nome dello schema globale, il nome della classe globale, il riferimento alla mapping table costruita su tale classe. Esiste un ulteriore attributo (*globalClasses*) che è una variabile globale contenente i riferimenti a tutte le classi globali presenti nello schema. Questo torna utile ai due metodi (*getMappingTable()* e *getExtensionalHierarchy()*) che, fornito il nome di una classe globale, ritornano rispettivamente la sua mapping table o la sua gerarchia estensionale se questa classe esiste, altrimenti ritornano un valore null.

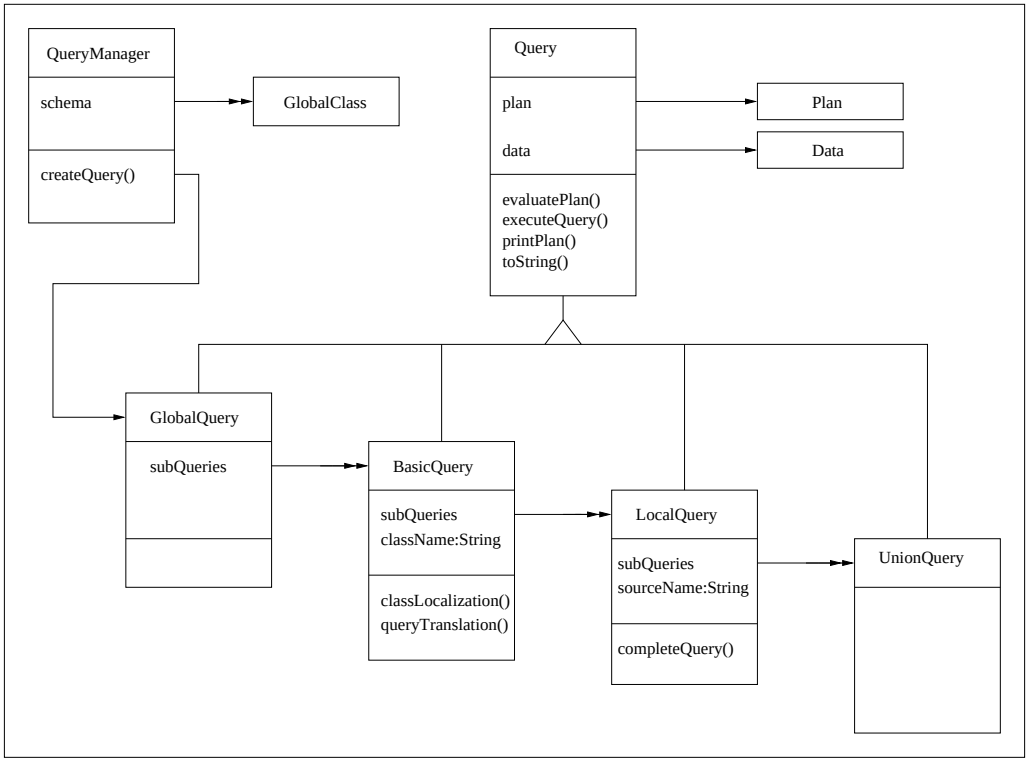


Figura 4.6: Modello ad oggetti del package queryman

Nel package *queryman* (Figura 4.6)[9] si è lavorato sulle classi **QueryManager** e **BasicQuery**. La prima genera un'istanza del Query Manager, la seconda è invece fondamentale per l'implementazione della fase di "Individuazione delle classi locali" (Figura 3.3). Vediamo meglio.

In **QueryManager** si è inserito un nuovo esempio completo[10]: quello dell'università utilizzato come riferimento anche in questo elaborato.

In **BasicQuery** i metodi *evaluatePlan()* e *classLocalization()* esistenti non sfruttavano la conoscenza estensionale, il secondo in particolare si limitava a fornire tutte la classi locali presenti nella mapping table. A questi due metodi ne sono stati aggiunti due con lo stesso nome, ma con attributi diversi (overloading). Le vecchie versioni richiedevano in input le strutture dati rappresentanti la Basic

[9] Quanto mostrato in figura (anche se non rappresentato in grigio) è stato implementato in [13]; di seguito saranno descritte le modifiche apportate

[10] Questa classe deve generare una nuova istanza del Query Manager caricando nelle sue strutture dati le informazioni fornite dal Global Schema Builder. Attualmente però il Global Schema Builder non è ancora stato completamente implementato e non è in grado di fornire tali dati, quindi la classe **QueryManager** genera l'istanza del Query Manager inserendo, mediante assegnamenti, un esempio di schema globale

Query e la mapping table, le nuove richiedono anche quella relativa alla gerarchia estensionale ed un vettore di stringhe contenente gli attributi della query; il risultato fornito è l'insieme ottimale di classi locali da interrogare. Sono stati creati inoltre altri due metodi: *getQueryAttributes()* e *baseExtRebuilding()*. Il primo, ricevendo la struttura dati contenente la Basic Query, restituisce gli attributi globali coinvolti e quindi implementa la fase di "Analisi della query" (Sezione 3.3.1). Il secondo genera la parte di piano di accesso nella quale è indicato come ricostruire una determinata base extension (in questo caso passata come parametro); il risultato sarà quindi un vettore di oggetti di tipo **PEClassJoin**.

4.3 Descrizione operativa del software

Il software del Query Manager si comporta esattamente come previsto nella teoria seguendo i passi descritti nel Capitolo 3, limitatamente alle fasi relative ai moduli fin qui implementati.

Per prima cosa viene creata dalla classe **QueryManager** un'istanza del Query Manager che ora è pronto per ricevere una query.

La query che arriva è già ottimizzata e validata (la fase di "Ottimizzazione e Validazione" viene difatti svolta da ODB-Tools), viene caricata nella struttura dati predisposta (la classe **GlobalQuery**) ed a questo punto, trattando solo Basic Query, non viene ovviamente svolta la fase di "Decomposizione da Global a Basic Query", ma al suo posto, mediante la classe **BQChecker** si verifica che si tratti realmente di una Basic Query e contestualmente viene svolta la fase di "Parsing" dalla classe **OQLAnalizer**; se la query supera tutti i controlli viene caricata in un'altra struttura dati: nella classe **BasicQuery**.

A questo punto il metodo *evaluatePlan()* svolge la fase di "Individuazione della classi locali", richiamando metodi che ne implementano le varie sottofasi:

- *getQueryAttributes()* realizza la sottofase di "Analisi della Query";
- *classLocalization()* è preposto alla fase di "Individuazione della classe virtuale target" e realizza la prima fase di semplificazione del piano ("Eliminazione di base extension"), inoltre richiama il metodo *classOptimization()* (descritto di seguito) e riempie le strutture dati relative al piano di accesso: quelle per l'unione di base extension direttamente, quelle per la ricostruzione delle stesse richiamando *baseExtRebuilding()*;
- *classOptimization()*, metodo della classe **BaseExtension**, realizza la seconda fase di semplificazione ("Eliminazione di classi locali") per ogni base extension rimasta.

Le classi locali così individuate costituiscono l'insieme minimo cercato e per ognuna di esse viene generata una query locale (fase di "Traduzione da Basic a Local Query" implementata dal metodo *translateQuery()*) che viene caricata nella struttura dati rappresentata dalla classe **LocalQuery**.

4.4 Ipotesi di sviluppi futuri

Come più volte ricordato, il Query Manager è preposto allo svolgimento di due attività ben distinte: la *Definizione del Query Plan* e la *Esecuzione della Query*.

Le fasi teoriche in cui è stata suddivisa la prima sono state descritte nelle sezioni dalla 3.1 alla 3.4 ed i moduli software individuati per la sua implementazione sono stati illustrati in Figura 4.1.

Per quanto riguarda la seconda invece, i passi che la costituiscono sono stati descritti in Sezione 3.5, mentre in Figura 4.7 vengono rappresentati i moduli software ad essa relativi.

Consideriamo la Figura 4.1: sono indicati cinque moduli che contribuiscono all'esecuzione della fase di *Definizione del Piano*. Di questi, i tre più importanti sono già stati implementati[11] e, analizzando i due rimanenti, si vede che in realtà anche *Local Semantic Optimizer* è già disponibile: questa fase difatti viene eseguita da ODB-Tools; dal punto di vista dello sviluppo futuro del software si tratterà quindi solamente di realizzare l'interazione tra il Query Manager ed ODB-Tools.
Resta da realizzare invece il modulo *Execution Plan Evaluator*, quello preposto alla fase di decomposizione da Global a Basic Query ed alla definizione del piano di esecuzione della query globale. È stata implementata la classe **GlobalQuery** e **Plan** dovrà essere specializzata per questo tipo di piano. L'implementazione di questa fase non risulta comunque urgente, essendo a tutt'oggi completamente gestibili le query di tipo Basic che costituiscono la stragrande maggioranza di quelle interessanti un modulo mediatore.

I prossimi sforzi nell'ambito dello sviluppo del software dovrebbero quindi essere orientati verso la fase di *Esecuzione della Query*. In Figura 4.7 sono rappresentati i moduli che la realizzano, da notare come il tutto dovrà coincidere con quanto previsto dalla teoria (Figura 3.5). È già stata implementata la classe **Data**: ogni query ha un attributo di questo tipo che conterrà le risposte all'interrogazione, in esso saranno quindi contenuti i dati ritornati dalle query di ordine inferiore. Questa classe dovrà essere specializzata diversamente per ogni tipo di

[11] *Query Parsing and Validator* e *Query Translator* in [13], *Access Plan Evaluator* nella presente tesi

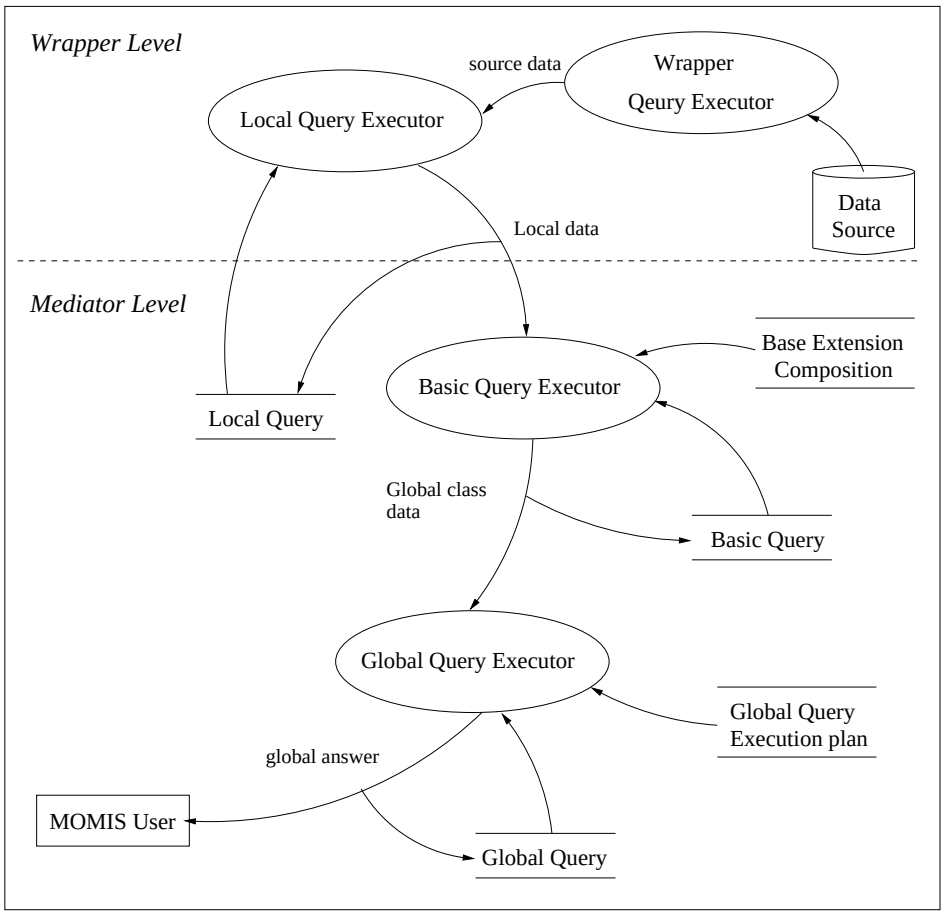


Figura 4.7: Esecuzione di una Query

query in quanto molto probabilmente i risultati delle Local Query saranno rappresentati con relazioni in database relazionali, i risultati delle Basic Query derivano dalla ricostruzione e dall'unione di base extension, ed infine i risultati delle Global Query sono i risultati finali attesi.

4.5 Considerazioni sul software prodotto

Il software realizzato per questa tesi è stato sviluppato utilizzando la versione 1.2 del Java Development Kit della Sun (jdk1.2), disponibile sul Web presso http://java.sun.com.

Tutto il software relativo al Query Manager di MOMIS[12] attualmente è disponibile nella directory /export/home/progetti.comuni/tesi/francesc/sw del server Sparc20 presso il dipartimento di Scienze dell'Ingegneria. È possibile mandarlo in esecuzione lanciando il comando:

```
java QMWindow
```

La classe QMWindow.java predispone un'interfaccia grafica attraverso la quale si possono porre interrogazioni sullo schema globale ottenuto dall'esempio di riferimento (Appendice B).

Una versione sempre aggiornata, ed in continuo sviluppo, del software del Query Manager si può invece trovare nella directory /export/home/progetti.comuni/Momis/prototype/CORBA_QueryManager, sempre su Sparc20.
In Appendice C viene riportato, a titolo di esempio, il codice relativo ad una classe Java particolarmente significativa (**BaseExtension**) unitamente alla corrispondente documentazione HTML ottenuta mediante *Javadoc*.

[12] Vale a dire il software realizzato in [13] ed in questa tesi

Capitolo 5

Le sorgenti semistrutturate: XML-QL

Buona parte delle informazioni reperibili sul Web ed utilizzabili in MOMIS si trovano in forma di dati semistrutturati, quindi anche se in un primo momento l'attenzione è stata esclusivamente rivolta alla gestione di sorgenti tradizionali o strutturate, successivamente si è cercato di estendere il contesto applicativo in modo da poter trattare anche dati multimediali. La necessità di gestire dati semistrutturati [38, 39] deriva dall'aumento dei formati in cui i dati possono essere rappresentati, tuttavia la loro trattazione apre problematiche di complessa soluzione.

Tra le varie tipologie di rappresentazione di questa categoria di sorgenti, una decisamente interessante risulta essere il linguaggio XML (eXtensible Markup Language). In questo capitolo si intende principalmente presentare un linguaggio di interrogazione per documenti XML, denominato XML-QL, ed i suoi possibili coinvolgimenti nel progetto MOMIS, specialmente nell'ambito del Query Manager.

5.1 Il ruolo del linguaggio OQL nel progetto MOMIS

Con il termine OQL (Object Query Language) si indica comunemente un linguaggio dichiarativo di interrogazione e manipolazione per Basi di Dati orientate agli oggetti basato sul modello ODMG, questo perché molti OODBMS si stanno uniformando allo standard di ODMG-93. Più in generale comunque, con il termine OQL ci si potrebbe riferire ad un generico linguaggio di interrogazione orientato agli oggetti, ma in seguito, se non diversamente specificato, lo si userà

con la prima accezione.

Come si può facilmente evincere dal nome, OQL permette di interrogare, ma anche manipolare, Basi di Dati ad oggetti; molto semplicisticamente si potrebbe affermare che OQL sta alle Basi di Dati ad oggetti come SQL sta alle Basi di Dati relazionali; in realtà questa affermazione necessita di alcune puntualizzazioni. Innanzi tutto nelle Basi di Dati orientate agli oggetti non esistono tabelle e quindi le query devono essere poste sulle classi, in quanto "contenitori di oggetti", e quindi l'operatore di selezione su una classe (o gerarchia) consiste nel recupero delle istanze della classe (o gerarchia) con la conseguenza che un'interrogazione apparentemente semplice possa in realtà risultare piuttosto complessa per via dei join impliciti. Inoltre nei DataBase relazionali il risultato di una query è ancora una relazione, mentre il risultato di una query su una classe (o un insieme di classi) presenta dei problemi di definizione.
Pur essendo necessaria una loro presentazione, queste considerazioni esulano dal contesto di questo capitolo, ai fini della comprensione del quale, l'assunto iniziale (OQL : OO-DataBase = SQL : Relational DB) può sufficientemente risultare soddisfacente.

Consentendo interrogazioni su DataBase ad oggetti, OQL è utilizzabile anche per interrogare sorgenti descritte con il linguaggio ODL (Object Definition Language) utilizzato per la specifica di schemi ad oggetti proposto anch'esso dal gruppo di standardizzazione ODMG-93 e universalmente riconosciuto come standard. Questo linguaggio, o meglio una sua estensione denominata ODL_{I^3} e finalizzata alla descrizione di Basi di Dati eterogenee in un contesto di integrazione, è quello utilizzato nel progetto MOMIS per le comunicazioni tra le sorgenti e il mediatore, come già evidenziato nella Sezione 1.2.1.

Limitandoci ad uno dei due grandi blocchi di cui il mediatore è costituito, vale a dire il Query Manager (l'altro è il Global Schema Builder), ODL_{I^3} permette ai suoi sviluppatori di prevedere interazioni solo con sorgenti descritte con questo formalismo, senza che ciò limiti in alcun modo le sue potenzialità o vada a ledere la sua generalità. Infatti sarà ugualmente possibile interrogare qualunque tipo di sorgente strutturata (DataBase ad oggetti o relazionali) e semistrutturata (ad es. documenti XML) in quanto sono presenti moduli, denominati Wrapper, che si occuperanno, in fase di integrazione dello schema, di fornire gli schemi delle singole sorgenti (a prescindere dal linguaggio con cui sono implementate) espressi in ODL_{I^3} e, in fase di interrogazione, di tradurre le query locali (generate dal Query Manager in OQL sullo schema ODL_{I^3} della sorgente interessata) nel linguaggio di interrogazione comprensibile dalla sorgente stessa. Il Query Manager quindi si preoccupa di prelevare la query formulata dall'utente sullo Schema Globale e ge-

nerare le query locali (in OQL) da mandare alle singole sorgenti per poi, una volta ottenuti da queste i dati per la risposta, integrare i risultati in modo da presentarli all'utente in un formato che rispecchi la struttura dello Schema Globale.

Questo metodo di lavoro è applicabile, come già accennato, ad ogni tipo di sorgente, sia essa strutturata che semistrutturata, con la limitazione tuttavia dovuta all'esistenza o meno del wrapper relativo alla tipologia di sorgente desiderata; e permette la generazione di una risposta intelligente alla query posta sullo Schema Globale ottenuto attraverso integrazione intensionale ed estensionale degli schemi locali.

5.2 Le sorgenti semistrutturate

Per quanto riguarda le sorgenti strutturate tradizionali (DB ad oggetti o relazionali) la descrizione dello schema è sempre disponibile e può essere tradotta quindi in ODL_{I^3} automaticamente da un wrapper specifico.

Nelle sorgenti semistrutturate invece non è definito a priori nessun schema, questo per la natura stessa dei dati semistrutturati che possono definirsi auto-descrittivi in quanto per questo tipo di dati le informazioni generalmente associate allo schema si possono direttamente estrarre dai dati stessi; la traduzione in ODL_{I^3} rimane comunque sempre possibile ma a tal fine risulta necessaria una procedura di estrazione dello schema per questo tipo di sorgenti.

Il modello di rappresentazione dei dati semistrutturati proposto in MOMIS, in pieno accordo con i numerosi proposti in letteratura, è esaurientemente descritto in [15]. Ma per le sorgenti semistrutturate, ed in particolar modo per i documenti in XML, si è pensato di rendere disponibile anche un'altra via per la loro interrogazione, trattata qui di seguito.

5.3 Ipotesi di utilizzo del linguaggio XML-QL nell'ambito del mediatore MOMIS

Il linguaggio **XML** (e**X**tensible **M**arkup **L**anguage) rappresenta un tentativo del W3C (World Wide Web Consortium) di creare un nuovo linguaggio per pagine Web che superi le limitazioni dell'HTML e sfrutti meglio di questo le potenzialità di SGML (Standard Generalized Markup Language, ISO 8879) tra cui la possibilità di specificare strutture necessarie per rappresentare schemi di DataBase e/o gerarchie di oggetti [40, 41]. Questo oltre a permettere di interrogare direttamente un documento XML permette anche di porre in relazione Basi di Dati eterogenee.

Un'importante applicazione dell'XML è difatti lo scambio di dati elettronici (EDI: Electronic Data Interchange), integrare e aggregare dati da più sorgenti sulla rete nonché trasformare e cancellare dati per facilitarne lo scambio sono bisogni sempre più sentiti; ma estrazione, conversione, trasformazione e integrazione di dati sono problemi di DataBase ben conosciuti. Da qui la necessità di disporre di un query language per sorgenti semistrutturate.

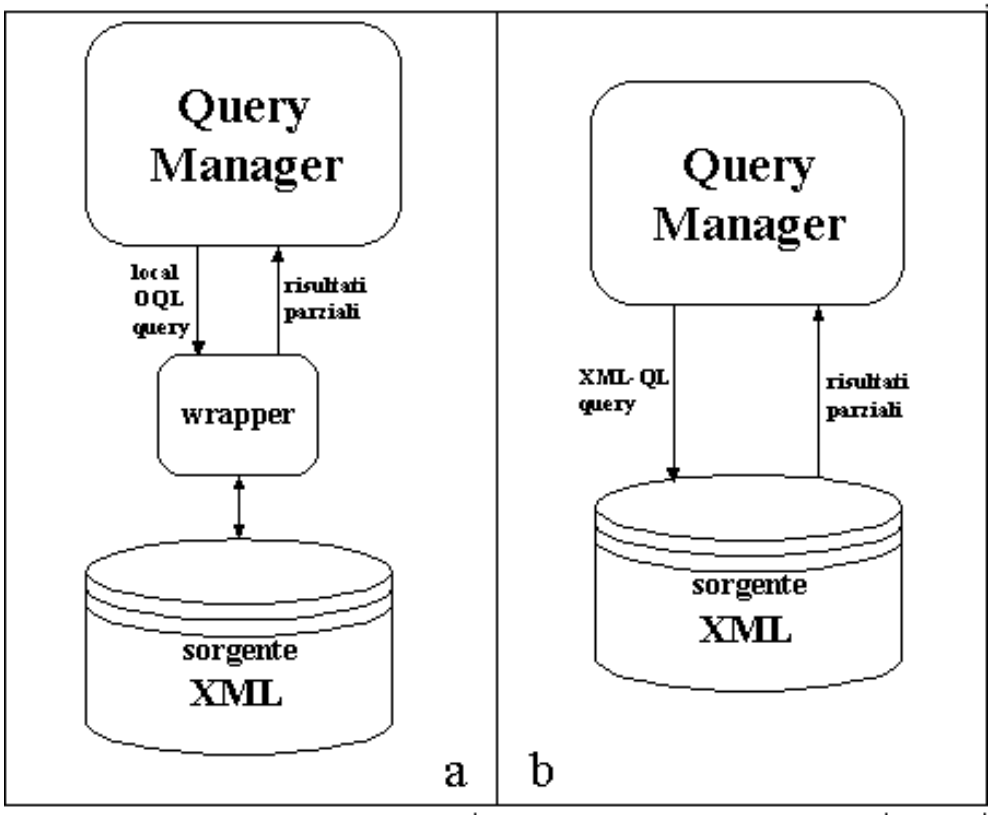


Figura 5.1: Possibili interazioni tra Query Manager e sorgenti XML

Non è ancora stato definito un linguaggio di interrogazione standard per documenti XML, numerose sono a tal riguardo le proposte e tra queste al momento la più accreditata per divenire lo standard sembra essere quella di AT&T Labs che ha presentato un linguaggio di interrogazione per documenti XML denominato **XML-QL** (**XML Q**uery **L**anguage).

Grazie all'utilizzo di questo linguaggio è quindi ipotizzabile, nell'ambito del progetto MOMIS, di affiancare, per quanto riguarda le sorgenti semistrutturate, al metodo di interrogazione basato sulla traduzione della sorgente in ODL_{I^3} e sulla

formulazione dell'interrogazione in OQL (Figura 5.1.a), un procedimento alternativo che permetta al Query Manager di interrogare direttamente i documenti XML utilizzando il linguaggio XML-QL e di interpretarne i risultati (Figura 5.1.b). Un comportamento del genere da parte del Query Manager risulterà sicuramente meno *intelligente* di quello descritto in precedenza, ma offrirà una veloce alternativa al procedimento generale che dovrà comunque restare sempre attuabile ed anzi preferibile.

Inoltre se, come sembra sempre più probabile, l'XML-QL sarà universalmente accettato come standard per l'interrogazione di documenti XML e quest'ultimo sostituirà, o perlomeno affiancherà massicciamente, l'HTML nella realizzazione di pagine Web, si può comprendere come essere dotati di un Query Manager che interagisca direttamente con questi linguaggi costituisca, se non un'irrinunciabile necessità, almeno una comodità notevole nonché un utile strumento addizionale che contribuirà al valore aggiunto del progetto MOMIS.

Ipotizzando quindi un'estensione futura del Query Manager in questa direzione, verrà qui di seguito descritto e rapidamente analizzato il linguaggio XML-QL al fine di dotare lo sviluppatore della conoscenza di base necessaria all'implementazione.

5.4 Introduzione all'XML-QL

Per trattare l'applicazione dell'XML a problematiche di EDI è necessario adottare una vista di tipo DataBase per una sorgente XML, opposta alla vista di tipo Documento. Così facendo si considera il documento come una Base di Dati e il DTD (Document Type Descriptor) ad esso associato come il DataBase Schema. E questo è proprio il punto di partenza che hanno adottato i ricercatori che hanno sviluppato XML-QL [42].
XML-QL presenta le seguenti caratteristiche principali:

- è dichiarativo;
- ha una sintassi che combina elementi di quella XML con quella di tradizionali query language (es. ha un costrutto Select-Where, tipo SQL);
- è "relational complete" (in particolare può esprimere join);
- può estrarre dati da documenti XML esistenti, modificarli e costruirne di nuovi;
- non è limitato ad interrogare un unico documento ma supporta query su più documenti;

- supporta viste sia ordinate che non ordinate;
- è infine abbastanza semplice da poter estendere ad esso le note tecniche di ottimizzazione di query, stima dei costi e riscrittura di query tipiche delle Basi di Dati.

5.4.1 Esempio introduttivo

Per meglio comprendere XML-QL introduciamo un esempio che ci servirà da riferimento per evidenziare alcune caratteristiche di questo linguaggio.

L'esempio scelto è lo stesso proposto dal gruppo di studio di AT&T in [42], documento sottoposto al W3C il 18 agosto 1998 nel quale si presentava XML-QL.

Supponiamo che il nostro input di esempio sia in www.a.b.c/bib.xml e che contenga voci di bibliografia conformi al seguente DTD:

```
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author+, title, year?, (shortversion |
          longversion))>
<!ATTLIST article type CDATA>
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname)>
```

Questo DTD specifica che un elemento *book* contiene uno o più elementi *author*, un *title*, un elemento *publisher* e ha un attributo *year*; un *article* è simile, ma il suo elemento year è opzionale, non ha il publisher e contiene un elemento *shortversion* o *longversion*; un article ha anche un attributo *type*; un pubblisher contiene gli elementi *name* e *address* mentre un author contiene un opzionale *firstname* e un *lastname* obbligatorio; si suppone inoltre che name, address, firstname e lastname siano tutti CDATA, cioè stringhe.

In XML-QL è possibile interrogare parti di documenti XML data una lista di constraint che devono essere definiti in termini di esistenza di elementi, matching di attributi, contenuto degli elementi.
Vediamo ora una semplice query che utilizza proprio il contenuto degli elementi per far corrispondere i dati:

```
WHERE <book>
        <publisher><name>Addison-Wesley</name></publisher>
        <title> $t </title>
        <author> $a </author>
```

```
      </book> IN "www.a.b.c/bib.xml"
CONSTRUCT $a
```

In via informale si può dire che questa query ci fornisce ogni elemento book nel documento www.a.b.c/bib.xml che ha almeno un title, un author e un publisher il cui name è uguale a ”Addison-Wesley”; per ogni occorrenza forza le variabili $t e $a ad ogni coppia di title e author. Si noti che i nomi delle variabili sono preceduti dal simbolo $ per non confonderle con stringhe e che in futuro si abbrevierà ⟨/elemento⟩ con ⟨/⟩ utilizzando un rilassamento della sintassi XML.

Con XML-QL è possibile anche costruire dati XML: la query precedente ad esempio produceva un documento XML contenente una lista di coppie (first-name, lastname) oppure un elemento lastname dal documento di input; mentre la seguente fornisce sia author che title e li raggruppa in un nuovo elemento *result*:

```
WHERE <book>
        <publisher><name>Addison-Wesley</></>
        <title> $t</>
        <author> $a</>
      </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>
            <author> $a</>
            <title> $t</>
          </>
```

Se consideriamo ad esempio i seguenti dati XML:

```
<bib>
  <book year="1995">
     <!-- A good introductory text -->
     <title> An Introduction to Database Systems </title>
     <author> <lastname> Date </lastname> </author>
     <publisher> <name> Addison-Wesley </name >  </publisher>
  </book>
  <book year="1998">
     <title> Foundation for Object/Relational Databases:
             The Third Manifesto </title>
     <author> <lastname> Date </lastname> </author>
     <author> <lastname> Darwen </lastname> </author>
     <publisher> <name> Addison-Wesley </name >  </publisher>
  </book>
</bib>
```

la query precedente produce:

```
<result>
   <author> <lastname> Date </lastname> </author>
   <title> An Introduction to Database Systems </title>
</result>
<result>
   <author> <lastname> Date </lastname> </author>
   <title> Foundation for Object/Relational Databases:
           The Third Manifesto </title>
</result>
<result>
   <author> <lastname> Darwen </lastname> </author>
   <title> Foundation for Object/Relational Databases:
           The Third Manifesto </title>
</result>
```

Per concludere la carrellata di esempi, vediamo adesso gli ultimi due (anch'essi proposti in [42]), il primo dei quali rappresenta un raggruppamento mediante query innestate. Per ottenere un raggruppamento per titolo di libro ed evitare che differenti autori dello stesso libro appaiano in elementi result diversi, la query precedente si può modificare con una query innestata, così:

```
WHERE <book > $p</> IN "www.a.b.c/bib.xml",
      <title > $t</>,
      <publisher><name>Addison-Wesley</>> IN $p
CONSTRUCT <result>
             <title> $t </>
             WHERE  <author> $a </> IN $p
             CONSTRUCT <author> $a</>
          </>
```

Facendo riferimento ai nostri dati di esempio, otterremo:

```
<result>
   <title> An Introduction to Database Systems </title>
   <author> <lastname> Date </lastname> </author>
</result>
<result>
   <title> Foundation for Object/Relational Databases:
           The Third Manifesto </title>
   <author> <lastname> Date </lastname> </author>
   <author> <lastname> Darwen </lastname> </author>
</result>
```

In XML-QL è inoltre possibile esprimere join. L'ultimo esempio proposto riguarda proprio un join per valore tra elementi diversi. La seguente query ci fornisce tutti gli articoli che hanno almeno un autore che ha inoltre scritto un libro a partire dal 1995:

```
WHERE <article>
        <author>
          <firstname> $f </> // firstname $f
          <lastname> $l </>  // lastname $l
        </>
      </> CONTENT_AS $a IN "www.a.b.c/bib.xml"
      <book year=$y>
        <author>
          <firstname> $f </> // join on same firstname $f
          <lastname> $l </>  // join on same lastname $l
        </>
      </> IN "www.a.b.c/bib.xml",
      y > 1995
CONSTRUCT <article> $a </>
```

Note: quelli preceduti da // sono commenti e CONTENT_AS serve per legare il contenuto dell'elemento che ha il match con la variabile ($) che segue.

5.4.2 Considerazioni aggiuntive su XML-QL

XML-QL prevede un modello di dati Ordinato ed uno Non Ordinato [43]. Il secondo permette di utilizzare alcuni compromessi che semplificano la semantica del linguaggio e aumentano l'efficacia e l'utilità dei processi di ottimizzazione di query. Vediamo meglio.

In un documento XML sono contenute informazioni addizionali che non sono direttamente attinenti ai dati stessi, quali (riferendoci al nostro esempio) commento all'inizio del primo elemento book, il fatto che il titolo precede l'autore o il fatto che un libro del '95 precede uno del'98, etc. Questi non sono rilevanti in un'applicazione EDI che ignora i commenti e tratta i dati come strutture non ordinate di valori.
Possiamo quindi distinguere tra informazioni intrinseche nei dati ed altre, come ad esempio le specifiche di layout, che non lo sono. Il modello Non Ordinato ignora i commenti e l'ordine relativo tra gli elementi, ma conserva tutti gli altri dati essenziali.

I dati XML possono quindi essere rappresentati attraverso un Grafo XML non ordinato che si può definire come un grafo, G, nel quale ogni nodo è rappre-

sentato da un'unica stringa chiamata Object Identifier (OID), gli archi di G sono etichettati con i tag degli elementi, i nodi con insiemi di coppie attributo-valore, le foglie con una stringa che ne rappresenta il valore ed infine G ha un nodo particolare denominato root. I grafi XML non derivano solo da documenti XML, ma possono anche essere generati da query.
Un Grafo XML ordinato invece è un grafo nel quale vi è un ordine totale tra tutti i nodi. Per quelli generati da documenti un ordinamento naturale può essere quello del documento stesso.

Bisogna notare infine che anche se i documenti XML sono sempre ordinati (DTD può imporre un particolare ordine sugli elementi) è più difficile interrogare dati ordinati. Bisogna prevedere costrutti addizionali per gestire l'ordine, la semantica risulterà quindi più complessa e il query optimizer avrà meno possibilità di generare un buon piano d'accesso. Per molte applicazioni EDI l'ordine non è importante, in questi casi lo si ignora e si usa un sottoinsieme di XML-QL riservando l'utilizzo del modello Ordinato solo ai casi in cui l'ordine è determinante.

Per supportare la condivisione di elementi XML prevede un attributo di tipo ID per specificare una chiave unica per un elemento e un attributo IDREF che permette ad un elemento di riferirsi ad un altro attraverso la chiave designata, mentre uno di tipo IDREFS può riferirsi a elementi multipli. Nel modello dei dati questi attributi vengono trattati in modo particolare. Come già accennato, in un grafo XML ogni nodo ha un OID che è l'attributo ID associato con il corrispondente elemento oppure è generato se non esiste nessun attributo ID.

5.4.3 Esempi più complessi (in un'ottica MOMIS)

XML-QL possiede anche alcune caratteristiche avanzate che potrebbero risultare interessanti dal punto di vista di un suo futuro utilizzo nell'ambito del progetto MOMIS. Due di queste (tag variables e regular path expressions) permettono di scrivere query applicabili a più sorgenti XML con DTD simili (ma non identici).

Esempio di tag variables:

```
WHERE <$p>
        <title> $t </title>
        <year>1995</>
        <$e> Smith </>
      </> IN "www.a.b.c/bib.xml",
      $e IN {author, publisher}
CONSTRUCT <$p>
```

```
            <title> $t </title>
            <$e> Smith </>
        </>
```

questa query fornisce tutte le pubblicazioni del 1995 delle quali Smith è o l'autore o l'editore. $p è una variabile tag legata ad ogni tag di livello superiore (book, article, . . .); anche $e è una variabile tag: la query la forza a legarsi a uno tra author o publisher.

Introduciamo per un momento un nuovo esempio di riferimento per vedere una regular path expression:

```
<!ELEMENT part (name brand part*)>
<!ELEMENT name CDATA>
<!ELEMENT brand CDATA>
```

Ogni elemento part può contenere altri elementi part innestati.

```
WHERE <part*> <name>$r</> <brand>Ford</> </> IN
"www.a.b.c/parts.xml" CONSTRUCT <result>$r</>
```

part* è una regular path expression e soddisfa ogni sequenza di archi etichettati con part. Difatti

```
<part*> <name>$r</> <brand>Ford</> </>
```

è equivalente agli infiniti:

```
<name>$r</> <brand>Ford</>
<part> <name>$r</> <brand>Ford</> </>
<part> <part> <name>$r</> <brand>Ford</> </> </>
<part> <part> <part> <name>$r</> <brand>Ford</> </> </> </>
...
```

Sono disponibili in XML-QL anche gli operatori di alternanza (|), concatenamento (.) e Kleene-star (*) simili a quelli delle espressioni regolari (l'operatore + significa "uno o più": part+ è equivalente a part.part*).

Una possibilità di XML-QL che può risultare molto importante ai nostri fini è quella di trasformare i dati XML.

Aggiungiamo all'esempio DTD originale della bibliografia un DTD che definisce una persona:

```
<!ELEMENT person (lastname, firstname, address?, phone?,
  publicationtitle*)>
```

La query seguente estrae degli autori dalla bibliografia e li trasforma in elementi *person* in conformità con il DTD person:

```
WHERE <$> <author> <firstname> $fn </>
                   <lastname> $ln </>
          </>
          <title> $t </>
      </> IN "www.a.b.c/bib.xml",
CONSTRUCT <person ID=PersonID($fn, $ln)>
              <firstname> $fn </>
              <lastname> $ln </>
              <publicationtitle> $t </>
          </>
```

Si possono anche interrogare più sorgenti contemporaneamente e produrre una vista integrata dei loro dati:

```
WHERE <person>
        <name></> ELEMENT_AS $n
        <ssn> $ssn</>
      </> IN "www.a.b.c/data.xml",

      <taxpayer>
        <ssn> $ssn</>
        <income></> ELEMENT_AS $i
      </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> $n $i </>
```

In questo modo abbiamo prodotto tutte le coppie di nome e numero della sicurezza sociale interrogando www.a.b.c/data.xml e www.irs.gov/taxpayers.xml (il join avviene sul numero di sicurezza sociale, il risultato contiene solo gli elementi che hanno sia una elemento name della prima sorgente che uno income dalla seconda). Avremmo potuto spezzare la query in due frammenti:

```
{ WHERE <person>
          <name> </> ELEMENT_AS $n
          <ssn> $ssn </>
        </> IN "www.a.b.c/data.xml"
  CONSTRUCT <result ID=SSNID($ssn)> $n </>
}

{ WHERE <taxpayer>
          <ssn> $ssn </>
```

```
          <income> </> ELEMENT_AS $i
        </> IN "www.irs.gov/taxpayers.xml"
  CONSTRUCT <result ID=SSNID($ssn)> $i </>
}
```

ottenendo una query simile ma non equivalente: in questo caso difatti otteniamo anche tutte le persone che non sono taxpayers e viceversa (è un outer join). Le query XML-QL sono strutturate in blocchi e semiblocchi, questi ultimi racchiusi tra parentesi graffe.

Infine, XML-QL supporta la definizione di funzioni che hanno come parametri documenti XML, esse aumentano il riuso delle query in quanto possono essere applicate a più documenti. Una funzione è definita nel modo seguente:

```
FUNCTION functionName
 ($firstParameter, $secondParameter, ...): funcResult
    { query }
```

Parametri e risultato possono essere limitati da DTD.

In Appendice D viene presentata una descrizione del linguaggio XML-QL mediante la notazione BNF.

5.5 XML-QL e OQL

Un confronto tra OQL e XML-QL non è direttamente possibile per via dei tipi di sorgenti che i due linguaggi sono chiamati ad interrogare: dati strutturati (DataBase ad oggetti o rappresentazioni ad oggetti di schemi) il primo, documenti XML (quindi dati semistrutturati) il secondo.

Un loro utilizzo nell'ambito del progetto MOMIS non può quindi intendersi in senso esclusivo in quanto una sostituzione di OQL con XML-QL è assolutamente, non solo impossibile, ma nemmeno ipotizzabile: OQL può anche essere utilizzato per interrogare sorgenti semistrutturate, previa una loro traduzione in ODL_{I^3} , mentre XML-QL è utilizzabile esclusivamente per interrogare documenti XML (quindi nemmeno tutti i possibili dati semistrutturati). Questi linguaggi possono essere quindi considerati alternativi nell'ambito di MOMIS solo per quanto riguarda l'interrogazione di documenti XML.

Un approccio che non prenda in considerazione la strada di una traduzione in ODL_{I^3} e un'interrogazione attraverso OQL di sorgenti XML, non è tuttavia conveniente in quanto le informazioni ottenibili da uno Schema Globale generato da un maggior numero e tipologie di sorgenti hanno una valenza ed un livello di

intelligenza della risposta maggiore.

XML-QL suscita dunque interesse nell'ambito di MOMIS in riferimento ad un suo possibile utilizzo parallelamente all'OQL. Come già accennato in precedenza, la possibilità di affiancare la prassi, definiamola, tradizionale di interrogazione adottata dal Query Manager e sempre applicabile con una più rapida, come può essere appunto l'utilizzo direttamente di XML-QL, per l'interrogazione di un certo tipo di sorgenti, quelle semistrutturate, che stanno assumendo sempre maggior rilievo, permette una più marcata flessibilità e completezza.

L'obiettivo è quello di dare all'utente la possibilità di interrogare direttamente un documento XML per ottenerne una risposta soddisfacente anche se non completamente integrata con la vista di insieme. Questo non si sposa perfettamente con la filosofia di MOMIS che invece basa la sua forza sulla possibilità di interrogazioni e risposte intelligenti grazie all'integrazione, non solo intensionale ma anche estensionale, delle sorgenti. Lo sforzo compiuto nella direzione XML-QL è quindi da intendersi nell'ottica di fornire all'utilizzatore anche alternative meno sofisticate ma più veloci.

Per evidenziare le differenze viene presentata adesso una ipotetica rappresentazione dell'esempio di riferimento (Bibliografia) in ODL_{I^3} ed alcune delle query viste in precedenza tradotte in OQL.

```
Bibliography (B):

interface Book
 ( source semistructured Bibliography )
 { attribute set<Author> author ;
  attribute string      title ;
  attribute Publisher publisher ;
  attribute integer   year; } ;

interface Article
 ( source semistructured Bibliography )
 { attribute set<Author>       author ;
  attribute string            title ;
  attribute integer           year* ;
  attribute enum {short,long} version ;
  attribute string            type; } ;

interface Publisher
 ( source semistructured Bibliography )
```

```
 { attribute string name ;
  attribute string address ; } ;

interface Author
 ( source semistructured Bibliography )
 { attribute string firstname* ;
  attribute string lastname ;} ;
```

La prima semplicissima query dell'esempio, quella che fornisce tutte le coppie autore - titolo di un libro con editore Addison-Wesley, posta in OQL sullo schema precedente si può esprimere come:

```
Select author, title
from Book
where name = 'Addison-Wesley'
```

In XML-QL per evitare che autori diversi dello stesso libro venissero conteggiati disgiuntamente si adottava una struttura a query innestate, in OQL basta utilizzare la clausola "Group By":

```
Select author, title
from Book
where name = 'Addison-Wesley'
group by title
```

Un'altra query ci permetteva, mediante un join per valore, di ottenere tutti gli articoli che hanno almeno un autore che ha inoltre scritto un libro a partire dal 1995, anche questo è ovviamente esprimibile in OQL:

```
Select a.author, a.title, a.year, a.version, a.type
from Article as a, Book as b
where a.firstname = b.firstname
and a.lastname = b.lastname
and b.year > 1995
```

Vediamo un ultimo esempio. In XML-QL mediante tag variables è semplice esprimere query su elementi che assumono significati diversi. Abbiamo visto quella che ci permette di ottenere le pubblicazioni del 1995 nelle quali Smith è o l'autore o l'editore. In OQL diventerebbe:

```
Select b.title, a.lastname
from Author as a, Book as b
where a.firstname = b.firstname
```

```
and a.lastname = b.lastname
and b.year = 1995
union
Select c.title, a.lastname
from Author as a, Article as c
where a.firstname = c.firstname
and a.lastname = c.lastname
and c.year = 1995
union
select b.title, p.name
from Publisher as p, Book as b
where p.name = b.name
and a.year = 1995
```

5.6 Commenti

Dagli esempi emerge una capacità espressiva e di sintesi notevole per XML-QL grazie all'utilizzo di tag variables ed anche regular path expressions e funzioni. Interrogazioni che richiedano accorgimenti del genere risultano quindi facilmente esprimibili in questo linguaggio come lo sono del resto trasformazione di dati ed integrazione di dati da sorgenti diverse (si riguardino a tal proposito i relativi esempi). Questi sono i tipi di query meno banali ma anche i più facilmente riscontrabili in applicazioni EDI quali la comunicazione e lo scambio di informazioni tra DataBase diversi possono essere, e quindi risultano interessanti anche nell'ambito del progetto MOMIS.

Altre query, banali in SQL o OQL, risultano esprimibili solo con un formalismo pesante per via della natura non strutturata dei dati che si devono interrogare e della mancanza di alcune caratteristiche tipiche invece dei due linguaggi di interrogazione per Basi di Dati strutturate, dovuta anche in parte all'incompletezza di XML-QL ancora in fase di evoluzione.

Predisporre strumenti per un utilizzo di XML-QL nell'ambito dell'interrogazione di sorgenti semistrutturate in MOMIS può risultare quindi utile, non solo per i vantaggi di flessibilità e completezza già trattati in precedenza, ma anche a livello pratico per la sua più naturale predisposizione ad interagire con dati di questo tipo rispetto ad OQL che deve pur sempre interrogare schemi ODL_{I^3} che, traducendo in un formalismo ad oggetti documenti XML, inevitabilmente tende a modificare la natura di questi dati semistrutturati.
Bisogna però sottolineare anche il fatto che OQL ha dimostrato di possedere tutte le caratteristiche necessarie per interrogare esaurientemente uno schema integra-

to, un’espansione del Query Manager in direzione di un utilizzo diretto di XML-QL quindi non risulta in questo momento particolarmente urgente, mentre lo è la creazione dei wrapper per XML e XML-QL al fine di poter integrare nello schema globale anche sorgenti XML.

Conclusioni

Il progetto MOMIS (Capitolo 1) si prefigge l'obiettivo di realizzare un mediatore in grado di integrare sorgenti eterogenee di dati, al fine di poter interrogare uno schema globale integrato permettendo all'utente di esulare completamente dal modo in cui sono organizzate effettivamente le diverse sorgenti. Come il sistema recupererà i dati dalle sorgenti e come li combinerà per ottenere la risposta globale richiesta deve risultare del tutto trasparente all'utente, che avrà quindi l'impressione di interagire con un unico DataBase. Le soluzioni dei problemi da risolvere affinché quanto appena esposto si realizzi risultano tutt'altro che di semplici. In alcune delle precedenti tesi su MOMIS ([15, 16, 17]) si sono individuati questi problemi e si sono proposte ed implementate soluzioni per l'integrazione delle informazioni in modo intelligente. Ma questo costituisce solo una delle due macrofunzioni di MOMIS: la *Creazione dello Schema Globale*, svolta dal modulo **Global Schema Builder**.

Una volta ottenuto lo schema globale integrato bisogna difatti poter essere in grado di interrogarlo; alla gestione delle interrogazioni è preposto l'altro macromodulo di MOMIS: il **Query Manager**, che svolge la funzione di *Query Processing*. Allo studio ed al progetto di questo modulo, nonché all'implementazione dei suoi principali sottomoduli, sono state dedicate altre tesi ([13, 14]).

Anche in questa tesi si è studiato principalmente il Query Manager e si è posta particolare attenzione alle implicazioni che in esso derivano dalla Conoscenza Estensionale (Capitolo 2). Il contributo fornito si è materializzato nell'ampliamento del software del Query Manager, mediante l'implementazione del modulo che permette di sfruttare tale conoscenza al fine di individuare l'insieme ottimale di classi locali alle quali effettivamente accedere per reperire i dati richiesti dall'utente. Contestualmente si arriva anche a definire un piano di accesso, vale a dire un insieme di indicazioni su come gestire le Local Query ed i risultati parziali da esse ritornati, per ottenere i risultati delle Basic Query. Per riuscire nell'intento prima si sono individuate e studiate le varie sottofasi dalle quali è composta quella

di *Query Processing* (Capitolo 3), poi si è analizzato il software esistente e lo si è integrato con le nuove funzionalità implementate (Capitolo 4).

Infine (Capitolo 5) si è studiata, sempre dal punto di vista del Query Manager, l'interazione tra MOMIS ed il linguaggio XML-QL, linguaggio per l'interrogazione di sorgenti semistrutturate descritte in XML. L'obiettivo era di individuare se fosse possibile interagire direttamente con questa tipologia di sorgenti senza l'utilizzo dei wrapper, per una risposta più veloce, ma anche inevitabilmente meno significativa. Il linguaggio di interrogazione utilizzato in MOMIS (OQL) ha dimostrato di possedere già tutte le potenzialità e funzionalità di XML-QL, quindi questa ipotesi è stata classificata come un marginale e non urgente, anche se possibile, ampliamento delle funzioni del Query Manager.

Concludendo è da sottolineare come si stia attualmente lavorando nell'ambito del progetto MOMIS per definire uno standard comune di comunicazione tra i vari moduli (Global Schema Builder, Query Manager, Wrapper) al fine di rendere ancora più agevole l'ampliamento futuro. Si stanno individuando le interfacce che i vari moduli dovranno presentare e si è deciso di adottare lo standard CORBA (Common ORB Architecture) per le comunicazioni tra i moduli [44]. CORBA è una tecnologia per l'integrazione, inoltre è ad oggetti ed una modellazione di questo tipo permette di ridurre la complessità di MOMIS: esistono difatti metodologie consolidate per la rappresentazione e progettazione di sistemi ad oggetti (OMT, UML), ma soprattutto per utilizzare un oggetto è sufficiente conoscerne l'interfaccia pubblica e questo favorisce il lavoro degli sviluppatori successivi.

Appendice A

Glossario I^3

Questo glossario ed il vocabolario sul quale si basa sono stati originariamente sviluppati durante l'I^3 Architecture Meeting in Boulder CO, 1994, sponsorizzato dall'ARPA, e rifiniti in un secondo incontro presso l'Università di Stanford, nel 1995. Il glossario è strutturato logicamente in diverse sezioni:

- Sezione 1: Architettura
- Sezione 2: Servizi
- Sezione 3: Risorse
- Sezione 4: Ontologie

Nota: poiché la versione originaria del glossario usa una terminologia inglese, in alcuni casi è riportato, a fianco del termine, il corrispettivo inglese, quando la traduzione dal termine originale all'italiano poteva essere ambigua o poco efficace.

A.1 Architettura

- Architettura = insieme di componenti.
- architettura di riferimento = linea guida ed insieme di regole da seguire per l'architettura.
- componente = uno dei blocchi sui quali si basa una applicazione o una configurazione. Incorpora strumenti e conoscenza specifica del dominio.
- applicazione = configurazione persistente o transitoria dei componenti, rivolta a risolvere un problema del cliente, e che può coprire diversi domini.

- configurazione = istanza particolare di una architettura per una applicazione o un cliente.
- collante (glue) = software o regole che servono per per collegare i componenti o per interoperare attraverso i domini.
- strato = grossolana categorizzazione dei componenti e degli strumenti in una configurazione. L'architettura I^3 distingue tre strati, ognuno dei quali fornisce una diversa categoria di servizi:
  1. Servizi di Coordinamento = coprono le fasi di scoperta delle risorse, distribuzione delle risorse, invocazione, scheduling . . .
  2. Servizi di Mediazione = coprono la fase di query processing e di trattamento dei risultati, nonché il filtraggio dei dati, la generazione di nuove informazioni, etc.
  3. Servizi di Wrapping = servono per l'utilizzo dei wrappers e degli altri strumenti simili utilizzati per adattarsi a standards di accesso ai dati e alle convenzioni adoperate per la mediazione e per il coordinamento.
- agente = strumento che realizza un servizio, sia per il suo proprietario, sia per un cliente del suo proprietario.
- facilitatore = componente che fornisce i servizi di coordinamento, come pure l'instradamento delle interrogazioni del cliente.
- mediatore = componente che fornisce i servizi di mediazione e che provvede a dare valore aggiunto alle informazioni che sono trasmesse al cliente in risposta ad una interrogazione.
- cliente (customer) = proprietario dell'applicazione che gestisce le interrogazioni, o utente finale, che usufruisce dei servizi.
- risorsa = base di dati accessibile, server ad oggetti, base di conoscenze . . .
- contenuto = risultato informativo ricavato da una sorgente.
- servizio = funzione fornita da uno strumento in un componente e diretta ad un cliente, direttamente od indirettamente.
- strumento (tool) = programma software che realizza un servizio, tipicamente indipendentemente dal dominio.
- wrapper = strumento utilizzato per accedere alle risorse conosciute, e per tradurre i suoi oggetti.

- regole limitative (constraint rules) = definizione di regole per l‘assegnamento di componenti o di protocolli a determinati strati.
- interoperare = combinare sorgenti e domini multipli.
- informazione = dato utile ad un cliente.
- informazione azionabile = informazione che forza il cliente ad iniziare un evento.
- dato = registrazione di un fatto.
- testo = dato, informazione o conoscenza in un formato relativamente non strutturato, basato sui caratteri.
- conoscenza = metadata, relazione tra termini, paradigmi . . . , utili per trasformare i dati in informazioni.
- dominio = area, argomento, caratterizzato da una semantica interna, per esempio la finanza, o i componenti elettronici . . .
- metadata = informazione descrittiva relativa ai dati di una risorsa, compresi il dominio, proprietà, le restrizioni, il modello di dati, . . .
- metaconoscenza = informazione descrittiva relativa alla conoscenza in una risorsa, includendo l‘ontologia, la rappresentazione . . .
- metainformazioni = informazione descritiiva sui servizi, sulle capacità, sui costi . . .

A.2 Servizi

- Servizio = funzionalità fornita da uno o più componenti, diretta ad un cliente.
- instradamento (routing) = servizio di coordinamento per localizzare ed invocare una risorsa o un servizio di mediazione, o per creare una configurazione. Fa uso di un direttorio.
- scheduling = servizio di coordinamento per determinare l‘ordine di invocazione degli accessi e di altri servizi; fa spesso uso dei costi stimati.
- accoppiamento (matchmaking) = servizio che accoppia i sottoscrittori di un servizio ai fornitori.

- intermediazione (brokering) = servizio di coordinamento per localizzare le risorse migliori.
- strumento di configurazione = programma usato nel coordinamento per aiutare a selezionare ed organizzare i componenti in una istanza particolare di una configurazione architetturale.
- servizi di descrizione = metaservizi che informano i clienti sui servizi, risorse . . .
- direttorio = servizio per localizzare e contattare le risorse disponibili, come le pagine gialle, pagine bianche . . .
- decomposizione dell'interrogazione (query decomposition) = determina le interrogazioni da spedire alle risorse o ai servizi disponibili.
- riformulazione dell'interrogazione (query reformulation) = programma per ottimizzare o rilassare le interrogazioni, tipicamente fa uso dello scheduling.
- contenuto = risultato prodotto da una risorsa in risposta ad interrogazioni.
- trattamento del contenuto (content processing) = servizio di mediazione che manipola i risultati ottenuti, tipicamente per incrementare il valore delle informazioni.
- trattamento del testo = servizio di mediazione che opera sul testo per ricerca, correzione . . .
- filtraggio = servizio di mediazione per aumentare la pertinenza delle informazioni ricevute in risposta ad interrogazioni.
- classificazione (ranking) = servizio di mediazione per assegnare dei valori agli oggetti ritrovati.
- spiegazione = servizio di mediazione per presentare i modelli ai clienti.
- amministrazione del modello = servizio di mediazione per permettere al cliente ed al proprietario del mediatore di aggiornare il modello.
- integrazione = servizio di mediazione che combina i contenuti ricevuti da una molteplicità di risorse, spesso eteorgenee.
- accoppiamento temporale = servizio di mediazione per riconoscere e risolvere differenze nelle unità di misura temporali utilizzate dalle risorse.

- accoppiamento spaziale = servizio di mediazione per riconoscere e risolvere differenze nelle unità di misura spaziali utilizzate dalle risorse.
- ragionamento (reasoning) = metodologia usata da alcuni componenti o servizi per realizzare inferenze logiche.
- browsing = servizio per permettere al cliente di spostarsi attraverso le risorse.
- scoperta delle risorse = servizio che ricerca le risorse.
- indicizzazione = creazione di una lista di oggetti (indice) per aumentare la velocità dei servizi di accesso.
- analisi del contenuto = trattamento degli oggetti testuali per creare informazioni.
- accesso = collegamento agli oggetti nelle risorse per realizzare interrogazioni, analisi o aggiornamenti.
- ottimizzazione = processo di manipolazione o di riorganizzazione delle interrogazioni per ridurne il costo o il tempo di risposta.
- rilassamento = servizio che fornisce un insieme di risposta maggiore rispetto a quello che l'interrogazione voleva selezionare.
- astrazione = servizio per ridurre le dimensioni del contenuto portandolo ad un livello superiore.
- pubblicità (advertising) = presentazione del modello di una risorsa o del mediatore ad un componente o ad un cliente.
- sottoscrizione = richiesta di un componente o di un cliente di essere informato su un evento.
- controllo (monitoring) = osservazione delle risorse o dei dati virtuali e creazione di impulsi da azionare ogniqualvolta avvenga un cambiamento di stato.
- aggiornamento = trasmissione dei cambiamenti dei dati alle risorse.
- instanziazione del mediatore = popolamento di uno strumento indipendente dal dominio con conoscenze dipendenti da un dominio.
- attivo (activeness) = abilità di un impulso di reagire ad un evento.

- servizio di transazione = servizio che assicura la consistenza temporale dei contenuti, realizzato attraverso l‘amministrazione delle transzioni.
- accertamento dell‘impatto = servizio che riporta quali risorse saranno interessate dalle interrogazioni o dagli aggiornamenti.
- stimatore = servizo di basso livello che stima i costi previsti e le prestazioni basandosi su un modello, o su statistiche.
- caching = mantenere le informazioni memorizzate in un livello intermedio per migliorare le prestazioni.
- traduzione = trasformazione dei dati nella forma e nella sintassi richiesta dal ricevente.
- controllo della concorrenza = assicurazione del sincronismo degli aggiornamenti delle risorse, tipicamente assegnato al sistema che amministra le transazioni.

A.3 Risorse

- Risorsa = base di dati accessibile, simulazione, base di conoscenza, . . . comprese le risorse ”legacy”.
- risorse ”legacy” = risorse preesistenti o autonome, non disegnate per interoperare con una architettura generale e flessibile.
- evento = ragione per il cambiamento di stato all‘interno di un componente o di una risorsa.
- oggetto = istanza particolare appartenente ad una risorsa, al modello del cliente, o ad un certo strumento.
- valore = contenuto metrico presente nel modello del cliente, come qualità, rilevanza, costo.
- proprietario = individuo o organizzazione che ha creato, o ha i diritti di un oggetto, e lo può sfruttare.
- proprietario di un servizio = individuo o organizazzione responsabile di un servizio.
- database = risorsa che comprende un insieme di dati con uno schema descrittivo.

- warehouse = database che contiene o dà accesso a dati selezionati, astratti e integrati da una molteplicità di sorgenti. Tipicamente ridondante rispetto alle sorgenti di dati.
- base di conoscenza = risorsa comprendente un insieme di conoscenze trattabili in modo automatico, spesso nella forma di regole e di metadata; permettono l'accesso alle risorse.
- simulazione = risorsa in grado di fare proiezioni future sui dati e generare nuove informazioni, basata su un modello.
- amministrazione della transazione = assicurare che la consistenza temporale del database non sia compromessa dagli aggiornamenti.
- impatto della transazione = riporta le risorse che sono state coinvolte in un aggiornamento.
- schema = lista dedlle relazioni, degli attributi e, quando possibile, degli oggetti, delle regole, e dei metadata di un database. Costituisce la base dell'ontologia della risorsa.
- dizionario = lista dei termini, fa parte dell'ontologia.
- modello del database = descrizione formalizzata della risorsa database, che include lo schema.
- interoperabilità = capacità di interoperare.
- eterogeneità = incompatibilità trovate tra risorse e servizi sviluppati autonamamente, che vanno dalla paittaforma utilizzata, sistema operativo, modello dei dati, alla semantica, ontologia, . . .
- costo = prezzo per fornire un servizio o un accesso ad un oggetto.
- database deduttivo = database in grado di utilizzare regole logiche per trattare i dati.
- regola = affermazione logica, unità della conoscenza trattabile in modo automatico.
- sistema di amministrazione delle regole = software indipendente dal dominio che raccoglie, seleziona ed agisce sulle regole.
- database attivo = database in grado di reagire a determinati eventi.
- dato virtuale = dato rappresentato attraverso referenze e procedure.

- stato = istanza o versione di una base di dati o informazioni.
- cambiamento di stato = stato successivo ad una azione di aggiornamento, inserimento o cancellazione.
- vista = sottoinsieme di un database, sottoposto a limiti, e ristrutturato.
- server di oggetti = fornisce dati oggetto.
- gerarchia = struttura di un modello che assegna ogni oggetto ad un livello, e definisce per ogni oggetto l'oggetto da cui deriva.
- network = struttura di un modello che fa uso di relazioni relativamente libere tra oggetti.
- ristrutturare = dare una struttura diversa ai dati seguendo un modello differente dall'originale.
- livello = categorizzazione concettuale , dove gli oggetti di un livello inferiore dipendono da un antenato di livello superiore.
- antenato (ancestor) = oggetto di livello superiore, dal quale derivano attributi ereditabili.
- oggetto root = oggetto da cui tutti gli altri derivano, all'interno di una gerarchia.
- datawarehouse = deposito di dati integrati provenienti da una molteplicità di risorse.
- deposito di metadata = database che contiene metadata o metainformazioni.

A.4 Ontologia

- Ontologia = descrizione particolareggiata di una concettualizzazione, i.e. l'insieme dei termini e delle relazioni usate in un dominio, per indicare oggetti e concetti, spesso ambigui tra domini diversi.
- concetto = definisce una astrazione o una aggregazione di oggetti per il cliente.
- semantico = che si riferisce al significato di un termine, espresso come un insieme di relazioni.

- sintattico = che si riferisce al formato di un termine, espresso come un insieme di limitazioni.
- classe = definisce metaconoscenze come metodi, attributi, ereditarietà, per gli oggetti in essa istanziati.
- relazione = collegamento tra termini, come *is-a*, *part-of*, . . .
- ontologia unita (merged) = ontologia creata combinando diverse ontologie, ottenuta mettendole in relazione tra loro (mapping).
- ontologia condivisa = sottoinsieme di diverse ontologie condiviso da una molteplicità di utenti.
- comparatore di ontologie = strumento per determinare relazioni tra ontologie, utilizzato per determinare le regole necessarie per la loro integrazione.
- mapping tra ontologie = trasformazione dei termini tra le ontologie, attraverso regole di accoppiamento, utilizzato per collegare utenti e risorse.
- regole di accoppiamento (matching rules) = dichiarazioni per definire l‘equivalenza tra termini di domini diversi.
- trasformazione dello schema = adattamento dello schema ad un‘altra ontologia.
- editing = trattamneto di un testo per assicurarne la conformità ad una ontologia.
- algebra dell‘ontologia = insieme delle operazioni per definire relazioni tra ontologie.
- consistenza temporale = è raggiunta se tutti i dati si riferiscono alla stessa istanza temporale ed utilizzano la stessa granularità temporale.
- specifico ad un dominio = relativo ad un singolo dominio, presuppone l‘assenza di incompatibilità semantiche.
- indipendente dal dominio = software, strumento o conoscenza globale applicabile ad una molteplicità di domini.

Appendice B

Esempio di riferimento in ODL$_{I^3}$

Quella che segue è la descrizione in linguaggio ODL$_{I^3}$ dell'esempio a cui si è fatto riferimento in questa tesi, quello dell'Università.

```
UNIVERSITY source:
interface University_Worker
( source relational University
  extent University_Worker
  key first_name, last_name
  foreign_key dept_code )
{ attribute string first_name;
  attribute string last_name;
  attribute integer dept_code;
  attribute integer pay; };

interface Research_Staff
( source relational University
  extent Research_Staffers
  keys first_name, last_name
  foreign_key dept_code, section_code )
{ attribute string first_name;
  attribute string last_name;
  attribute string relation;
  attribute string e_mail;
  attribute integer dept_code;
  attribute integer section_code;
  attribute integer pay; };

interface School_Member
( source relational Universit
  extent School_Members
  keys first_name, last_name
{ attribute string  first_nam
  attribute string  last_name
  attribute string  faculty;
  attribute integer year; }

interface Department

interface Section
```

```
( source relational University
  extent Departments
  key dept_code )
{ attribute string dept_name;
  attribute integer dept_code;
  attribute integer budget;
  attribute string dept_area; };
```

```
( source relational University
  extent Sections
  key section_code
  foreign_key room_code )
{ attribute string section_name;
  attribute integer section_code;
  attribute integer length;
  attribute integer room_code; };
```

```
interface Room
( source relational University
  extent Room
  key room_code )
{ attribute integer room_code;
  attribute integer seats_number;
  attribute string notes; };
```

COMPUTER_SCIENCE source:

```
interface CS_Person
( source object Computer_Science
  extent CS_Persons
  key name )
{ attribute string name; };
```

```
interface Professor : CS_Person
( source object Computer_Science
  extent Professors )
{ attribute string title;
  attribute Division belongs_to;
  attribute string rank; };
```

```
interface Student : CS_Person
( source object Computer_Science
  extent Students )
{ attribute integer year;
  attribute set<Course> takes;
  attribute string rank; };
```

```
interface Division
( source object Computer_Science
  extent Divisions
  key description )
{ attribute string description;
  attribute Location address;
  attribute integer fund;
  attribute integer employee_nr;
  attribute string sector; };
```

```
interface Location
( source object Computer_Science
```

```
interface Course
( source object Computer_Science
```

```
  extent Locations                    extent Courses
  keys city, street, county, number)  key course_name )
{ attribute string city;            { attribute string course_name;
  attribute string street;            attribute Professor taught_by;
  attribute string county;
  attribute integer number; };
```

Tax_Position source:

```
interface University_Student
( source file Tax_Position
  extent University_Students
  key student_code )
{ attribute string name;
  attribute integer student_code;
  attribute string faculty_name;
  attribute integer tax_fee; };
```

Appendice C

La classe Java BaseExtension

Come già accennato nel Capitolo 4, il software prodotto in questa sede è disponibile nella directory /export/home/progetti.comuni/tesi/francesc/sw del server Sparc20 presso il dipartimento di Ingegneria, unitamente al software prodotto in [13]. Eseguendo la classe Java QMWindow è possibile utilizzare il Query Manager di MOMIS interrogando lo schema globale fornito dall'esempio citato in questa tesi (Capitolo 2 ed Appendice B).

C.1 Il codice

A titolo esemplificativo viene riportato qui di seguito il codice relativo ad una classe Java particolarmente significativa tra quelle implementate: la classe *BaseExtension.*

BaseExtension.java

```
package globalschema;

import java.util.Vector;
import java.util.TreeMap;

/** questa classe implementa la singola Base Extension*/
public class BaseExtension
{

//
// PROPERTIES
//
```

```
/** numero di riferimento della base extension; deve essere
 *  unico all'interno di una stessa <i>ExtensionalHierarchy</i>
 */
 public int baseExtNumber;

/** vettore di stringhe contenenti i nomi degli attributi */
 public Vector attributes;

/** vettore di SourceClass: classi locali presenti
 *  nella base extension
 */
 public Vector localClasses;

/** vettore di JoinMap, praticamente matrice di SourceClass */
 public Vector joinMap;

 //
 // CONSTRUCTORS
 //

/** inizializza una classe vuota
 *  @param num e' il numero di riferimento
 *         della base extension
 */
 public BaseExtension(int num)
  {
   baseExtNumber = num;
   attributes = new Vector(0);
   localClasses = new Vector(0);
   joinMap = new Vector(0);
  }

/** genera un'istanza della classe inizializzandone i campi;
 *  se uno dei vettori rappresentanti gli attributi o le
 *  classi locali dovesse essere un campo null, viene creato
 *  un vettore vuoto
 *  @param num e' il numero di riferimento della
 *         base extension
 *  @param a e' un vettore di stringhe contenenti i nomi
 *         degli attributi
 *  @param lc e' un vettore i cui elementi sono oggetti
```

```
 *         istanza di <i>SourceClass</i> che rappresentano
 *         le classi locali della base extension
 */
 public BaseExtension(int num, Vector a, Vector lc)
  {
   baseExtNumber = num;
   if (a == null) attributes = new Vector(0);
   else attributes = a;
   if (lc == null) localClasses = new Vector(0);
   else localClasses = lc;
   joinMap = new Vector(0);
  }

// inizializza una classe completamente vuota
 public BaseExtension()
  {
   baseExtNumber = 0;
   attributes = new Vector(0);
   localClasses = new Vector(0);
   joinMap = new Vector(0);
  }

//
// METHODS
//

// NOTA: se si "scommentano" i println si ottengono dei
//       commenti di verifica su standard output

/** restituisce l'insieme delle classi locali relative
 *  alla Base Extension specificata
 */
 public Vector getLocalClasses()
  {
   return localClasses;
  }

/** Questo metodo fornisce l'oggetto di tipo JoinMap
 *  riguardante classi passate come parametri; se queste
 *  classi non sono joineable in le due questa base
 *  extension, dovrebbe essere ritornato un valore null
```

```
 *  @param class1 prima classe sorgente da controllare
 *  @param class2 seconda classe sorgente da controllare
 *  @return restituisce un oggetto di tipo JoinMap se le
 *          due classi sono joineable in questa base
 *          extension, se no restituisce null
 */
 public JoinMap getJoinMap(SourceClass class1,
                           SourceClass class2)
  {
   int i;
   JoinMap jM, returnedMap=null;
   for (i=0;i<this.joinMap.size();i++)
    {
     jM = (JoinMap)this.joinMap.get(i);
     if ((jM.firstElement == class1)
       && (jM.secondElement == class2))
      {
       returnedMap = jM;
       break;
      }
     else
      if ((jM.firstElement == class2)
        && (jM.secondElement == class1))
       {
        returnedMap = jM;
        break;
       }
    }
   return returnedMap;
  }

/** Questo metodo fornisce l'insieme di classi da
 *  interrogare ottimizzato rispetto ad una determinata
 *  <i>BaseExtension</i> ed una <i>BasicQuery</i> i cui attributi
 *  sono passati come parametro
 *  @param qAttr e' un vettore contenente gli attributi
 *         presenti nella query
 *  @return restituisce un vettore di oggetti di tipo
 *          <i>SourceClass</i> che rappresenta l'insieme
 *          ottimizzato di classi da interrogare
 */
```

```
public Vector classOptimization(Vector qAttr)
 {
  int i,j;
  boolean joins,classJoin[],toBeKept[], check[];
  SourceClass localClass,otherClass = null;
  Vector otherClasses,returnedClasses, classesToAdd;
  JoinMap jMap;
  toBeKept = new boolean[localClasses.size()];
  check = new boolean[localClasses.size()];
  returnedClasses = new Vector(0);
  classesToAdd = new Vector(0);
 //System.out.println("Base Extension: "
 // + this.baseExtNumber);
  for (i=0;i<localClasses.size();i++)
   {toBeKept[i] = true; check[i] = true;}
  for (i=0;i<localClasses.size();i++)
   {
    localClass = (SourceClass)localClasses.get(i);
    if (thereIsNoOne(qAttr,localClass.globalAttributes))
    toBeKept[i] = false;
    else
     {
      otherClasses = new Vector(this.localClasses);
      otherClasses.remove(i);
     // si guarda se una classe e' sopraclasse di un'altra
     // di questa base extension
      for (j=0;j<otherClasses.size();j++)
       {
        otherClass = (SourceClass)otherClasses.get(j);
       //System.out.println(localClass.name +
       // " superclasse di " + otherClass.name + ": "
       // + localClass.superClassOf(otherClass));
        if (localClass.superClassOf(otherClass))
         { // se e' sopraclasse e' da scartare se:
           // la sorgente e' Object
          if (localClass.source.type.equalsIgnoreCase("Object"))
           toBeKept[i] = false;
           // oppure la sottoclasse ha tutti gli attributi
           // che interessano o comunque almeno tutti
           // quelli della sopraclasse
          else
```

```
            if (otherClass.globalAttributes.containsAll(qAttr)
             || thereAreMore(localClass.globalAttributes,
                             otherClass.globalAttributes,qAttr))
              toBeKept[i] = false;
           }
         }
       // si guarda se una classe ha delle classi equivalenti
        for (j=0;j<otherClasses.size();j++)
         {
          otherClass = (SourceClass)otherClasses.get(j);
         //System.out.println(localClass.name +
         // " equivalente a " + otherClass.name + ": "
         // + localClass.equivalentTo(otherClass));
         //System.out.println("and check: " + check[i]);
          if (localClass.equivalentTo(otherClass) && check[i])
           { // la classe in esame e' da scartare se
             // l'equivalente ha tutti gli attributi presenti
             // nella query o se ha almeno tutti quelli che
             // ha anche quella in esame
            if (otherClass.globalAttributes.containsAll(qAttr)
             || thereAreMore(localClass.globalAttributes,
                          otherClass.globalAttributes, qAttr))
             {
              toBeKept[i] = false;
              check[(localClasses.indexOf(otherClass))]=false;
             // non dobbiamo piu' controllare l'altra
             // equivalente a questa
             }
           }
         }
       }
     //System.out.println(localClass.name + " da tenere: "
     // + toBeKept[i]);
      if (toBeKept[i]) returnedClasses.add(localClass);
     }
   // controlliamo che con le classi ottenute sia
   // effettivamente possibile ricostruire la base extension
   // in questione, se no aggiungiamo alle classi da
   // interrogare quelle necessarie
    if (returnedClasses.size() > 1)
     for (i=0;i<returnedClasses.size();i++)
```

```
      {
       localClass = (SourceClass)returnedClasses.get(i);
       otherClasses = new Vector(returnedClasses);
       otherClasses.remove(i);
       if (!joinClass(localClass,otherClasses))
      // se una classe non e' joineable bisogna prendere
      //  un'altra classe
       if (! classesToAdd.contains(findJoin(localClass,
                                            returnedClasses)))
        classesToAdd.add(findJoin(localClass,
                                            returnedClasses));
      }
    if (classesToAdd.capacity() > 0)
     returnedClasses.addAll(classesToAdd);
    return returnedClasses;
   }

 /** Questo metodo indica se la base extension in questione
  *  domina o meno quella passata come parametro
  *  @param dominated oggetto di tipo base extension per il
  *         quale si deve verificare la dominanza o meno
  *  @return restituisce true se la base extension passata
  *          come parametro e' dominata da quella in
  *          questione, se no restituisce false
  */
  public boolean dominates(BaseExtension dominated)
   {
    if (this.localClasses.size() <
        dominated.localClasses.size())
     { // se la base extension passata ha un'estensione
       // maggiore: si procede
      boolean classFound[];
      int j,k;
      classFound = new boolean[this.localClasses.size()];
      for (j=0;j<this.localClasses.size();j++)
         classFound[j] = false;
      for (j=0;j<this.localClasses.size();j++)
       for (k=0;k<dominated.localClasses.size();k++)
         if ((SourceClass)this.localClasses.get(j) ==
             (SourceClass)dominated.localClasses.get(k))
           {
```

```
            classFound[j] = true;
             // questa classe locale c'e' in tutte e 2
            break;
           }
        boolean answer = true;
      for (j=0;j<this.localClasses.size();j++)
        if (!classFound[j])
          { // se manca una classe la base extension piu'
            // grande non e' comunque dominata
            answer = false;
            break;
          }
      return answer;
     }
    // altrimenti non e' sicuramente dominata
    else return false;
   }

 /** Questo metodo fornisce un oggetto di tipo SourceClass
  *  contenente una classe locale appartenente all'insieme
  *  di classi della base extension in questione meno
  *  l'insieme di classi passato come secondo parametro
  *  e joineable con quella passata come primo
  *  @param classToJoin classe alla quale trovare un'altra
  *         joineable
  *  @param chosenClasses insieme di classi da sottrarre
  *         da quelle della base extension
  *  @return restituisce un oggetto di tipo SourceClass
  *          contenente una classe joineable con quella
  *          passata come parametro
  */
  SourceClass findJoin(SourceClass classToJoin,
                       Vector chosenClasses)
   {
    int i;
    SourceClass thisClass,joinWithThis = null;
    JoinMap jM;
    Vector notChosenClasses = new Vector(0);
    notChosenClasses = this.localClasses;
    notChosenClasses.removeAll(chosenClasses);
    for (i=0;i<notChosenClasses.size();i++)
```

```
    {
     thisClass = (SourceClass)notChosenClasses.get(i);
     jM = (JoinMap)getJoinMap(classToJoin,thisClass);
     if (jM.joineable)
      {
       joinWithThis = thisClass;
       break;
      }
    }
   return joinWithThis;
  }

/** Questo metodo indica se sono presenti nel secondo
 *  vettore passato tutti gli oggetti contenuti nel primo
 *  limitatamente ai loro sottoinsiemi determinati
 *  dall'intersezione con il terzo vettore
 *  @param toFind e' un vettore contenente gli attributi
 *         da cercare nel secondo vettore passato
 *  @param inThere e' un vettore contenente un insieme
 *         di attributi nel quale cercare quelli del
 *         primo vettore passato
 *  @param subSet e' un vettore che limita gli inisiemi
 *         precedenti
 *  @return restituisce true se gli attributi cercati sono
 *          TUTTI presenti nel secondo vettore, altrimenti
 *          restituisce false
 */
 boolean thereAreMore(Vector toFind, Vector inThere,
                      Vector subSet)
  {
   toFind.retainAll(subSet);
   inThere.retainAll(subSet);
   if (inThere.containsAll(toFind)) return true;
   else return false;
  }

/** Questo metodo indica se nel secondo vettore passato non
 *  e' presente nessuna stringa contenuta nel primo
 *  @param toFind e' un vettore contenente gli attributi
 *         da cercare nel secondo vettore passato
 *  @param inThere e' un vettore contenente un insieme
```

```
 *         di attributi nel quale cercare quelli del primo
 *         vettore passato
 *  @return restituisce true se nessuno degli attributi
 *          cercati e' presente nel secondo vettore,
 *          altrimenti restituisce false
 */
 boolean thereIsNoOne(Vector toFind, Vector inThere)
  {
   int i,j;
   boolean answer = true, stopAll = false;
   String iT,tF;
   for (i=0;i<inThere.size();i++)
    {
     if (stopAll) break;
     for (j=0;j<toFind.size();j++)
      {
       iT = (String)inThere.get(i);
       tF = (String)toFind.get(j);
       if (iT.equals(tF))
        {
         answer = false;
         stopAll = true;
         break;
        }
      }
    }
   return answer;
  }

/** Questo metodo indica se nel secondo vettore passato
 *  e' presente almeno un oggetto contenuto nel primo
 *  @param toFind e' un vettore contenente gli oggetti da
 *         cercare nel secondo vettore passato
 *  @param inThere e' un vettore contenente un insieme
 *         di oggetti nel quale cercare quelli del primo
 *         vettore passato
 *  @return restituisce true se almeno uno degli attributi
 *          cercati e' presente nel secondo vettore,
 *          altrimenti restituisce false
 */
 boolean thereIsOne(Vector toFind,Vector inThere)
```

```
  {
   int i;
   Object obj = new Object();
   boolean answer = false;
   for(i=0;i<toFind.size();i++)
    {
     obj = (Object)toFind.get(i);
     if (inThere.contains(obj))
      {
       answer = true;
       break;
      }
    }
   return answer;
  }

/** Questo metodo indica se la classe passata come
 *  primo parametro e' joineable con almeno una tra
 *  quelle presenti nel secondo parametro attraverso
 *  almeno un attributo tra quelli del terzo parametro
 *  @param thisClass e' la SourceClass in questione
 *  @param classes e' un vettore contenente un insieme
 *         di oggetti di tipo SourceClass
 *  @param qAttr e' un vettore contenente un insieme di
 *         stringhe rappresentanti gli attributi della query
 *  @return restituisce true se la classe in questione
 *          e' joineable con almeno una di quelle passate come
 *          secondo parametro attraverso almeno un attributo
 *          passato come terzo, altrimenti restituisce false
 */
 boolean joinQuery (SourceClass thisClass,Vector classes,
                    Vector qAttr)
  {
   int j;
   SourceClass otherClass;
   boolean classJoin[],joins;
   JoinMap jMap;
   classJoin = new boolean[classes.size()];
   for (j=0;j<classes.size();j++) classJoin[j] = false;
   for (j=0;j<classes.size();j++)
    {
```

```
      otherClass = (SourceClass)classes.get(j);
      jMap = getJoinMap(thisClass,otherClass);
      if (thereIsOne(jMap.joinAttr,qAttr))
       { // e' joineable attraverso attributi presenti
         // nella query
        classJoin[j] = true;
        break;
       }
     }
    joins = false;
    for (j=0;j<classes.size();j++)
     if (classJoin[j])
      { // questa classe e' joineable con almeno un'altra
        // tra quelle scelte
       joins = true;
       break;
      }
    return joins;
   }

 /** Questo metodo indica se la classe passata come
  *  primo parametro e' joineable con almeno una tra quelle
  *  presenti nel secondo parametro
  *  @param thisClass e' la SourceClass in questione
  *  @param classes e' un vettore contenente un insieme di
  *         oggetti di tipo SourceClass
  *  @return restituisce true se la classe in questione e'
  *          joineable con almeno una di quelle passate come
  *          secondo parametro, altrimenti restituisce false
  */
  boolean joinClass (SourceClass thisClass,Vector classes)
   {
    int j;
    JoinMap jMap;
    SourceClass otherClass;
    boolean classJoin[],joins;
    classJoin = new boolean[classes.size()];
    for (j=0;j<classes.size();j++) classJoin[j] = false;
    for (j=0;j<classes.size();j++)
     {
      otherClass = (SourceClass)classes.get(j);
```

```
      jMap = getJoinMap(thisClass,otherClass);
      if (jMap.joineable)
       {
        classJoin[j] = true;
        break;
       }
     }
    joins = false;
    for (j=0;j<classes.size();j++)
     if (classJoin[j])
      { // questa classe e' joineable con almeno
        // un'altra tra quelle scelte
       joins = true;
       break;
      }
    return joins;
   }
 }
```

C.2 La documentazione

Come si può notare dal listato riportato nella sezione precedente, numerosi commenti sono stati redatti rispettando un preciso formalismo. Questo permette, utilizzando il comando *javadoc*, di ottenere una significativa documentazione in formato HTML.
Più precisamente il comando che permette di produrre la documentazione JavaDoc relativa a tutte le classi è:

```
javadoc -d doc ‘find . -name ’*.java’‘
```

Nella subdirectory *doc* sono adesso contenuti tutti i documenti HTML relativi al software del Query Manager.

In questa sezione viene presentato un esempio di documentazione, sempre relativo alla classe BaseExtension.java; come si potrà notare sono presenti solo i metodi definiti *public*, se si volesse invece la descrizione di tutti bisognerebbe aggiungere l'opzione `-private` nel comando precedentemente riportato.

Appendice D

Grammatica XML-QL

Viene descritta in questa parte la grammatica di XML-QL utilizzando la notazione BNF.
Si adotta inoltre la seguente convenzione: i simboli terminali sono rappresentati tra parentesi angolari, mentre la loro struttura lessicale non viene specificata ulteriormente.
Da notare infine che essendo il linguaggio in evoluzione, la grammatica potrebbe subire dei mutamenti.

```
XML-QL       : : =  (Function | Query) ⟨EOF⟩
Function     : : =  'FUNCTION' ⟨FUN-ID⟩
                    '(' (⟨VAR⟩(':' ⟨DTD⟩)?)*')'(':' ⟨DTD⟩)?
                    '{' Query '}'
Query        : : =  Element | Literal | ⟨VAR⟩ | QueryBlock
Element      : : =  StartTag Query EndTag
StartTag     : : =  '⟨'(⟨ID⟩|⟨VAR⟩) SkolemID? Attribute* '⟩'
SkolemID     : : =  ⟨ID⟩ '(' ⟨VAR⟩ (',' ⟨VAR⟩)* ')'
Attribute    : : =  ⟨ID⟩ '=' ('"' ⟨STRING⟩ '"' | ⟨VAR⟩ )
EndTag       : : =  '⟨' / ⟨ID⟩? '⟩'
Literal      : : =  ⟨STRING⟩
QueryBlock   : : =  Where Construct ('{' QueryBlock '}')*
Where        : : =  'WHERE' Condition (',' Condition)*
Construct    : : =  OrderedBy? 'CONSTRUCT' Query
Condition    : : =  Pattern BindingAs* 'IN' DataSource | Predicate
Pattern      : : =  StartTagPattern Pattern* EndTag
StartTagPattern : = '⟨' RegExpr Attribute* '⟩'
RegExpr      : : =  RegExpr '*' | RegExpr '+' | RegExpr '.' RegExpr |
                    RegExpr '|' RegExpr |
                    ⟨VAR⟩ ('['⟨INDEX-VAR⟩']')? |
                    ⟨ID⟩('['⟨INDEX-VAR⟩']')?
BindingAs    : : =  'ELEMENT_AS' ⟨VAR⟩ | 'CONTENT_AS' ⟨VAR⟩
Predicate    : : =  Expression OpRel Expression
Expression   : : =  ⟨VAR⟩ | ⟨CONSTANT⟩
OpRel        : : =  '<'|'<='|'>'|'>='|'='|'! ='
OrderBy      : : =  'ORDER-BY' ⟨VAR⟩+ ('DESCENDING')?
DataSource   : : =  ⟨VAR⟩ | ⟨URI⟩ | ⟨FUN-ID⟩
                    '('DataSource (',' DataSource)*')'
```

Bibliografia

[1] Gio Wiederhold et al. *Integrating Artificial Intelligence and Database Technology*, volume 2/3. Journal of Intelligent Information Systems, June 1996.

[2] Daniel P.Miranker and Vasilis Samoladas. Alamo: an architecture for integrating heterogenous data sources. In *Proceedings of the 4th KRDB Workshop*, Athens, Greece, August 1997.

[3] Oliver M.Duschka and Micheal R.Genesereth. Infomaster - an information integration toolkit. Technical report, Department of Computer Science. Stanford University, 1996.

[4] V.S. Subrahmanian, Sibel Adali, Anne Brink, J.ames, J. Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, and Charles Ward. Hermes: A heterogeneous reasoning and mediator system. Available at http://www.cs.umd.edu/projects/hermes/overview/paper/index.html.

[5] Alon Levy, Dana Florescu, Jaewoo Kang, Anand Rajaraman, and Joanne J. Ordille. The information manifold project.

[6] H. Garcia-Molina et al. The tsimmis approach to mediation: Data models and languages. In *NGITS workshop*, 1995.

[7] Y.Papakonstantinou H.Garcia-Molina and J.Ullman. Medmaker: a mediation system based on declarative specification. Technical report, Stanford University, 1995.

[8] H. Garcia-Molina Y.Papakonstantinou. Object fusion in mediator systems (extended version). 1997.

[9] M.J.Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. Object exchange across heterogeneous information sources. Technical report, Stanford University, 1994.

[10] M.T. Roth and P. Scharz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. of the 23rd Int. Conf. on Very Large Databases*, Athens, Greece, March 1995.

[11] Y. Arens, C.Y. Chee, C. Hsu, and C. A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[12] Y. Arens, C. A. Knoblock, and C. Hsu. Query processing in the sims information mediator. *Advanced Planning Technology*, 1996.

[13] Andrea Zaccaria. MOMIS: Il componente Query Manager. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[14] Alberta Rabitti. Architettura di un mediatore per un sistema di integrazione di sorgenti distribuite ed autonome. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[15] Alberto Zanoli. Si-designer, un tool di ausilio all'integrazione di sorgenti di dati eterogenee distribuite: progetto e realizzazione. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[16] Simone Montanari. Un approccio intelligente all' integrazione di sorgenti eterogenee di informazione. Tesi di laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1996-1997.

[17] Gianni Pio Grifa. Analisi di affinità strutturali fra classi ODL_{I^3} nel sistema MOMIS. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[18] Maurizio Vincini. Utilizzo di tecniche di intelligenza artificiale nell'integrazione di sorgenti informative eterogenee. Tesi di dottorato, Università di Modena, Facoltà di Ingegneria, dottorato di ricerca in Ingegneria Informatica, 1997-1998.

[19] S.Bergamaschi, S.Castano, S.De Capitani di Vimercati, S.Montanari, and M.Vincini. An intelligent approach to information integration. *Accepted for: Formal Ontology in Information Systems FOIS98*.

[20] S.Bergamaschi, S.Castano, S.De Capitani di Vimercati, S.Montanari, and M.Vincini. Exploiting schema knowledge for the integration of heterogeneous sources. *Accepted for: Sistemi Evoluti per Basi di Dati, SEBD98*.

[21] Arpa i^3 reference architecture. Available at http://dc.isx.com/I3/.

[22] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

[23] N.Guarino. Semantic matching: Formal ontological distinctions for information organization, extraction, and integration. Technical report, Summer School on Information Extraction, Frascati, Italy, July 1997.

[24] N.Guarino. Understanding, building, and using ontologies. A commentary to 'Using Explicit Ontologies in KBS Development', by van Heijst, Schreiber, and Wielinga.

[25] E.Rodriguez F.Saltor. On intelligent access to heterogeneous information. In *Proceedings of the 4th KRDB Workshop*, Athens, Greece, August 1997.

[26] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. Technical report, Bell Laboratories, 1996.

[27] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. Odb-qOptimizer: a tool for semantic query optimization in oodb. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, April 1997.

[28] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. a description logics based tool for schema validation and semanti query optimization in object oriented databases. Technical report, sesto convegno AIIA, 1997.

[29] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. Odb-tools: a description logics based tool for schema validation and semantic query optimization in object oriented databases. In *Proc. of Int. Conference of the Italian Association for Artificial Intelligence (AI*IA97)*, Rome, 1997.

[30] A.G. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[31] S. Castano and V. De Antonellis. Deriving global conceptual views from multiple information sources. In *preProc. of ER'97 Preconference*

Symposium on Conceptual Modeling, Historical Perspectives and Future Directions, 1997.

[32] M.R. Cohen and E. Nagel. An itroduction to logic. Indianapolis, Indiana: Hackett Publishing Company, 1993.

[33] S. Castano and V. De Antonellis. Semantic dictionary design for database interoperability. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, April 1997.

[34] I. Schmitt and G. Saake. Merging inheritance hierarchies for database integration. *IEEE Trans. on Knowledge and Data Engineering*.

[35] C. Carpineto and G. Romano. Galois: An order-theoretic approach to conceptual clustering. In *Macine Learning Conference*, pages 33–40, 1993.

[36] Byacc/java. Available at http://www.lincom-asg.com/ rjamison/byacc/.

[37] A. Cerfogli MOMIS: stato di lavori. Available in the directory /export/home/progetti.comuni/Momis/documenti/ of the sparc20.dsi.unimo.it Server.

[38] S. De Capitani di Vimercati S. Bergamaschi, S. Castano and M. Vincini. Momis: An intelligent system for the integration of semistructured data, November 1998.

[39] S.Bergamaschi, S.Castano, S.De Capitani di Vimercati, S.Montanari, and M.Vincini. An intelligent system for the integration of semistructured and strucurer data. *Submitted for: Information Systems. special Issue on Semistructured Data*.

[40] Norman Walsh. What is XML-QL? Available at http://www.xml.com/pub/98/10/guide0.html

[41] Jon Bosak. XML, Java, and the future of the Web. Available at http://www.xml.com/xml/pub/r/XML,_Java,_and_the_future_of_the_Web/

[42] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. A Query Language for XML. Available at http://www.w3.org/TR/NOTE-xml-ql/

[43] A. Layman, E. Jung, E. Maler, H. S. Thompson, J. Paoli, J. Tigue, N. H. Mikula, S. De Rose. XML-Data. Available at http://www.w3.org/TR/1998/NOTE-XML-data/

[44] Introduction to CORBA Available at http://developer.java.sun.com/developer/onlineTraining/corba/corba.html