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

# Il componente Query Manager di MOMIS: esecuzione di interrogazioni

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Silvia Zanni

Correlatore
Dott. Ing. Domenico Beneventano

Controrelatore
Chiar.mo Prof. Michele Colajanni

Anno Accademico 1999 - 2000

Parole chiave:

Query Execution
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 gruppo MOMIS, in particolare l'Ing. Alberto Corni e l'Ing. Domenico Beneventano per i consigli di ordine pratico e teorico.*

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Introduzione

Lo sviluppo delle tecnologie nell'ambito delle telecomunicazioni, delle reti di calcolatori e dei sistemi di elaborazione, ha portato ad una crescita esponenziale del numero e della varietá di sorgenti di informazioni disponibili e di conseguenza del quantitativo di dati reperibili.
Tale eterogeneitá di sistemi si manifesta a diversi livelli: differenti piattaforme hardware e software su cui la sorgente é implementata (diversi DBMS e linguaggi di interrogazione, . . . ), differenti modelli di dati (relazionale, object-oriented, . . . ), differenti schemi usati per la rappresentazione logica dei dati memorizzati.
In un contesto di questo tipo, per poter gestire correttamente tutte le informazioni disponibili, l'utente dovrebbe possedere una conoscenza specifica sia delle strutture che dei linguaggi di interrogazione delle singole sorgenti. Inoltre, a causa del fenomeno dell'*information overload*, risulta difficile all'utente gestire l'inevitabile duplicazione e ridondanza di contenuti al fine di ottenere un'informazione significativa.
É quindi diventata indispensabile la progettazione di meccanismi in grado di automatizzare il processo di interrogazione e reperimento delle informazioni, permettendo all'utente di porre la propria richiesta senza avere un'effettiva conoscenza della natura e organizzazione delle sorgenti.

Questa tesi si inserisce in un progetto più ampio denominato **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources) [8, 10, 11, 12, 13, 14, 15, 16], sviluppato con l'obiettivo di realizzare l'integrazione semi-automatica delle informazioni contenute all'interno di sorgenti eterogenee e distribuite.
**MOMIS** adotta un'architettura a tre livelli con un *Mediatore* che si occupa della creazione della vista aggregata degli schemi costitutivi le singole sorgenti, e della gestione delle query poste dall'utente sullo schema globale.
Gli elementi innovativi introdotti in questo progetto sono rappresentati dall'impiego di un approccio semantico e dall'uso di logighe descrittive per la rappresentazione degli schemi locali, elementi che introducono comportamenti intelligenti in grado di rendere semi-automatico il processo di integrazione.

L'obittivo della presente tesi é stato l'ampliamento del modulo Query Manager del *Mediatore* di Momis, estendendo il progetto presentato in [8, 9, 10]. Piú precisamente é stata introdotta la gestione della clausola where di una generica query posta dall'utente ed é stata implementa la fase di ricomposizione dei risultati restituiti dalle sorgenti locali, rendendo possibile la generazione di una risposta coretta e, quanto piú possibile, completa e minima.

La tesi é organizzata nel modo seguente:

Nel **Capitolo 1** si introduce il concetto di Integrazione Intelligente di Informazioni, descrivendo l'architettura di riferimento $I_3$ e la struttura di un *Mediatore*. Si elencheranno le scelte implementative adottate per il sistema MOMIS soffermandosi sulla sua architettura.

Nel **Capitolo 2** viene descritto il processo di integrazione degli schemi e le strutture rappresentanti la conoscenza intensionale ed estensionale utilizzate dal Query Manager per la fase di *Query Processing*.

Nel **Capitolo 3** é introdotto il modulo Query Manager di MOMIS, in particolare sono descritti i passi da esso compiuti per l'elaborazione delle query poste dall'utente. All'interno di questo capitolo sono approfonditi gli argomenti trattati da questa tesi, riguardanti la gestione della clausola where e la fase di ricomposizione dei risultati restituiti dalle sorgenti.

Il **Capitolo 4** descrive la progettazione e realizzazione del software del modulo Query Manager. In particolare vengono esaminate le classi ed i metodi introdotti per l'implementazione delle fasi descritte nel capitolo precedente.

Sono inoltre presenti cinque appendici: in Appendice A viene riportato un glossario dei termini usati in ambito $I_3$, in Appendice B l'esempio in $ODL_{I^3}$ che sará utilizzato come riferimento, nelle Appendici C e D vengono mostrate rispettivamente le rappresentazioni BNF della grammatica OQL e della versione ristretta per le Basic Query ed infine in Appendice E é riportato il codice relativo alla classe java *BasicQuery*, la cui implementazione é fondamentale per la fase di Query Processing.

# Capitolo 1

# Un sistema intelligente di integrazione

Le problematiche connesse all'integrazione di informazioni sono legate alla grande eterogeneità dei dati disponibili, sia per quanto riguarda la natura (testi, immagini, etc.), sia il modo in cui vengono descritti nelle diverse sorgenti. 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.
Problematiche aggiuntive sono dovute al cosiddetto sovraccarico di informazioni, o *information overload*, infatti la enorme mole di dati disponibili impedisce all'utente di discernere e isolare le informazioni significative.
Altri problemi sono dovuti ai tempi di accesso, alla salvaguardia della sicurezza e agli elevati costi per il mantenimento e la consistenza delle informazioni.

Per far fronte alla molteplicitá e complessitá degli aspetti appena descritti, le architetture dedicate all'integrazione di sorgenti eterogenee devono essere necessariamente flessibili e modulari.
Gli approcci all'integrazione presentano diverse metodologie: la *rengegnerizzazione* delle sorgenti mediante standardizzazione degli schemi e la creazione di un database distribuito; i *datawarehouse* che realizzano delle viste presso l'utente finale, replicando fisicamente i dati e utilizzando algoritmi di allineamento per assicurarne la consistenza a fronte di modifiche nelle sorgenti.
Nel seguito verrá descritto un tipo di approccio differente, che non ricorre alla duplicazione fisica dei dati, quello che in letteratura viene indicato come Integrazione di Informazioni ($I^2$) [1].

## 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*, $I^3$).

### 1.1.1 Il programma $I^3$

Dal 1992 è operativo il Programma $I^3$, un progetto di ricerca fondato e sponsorizzato dall'ARPA (Advanced Research Projects Agency), che si prefigge di individuare un'architettura di riferimento che realizzi in maniera automatica l'integrazione di sorgenti di dati eterogenee [2].
$I^3$ 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, inoltre la riutilizzazione della tecnologia già sviluppata, rende la costruzione di nuovi sistemi 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. Questo livello deve offrire servizi dinamici quali la selezione delle sorgenti, la gestione degli accessi e delle interrogazioni, l'analisi e sintesi dei dati.

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.

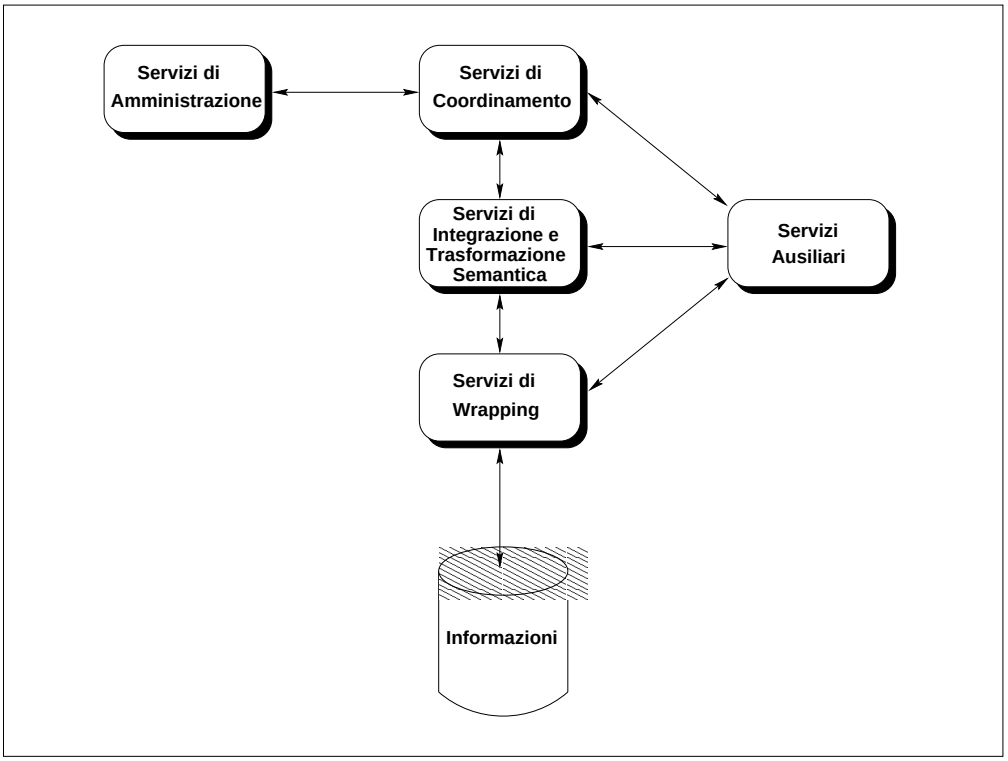


Figura 1.1: Diagramma dei servizi $I^3$

### 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, fornendo una raccolta e una formalizzazione delle soluzioni prevalenti finora nel campo della ricerca. Come abbiamo visto, la complessitá del processo di integrazione é tale da rendere estremamente utile la proposta di un'architettura di riferimento standard, che rappresenti alcuni dei servizi che un integratore di informazioni deve contenere e le possibili interconnessioni fra di essi. Il programma $I^3$ individua cinque famiglie di attività omogenee, illustrate in Figura 1.1 unitamente ai loro legami. La reciproca interazione tra queste attività consente di eseguire le operazioni di comunicazione, traduzione ed integrazione dei dati nelle sorgenti.
Analizziamo nel dettaglio questi servizi:

I **Servizi di Coordinamento** sono servizi ad alto livello che costituiscono l'interfaccia con l'utente, dandogli l'impressione di trattare con un sistema omogeneo. Grazie alle funzionalitá messe a disposizione dalle altre famiglie, essi per-

mettono l'individuazione delle sorgenti di dati *interessanti*, ovvero delle sorgenti che possono fornire una risposta ad una determinata richiesta dell'utente.
I moduli di coordinamento possono essere piú o meno complessi in funzione delle modalitá di selezione delle sorgenti:

- *Facilitator e Brokering Services*: forniscono una selezione dinamica delle sorgenti in grado di soddisfare la richiesta dell'utente. Il sistema usa un deposito di metadadati per individuare il modulo che puó trattare direttamente questa richiesta, in particolare si parla di Brokering quando é coinvolto un modulo alla volta, oppure di Facilitatori o Mediatori se vi sono piú moduli interessati. In quest'ultimo caso la query iniziale viene decomposta in un insieme di sottoquery da inviare a differenti moduli che gestiscono sorgenti distinte, successivamente vengono integrate le risposte per fornirne una prentazione globale all'utente.

- *Matchmaking*: il mappaggio fra informazioni integrate e locali é effettuato manualmente da un'operatore in fase di inizializzazione. In questo caso tutte le richieste vengono trattate allo stesso modo.

I **Servizi di Amministrazione** sono utilizzati da quelli di Coordinamento per individuare le sorgenti, determinarne le capacità, creare ed interpretare *template*. I template sono strutture dati che descrivono i servizi, le sorgenti ed i moduli da utilizzare per portare a termine un determinato task; essi 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*, che aiutano l'utente a rilassare o specificare meglio alcuni vincoli dell'interrogazione al fine di ottenere risposte più precise.

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** si comportano come traduttori dai sistemi locali ai servizi di alto livello dell'integratore e viceversa quando si interroga la sorgente dati. Essi realizzano il primo passo verso l'integrazione, rendendo le informazioni provenienti dalle sorgenti omogenee grazie alla loro traduzione in un linguaggio standard. Inoltre, fornendo interfacce che seguono gli standard piú diffusi (ad esempio il linguaggio SQL come linguaggio di interrogazione di basi di dati e CORBA come protocollo di scambio di oggetti), consentono alle sorgenti estratte da questi wrapper "universali" di essere accedute dal maggior numero possibile di moduli mediatori.

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.

In Figura 1.1 risultano evidenti i due assi orizzontale e verticale, che rappresentano le diverse interazioni fra i servizi appena descritti.
Percorrendo l'asse orizzontale, si nota il rapporto fra servizi di Coordinamento ed Amministrazione, mentre sull'asse verticale viene messo in evidenza lo scambio di informazioni all'interno del sistema: i dati estratti dalle sorgenti per mezzo dei servizi di Wrapping sono integrati dai servizi di Integrazione e Trasformazione semantica per poi essere passate ai servizi di Coordinamento che ne avevano fatto richiesta.

Concludendo questa rapida descrizione dell'architettura di riferimento per sistemi $I^3$ è opportuno sottolineare che, essendo la casistica dei campi applicativi interessata molto vasta, le funzionalitá di base descritte non sono esaustive, ogni ambiente di sviluppo individuerá obiettivi e problematiche differenti e di conseguenza anche funzionalitá specifiche.

### 1.1.3 Il mediatore

Secondo la definizione proposta da Wiederhold in [3] "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 nel caso di cambiamento delle 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.

Il progetto MOMIS, di cui questa tesi fa parte, ha come obiettivo la progettazione e realizzazione di un **Mediatore**, come descritto in [11, 12, 13].
L'ipotesi di avere a che fare esclusivamente con sorgenti di dati strutturati e semistrutturati, ha consentito di restringere il campo applicativo del sistema con una conseguente diminuzione delle problematiche riscontrate in fase di progettazione e realizzazione. L'approccio architetturale scelto é quello classico, che si sviluppa su tre livelli principali:

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 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. L'approccio strutturale adottato da sistemi quali TSIMMIS [4, 5, 6], 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

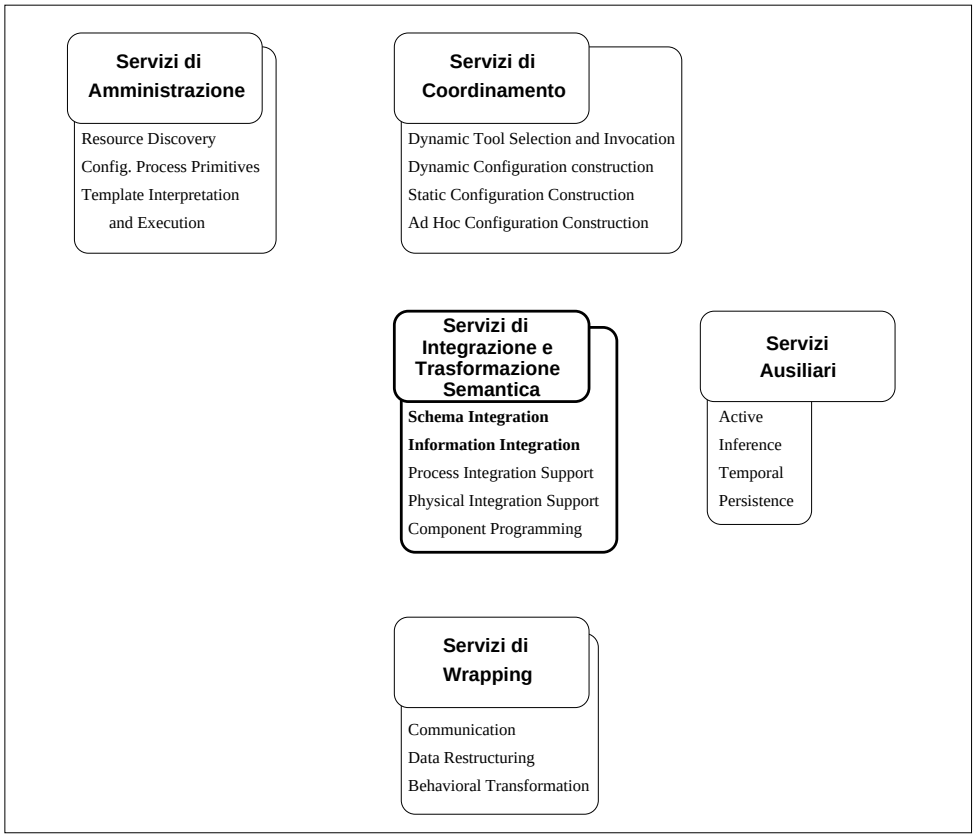


Figura 1.2: Servizi $I^3$ presenti nel mediatore

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 consentono l' integrazione, in modo completamente trasparente al mediatore, di 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. 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.

### 1.1.4 Problematiche da affrontare

Pur avendo a disposizione gli schemi concettuali delle varie sorgenti non é un compito banale realizzarne una coerente integrazione, individuare i concetti comuni e le relazioni che possono legarli. Oltre ai problemi dovuti alle differenze fra i sistemi fisici , di cui si occupano i wrapper, esistono due tipologie di problemi che devono essere affrontati a livello di mediazione:

**Problemi ontologici**
: Come riportato in Appendice A, per *ontologia* si intende, in questo ambito, "l'insieme dei termini e delle relazioni usate in un dominio, per indicare oggetti e concetti". In sostanza con ontologia ci si riferisce a quell'insieme di termini che, in un particolare dominio applicativo, denotano una particolare conoscenza e fra i quali non esiste ambiguitá perché sono condivisi dall'intera comunitá di utenti del dominio applicativo stesso.
Fra i diversi livelli di ontologia esistenti [17, 18], ognuno con le proprie problematiche, 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, é improbabile che usino la stessa semantica, cioé gli stessi vocaboli e le stesse strutture dati per rappresentare questi concetti.
Come riportato in [19] 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 possibile 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 [4, 5, 6, 20, 21, 22, 23, 24, 25, 26, 27], 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 **I**nformation **S**ources). Il contributo innovativo di questo progetto,

rispetto ad altri similari, risiede nell'impiego di un approccio semantico e nell'uso di logiche descrittive per la rappresentazione degli schemi delle sorgenti, elementi che introducono comportamenti intelligenti in grado di rendere semi-automatica la fase di integrazione [11, 12, 13]. Un lavoro approfondito è stato svolto anche riguardo alla fase di *query processing* ([8, 9, 10], 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* [8]. Il concetto di semantico è stato illustrato nella sezione 1.1.3. Con virtuale [28] 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:

1. la presenza di uno schema globale permette all'utente di formulare qualsiasi interrogazione che sia con esso consistente;

2. le informazioni semantiche che comprende possono contribuire ad una eventuale ottimizzazione delle interrogazioni;

3. l'adozione di una semantica *type as a set* per gli schemi permette di controllarne la consistenza facendo riferimento alle loro descrizioni;

4. 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);

Parallelamente a questa impostazione si é deciso di adottare, sia per la rappresentazione degli schemi che per la formulazione delle interrogazioni, un unico modello dei dati basato sul paradigma ad oggetti. 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 $ODL_{I^3}$[10, 11, 12, 13] che si presenta come estensione del linguaggio standard ODL proposto dal gruppo di standardizzazione ODMG-93.

Per quanto riguarda il linguaggio di interrogazione si è adottato $OQL_{I^3}$ che adotta la sintassi OQL senza discostarsi dallo standard. Questo linguaggio risulta estremamente versatile ed espressivo fornendo la possibiltà di sfruttare le informazioni rappresentate nello schema globale.

Inoltre si é cercato di definire uno standard comune di comunicazione tra i vari moduli MOMIS al fine di rendere ancora piú agevole l'ampliamento futuro. Si é deciso di adottare lo standard CORBA (Common ORB Architecture) per le comunicazioni tra i moduli [29]. 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.

### 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 Dati.** Qui si trovano i Wrapper, che costituiscono l'interfaccia fra il Mediatore e le singole sorgenti. La loro funzione é duplice:

- in fase di integrazione, forniscono una descrizione delle informazioni contenute nella sorgente, in formato $ODL_{I^3}$;
- in fase di Query Processing, traducono ogni subquery (espressa in $OQL_{I^3}$) in una interrogazione comprensibile ed eseguibile dalla sorgente. Inoltre forniscono al Mediatore i dati ottenuti come risposta, nel formalismo comune ($ODL_{I^3}$).

Collegate ai wrapper ci sono le sorgenti, per questo spesso si parla di quattro livelli. Esse contengono le informazioni da integrare, possono essere

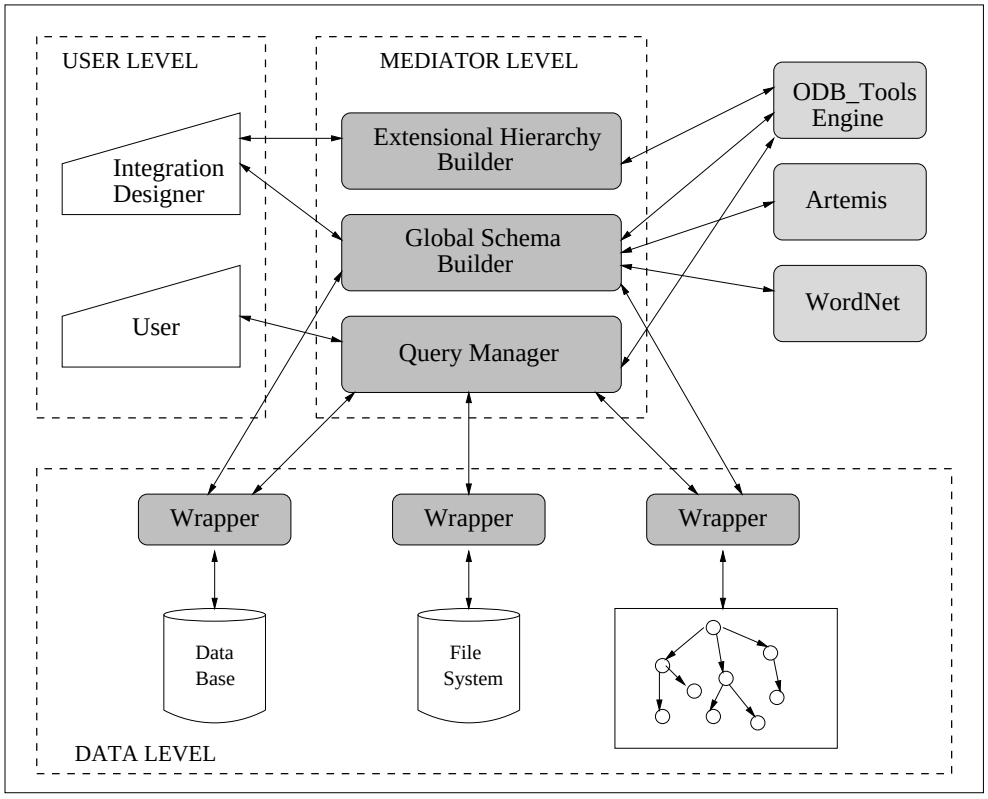


Figura 1.3: Architettura generale di MOMIS

di vario genere: database (reazionale o ad oggetti), file system, sorgenti semistrutturate, . . .

**Livello Mediatore.** Il nucleo centrale del sistema è costituito dal **Mediatore** (o *Mediator*) che presiede all'esecuzione di diverse operazioni.
La sua prima funzionalità 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 $\text{ODL}_{I^3}$ e forniti ognuno dal relativo wrapper. A questo punto (utilizzando 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, . . .

).

Il modulo **Extensional Hierarchy Builder** del Mediatore, si occupa della generazione della Conoscenza Estensionale (Gerarchie Estensionali e Base Extension), come vedremo nel dettaglio in 2.4.

L'altro importante modulo del mediatore è il **Query Manager** che presiede alla fase di Query Processing, cioé alla seconda funzionalitá del Mediatore. 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* (anch'esse espresse in $OQL_{I^3}$) da inviare alle varie sorgenti, o meglio ai wrapper predisposti alla loro traduzione, come abbiamo visto. Questa traduzione avviene in maniera automatica da parte del Query Manager utilizzando la conoscenza intensionale ed estensionale definite nella precedente fase di integrazione. Le operazioni svolte dal Query Manager, essendo argomento di questa tesi, saranno più approfonditamente illustrate nel Capitolo 3, mentre in figura 1.4 ne sono rappresentate le interazioni con gli altri moduli del sistema MOMIS.

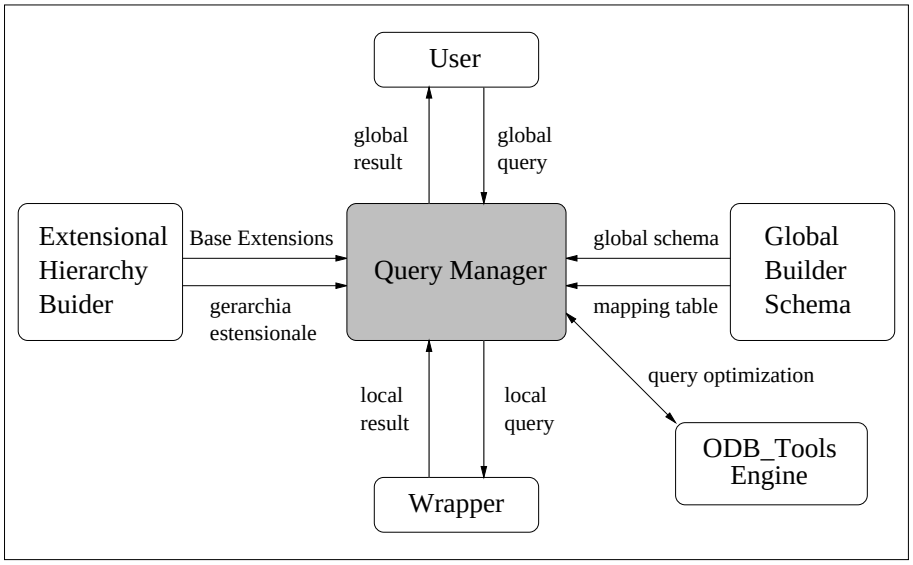


Figura 1.4: Interazioni del Query Manager

**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 combinarle in un'unica risposta corretta, completa e non ridondante.

Riportiamo una breve descrizione dei tool che forniscono un supporto alle attivitá del Mediatore:

- **ODB_Tools** è uno strumento software sviluppato presso il dipartimento di Ingegneria dell'Università di Modena e Reggio Emilia [30, 31, 32]. Esso é utilizzato sia durante la fase d'integrazione degli schemi delle sorgenti per costruire il Thesaurus Comune, sia durante la fase di Query Processing per l'ottimizzazione di query. Attraverso i suoi due componenti ODB_Designer e ODB_QOptimizer si occupa rispettivamente dell'acquisizione e della verifica di consistenza di schemi di basi di dati e dell'ottimizzazione semantica di interrogazioni su basi di dati orientate agli oggetti (OODB).

- *WordNet* [33] 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* [34] 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

# Processo di integrazione degli schemi

MOMIS è un sistema di mediazione sviluppato per assistere l'utente nel reperimento di informazioni su di un insieme di sorgenti eterogenee e distribuite. Il processo semi-automatico di integrazione degli schemi locali porta alla creazione di una visione integrata delle informazioni che consente all'utente di effettuare interrogazioni prescindendo dalla conoscenza della specifica sorgente, mentre il Query Manager automatizza il reperimento e l'integrazione delle informazioni.

Entrambe queste fasi sfruttano un insieme di conoscenze relative alle sovrapposizioni delle estensioni delle classi, alle relazioni intensionali e semantiche degli schemi.

In questo capitolo si intende studiare l'integrazione tra gli schemi.

## 2.1 Conoscenza intensionale ed estensionale

Gli schemi locali vengono integrati in MOMIS secondo criteri sia *intensionali* che *estensionali*.
Schemi locali parzialmente sovrapposti possono rappresentare uno stesso concetto adottando strutture diverse, pertanto il presupposto fondamentale per la realizzazione di un Mediatore consiste nella capacità di individuare e risolvere questi conflitti, che possono essere definiti "$intensionali$", fornendo una rappresentazione unificata ed omogenea dei medesimi concetti descritti nelle varie sorgenti.

L'integrazione degli schemi non è però l'unico aspetto che occorre gestire per ottenere un'effettiva integrazione di sorgenti eterogenee, infatti, come descritto in [10, 39, 37], è necessario risolvere anche i conflitti derivanti dalle sovrapposizioni delle estensioni, cioè dalla presenza, in sorgenti diverse, di informazioni

relative alla stessa entità del "$mondo\ reale$".
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.
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.

## 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 (`e_mail`), 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,e_mail,
            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 databasead 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 quest'ultimo caso non si tratta di un database ma di un file system, che contiene quindi semplici tracciati record.

Il processo di integrazione, sfruttando sia la conoscenza intensionale che quella estensionale, avviene in due fasi distinte:

1. **Unificazione degli schemi** [15]: in questa fase l'uso della logica descrittiva OCDL e di tecniche di *clustering* [36] permette di realizzare un processo semi-automatico di integrazione di schemi, fino a pervenire alla definizione dello Schema Globale, direttamente interrogabile dall'utente.

2. **Fusione delle istanze**: lo scopo di questa fase é la generazione degli oggetti *virtuali* fusi dalla sintesi di quelli *reali* provenienti dalle sorgenti, il progettista può introdurre conoscenza sulle relazioni fra le estensioni di classi simili in sorgenti distinte,[10] per mezzo dell'introduzione dei cosiddetti *assiomi estensionali*.

## 2.3 Integrazione intensionale

Il Global Schema Builder e' il modulo che si occupa dell'integrazione degli schemi locali per la generazione dell'unico schema globale da presentare all'utente. La sua interfaccia grafica, l'SI-Designer (Source Integration Designer), e' il tool di ausilio che supporta il progettista nella fase semi-automatica di integrazione che, partendo dalla descrizione $ODL_{I^3}$ delle sorgenti porta alla creazione dello schema globale attraverso una serie di fasi successive descritte in seguito.

1. *Estrazione di relazioni terminologiche*

2. *Analisi delle affinità intensionali fra classi*

3. *Creazione dei cluster*

In Figura 2.2 sono rappresentate le fasi del processo di integrazione intensionale evidenziando l'intervento del progettista da un lato e l'impiego di ODB_Tools dall'altro.

### 2.3.1 Estrazione di relazioni terminologiche

Durante questa fase viene generato un *Common Thesaurus* di relazioni terminologiche, che vengono derivate in modo semi-automatico dalla descrizione $ODL_{I^3}$ delle sorgenti attraverso l'analisi strutturale e di contesto delle classi coinvolte.
Nel seguito si indicherá genericamente con *termine* $t_i$, il nome di una

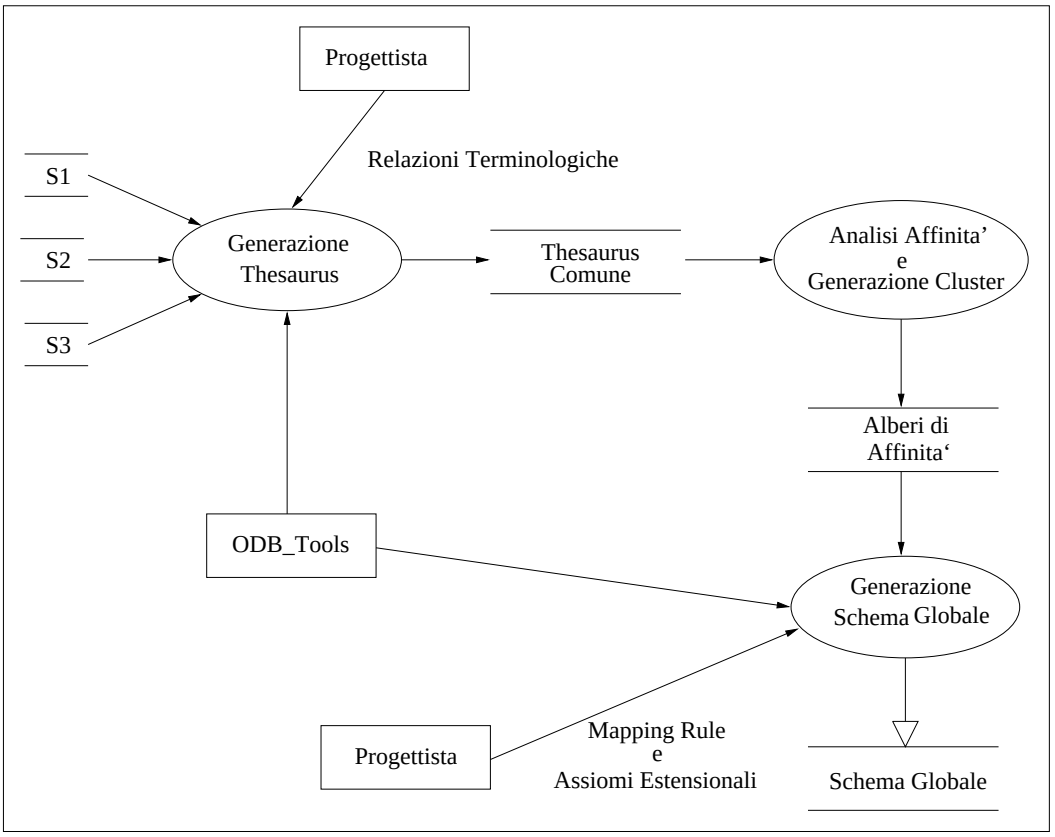


Figura 2.2: Fasi dell' Integrazione Intensionale

classe o di un attributo rispettivamente come *nome_sorgente.nome_classe* e *nome_sorgente.nome_attributo* .

Dati due termini $t_i$ e $t_j$, con $t_i \neq t_j$ le relazioni che si possono definire all'interno del Thesaurus sono le seguenti:

- *Relazioni di sinonimia*:

  $t_i$ **SYN**(synonim-of)$t_j$
  indica che i due termini sono sinonimi, cioé che identificano lo stesso concetto del mondo reale.
  Un esempio di relazione di sinonimia è $\langle$`SU.Section` SYN `SCS.Course`$\rangle$.

- *Relazioni di specializzazione*:

$t_i$ **BT** (broader-term) $t_j$
: indica che $t_i$ ha un significato più ampio, più generale di $t_j$.
Un caso di BT, nel nostro esempio, può essere $\langle$`SU.Research_Staff` BT `SCS.Professor`$\rangle$.

$t_i$ **NT** (narrower-term) $t_j$
: concettualmente è la stessa relazione espressa con una BT, intesa dall'altro punto di vista, dunque $t_i$ BT $t_j \rightarrow t_j$ NT $t_i$.
Lo stesso esempio potrebbe infatti essere $\langle$`SCS.Professor` NT `SU.Research_Staff`$\rangle$.

- *Relazioni di aggregazione*:

$t_i$ **RT** (related-term) $t_j$
: indica due termini che sono generalmente usati nello stesso contesto, tra i quali esiste comunque un legame generico.
Per esempio, possiamo avere la seguente relazione RT $\langle$`SCS.Student` RT `SCS.Course`$\rangle$.

La scoperta di queste relazioni terminologiche avviene con un processo semi-automatico in cui il progettista interagisce con ODB_Tools.

### 2.3.2 Analisi delle affinità intensionali fra classi

In questa fase vengono individuate classi, appartenenti a sorgenti diverse, che descrivono informazioni semanticamente equivalenti. Per questo scopo le classi vengono analizzate e valutate in base al concetto di *affinita'* in modo da individuare il livello di *similaritá*. In particolare, vengono analizzate le relazioni che esistono tra i loro nomi delle classi (attraverso il *Name Affinity Coefficient*) e tra i loro attributi (per mezzo dello *Structural affinity Coefficient*), per arrivare ad un valore globale denominato *Global Affinity Coefficient*.
Questa attività è compiuta con il supporto dell'ambiente di ARTEMIS [7], attraverso un processo interattivo che permette di modificare i parametri di valutazione delle affinitá e validare le scelte fatte.

### 2.3.3 Creazione dei cluster

Le classi vengono raggruppate in gruppi aventi elevato livello di affinitá, con l'utilizzo di tecniche di clustering. La procedura di clustering adottata [12] opera in modo iterativo andando a creare insiemi di classi, *cluster* appunto, di dimensioni via via crescenti. Ad ogni passo viene quindi costruito un nuovo cluster unendo quelli ottenuti al passo precedente ed aventi il valore massimo di affinità. Questo

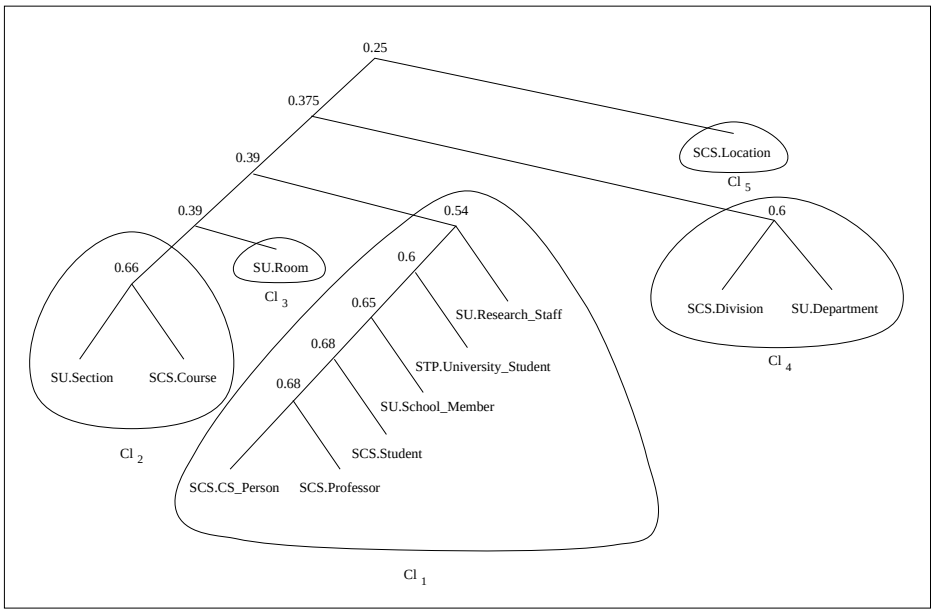


Figura 2.3: Albero di affinità

processo porta alla creazione di un *albero di affinità*, caratterizzato dall'avere come foglie le classi locali ed i cui nodi sono proprio i *cluster* individuati (con valore di affinità calante spostandosi verso la radice).

Un esempio di come le classi vengano organizzate é riportato nell' *Albero di Affinità* in Figura 2.3 in cui l'algoritmo di *Clustering* è stato applicato all' esempio di riferimento.

Dopo aver costruito l'albero di affinitá, il problema è quello di selezionare i cluster più appropriati per la definizione delle classi nello schema globale. La procedura di selezione dei cluster, in ARTEMIS, viene mantenuta interattiva attraverso la modifica del valore di soglia. Il progettista specifica il valore della soglia $T$ ed i cluster caratterizzati da un valore del *Global Affinity Coefficient*, superiore o uguale a $T$ sono selezionati e proposti. Per alti valori di $T$ si ottengono cluster piccoli e molto omogenei tra loro. Diminuendo il valore di $T$, i cluster ottenuti contengono più classi e sono più eterogenei. Nel tool il valore di default di $T$ viene posto pari a $0.5$. Tale valore può essere modificato dinamicamente, sulla base della specifica applicazione in esame.

### 2.3.4 Costruzione dello Schema Globale

In questa fase si definisce lo *Schema Globale* partendo dai cluster creati al passo precedente, piú precisamente si ricava la descrizione in $ODL_{I^3}$ delle *Classi Globali* con le relative regole di *mapping* fra attributi globali e attributi locali.

Per ogni cluster viene creata una classe globale e ne vengono determinati gli attributi globali in maniera automatica, basandosi sulle relazioni contenute nel Thesaurus.

Nell' esempio in corrispondenza del cluster $\texttt{Cl}_1$ viene definita la classe globale `University_Person` :

University_Person = (`University_Person,{ name, rank, title, dept_code, year, takes, relation, email, student_code, tax_fee, section_code, faculty}`)

Dopo questa prima fase l'intervento interattivo del progettista porta alla creazione, per ogni classe globale, della corrispondente *Mapping Table*, cioé quella struttura dati che riporta le corrispondenze fra gli schemi locali e lo schema globale. Queste informazioni servono da un lato all'utente finale per poter utilizzare in modo efficace la vista globale, dall'altro al Query Manager per svolgere in maniera automatica la fase di Query Processing.

I tipi di mapping gestiti fino ad ora sono i seguenti:

- *mapping semplice:* l'attributo globale corrispone ad un attributo locale;
- *null mapping:* l'attributo globale non ha corrispondenza tra gli attributi locali;
- *default mapping:* l'attributo globale ha un valore predefinito;
- *and mapping:* l'attributo globale corrisponde alla concatenazione, in uno specificato ordine, di piú attributi locali;
- *complex mapping:* l'attributo globale mappa su altre classi;
- *union mapping:* descrive la regola di *or mapping* che prevede la sostituzione dell'attributo globale con uno scelto fra i possibili candidati della classe locale.

Il *processo di unificazione degli schemi* applicato all'esempio porta all'individuazione di cinque classi globali:

Global Schema: **University_Schema**

- University_Person (name, pay, faculty, year, tax, rank, email, relation, takes, title, section, studcode)

| 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 2.4: Mapping Table della classe globale University_Person

- Workplace (name, area, budget, employee_nr, address, code)
- Course (name, lecturer, length, site, code)
- Location (city, street, country, number)
- Room (seats_number, notes, code)

In Figura 2.4 è riportata la mapping table relativa alla global class University_Person, in Figura 2.5 quelle delle altre classi globali.

## 2.4 Integrazione estensionale

L'integrazione degli schemi non e' l'unico aspetto da gestire, ma occorre risolvere anche i *conflitti estensionali* che, come descritto in 2.1 sono causati dalla presen-

| 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 2.5: Mapping Table delle altre classi globali

za, in sorgenti diverse, di informazioni relative alla stessa entitá del *mondo reale*. L'approccio teorico adottato in MOMIS [10] si basa sulla teoria della *Formal Context Analysis* [38] 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. Definizione degli *assiomi estensionali*
2. Individuazione delle *base extension*
3. Generazione della *gerarchia estensionale*

Queste tre fasi vengono analizzate nelle sezioni seguenti.

### 2.4.1 Definizione degli assiomi estensionali

Gli assiomi estensionali descrivono le relazioni insiemistiche esistenti tra le estensioni delle sorgenti.

**Definizione 1 (Stato di una classe)** *Lo stato di una classe* **C**[1] *all'istante t, scritto* $Stato_C^t$*, é costituito dall'insieme degli oggetti che popolano la classe* **C** *all'istante t. Lo stato di una classe viene spesso indicato come* l'estensione della classe.

Date due classi (A e B) sono individuabili quattro possibili tipologie di relazioni tra esse:

- *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*: $\forall t : S_A^t \cap S_B^t = \emptyset$

Una parte degli assiomi estensionali vengono ricavati direttamente dalle descrizioni degli schemi, ad esempio una relazione IS_A viene espressa tramite l'assioma di inclusione, ma é essenziale l'intervento del progettista. 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.

Ogni assioma definito *vincola* le classi coinvolte ad avere anche un legame intensionale.
Estendendo la notazione usata per le relazioni intensionali **NT, BT, SYN**, gli assiomi estensionali possono essere definiti in $ODL_{I^3}$come:

- $A \;\; SYN_{Ext} \;\; B$: le istanze della classe A sono le stesse della classe B. Questo implica una relazione intensionale di tipo SYN tra le due classi e due relazioni IS_A;
- $A \;\; NT_{Ext} \;\; B$: le istanze della classe A sono un sottoinsieme di quelle della classe B. Questo assioma viene scomposto in una relazione intensionale di tipo NT e una relazione IS_A;
- $A \;\; BT_{Ext} \;\; B$: le istanze della classe A sono un sovrainsieme di quelle della classe B. Viene generata una relazione intensionale di tipo BT ed una ISA tra le classi;

[1] C puó essere definito anche come un espresione logica che coinvolge piú classi.

- $A\ DISJ_{Ext}$[2] $B$: le istanze della classe A sono diverse da quelle della classe B. Questo assioma non implica nessun tipo di relazione intensionale, esiste solamente un legame di tipo BOTTOM[3] tra le due classi coinvolte.

L'assioma che definisce la *sovrapposizione estensionale* non necessita di notazione in quanto, come giá specificato, viene considerato di default per le classi appartenenti allo stesso cluster.

Gli assiomi estensionali sono epressi tramite rule in $ODL_{I^3}$, come segue:

- 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
  ```
- 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.

Per come sono stati definiti gli assiomi estensionali di tipo $SYN_{Ext}$, $NT_{Ext}$ e $BT_{Ext}$, si intuisce che il legame che generano tra le classi coinvolte é molto forte poiché implica sia un legame tra gli schemi che un legame tra le istanze. Per questo motivo le classi legate da relazioni di questo tipo devono necessariamente appartenere allo stesso cluster. Infatti le relazioni intensionali logicamente implicate dagli assiomi estensionali appena visti vengono registrate nel *Common Thesaurus* e gli viene assegnato un peso uguale a uno in modo da forzare le classi nel medesimo cluster.

La conoscenza estensionale quindi, non viene utilizzata solamente per la costruzione dell'insieme delle *Base Extension* e della *Gerarchia Estensionale* che vedremo in seguito, ma anche per la definizione dei cluster.

### 2.4.2 Individuazione delle base extension

Una base extension è un sottoinsieme dell'insieme complessivo delle estensioni ed identifica oggetti realmente esistenti in una o più sorgenti. Presa quindi una classe

[2]Questa notazione non era presente nella rappesentazione delle relazioni intensionali ma é stata introdotta per poter rappresentare il concetto di *disgiunzione estensionale*.

[3]Nella logica descrittiva un tipo o classe *bottom* rappresenta un concetto incongruente, cioé che non puó essere in nessun caso popolato da dati o istanze.

di entità, un insieme di base extension ne rappresenta il partizionamento in modo che ogni istanza appartenga ad una ed una sola di esse e che tutte le istanze in una stessa base extension abbiano lo stesso insieme di proprietà. L'intera struttura delle base extension può quindi essere ricavata a partire dall'insieme di classi che la compongono, in particolare:

- l'*intensione* è data dall'unione degli attributi globali descritti nelle classi locali dell'insieme;

- l'*estensione* è costituita partendo dall'intersezione delle estensioni delle classi locali che la compongono.

Il procedimento adottato per il calcolo delle base extension, si basa sull'uso delle relazioni estensionali definite in precedenza.
Un prerequisito fondamentale per una corretta integrazione é rappresentato dalla corretta e completa conoscenza da parte del progettista delle relazioni estensionali che legano le diverse classi coinvolte.
Raramente il progettista é in grado di specificare, sin dal principio, tutte le rule estensionali esistenti, per questo spesso si preferisce generare una soluzione incompleta ma di immediata definizione piuttosto che investire grosse risorse per determinare tutta la conoscenza estensionale. Questa metodologia introduce un approccio diverso, di tipo incrementale. Partendo da un insieme non completo di rule é possibile calcolare il relativo insieme di base extension e la gerarchia estensionale che possono essere utilizzati per la deduzione di nuova conoscenza da integrare. Il processo viene ripetuto iterativamente fino al raggiungimento di una soluzione soddisfacente. Inoltre il procedimento é in grado di controllare la consistenza delle rule e rilevare e segnalare eventuali relazioni in conflitto tra di loro.
L'algoritmo utilizzato é schematizzato in Figura 2.4.2.

Analizzando la figura si vede come in base agli assiomi estensionali, definiti dal progettista, vengono ricavati gli *existence requirement*[4] e successivamente la gerarchia estensionale. Una volta determinata tale gerarchia é possibile, in base al risultato ottenuto, modificare l'insieme iniziale degli assiomi fino al raggiungimento del risultato voluto.

In figura 2.7 é rappresentata la tabella delle base extension ricavate per l'esempio, adottando la metodologia appena descritta.

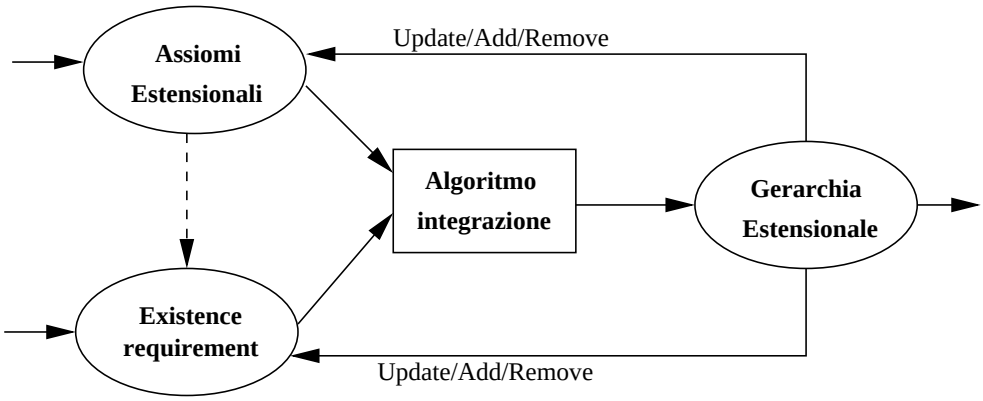


Figura 2.6: Procedimento di integrazione

| Base Extension | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| University_Worker | | | | X | X | X | X | X | | X | X | X |
| University_Student | X | X | X | X | X | X | | | | | | |
| School_Member | X | X | X | X | X | X | | | | | | |
| CS_Person | | X | X | X | X | | | X | X | X | | X |
| Student | | X | | X | | | | | | | | |
| Professor | | | | | | | | | | | | X |
| Research_Staff | | | | | | | | | | X | X | X |

Figura 2.7: Tabella delle base extension della classe University_Person

### 2.4.3 Generazione della gerarchia estensionale

Partendo dalle base extension definite in precedenza, viene costruita la gerarchia estensionale, che rappresenta le relazioni di specializzazione che legano fra loro le intensioni delle varie base extension. Avendo a disposizione gli attributi globali

[4]Un *existence requirement* é rappresentato da una espressione logica che deve essere soddisfatta da almeno una *base extension* dell'insieme calcolato in base alla rule estensionali definite. Deve essere verificata in ogni momento la: $\text{State}^{t}_{ER} \neq \emptyset$, dove ER é un requisito di esistenza.

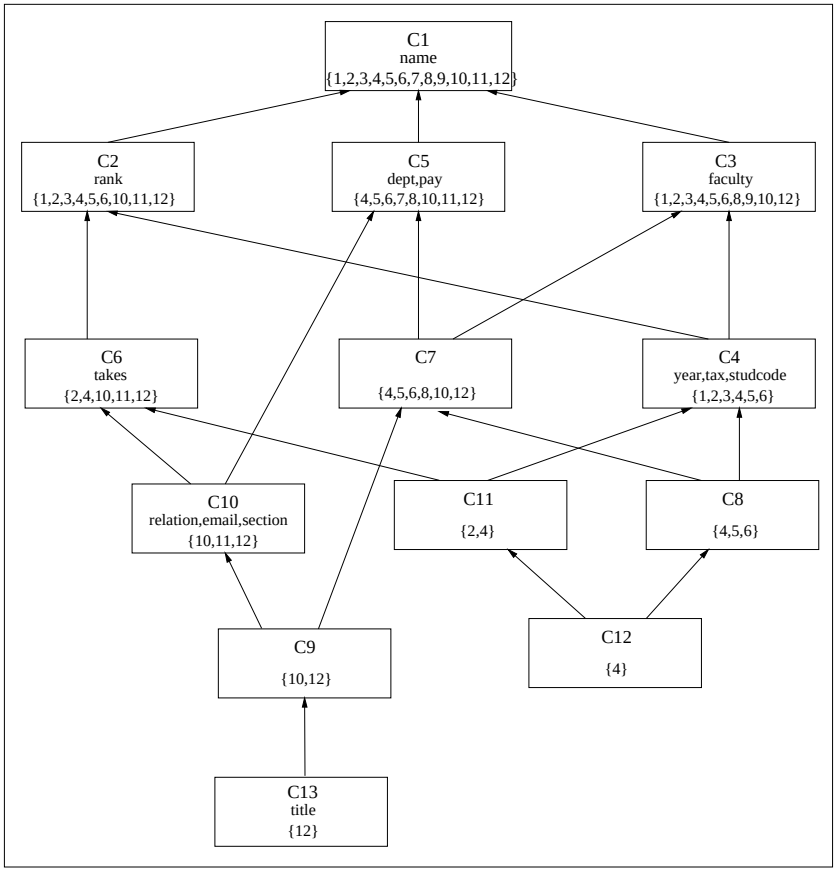


Figura 2.8: Rappresentazione della gerarchia estensionale della classe University_Person

presenti nella query posta dall'utente, é possibile individuare le classi virtuali e di conseguenza le base extension interessate.
Questa struttura consente quindi il passaggio da un'informazione di tipo intensionale ad una di tipo estensionale, in modo tale da minimizzare i tempi impiegati dal Query Manager per la procedura di ricerca della *Classe Virtuale Target*, che verrá descritta in 3.2.2. Le classi virtuali che la compongono sono caratterizzate dalla seguente struttura:

- l' *intensione* corrisponde agli schemi delle base extension presenti nella classe globale;

- l' *estensione* é data dall'unione di tutte le base extension che hanno almeno tutti gli attributi presenti nell'intensione della classe.

| Virtuale Classe | Intensione | Estensione |
|---|---|---|
| C1 | name | 1,2,3,4,5,6,7,8,9,10,11,12 |
| C2 | name, rank | 1,2,3,4,5,6,10,11,12 |
| C3 | name, faculty | 1,2,3,4,5,6,8,9,10,12 |
| C4 | name, rank, faculty, year, tax, studcode | 1,2,3,4,5,6 |
| C5 | name , dept, pay | 4,5,6,7,8,10,11,12 |
| C6 | name, rank, takes | 2,4,10,11,12 |
| C7 | name , dept, pay, faculty | 4,5,6,8,10,12 |
| C8 | name, dept, pay, faculty, rank, year, tax, studcode | 4,5,6 |
| C9 | name, dept, pay, faculty, relation, email, section, rank, takes | 10,12 |
| C10 | name, rank, takes , dept, pay, relation, email, section | 10,11,12 |
| C11 | name, rank, takes, faculty, year, tax, studcode | 2,4 |
| C12 | name, rank, takes, faculty, year, tax, studcode, dept, pay | 4 |
| C13 | name, dept, pay, faculty, relation, email, title, section, rank, takes | 12 |

Figura 2.9: Tabella della gerarchia estensionale della classe University_Person

In Figura 2.8 è illustrata la gerarchia estensionale relativa alla classe `University_Person`[5], mentre in figura 2.9 viene presentata una tabella nella quale, per ogni classe virtuale, viene mostrata l'intensione e l'estensione.

[5]C1, . . . ,C13 sono gli identificativi di ogni **classe virtuale** costituente la gerarchia estensionale associata ad una classe globale

## 2.5 Definizione di un modello di rappresentazione dello schema globale

In questa sezione verrá definita una formalizzazione della conoscenza generata all'interno della fase di integrazione degli schemi. I concetti e le strutture definite nelle sezioni 2.3 e 2.4 vengono descritti attraverso una rappresentazione matematica che consentirá di esprimere in modo formale i passi eseguiti dal Query Manager nella fase di *Definizione del Query Plan*.

### 2.5.1 Schema Globale

Sia **L** un insieme di *nomi di classi locali* (denotati da $L_1, L_2, \ldots$) e **AL** un insieme di *nomi di attributi locali* (denotati da $al_1, al_2, \ldots$). Gli attributi locali di un classe locale sono determinati tramite la funzione $A_L : \mathbf{L} \rightarrow 2^{\mathbf{AL}}$.

Sia **G** un insieme di *nomi di classi globali* (denotati da $G_1, G_2, \ldots$) e **AG** un insieme di *nomi di attributi globali* (denotati da $ag_1, ag_2, \ldots$). Gli attributi globali di un classe globale sono determinati tramite la funzione $A_G : \mathbf{G} \rightarrow 2^{\mathbf{AG}}$.

**Definizione 2 (Schema Globale)** *Data un insieme **G** ed un insieme **L**, uno* schema globale $SG$ *di **L**, è una funzione* $SG$

$$SG : \boldsymbol{G} \rightarrow 2^{\boldsymbol{L}}$$

*tale che* $SG(G_1), SG(G_2), \ldots, SG(G_k)$ *sia un partizionamento di **L**.*

La *Mapping Table* è la struttura dati che contiene tutte le informazioni riguardanti il passaggio dalla rappresentazione globale agli schemi locali, definisce quindi la conoscenza intensionale di una classe globale $G$. É rappresentata da una matrice, le cui colonne sono gli attributi globali di $G$, $ag$, e le righe sono le classi locali di $G$, $L$; i suoi elementi rappresentano la corrispondenza fra la classe locale $L$ e l'attributo globale $ag$.

**Definizione 3 (Mapping Table)** *Data una classe globale $G$, una* Mapping Table *di $G$, $MT$ è una matrice* $||SG(G)|| \times ||A_G(G)||$

$$MT = \begin{pmatrix} m_{11} & m_{12} & \ldots \\ m_{21} & m_{22} & \ldots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

*i cui elementi* $m_{kh} = MT[L_k][ag_h]$ *possono assumere i seguenti valori:*

- $al_i \in A_L(L_k)$
  mapping semplice: $\exists!\ al_i \in A_L(L_k) : al_i \leftrightarrow ag_h$
- `null`
  null mapping: *not* $\exists\ al_i \in A_L(L_k) : al_i \leftrightarrow ag_h$
- `<costante>`
  default mapping: (*not* $\exists^6\ a_i \in A(c_k) : a_i \leftrightarrow ga_h) \vee (\exists\ a_i \in A(c_k) : a_i = ga_h =$ `<costante>`)
- $al_1$ `and` ... `and` $al_M$
  and mapping: $\exists\{al_1, \cdots, al_M\} \subseteq A_L(L_k) : \{al_1, \cdots, al_M\} \leftrightarrow ag_h$
- $al_1, \ldots, al_M$
  complex mapping: $\exists\{al_1, \cdots, al_M\} \subseteq A_L(L_k) : \{al_1, \cdots, al_M\} \leftrightarrow ag_h$
- **`case`** *al* **`of`** `<costante>`$_1$ *:* $al_1$, ... `<costante>`$_M$ *:* $al_M$
  union mapping: $\exists\{al, al_1, \cdots, al_M\} \subseteq A_L(L_k)$ : *se* $al =$ `<costante>`$_i$,
  *con* $i = 1, \ldots, M \Rightarrow al_i \leftrightarrow ag_h$

### 2.5.2 Base Extension

Come abbiamo giá visto in 2.4.2, informalmente per *Base Extension* si intende un partizionamento dell'insieme complessivo di tutti gli oggetti rappresentati dalle sorgenti: sono sottoinsiemi disgiunti delle estensioni delle classi e sono ottenute dall'intersezione delle stesse. Più formalmente, un *insieme di Base Extension*, scritto $BES_{A1,A2,...,An}$ delle classi $A1, A2, ..., An$ viene definito da una formula booleana in DCF (Forma Canonica Disgiuntiva) sulle variabili $A1, A2, ...., An$. Ogni *minterm* di detta formula rappresenta una singola Base Extension.

Per applicare tale definizione nel contesto di una classe globale costituito da un insieme di classi locali, si parte dalla definizione di *Istanza di una classe globale*.

**Definizione 4 (Istanza di una classe globale)** *Data una classe globale $G$ ed un dominio **D**, un'istanza di $G$ sul dominio **D** è una funzione $\mathcal{I}$*

$$\mathcal{I} : SG(G) \rightarrow 2^{\boldsymbol{D}}$$

Un'istanza della classe globale $G$ verrà detta *legale* se soddisfa gli assiomi estensionali definiti sulle classi locali $SG(G)$.

[6] In questo caso il valore di default é inserito dal progettista.

**Definizione 5 (Base Extension)** *Data una classe globale $G$ ed una sua istanza legale $\mathcal{I}$, un* set di base extension *di $G$ rispetto ad $\mathcal{I}$ è una coppia $(\boldsymbol{B}, F)$, dove $\boldsymbol{B}$ è un insieme di* nomi di base extension *(denotati da $B_1, B_2, \ldots$ ,) e $F$ è una funzione*

$$F : \boldsymbol{B} \rightarrow 2^{SG(G)}$$

*tale che*

1. $\bigcup_{B \in \boldsymbol{B}} F(B) = SG(G)$
2. $$\bigcup_{B \in \boldsymbol{B}} \Big( \bigcap_{L \in F(B)} \mathcal{I}(L) - \bigcup_{L \in (SG(G) - F(B))} \mathcal{I}(L) \Big)$$

   *sia un partizionamento di $\mathcal{I}(G)$.*

Un set di base extension di una classe globale $G$ viene rappresentato tramite una tabella le cui righe riportano le classi locali della classe globale, cioè $SG(G)$, le colonne riportano le base extension, cioè gli elementi di **B**: una X in corrispondenza dell'elemento della tabella $(L, B)$ indicherà che $L \in F(B)$; un esempio è riportato in 2.7.

La parte intensionale di una base extension viene determinata a partire dall'insieme di classi locali che la compongono, considerando come attributi della base extension l'unione degli attributi globali che hanno un mapping non nullo in tali classi locali. Formalmente:

**Definizione 6 (Attributi di una base extension)** *Data una classe globale $G$, un suo set di base extension $(\boldsymbol{B}, F)$ ed una Mapping Table di $G$, $MT$, si definiscono gli attributi di $B \in \boldsymbol{B}$ come segue:*

$$A_{BE}(B) = \{ag \in A_G(G) \mid \exists L \in F(B), MT[L][ag] \textit{è un mapping non nullo}\}.$$

### 2.5.3 Schema Virtuale e Gerarchia Estensionale

Nella sezione precedente una classe globale è stata caratterizzata tramite il suo set di base extension e ad ogni base extension sono stati associati un insieme di attributi globali. Ora si considera una query sulla classe globale ed il problema che si vuole affrontare è il seguente: quali solo le base extension che occorre considerare per rispondere all'interrogazione, cioè quali sono le base extension che hanno almeno tutti gli attributi specificati nell'interrogazione? Il problema può essere risolto semplicemente controllando per tutte le base extension il rispettivo insieme di attributi. Un'altra possibilità che velocizza la ricerca è quella di

creare un *indice* che dato l'insieme di attributi dell'interrogazione restituisca direttamente l'insieme delle base extension interessate. A tale scopo, le base extension di una classe globale vengono raggruppate in *classi virtuali*; una classe virtuale è descritta da un insieme di attributi globali (*intensione*) e da un insieme di base extension (*estensione*) in modo tale che ogni attributo è comune a tutte le base extension e, viceversa, ogni base extension ha tutti gli attributi. Formalmente:

**Definizione 7 (Schema Virtuale)** *Data una classe globale $G$ ed un suo* set di base extension $BE = (\boldsymbol{B}, F)$*, lo schema virtuale di $G$ rispetto a $BE$ è una tripla* $(\boldsymbol{V}, INT, EST)$*, dove*

- $\boldsymbol{V}$ *è un insieme di* nomi di classi virtuali *(denotati da $V_1, V_2, \ldots, V_k$)*
- $INT : \boldsymbol{V} \rightarrow 2^{A_G(G)}$*,* intensione *dello schema virtuale*
- $EST : \boldsymbol{V} \rightarrow 2^{\boldsymbol{B}}$*,* estensione *dello schema virtuale*

*tale che*

1. $\forall V \in \boldsymbol{V}, \forall ag \in INT(V), \forall B \in EST(V)$ *si ha che* $ag \in A_{BE}(B)$
2. $\forall ag \in A_G(G), \exists V \in \boldsymbol{V} : ag \in INT(V)$
3. $\forall B \in \boldsymbol{B}, \exists V \in \boldsymbol{V} : B \in EST(V)$

Le classi virtuali di una classe globale vengono organizzate in una gerarchia, detta *gerarchia estensionale,* sulla base della relazione di inclusione tra i rispettivi insiemi di attributi:

**Definizione 8 (Gerarchia Estensionale)** *Dato uno schema virtuale* $(\boldsymbol{V}, INT, EST)$ *di una classe globale $G$, la* Gerarchia Estensionale *è costruita sulla base della relazione* $ISA_{EXT} \subseteq \boldsymbol{V} \times \boldsymbol{V}$ *tale che,* $\forall V, V' \in \boldsymbol{V}$*:*

$$V \; ISA_{EXT} \; V' \;\; \texttt{iff} \;\; INT(V) \supseteq INT(V')$$

Dalle definizioni date si può verificare che:

$$V \; ISA_{EXT} \; V' \;\; \texttt{iff} \;\; EST(V) \subseteq EST(V')$$

Nel Capitolo 3 verrá illustrato in che modo lo schema virtuale e la relativa gerarchia estensionale di una classe globale sono utili per l'elaborazione di una query posta dall'utente sulla classe globale.

## 2.6 Uso della conoscenza estensionale e intensionale nel Query Manager

Si é visto come, in fase di integrazione degli schemi, i due moduli Global Schema Builder ed Extensional Hierarchy Builder portano all'ottenimento delle informazioni necessarie per la fase di Query Processing. Da una parte il primo restituisce lo schema globale e le mapping table, che rappresentano la conoscenza intensionale, mentre il secondo ottiene la la gerarchia estensionale e le descrizioni delle base extension, che costituiscono la conoscenza estensionale.

Il Query Manager utilizza queste informazioni per svolgere la fase di Query Processing, in particolare nelle due sottofasi di *Definizione del Query Plan* (Query Plan) e *Esecuzione della Query* (Query Execution).

**Query Plan**:
si individua come e a quali sorgenti accedere per reperire i dati richiesti dall'utente tramite la query, per fare ció il Query Manager utilizza la gerarchia estensionale decidendo la classe virtuale contenente tutte le proprietá richieste e, dall'estensione di quest'ultima si stabiliscono le classi locali da interrogare.
Viene poi utilizzata la mapping table per tradurre la query iniziale in funzione delle tipologie di mapping esistenti.
La risposta generata per la query posta dall'utente deve essere corretta e, quanto più possibile, completa e minima e, come vedremo in 3, 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.

**Query Execution**:
si esegue la query, cioé si accede alle sorgenti, si recuperano e integrano i dati in modo da presentare all'utente una risposta globale. A questo punto si pone il problema dell'*Object Fusion*, cioé l'individuazione univoca delle entita' per consentire la corretta ricostruzione della risposta.

### 2.6.1 Object Fusion

Il problema dell'Object Fusion consiste nel determinare una o piu' chiavi semantiche, cioe' insiemi di attributi che consentano di individuare le entita' del dominio in modo univoco. Queste informazioni devono essere definite in fase di integrazione degli schemi locali e richiedono una profonda conoscenza della semantica sia del contesto applicativo, sia delle singole sorgenti.

Si é quindi resa necessaria l'introduzione delle **Regole di Join**, cioe' regole di fusione tra classi sorgenti, definite all'interno di una stessa classe globale. Ció significa stabilire, per ogni coppia di classi locali, l'eventuale presenza di almeno una chiave semantica comune che permetta il join. La definizione di queste regole e' solo in parte un processo automatico, l'intervento del progettista rimane fondamentale in questo caso. Il reperimento delle informazioni necessarie e le metodologie da utilizzarsi per l'individuazione delle regole di join, sono tuttora in fase di studio. L'ipotesi di sviluppo attuale prevede l'uso delle seguenti informazioni significative:

- relazioni di SYN presenti all'interno del Common Thesaurus;
- relazioni BT,NT presenti nel Common Thesaurus e che portano alla definizione di un unico attributo globale;
- legame tra attributi locali e attributi globali presente nella Mapping Table;
- assiomi estensionali definiti dal progettista;
- selezione di campi definiti *chiave* all'interno degli schemi locali.

Vengono quindi definite a livello di schema globale delle chiavi che possono essere di due tipi:

*chiavi semanticamente omogenee*;

*chiavi semanticamente disomogenee*.

La regole di join sono indispensabili nella fase di esecuzione delle query, in particolare i momenti in cui vengono utilizzate sono:

**Ricostruzione della base extension**:
: i risultati provenienti dalle classi locali vengono integrati utilizzando le regole di join. Tutte le classi appartenenti alla base extension devono essere fuse, nel caso in cui non sia possibile eseguire il join diretto fra due classi, é necessario ricorrere ad un una terza classe di collegamento.

**Unione delle base extension**:
: occorre unire gli oggetti ottenuti dalle ricostruzioni delle singole base extension evitando ridondanze.

Come vengono utilizzate le regole di join in queste due situazioni verrá descritto nel capitolo seguente, in cui viene analizzata la fase di Query Processing svolta dal Query Manager.

# Capitolo 3

# Il Query Manager

Il Query Manager, come giá visto nel Capitolo 1, é il modulo che gestisce la fase di *Query Processing*, in cui viene eseguita la query posta dall'utente sulla vista globale. Per svolgere il suo compito il Query Manager utilizza le strutture dati create nella fase di integrazione, descritta nel capitolo precedente, strutture che rappresentano la conoscenza intensionale ed estensionale necessaria per ottenere una risposta corretta e quanto piú possibile completa e minima.

Il processo di gestione delle interrogazioni puo' essere scomposto in tre fasi principali che vengono svolte in sequenza dal Query Manager:

1. *Acquisizione della Query*:
   in questa fase la query viene caricata nelle apposite strutture dati che ne rappresentano il contenuto. Il Query Manager ne verifica quindi la correttezza sintattica e semantica e ne esegue l'ottimizzazione. A questo punto la clausola where, acquisita inizialmente come una generica espressione booleana, viene posta in forma normale congiuntiva.

2. *Definizione del Query Plan*:
   la query acquisita al passo precedente é espressa in termini globali, in quanto posta dall'utente sulla vista integrata, é quindi necessario ricondurla agli schemi locali delle singole sorgenti. In questa fase viene generato il *plan* da associare alla query, che contiene non solo le informazioni relative alla generazione delle *Local Query*, ma anche quelle che consentiranno la successiva ricomposizione dei risultati.

3. *Esecuzione della Query*:
   in questa fase vengono eseguite dai wrapper le *Local Query* generate in precedenza ed i risultati restituiti vengono rielaborati seguendo le indicazioni contenute nel piano.

Le tre fasi appena citate verranno analizzate nelle sezioni seguenti, ponendo particolare attenzione alla gestione della clausola where della query ( sia per quanto riguarda la normalizzazione che per la traduzione) e all'esecuzione della query, che costituiscono gli argomenti approfonditi da questa tesi.

## 3.1 Acquisizione della Query

Come giá detto in precedenza, la query posta dall'utente é sottoposta ad alcune trasformazioni che la rendono eseguibile dal Query Manager.
Questa fase iniziale é suddivisa in tre momenti distinti:

- *parsing e validazione*;
- *ottimizzazione semantica globale*;
- *normalizzazione della clausola where*.

L'acquisizione della query comporta la verifica di correttezza sintattica e semantica e la successiva ottimizzazione, anche se in realtá all'interno di MOMIS le fasi di ottimizzazione e parsing sono invertite per sfruttare al meglio il software esistente, come vedremo in 4.

### 3.1.1 Parsing e validazione

A livello teorico, prima di effettuare il processo di ottimizzazione deve essere verificata la correttezza della query, sia da un punto di vista sintattico che semantico. Per questo viene utilizzato un modulo di riconoscimento grammaticale, il modulo di *parsing*, che ha il compito di verificare la correttezza sintattica della query, rispetto alla grammatica $OQL$ (riportata in Appendice C) e ne produce un'immagine in memoria centale. A questo punto la query viene validata, cioé ne viene accertata la correttezza semantica verificando l'effettiva appartenenza delle classi e degli attributi coinvolti allo schema integrato a cui é rivolta la query.

### 3.1.2 Ottimizzazione semantica globale

L'ottimizzazione é realizzabile se sono presenti regole di integritá inter-sorgenti definite sullo schema globale. Questi vincoli sono espressi con rule $ODL_{I^3}$ e vengono impiegati da ODB_Tools per riformulare la query producendone una semanticamente equivalente, ma eseguibile in modo piú efficiente.
Ad esempio vengono eliminati predicati ridondanti oppure vengono aggiunte nuove condizioni che possono abbreviare i tempi di risposta per la possibile presenza, nelle sorgenti, di indici sui predicati introdotti. In riferimento all'esempio

introdotto in 2.2, supponiamo che esista una relazione tra il numero di impiegati (`employee_nr`) e i fondi a disposizione del dipartimento (`budget`). Questo legame viene espresso dal progettista per mezzo di una rule $ODL_{I^3}$ che definisce la seguente regola di integritá sulla classe globale `Workplace`:

```
rule RG1  for all X in Workplace: (X.employee_nr > 100)
             then X.budget > 20000
```

Consideriamo la query seguente: *"Selezionare i nomi dei dipartimenti appartenenti al settore 'Engineering', con numero di impiegati maggiore di 150"*

```
Q:  select name
    from Workplace
    where area = 'Engineering'
    and employee_nr > 150
```

ODB-Tools espande semanticamente la query `Q` sfruttando la regola `RG1` ed ottiene la seguente query:

```
Q':  select name
     from Workplace
     where area = 'Engineering'
     and employee_nr > 150
     and budget > 20000
```

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 *budget* fossero definiti degli indici. Il prezzo da pagare è un aumento della complessità della fase di *Definizione del Query Plan* in quanto deve essere riformulata una query più complessa.

### 3.1.3 Normalizzazione della clausola where

La clausola where della query, dopo le fasi descritte in 3.1.2 e 3.1.1, é rappresentata da un'espressione booleana innestata ricorsivamente.
Il Query Manager, prima di iniziare la *Definizione del Query Plan*, trasforma questa generica struttura booleana in **forma normale congiuntiva**:

**Definizione 9 (Forma Normale Congiuntiva)** *Date n variabili di ingresso, una formula booleana viene detta in* forma normale congiuntiva *o CNF quando é formata da una produttoria di termini, ciascuno dei quali costituito da una sommatoria di letterali costituiti da un sottoinsieme delle n variabili di ingresso oppure da una loro negazione.*

Avere la clausola where in questa forma, consente di ottimizzare la fase di esecuzione della query, in quanto vengono valutati il maggior numero di predicati in and e quindi viene minimizzata la quantitá di risultati restituiti dalle sorgenti locali, con una conseguente velocizzazione della fase di integrazione.

Per ottenere la forma normale, la clausola where viene analizzata ricorsivamente e trasformata utilizzando i **teoremi dell'algebra di commutazione**:

- Valutazione di una clausola *NOT*:

  (T1) $\overline{x \vee y} = \bar{x} \wedge \bar{y}$ (De Morgan)
  (T1') $\overline{x \wedge y} = \bar{x} \vee \bar{y}$
  (T2) $\bar{\bar{x}} = x$ (Involuzione)

  Per quanto riguarda le espressioni not, utilizzando i teoremi elencati, queste possono essere considerate and o or e valutate come i due casi descritti in seguito.

- Valutazione di una clausola *AND*:

  (T3) $x \wedge 1 = x$ (Identitá)
  (T4) $x \wedge 0 = 0$ (Elemento nullo)
  (T5) $x \wedge x = x$ (Idempotenza)
  (T6) $x \wedge \bar{x} = 0$ (Complemento)
  (T7) $x \wedge (x \vee y) = x$ (Assorbimento)

  I teoremi precedenti sono valutati nell'ordine descritto e, se nessuno é applicabile, i due fattori in and vengono trasformati in modo indipendente.

- Valutazione di una clausola *OR*:

  (T3') $x \vee 0 = x$ (Identitá)
  (T4') $x \vee 1 = 1$ (Elemento nullo)
  (T5') $x \vee x = x$ (Idempotenza)
  (T6') $x \vee \bar{x} = 1$ (Complemento)
  (T7') $x \vee (x \wedge y) = x$ (Assorbimento)

  Nel caso in cui questi teoremi non siano applicabili viene utilizzata, quando é possibile, la proprietá distributiva che consente la scomposizione della clausola in due fattori in and analizzabili separatamente:

  (T8) $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$ (Proprietá distributiva)
  (T8') $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

Se (T8) e (T8') non sono applicabili, i due operandi vengono tradotti ricorsivamente, prima di applicare nuovamente la proprietá distributiva alla nuova clausola or cosí ottenuta.

Il seguente esempio evidenzia la trasformazione in forma normale congiuntiva di una espressione booleana, dove $x, y, v, w$ rappresentano i predicati semplici .

$$((\overline{x \vee y}) \vee v) \vee (w)$$

1. l'analisi inizia dall'or esterno e, non essendo applicabile la proprietá distributiva (T8), i due operandi vengono tradotti ricorsivamente;
   (a) il primo fattore $((\overline{x \vee y}) \vee v)$ viene trasformato tramite (T1) e successivamente (T8) generando :
   $(\bar{x} \vee v) \wedge (\bar{y} \vee v)$
   (b) il secondo fattore rimane $w$;
2. l'or iniziale diventa quindi:
   $((\bar{x} \vee v) \wedge (\bar{y} \vee v)) \vee w$
3. viene riapplicata la proprietá (T8) che genera la forma finale:
   $(\bar{x} \vee v \vee w) \wedge (\bar{y} \vee v \vee w)$

A questo punto la clausola where viene ulteriormente semplificata eliminando completamente la negazione: la negazione di una condizione composta é stata trasformata utilizzando i teoremi (T1), (T1')e (T2) visti in precedenza, se invece l'operatore not precede una condizione semplice si sostituisce la condizione con la sua complementare.
Per fare ció devono essere valutati sia l'operatore che il quantificatore presenti all'interno dell'espressione di confronto:

- sostituzione del quantificatore:

  *some, any* $\Leftrightarrow$ *all*

- sostituzione dell'operatore:

  ' $>$ ' $\Leftrightarrow$ ' $<=$ '
  ' $<$ ' $\Leftrightarrow$ ' $>=$ '
  ' $=$ ' $\Leftrightarrow$ '$!=$'
  'like' $\Leftrightarrow$ 'not like'

Un esempio di normalizzazione di una clausola WHERE é il seguente:

```
Q:  select name
    from University_Person
    where not((faculty = 'CS') or (pay > 10000))
    and (title = 'full professor')
```

La clausola where della query `Q` viene posta in forma normale congiuntiva, generando `Q'`:

```
Q':  select name
     from University_Person
     where (faculty != 'CS')
     and  (pay <= 10000)
     and (title = 'full professor')
```

## 3.2 Definizione del query plan

La query posta dall'utente viene validata e ottimizzata durante la fase descritta in 3.1, la struttura cosí generata prende il nome di Global Query. In Figura 3.1 sono rappresentate le operazioni effettuate dal Query Manager che, partendo dalla Global Query, portano alla produzione del risultato finale da restituire all'utente.

In questa sezione verrá descritta la fase di *Definizione del Query Plan*, durante la quale vengono generate le Basic Query, individuate le classi locali coinvolte e preparate le Local Query da inviare alle sorgenti. Parallelamente a queste attivitá il Query Manager raccoglie le informazioni necessarie per la fase successiva di *Esecuzione della Query*.

### 3.2.1 Decomposizione della Global Query in Basic Query

Una query, in generale, conterrá richieste rivolte a piú classi globali appartenenti allo stesso schema, di conseguenza il primo passo nella *Definizione del Query Plan* é la decomposizione della Global Query in Basic Query, la cui definizione é riportata di seguito:

**Definizione 10 (Basic Query)** *Una Basic Query è una query contenente tutte le richieste rivolte ad una singola classe globale.*

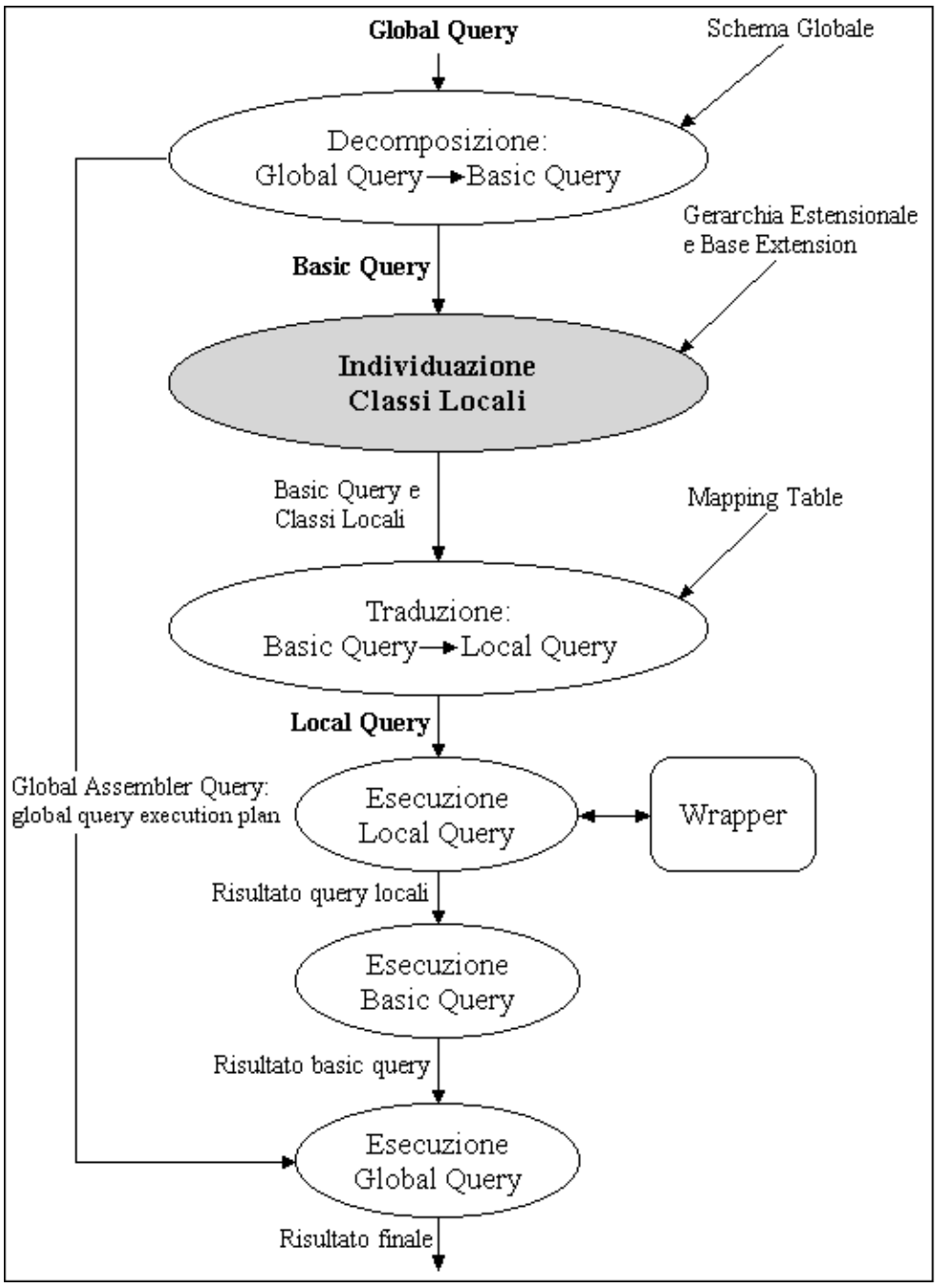


Figura 3.1: Operazioni del Query Manager

Le Basic Query costituiscono quindi gli elementi di base per le fasi successive, senza contare che di fatto questa tipologia di interrogazioni costituisce la

percentuale maggiore di query rivolte ad un Mediatore.

Le Basic Query sono espresse mediante un sottoinsieme del linguaggio OQL, riportato in Appendice D, in particolare non sono ammessi:

- join espliciti fra classi diverse, ma é possibile la navigazione attraverso aggregazioni ed associazioni per ricostruire oggetti complessi;
- subquery;
- operatori di ordinamento (order by) o di conversione (listtoset, element, flatten);
- restituzione di strutture complesse (list, array, struct).

Questi operatori devono essere implementati ad un livello superiore, in quanto agiscono su insiemi di informazioni giá integrate.

A livello Global Query, il Query Manager si deve occupare di due attivitá:

1. determinazione delle Basic Query;
2. ricostruzione della Global Query a partire dai dati restituiti dalle Basic Query.

Per rappresentare un esempio di decomposizione da Global Query a Basic Query, facciamo per il momento una variazione alla Mapping Table riportata in 2.7, e immaginiamo che non esista la navigazione implicita consentita dal mapping complesso dell'attributo globale *dept*, ma che il collegamento fra le classi globali `University_Person e Workplace`, sia fornito dal join esplicito con l'attributo globale *code*.
Con le modifiche apportate, cosideriamo la seguente Global Query `Q`: *"selezionare lo stipendio medio dei professori che lavorano in dipartimenti con budget maggiore di 10.000 dollari"*.

```
Q: select avg(University_Person.pay)
   from University_Person,Workplace
   where University_Person.dept = Workplace.code
   and University_Person.rank = 'professor'
   and Workplace.budget > 10000
```

Questa Global Query viene scomposta nelle due Basic Query `Qa` e `Qb`, rivolte rispettivamente alle classi globali `University_Person` e `Workplace`.

```
Qa: select pay, dept
    from University_Person
    where rank = 'professor'


Qb: select code
    from Workplace
    where budget > 10000
```

Viene generata una terza query `Qc`, chiamata *Global Query Assembler*, che rappresenta l'operazione di join che il Query Manager dovrá compiere per ricostruire il risultato.

```
Qc: select avg(Qa.pay)
    from Qa,Qb
    where Qa.dept = Qb.code
```

Le informazione da utilizzare per la fase di integrazione dei risultati, e le Basic Query generate vengono inserite in *execution plan evaluator*, consultato dal Query Manager durante la fase di esecuzione della Global Query.

### 3.2.2 Individuazione delle classi locali

Per ogni Basic Query generata al passo precedente, bisogna determinare l'insieme ottimo di classi locali a cui inviare le Local Query, per pervenire ad una risposta corretta e il piú possibile completa e minima. Per *correttezza* si intende la possibilitá di reperire le sole istanze che godono delle proprietá richieste e soddisfano tutte le condizioni imposte, la *completezza* indica la possibilitá di individuare tutte le sorgenti che possono contribuire alla risposta ed infine per *minimalitá* si intende l'eliminazione di inutili duplicazioni.
Per poter eseguire correttamente questo passaggio é necessario utilizzare entrambi i tipi di conoscenza individuati in fase di integrazione degli schemi. L'uso della sola mapping table infatti consente di individuare quali classi locali contengono le proprietá richieste dalla query, ma non consente la ricostruzione degli oggetti virtuali che rappresentano le entitá descritte in termini globali.
L'analisi effettuata esclusivamente in funzione della conoscenza intensionale porta all'ottenimento di una risposta piú o meno corretta, ma sicuramente non completa, in quanto le sorgenti vengono considerate singolarmente perdendo la possibilitá di ricostruire le entitá proprie del contesto applicativo. Queste entitá sono caratterizzate dall'avere proprietá distribuite su piú classi locali appartenenti a sorgenti differenti, di conseguenza la completezza é garantita solo utilizzando anche

le informazioni estensionali rappresentate dalle base extension e dalla gerarchia virtuale.
Inoltre l'impiego di tale conoscenza consente di introdurre elementi di ottimizzazione che permettono di soddisfare il requisito di minimalitá: sfruttando le informazioni estensionali é possibile evitare le duplicazioni, sia eliminando le base extension dominate, sia semplificando l'insieme di classi da interrogare, come vedremo nel seguito.
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.

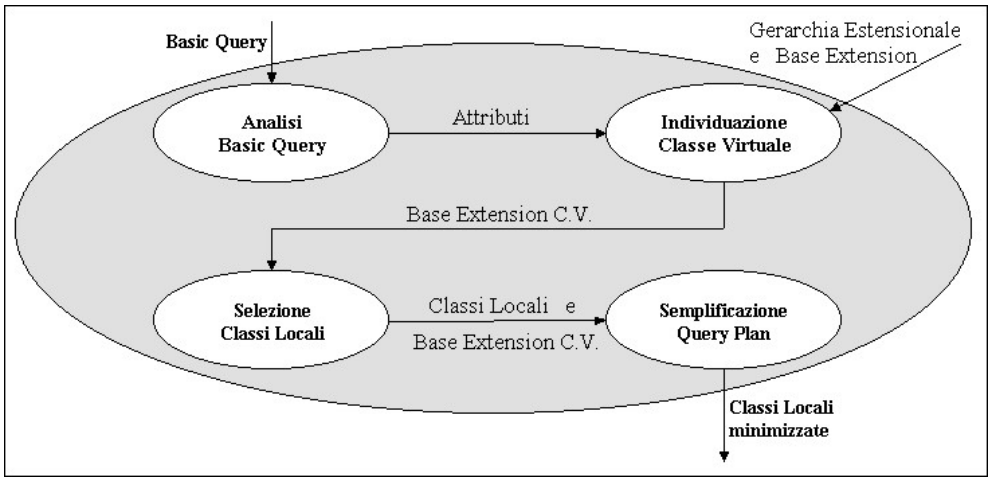


Figura 3.2: Passi della fase di individuazione delle classi locali

I passi costituenti questa fase di individuazione delle classi locali, cioè dell'insieme minimo ed ottimale di classi locali a cui accedere per ottenere una risposta corretta, completa e minima, sono schematizzati in Figura 3.2 e spiegati qui di seguito, utilizzando la simbologia introdotta in 2.5.

1. **Analisi della Basic Query**
   Il testo della Basic Query viene analizzato per individuare tutti gli attributi globali in esso contenuti.
   Una generica Basic Query puó essere schematizzata come segue:

```
SELECT <select-list>
FROM G
WHERE <where-clause>
```

Dove `<select-list>` e' un insieme di attributi globali: $f_1(A_G(G))$ , mentre `<where-clause>` e' un'espressione booleana, sempre funzione degli attributi globali, giá posta in forma normale congiuntiva $f_2(A_G(G))$. Viene individuato l'insieme $AG_Q$ di attributi globali contenuti nella `<select-list>` e nella `<where-clause>`.

Consideriamo ad esempio la query seguente:

```
Q_1: select name,dept
     from University_Person
     where faculty = 'CS'
     and (pay > 20000)
```

Per la query `Q_1` viene determinato l'insieme:
$AG_Q = \{$`name, dept, faculty, pay`$\}$, che costituisce il punto di partenza per i passi successivi.

2. **Individuazione delle Classi Virtuali Target**
   A questo punto bisogna utilizzare la gerachia estensionale $ISA_{EXT}$ dello schema virtuale $(\mathbf{V}, INT, EST)$, associata alla classe globale $G$, per individuare le *Classi Virtuali Target*, cioe' le classi virtuali piú generali che dispongono di tutte le proprietá richieste.

$$VC = \{V \in \mathbf{V} \mid AG_Q \subseteq INT(V), \text{not } \exists\, V' \neq V \mid AG_Q \subseteq INT(V) \wedge V\ ISA_{EXT}\ V'\}$$

Le classi virtuali vengono organizzate in una gerarchia di ereditarietá proprio per minimizzare i tempi di ricerca, infatti il Query Manager deve esaminare la gerarchia partendo dalla classe padre e passare al livello inferiore solo se le classe virtuale presa in considerazione non contiene le proprietá richieste.

In realtá, allo stato attuale, le classi Virtuali sono organizzate in una lista che non consente di effettuare il processo di ricerca appena descritto.

Per ora vengono eseguiti due passi successivi: in primo luogo si determina un insieme iniziale di classi virtuali $VC_{in}$, la cui intensione contiene tutti gli attributi $AG_Q$ della query:

$$VC_{in} = \{V \in \mathbf{V} \mid AG_Q \subseteq INT(V)\}$$

Nell'esempio vengono selezionate: $VC_{in} = \{C7, C8, C9, C12, C13\}$. Da questo insieme iniziale, sono individuate le classi virtuali con estensione massima, quelle cioé che contengono il maggior numero di base extension.

$$VC_{opt} = \{V \in VC_{in} \mid EST(V) \supseteq EST(V_h), \forall V_h \in VC_{in}\}$$

Nell'esempio si ottiene: $VC_{opt} = \{C7\}$.

3. **Individuazione delle Base Extension**
   Nel caso in cui le classi virtuali siano piú di una, bisogna considerare l'unione delle base extension, in caso contrario l'insieme $BE_{in}$ di base extension iniziali é determinato automaticamente dall'estensione della classe virtuale. Nell'esempio $VC_{opt} = \{C7\}$ e quindi $BE_{in} = EST(C11) = \{4, 5, 6, 8, 10, 12\}$. In generale vale la formula seguente:

   $$BE_{in} = \bigcup_{V \in VC_{opt}} EST(V)$$

   Vediamo un altro esempio:

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

   L'analisi di questa semplicissima query ci fornisce l'insieme di attributi {`pay`, `rank`}; le classi virtuali piú generali che li contengono entrambi nella loro intensione sono C8 e C10, la prima ha estensione formata dalle base extension $\{4, 5, 6\}$, la seconda dalle $\{10, 11, 12\}$. 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.

4. **Selezione delle Classi Locali**
   Individuate le base extension sono note automaticamente anche l'insieme $CL_{in}$ di classi da interrogare, definito come:

   $$CL_{in} = \{L \in SG(G) \mid \exists B \in BE_{in}, L \in F(B)\}$$

Relativamente alla nostra query di esempio `Q_1`, inizialmente le base extension selezionate sono: $BE_{in} = \{4, 5, 6, 8, 10, 12\}$, esaminando la tabella riportata in Figura 2.7, dalle colonne delle base extension in questione si evince l'insieme $CL_{in}$ delle classi locali alle quali accedere.

5. **Semplificazione del Query Plan**
   Gli insiemi $BE_{in}$ e $CL_{in}$ determinati devono essere ulteriormente semplificati per minimizzare il numero di query da inviare alle sorgenti. Questa semplificazione avviene in due momenti distinti:

   - **Eliminazione di base extension**:
     Puó succedere che non tutte le base extension contenute in $BE_{in}$ debbano essere ricostruite per ottenere la risposta minima, per valuare le eventuali ridondanze viene introdotto il concetto di *dominazione*:

     **Definizione 11 (Dominazione)** *Date due Base Extension $B_1$, $B_2$ ed un insieme di attributi A = $\{a_1, \ldots, a_n\}$, si dice che $B_1$* domina *$B_2$ rispetto ad A, se A è compreso nelle intensioni di entrambe e se le classi che compongono $B_1$ sono un sottoinsieme di quelle che formano $B_2$.*

     $$B_1 \text{ domina } B_2 \textit{ rispetto ad } A \Leftrightarrow$$
     $$(A \subseteq A_{BE}(B_1)) \wedge (A \subseteq A_{BE}(B_2)) \wedge (F(B_1) \subset F(B_2))$$

     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 3.3 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 $b_2$[1] (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 $b_5$ è data da: $A \bigcap B \bigcap C$, quindi $b_2 = (A \bigcap B) - b_5$. Ricostruendo $b_2$ dunque inevitabilmente si ricostruisce anche una parte di $b_5$, quella limitata agli attributi di A e B. La definizione su riportata ci dice che, rispetto a questo insieme di attributi, $b_2$ domina $b_5$.

[1] In questo caso le base extension vengono indicate con la "*b*" minuscola per non creare confusione con i nomi delle classi locali.

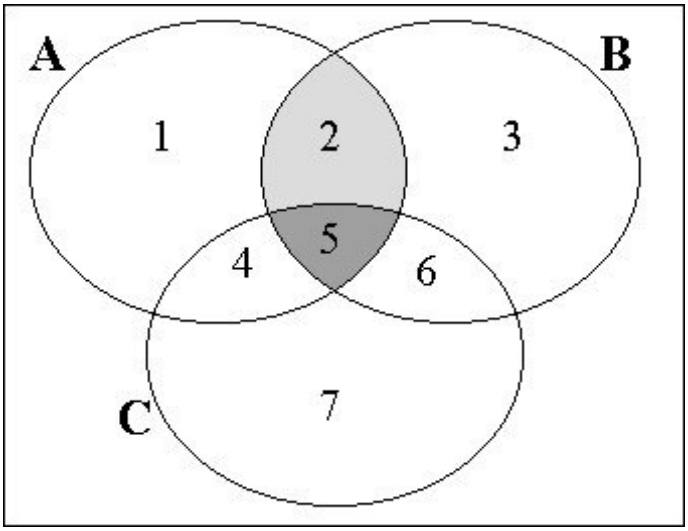


Figura 3.3: Esempio di dominazione tra Base Extension

Quindi si possono, anzi si devono, non prendere in considerazione le base extension dominate, ma si devono ricostruire solo quelle dominanti per evitare duplicazioni.
Di conseguenza sfruttando la definizione precedente, si ottiene l'insieme di base extension dominate:

$$BE_{dom} = \{B_2 \in BE_{in} \mid \exists B_1 \in BE_{in} \wedge B_1 \text{ domina } B_2 \text{ rispetto a } AG_Q\}$$

Con questo primo passo di semplificazione si ottiene l'insieme ottimo di base extension:

$$BE_{opt} = BE_{in} - BE_{dom}$$

Per la query di esempio `Q_1`, l'insieme di base extension iniziali $BE_{in} = \{4, 5, 6, 8, 10, 12\}$ viene semplificato in $BE_{opt} = \{6, 8\}$, infatti, rispetto all'insieme di attributi della query, si vede che la 6 domina sia la 4 che la 5, mentre la 8 domina l'insieme $\{4, 5, 10, 11\}$. Devono quindi essere ricostruite solo le base extension 6 e 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, 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. Nel nostro esempio `Q_1` entrambe le base extension selezionate dominano sia la 4 che la 5, di conseguenza in fase di esecuzione devono essere fuse tramite un outer join , come verrá approfondito in 3.3.2.

- **Eliminazione di classi locali**:
  una volta individuato l'insieme ottimo di base extension, ognuna di queste viene esaminata per ottenere il numero minimo di classi a cui accedere. Questa semplificazione del Query Plan consente di diminuire il numero di Local Query da generare e potrebbe anche evitare l'accesso ad una sorgente.

  In una base extension potrebbe accadere che due classi locali siano estensionalmente equivalenti, se questo accade e nessuna delle due classi aggiunge informazioni rispetto all'altra, puo' essere eliminata una delle due in modo indifferente. Se invece una classe contiene attributi della query non presenti nell'altra deve essere eliminata la seconda.

  Un altro caso in cui é possibile minimizzare l'insieme di classi locali, si verifica quando due classi appartengono allo stesso database ed esiste fra loro una relazione di specializzazione: se la sorgente é 'object' o se in ogni caso la classe specializzata contiene tutte le proprietá richieste, é possibile scartare la superclasse.

  Nell'esempio introdotto, la base extension 8 prevede l'interrogazione delle seguenti classi locali $CL_8$ = { `University_Worker,CS_Person` }, come si puo' vedere questo insieme non puó essere ulteriormente ridotto in quanto non ricorre uno dei due casi visti in precedenza. La base extension 6 prevede inizialamente $CL_6$ = { `University_Worker, School_Member, University_Student`}, in questo caso `School_Member` e `University_Student` risultano estensionalmente equivalenti e nessuna delle due aggiunge informazioni rispetto all'altra, quindi una delle due puó essere eliminata indifferentemente, ma eliminanado `University_Student` si evita di accedere alla terza sorgente, `TAX_POSITION`. L'insieme ottimo di classi a cui inviare le Local Query risulta essere: $CL_{opt}$ = {`University_Worker, CS_Person, School_Member`}.

  Prima di eseguire la fase di semplificazione delle classi locali, devono

essere eventualmente aggiunti alla `<select-list>` della query, gli attributi aggiuntivi per la fusione delle base extension. Come verrá descritto nel dettaglio in 3.3.2, per due base extension da fondere in outer join, é necessario individuare una chiave comune. Per questo é necessario considerare, per determinare l'insieme ottimo delle classi locali, non solo gli attributi originari della query, ma anche eventuali attributi aggiuntivi necessari alla fusione delle base extension.

### 3.2.3 Generazione delle Local Query

Conclusa la fase di individuazione delle classi locali, il Query Manager ha a disposizione l'insieme minimo di classi da interrogare, gli attributi della Basic Query ed eventualmente gli attributi aggiunti per ricostruire le base extension. Questi elementi costituiscono l'input per l'ultimo passo della fase di *Definizione del Query Plan* che consiste nella generazione delle Local Query da inviare alle varie sorgenti. Con l'ausilio della Mapping Table, contenente tutte le informazioni necessarie per risolvere i conflitti intensionali, la Basic Query, posta sulla schema globale, viene *''riscritta''* in funzione degli schemi locali.

Grazie al modello comune di dati $ODL_{I^3}$, con il quale il Mediatore comunica con i Wrapper, il Query Manager non si deve preoccupare dei linguaggi e dei formalismi utilizzati dalle singole sorgenti, ma saranno i Wrapper stessi a tradurre la query, formulata in OQL, in un linguaggio comprensibile alla sorgente. Prima di effettuare la traduzione vera e propria in funzione degli schemi locali, il Query Manager si occupa della scomposizione della clausola where e dell'individuazione di eventuali attributi aggiuntivi necessari per la successiva fase di *Esecuzione della Query*.

Introduciamo la Basic Query `Q_2`, che ci servirá da esempio per descrivere le diverse fasi della generazione delle Local Query:

```
Q_2: select name, email
     from University_Person
     where faculty = 'cs'
     and rank = 'professor'
     and ( pay > 10000 or title = 'full professor')
```

- **Scomposizione della clausola where e generazione della Basic Query Assembler**
  La clausola WHERE di uno statement OQL é una espressione booleana composta da attributi, tipi base, operatori logici e relazionali, che deve

essere soddisfatta dalle tuple di una relazione. Conclusa la fase di normalizzazione descritta in 3.1.3, la clausola e' posta in forma normale congiuntiva.

Ai fini della traduzione in funzione delle sorgenti locali, una *congiunzione* puo' essere scomposta ed i suoi due termini analizzati in modo indipendente, mentre una *disgiunzione* deve essere analizzata nel suo complesso.
Consideriamo la query `Q_2` riguardante la classe globale `University_Person`: la fase di *Definizione del Query Plan* individua una sola base extension da ricostruire $BE_{opt} = \{12\}$ , e l'insieme ottimo di classi locali da interrogare [2]:

- `Research_Staff(name, dept, pay, rank, email, relation, section)`
- `Professor(name, dept, faculty, rank, title)`

In questo caso il predicato di selezione:

```
(pay > 10000 or title = 'full professor')
```

non é compreso interamente in nessuna delle due classi locali, la scomposizione della clausola nei suoi due fattori porterebbe all'ottenimento di una risultato incompleto e deve quindi essere valutata nel suo insieme.
Nel caso in cui la Basic Query precedente avesse avuto la seguente clausola in *and*:

```
(pay > 10000 and title = 'full professor')
```

si sarebbe potuta scomporre per valutare i suoi due termini in modo indipendente. Le due Local Query ottenute sarebbero[3]:

```
Q_21: select first_ name, last_name, e_mail
      from Research_Staff
      where pay > 10000
```

[2] Vengono indicati, per ogni classe locale, gli attributi globali con mapping non nullo.

[3] Allo stato attuale il Query Manager non si occupa della gestione dei predicati contenenti valori di default, che vengono semplicemente tradotti col valore predefinito. Nelle rappresentazioni successive questi predicati non vengono riportati se hanno valore *true*, ad esempio le query `Q_21` e `Q_22` conterrebbero rispettivamente `('professor' = 'professor')` e `('cs' ='cs')` che non sono riportati.

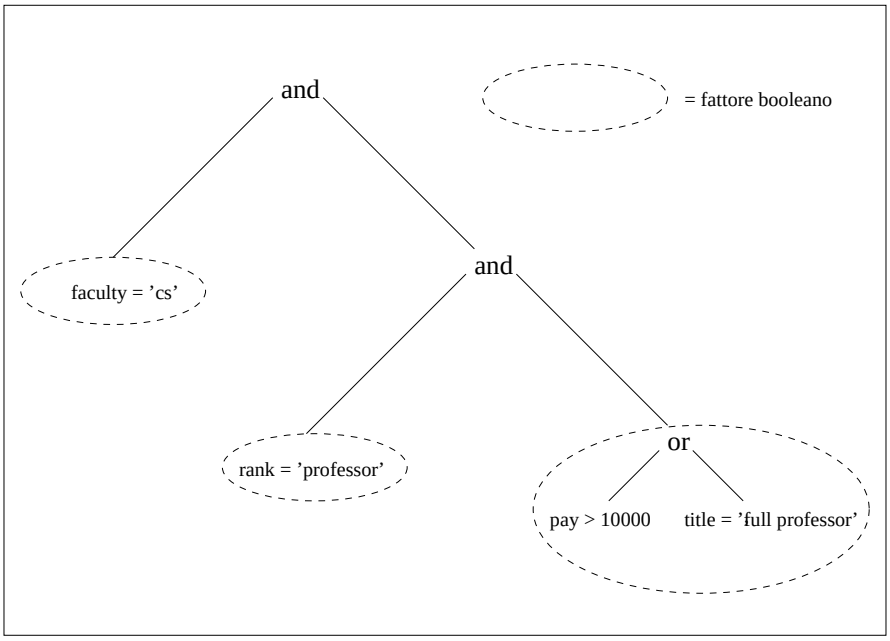


Figura 3.4: Albero dei predicati di selezione

```
Q_22: select name
      from Professor
      where rank = ’professor’
      and title = ’full professor’
```

Il Query Manager ha a disposizione la clausola where in forma normale congiuntiva, rappresentabile da un albero di predicati che evidenzia i fattori booleani da considerarsi in modo indipendente in fase di traduzione.

Nella figura 3.4 é rappresentato l’albero associato alla query di esempio `Q_2`; viene evidenziato il modo in cui il Query Manager analizza la clausola ricorsiva e i predicati che vengono valutati nel loro complesso. Ognuno di questi fattori viene inserito nei *predicati da tradurre*, se gli attributi globali in esso contenuti hanno un mapping non nullo in almeno una delle classi locali considerate.

É necessario peró considerare anche quei fattori che rimangono esclusi in fase di generazione delle query locali, quelli cioé che non sono mappati interamente da nessuna classe. Per questo motivo ad ogni Basic Query viene associata una struttura detta *Basic Query Assembler* che contiene all’interno della propria clausola where i predicati “esclusi”. Il Query Manager, dopo

aver concluso la fase di *Esecuzione della Query* ed aver quindi integrato i risultati provenienti dalle sorgenti locali, dovrá eseguire su di essi la Basic Query Assembler per ottenere la risposta desiderata.

Per poter svolgere correttamente questa query finale, devono essere recuperati dalle sorgenti locali anche gli attributi presenti nella clausola where della Query Assembler, aggiungendoli a quelli della Basic Query originale.

Valutando la query precedente, si nota che l'ultimo fattore booleano non é interamente mappato da nessuna delle due classi locali, la Basic Query Assembler diventa quindi:

```
Q_2A: select name, email
      from University_Person
      where ( pay > 10000 or title = 'full professor')
```

Per poter eseguire questa interrogazione sui risultati finali, la Basic Query viene completata aggiungendo agli attributi di selezione **pay** e **title**, dando origine a `Q_2'`:

```
Q_2':  select name, email, pay, title
       from University_Person
       where faculty = 'cs'
       and rank = 'professor'
       and ( pay > 10000 or title = 'full professor')
```

Nella Tabella 3.1 é rappresentato schematicamente l'algoritmo che implementa i passi appena descritti a livello teorico, in cui con $AG(P_i)$ é indicato l'insieme di attributi globali contenuti nel predicato $P_i$, mentre $AG(L_j)$ rappresenta l'insieme di attributi globali con mapping non nullo sulla classe locale considerata, vale a dire: $AG(L_j) = \{ag \in A_G(G) \mid MT[L_j][ag]$ è un mapping non nullo$\}$.

- **Traduzione da Basic Query a Local Query**
  La fase di traduzione riguarda gli attributi di proiezione e i predicati di selezione della Basic Query.

  *Attributi di proiezione*
  Il Query Manager analizza gli attributi presenti nella <select-clause> della Basic Query e quelli aggiunti per la fusione delle base extension, sostituendoli con i corrispondenti attributi originali della classe locale. Come visto in 2.3.4, la Mapping Table contiene le informazioni riguardanti il tipo di mapping fra attributo

♢ *input*: <where-clause>$= P_1 \wedge P2 \wedge \ldots \wedge P_m$, con $P_j = f_1(A_G(G))$;
$MT$ Mapping Table;
$CL_{opt} = \{L_1, L_2, \cdots, L_k\}$ insieme ottimo di classi da interrogare;
<select-clause> della Basic Query iniziale;
♢ *output*: <where-clause>$_{ass} = f_2(A_G(G))$ clausola where della Basic Query Assembler;
$P = f_3(A_G(G))$ insieme dei predicati da tradurre.

```
begin
    <where-clause>_ass := true;
    P := {};
    for i := 1 to m do
     begin
        ok:= false;
         for j := 1 to k do
             if (AG(P_i) ⊆ AG(L_j))
             then
             begin
                 ok := true;
                 break;
             end
         if ( ok= true )
         then P := P ⋃ {P_i};
         else
           begin
              <where-clause>_ass =<where-clause>_ass ∧ P_i;
              <select-clause>=<select-clause> ⋃ AG(P_i);
           end
     end
end
```

Tabella 3.1: Generazione della Basic Query Assembler

globale e attributo locale, informazioni definite in fase di integrazione degli schemi dal Global Schema Builder.

Per ognuna delle tipologie di mapping sono definite le apposite regole di traduzione che consentono di riscrivere gli attributi presenti

nella clausola di selezione della Basic Query in funzione degli schemi locali.

É necessario per ogni elemento tradotto, tener traccia dell'attributo globale originale, con il rispettivo mapping; queste informazioni vengono inserite nel *plan* per essere utilizzate nella fase di esecuzione delle Local Query descritta in 3.3.1.
Quando si incontra un *null mapping* o *default mapping* non viene eseguita nessuna traduzione, ma nell'ultimo caso viene comunque registrata l'informazione nel piano per consentire la corretta ricostruzione della risposta.
I metodi relativi alla *union mapping* non sono ancora stati implementati, saranno elementi necessari il vettore contenente gli attributi tra cui effettuare la scelta e una condizione che permetta di discriminare fra le diverse alternative.

Il Query Manager deve verificare la possibilitá di ricostruire
ogni base extension partendo dai risultati delle classi locali interrogate, per questo motivo devono essere eventualmente aggiunti alla `<select-clause>` della Local Query gli attributi mancanti, necessari per effettuare i join durante la fase di esecuzione delle Basic Query, come descritto in 3.3.2.

*Predicati di selezione*

Per ogni classe locale appartenente all'insieme ottimo di classi da interrogare, vengono analizzati i *predicati* individuati in precedenza.

Ognuno di questi fattori viene riportato nella Local Query se tutti gli attributi globali in esso contenuti hanno un mapping non nullo nella classe locale considerata. L'algoritmo riportato nella Tabella 3.2 rappresenta, in modo schematico, questo processo di traduzione per una generica Basic Query posta sulla classe globale $G$ in funzione di ogni classe locale definita in 3.2.2 ($\forall L \in CL_{opt}$).

`<where-clause>`$_L$ rappresenta la clausola where della generica classe locale $L$, $P_j$ é il predicato espresso in termini globali, mentre $P'_j$ é la sua traduzione ottenuta sfruttando le opportune regole definite in funzione della tipologia di mapping riscontrato.

Devono quindi essere definiti metodi di traduzione oltre che per i singoli attributi anche per i predicati di confronto nel loro complesso.

Per quanto riguarda la query di esempio `Q_2`, le due query locali generate in fase di traduzione sono le seguenti:

$\diamondsuit$*input*: $\{P_1, P2, \ldots, P_n\}$, con $P_j = f_1(A_G(G))$
$\diamondsuit$*output*: <where-clause>$_L = f_2(A_L(L))$

```
begin
    <where-clause>_L := true;
    for j := 1 to n do
     if (AG(P_j) ⊆ AG(L))
     then <where-clause>_L := <where-clause>_L ∧ P'_j;
end
```

Tabella 3.2: Algoritmo di traduzione della clausola where

```
Q_21:  select first_name, last_name, e_mail, pay
       from Research_Staff

Q_22:  select name, title
       from Professor
       and rank = 'professor'
```

### 3.2.4 Semplificazioni del piano di accesso

Un miglioramento al piano di accesso si puó ottenere analizzando eventuali predicati di selezione della Basic Query, contenenti valori di default.
Se ad esempio fosse formulata un'interrogazione contenente la clausola `where faculty = 'biology'`, prima di affrontare la fase di traduzione delle query locali, si potrebbero eliminare dal piano tutte le base extension che coinvolgono le classi locali in cui é predefinito un valore di *faculty* diverso, riducendo cosí il numero di query da inviare alle sorgenti.
Nel nostro esempio intodotto in 2.2 sarebbe possibile eliminare immediatamente le classi locali della sorgente `COMPUTER_SCIENCE`, in cui é predefinito il valore di default `faculty = 'cs'`.

Un'altra semplificazione del piano di accesso é ottenibile nel caso in cui sia possibile raggruppare piú Local Query da inviare ad una stessa sorgente, effettuando localmente i join. Per applicare questa semplificazione é necessario che i risultati provenienti dalle classi locali coinvolte siano utilizzati per la ricostruzione delle stesse base extension. Ad esempio, considerando la query seguente:

```
Q_3:  select name, year
      from University_Person
```

```
where faculty = 'CS'
and pay > 30000
```

La fase di *Definizione del Query Plan* individua la classe virtuale C8, l'insieme finale $BE_{opt} = \{6\}$ e quindi le classi locali `University_Worker` e `School_Member`[4]. Queste classi appartengono alla stessa sorgente `UNIVERSITY` ed inoltre vengono utilizzate entrambe per la ricostruzione della base extension 6. Si può quindi migliorare il plan raggruppando le Local Query `Q_31` e `Q_32`, generate dal normale processo di traduzione, nell'unica query `Q_31'`:

```
Q_31:  select firt_name, last_name
       from University_Worker
       where pay > 30000

Q_32:  select firt_name, last_name, year
       from School_Member
       where faculty = 'CS'

Q_31':  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'
```

Al momento queste due semplificazioni al piano di accesso sono due ipotesi di sviluppi futuri.

### 3.2.5 Ottimizzazione semantica locale

Un'altra tipologia di semplificazione, diversa da quella vista in fase di traduzione da Basic Query a Local Query, e' l'*ottimizzazione semantica locale*. Come per l'*ottimizzazione semantica globale* vista in 3.1.2, vengono sfruttate, tramite l'impiego di ODB_Tools, le regole di integritá che in questo caso sono definite sulle singole sorgenti.
Questa fase di ottimizzazione é posta a livello Mediatore, in quanto non tutte le sorgenti potrebbero essere in grado di realizzarla.

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

L'ottimizzazione semantica consente di creare query meno onerose, cioé query la cui esecuzione su di un database locale é migliorata da:

- aumenta la possibilità di utilizzare degli 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 consente di riconoscere se due condizioni sono ridondanti: in questo caso, se quella implicata comporta l'esecuzione di un join, viene eliminata;

- permette di evitare o ridurre l'accesso a dati inutili:

  questo caso è generato da una query che possa essere trasformata in una equivalente su di una sottoclasse;

- può determinare l'accesso a dati senza necessità di valutazione di predicati:

  questa possibilità è realizzata qualora si riconosca che una query sussuma una intera classe dello schema.

Ad esempio supponiamo che sulla classe locale `University_Student`, sia definito un vincolo di integritá espresso dalla suguente rule $ODL_{I^3}$:

```
rule RG2: forall x in University_Student : x.faculty = 'cs'
            then x.tax_fee > 12000
```

porta all'espansione della query `Q` in `Q'`:

```
Q: select name
   from University_Student
   where faculty_name = 'cs'

Q': select name
    from University_Student
    where faculty_name = 'cs'
    and tax_fee > 12000
```

L'aggiunta di questo predicato può essere conveniente purché sia presente nella sorgente un indice sull'attributo `tax_fee`. L'espansione ottenuta porta ad un aumento del costo di analisi dell'interrogazione, ma risultati sperimentali [14] hanno dimostrato che il costo complessivo di esecuzione della query ottimizzata decresce rapidamente all'aumentare del numero di istanze nel database e, mediamente, all'aumentare del numero di query fatte.

## 3.3 Esecuzione della query

L'ultimo passo del *Query Processing* é l' *Esecuzione della Query*: il Query Manager utilizza le informazioni del *plan*, generate in precedenza, per ricomporre i risultati ottenuti dalle sottoquery, presentando all'utente una risposta integrata.
Questa fase si svolge a tre livelli differenti, rappresentati in Figura 3.5: inizialmente vengono eseguite le Local Query, successivamente le Basic Query ed infine la ricomposizione della Global Query.

Con l'ausilio di un DBMS, il Query Manager crea delle tabelle in grado di memorizzare i risultati temporanei degli stadi intermedi della ricostruzione della risposta. In questo modo il Query Manager potrá eseguire le varie operazioni di ricostruzione e fusione attraverso query ( join e outer join) poste su queste tabelle temporanee.
Poiché il database "interno" sará a disposizione, in uno stesso istante, di varie istanze del Query Manager, ognuna delle quali eseguirá delle Global Query, per limitare i problemi dovuti allo spazio disponibile sul database, ogni tabella viene immediatamente eliminata non appena i dati temporanei in essa contenuti sono stati elaborati.

### 3.3.1 Esecuzione delle Local Query

Le Local Query generate in 3.2.3 sono espresse in OQL, sará poi compito del Wrapper tradurre la sintassi OQL in un linguaggio di interrogazione comprensibile alla relativa sorgente.

Le Local Query vengono inviate parallelamente ai Wrapper che le eseguono e restituiscono i risultati momentaneamente memorizzati in relazioni temporanee. Il Query Manager accede a questi dati e, sfruttando le informazioni contenute nel *plan*, deve restituirne una rappresentazione globale, traducendo i valori locali con un processo inverso rispetto a quello visto in precedenza. In questo momento vengono inseriti eventuali valori di default, non presenti all'interno dei risultati locali, ma di cui si era tenuto traccia nel piano. Il Query Manager crea quindi, per ogni Local Query, la corrispondente tabella sul database interno, in cui inserisce tutti i valori ottenuti dall'elaborazione precedente, valori che sono espressi in

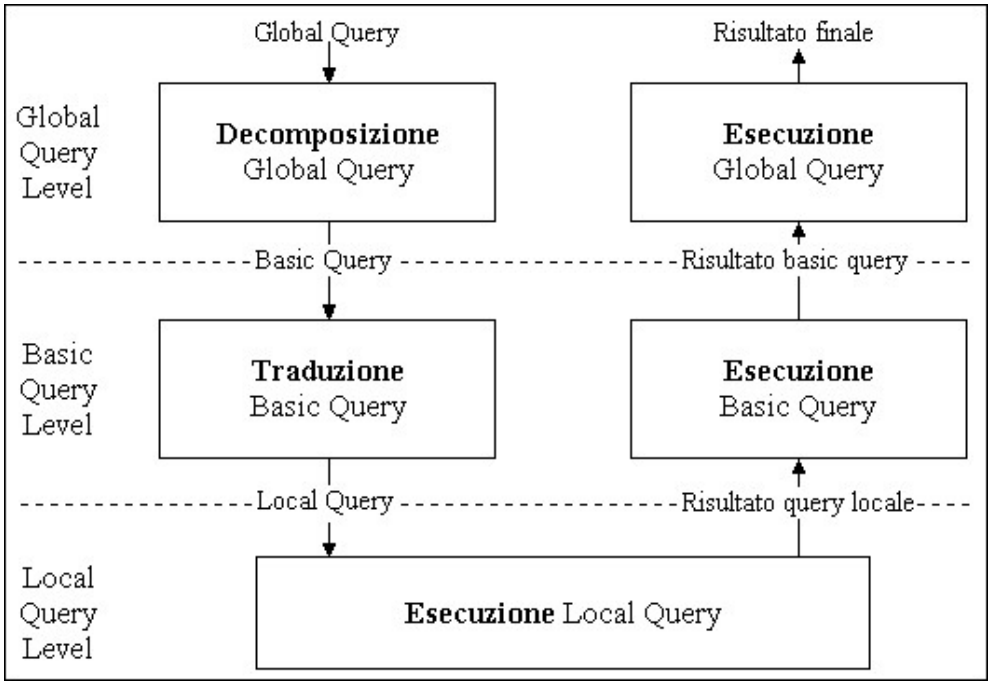


Figura 3.5: I diversi livelli di query processing

termini globali.

Abbiamo visto che per `Q_1`, intodotta in precedenza, le classi locali da interrogare sono:

- `University_Worker`
- `School_Member`
- `CS_Person`

Le Local Query generate dal processo di traduzione descritto in 3.2.3 sono[5]:

```
Q_11: select first_name, last_name, dept_code
      from Universty_Worker
      where  pay > 20000
```

[5]Le prime due classi locali appartengono alla stessa sorgente `UNIVERSITY`, ma in questo caso non si puó effettuare la semplificazione vista in 3.2.4 in quanto i risultati della Local Query `Q_11` vengono utilizzati per la ricostruzione, oltre che della base extension 6 insieme a `Q_12`, anche della base extension 8.

```
Q_12: select first_name, last_name, dept_code
      from School_Member
      where  faculty = 'cs'


Q_13: select name
      from CS_Person
```

Il Query Manager esegue, tramite i Wrapper, le tre Local Query e crea sul database le tre tabelle corrispondenti[6]:

- TABLE University_Worker (name, dept)
- TABLE School_Member (name, dept)
- TABLE CS_Person (name)

All'interno di queste tabelle vengono inseriti i risultati restituiti dalle sorgenti, ma espressi in termini globali, quindi ad esempio la tupla `('Maria', 'Rossi', 'd1')`, restituita dalla classe locale `University_Worker`, viene inserita nella tabella corrispondente nella seguente forma: `('Maria Rossi', 'd1')`.

### 3.3.2 Esecuzione delle Basic Query

Come é evidenziato in figura 3.6, l'esecuzione della Basic Query é composta da tre passi temporalmente successivi:

1. **Ricostruzione di ogni base extension**
   Questo passo viene effettuato attraverso l'unione dei risultati provenienti dalle classi locali. Vengono utilizzate le *regole di join*, che, come abbiamo giá visto in 2.6.1, stabiliscono per ogni coppia di classi locali gli attributi su cui effettuare il join. Questi attributi identificano in modo univoco gli elementi all'interno di entrambe le classi e costituiscono quindi una chiave comune utilizzabile per fondere i risultati provenienti dalle classi locali.
   Durante la fase di *Definizione del Query Plan*, vengono inserite queste informazioni nel piano di accesso e vengono eventualmente aggiunti alla clausola di selezione delle Local Query gli attributi necessari per la fusione di tutte le classi della base extension.

[6] In realtá, come si vedrá nel Capitolo 4, il nome delle tabelle é costruito aggiungendo al nome della classe locale un prefisso che rappresenta l'iteratore univoco associato alla Global Query.

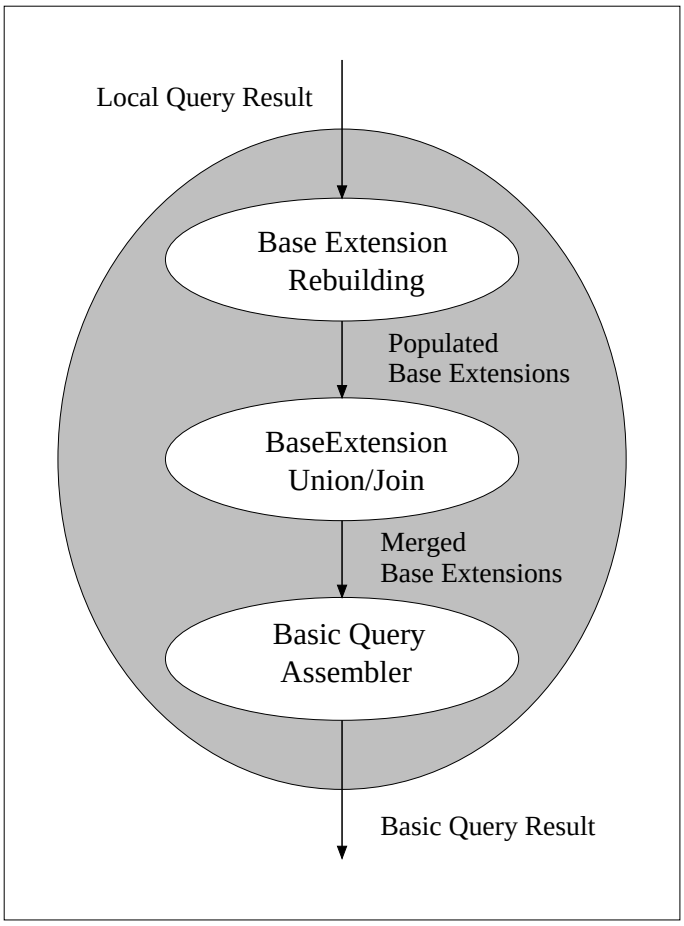


Figura 3.6: Esecuzione delle Basic Query

A livello pratico il Query Manager genera, per ogni base extension da ricostruire, una query che riporta i join da effettuare fra le tabelle contenenti i risultati dell'esecuzione delle Local Query e pone i risultati ottenti in altre tabelle temporanee.
Nella nostra query di esempio `Q_1` l'insieme di base extension ottimo da interrogare é $BE_{opt} = \{6, 8\}$, supponendo che l'attributo globale *name* costituisca la chiave semantica comune alle classi coinvolte, le regole di join individuate nel piano sono le seguenti:

- `R1 (University_Worker,School_Member)` $\Rightarrow k =$ `name`
- `R2 (University_Worker,CS_Person)` $\Rightarrow k =$ `name`

Quindi le query generate dal Query Manager sono:

```
B6: select a.name, a.dept
    from University_Worker as a,
         School_Member as b
    where a.name = b.name

B8: select a.name, a.dept
    from University_Worker as a,
         CS_Person as b
    where a.name = b.name
```

Il Query Manager, dopo aver eseguito queste due query, crea le due tabelle corrispondenti in cui inserire i dati ottenuti.

2. **Fusione delle base extension**
   Si sono quindi ottenute base extension popolate che a questo punto devono essere fuse per ricostruire il risultato della Basic Query. I criteri di fusione sono stati definiti durante la fase di *Definizione del Query Plan*: per ogni coppia di base extension dell'insieme ottimo viene indicato, all'interno del piano, il tipo di fusione da effettuare. Di default viene effettuata l'*unione*, ma nel caso in cui le due base extension ne dominino una terza, é necessario un *outer join* per evitare duplicazioni. In figura 3.7 viene rappresentato un caso di *outer join*: le base extension iniziali sono: $BE_{in} = \{4, 5, 6\}$. Durante la fase di semplificazione, la base extension 5 viene scartata in quanto é dominata sia dalla 4 che dalla 6. Questo peró non é sufficiente per eliminare completamente le duplicazioni, infatti la ricostruzione delle base extension 4 e 6 genera insiemi di entitá parzialmente sovrapposti e la sovrapposizione é rappresentata proprio dall'estensione della base extension eliminata. In questo caso quindi é necessario un *outer join* fra i risultati ottenuti dalla restituzione delle due base extension, ma per fare ció bisogna poter identificare gli elementi comuni, quelli cioé appartenenti alla base extension 5.

   Una proposta di risoluzione del problema é basata, anche in questo caso, sull'uso delle *regole di join*, che individuano per ogni coppia di classi locali una chiave comune, utilizzabile, oltre che per la ricostruzione della singola base extension anche per effettuare eventuali outer join.
   É possibile fondere due base extension se si determina una regola di join che indica un identificatore comune ad una classe locale di una base extension e ad una classe locale dell'altra, o eventualmente usando la chiave semantica di una classe comune ad entrambe.

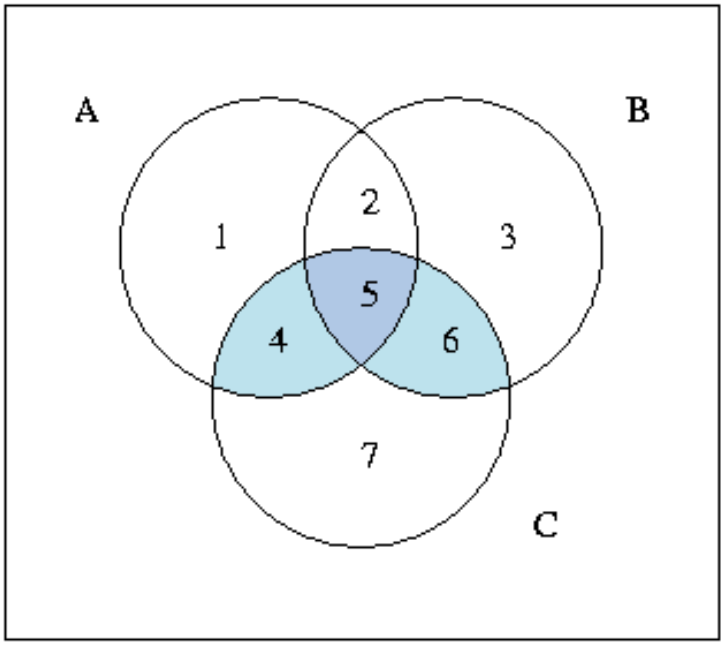


Figura 3.7: Fusione tramite outer join

Nell'esempio riportato in 3.7, il risultato della base extension 4 é fornito dalla fusione delle classi A e C, mentre la base extension 6 é costituita da B e C. Supponiamo che esistano le seguenti possibili regole di join:

- R1 (classe A, classe B) $\Rightarrow$ joineable = false
- R2 (classe A, classe C) $\Rightarrow$ k1
- R3 (classe C, classe B) $\Rightarrow$ k2

Inoltre avendo la classe C in comune :

- (classe C) $\Rightarrow$ k1,k2

Il Query Manager analizza quelle che in definitiva rappresentano le chiavi alternative per le due base extension, con lo scopo di trovare un possibile identificatore comune con cui effettuare l'outer join.

Come giá visto in 3.2.2, questa analisi deve essere effettuata prima della *semplificazione dell'insieme di classi locali*, infatti, nel caso in cui fra

le possibili chiavi, non ne esista una giá compresa fra gli attributi della query, bisogna considerare anche gli attributi aggiuntivi prima di procedere all'eliminazione di classi superflue.

Per la nostra query di esempio le due base extension 6 e 8 dominavano entrambe sia la 4 che la 5, di conseguenza una semplice unione dei loro risultati genererebbe insiemi di entitá parzialmente sovrapposti, quindi il Query Manager deve creare una query che esegua l'*outer join* fra i risultati della ricostruzione delle due base extension, contenuti nelle due tabelle temporanee B6 e B7.
Il piano di accesso indica l'attributo *name*, come chiave utilizzabile per la fusione, in quanto la classe `University_Worker` appartiene ad entrambe le base extension e la sua chiave, *name* appunto, é compresa fra gli attributi della query.
Si ottiene quindi:

```
Q_1outer: select B6.name, B6.dept
          from B6 full join  B8 on (B6.name = B8.name)
```

In tutti i casi in cui non si ha dominazione di due base extension su di una terza, si puó procedere all'*unione* dei risultati senza rischi di ridondanze, avendo eliminato le base extension dominate.

3. **Basic Query Assembler**
   L'ultimo passo che rimane per l'ottenimento del risultato della Basic Query é l'esecuzione della Basic Query Assembler, contenente i predicati non verificati dalle Local Query e la `<select-clause>` della query originaria. Infatti durante la fase di *Definizione del Query Plan*, la query é stata aggiornata aggiungendo eventuali attributi necessari per la fusione delle base extension, per la valutazione dei predicati "esclusi" ed inoltre i risultati delle Local Query possono contenere campi aggiunti per effettuare la ricostruzione delle base extension. Con l'esecuzione della Basic Query Assembler si ottiene quindi il risultato finale della Basic Query, comprendente sia la valutazione completa dei predicati, sia il reperimento delle informazioni effettivamente richieste inizialmente.

   Vediamo la fase di esecuzione della query `Q_2`:
   in fase di *Definizione del Query Plan* viene selezionata la sola base extension 1, quindi il Query Manager genera la query seguente per effettuare

la ricostruzione dei risultati ottenuti dalle due Local Query `Q_21, Q_22` descritte in 3.2.3:

```
B1: select a.name, a.email, a.pay, b.title
    from Research_Staff as a
         Professor as b
    where a.name = b.name
```

La Basic Query Assembler `Q_2A`, riportata in 3.2.3, restituisce il risultato finale della Basic Query:

```
Q_2A: select name, email
      from B1
      where ( pay > 10000 or title = 'full professor')
```

### 3.3.3 Esecuzione della Global Query

L'esecuzione di una Global Query consiste nell'esecuzione della *Global Query Assembler*, introdotta in 3.2.1, sui risultati ottenuti dall'esecuzione delle Basic Query coinvolte.
Oltre ai join fra le classi globali, vengono eseguite tutte le operazioni complesse che, dovendo agire su insiemi di informazioni giá integrate, non sono trattabili a livello Basic Query.

# Capitolo 4

# Progetto e realizzazione del software

La progettazione del Query Manager é stata eseguita con l'obiettivo di realizzare un modulo software componibile ed estendibile, caratteristiche indispensabili nell'ambito di un progetto di ricerca a lungo termine come MOMIS, che si avvale del contributo di numerose persone.
Per questo motivo é stata effettuata una modellazione di tipo object-oriented ed é stato adottato il linguaggio java per la fase di implementazione. Java, essendo un linguaggio completamente orientato agli oggetti, favorisce la modularitá e componibilitá del software ed inoltre, tramite il componente *javadoc*, consente di creare una documentazione completa e di veloce consultazione. In questo capitolo viene descritta la modellazione del Query Manager a livello software, illustrando il progetto presentato in [8] e esteso in [9]; inoltre verrá esaminato il software realizzato per questa tesi, relativo alla gestione della clausola where e all'esecuzione della query.

## 4.1 Organizzazione del software

Il software del Query Manager é organizzato in quattro package che raccolgono le classi java implementate per la realizzazione della fase di *Query Processing*, di cui il Query Manager si occupa.
Vengono ora analizzate le funzionalitá e la struttura di questi package, ricordando che, nel linguaggio java, i package consentono di raggruppare classi per scopo o relazioni di ereditarietá fornendo una maggiore protezione.

### 4.1.1 Package queryman

In questo package sono contenute le classi fondamentali per le fasi di *Definizione del Query Plan* e *Esecuzione della Query*, in particolare é presente la classe

**Query** che attraverso le sue specializzazioni consente di rappresentare le strutture utilizzate nei diversi stadi in cui si articola il *Query Processing*. In Figura 4.1 sono

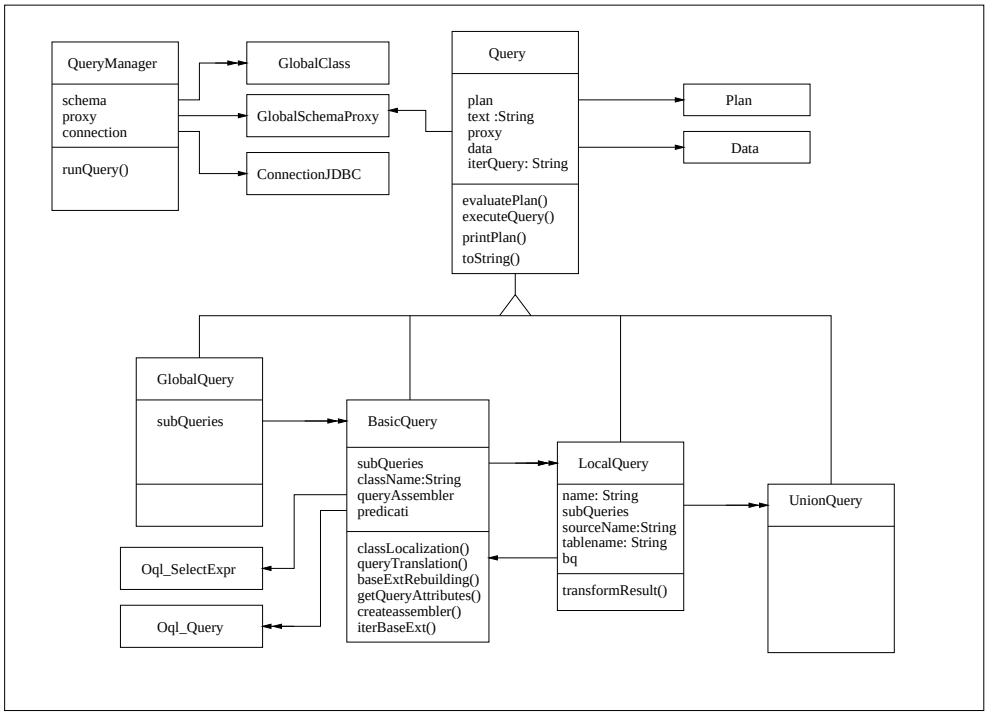


Figura 4.1: Modello ad oggetti del package queryman

rappresentete le: **Global Query**, **Basic Query** e **LocalQuery**, classi che presentano tutte le proprietá e i metodi ereditati dalla classe generica Query, specializzati in base alle esigenze specifiche.
La classe **UnionQuery**, di cui al momento manca l'implementazione, consentirá la gestione di sorgenti semistrutturate, infatti per questa tipologia di sorgenti, puó accadere di dover generare piú query rivolte ad una singola classe locale per poter considerare tutte le possibili rappresentazioni associate ad un unico oggetto.
Il metodo *evaluatePlan()*, la cui realizzazione é descritta in [9], implementa la fase di *Definizione del Query Plan*, il cui risultato é il riempimento dell'attributo *subqueries* con le Query di livello inferiore e della classe **Plan** con le informazioni da utilizzare per la ricomposizione dei risultati. In Figura 4.2 sono rappresentate le classi che implementano le diverse tipologie di informazioni contenute nel piano di accesso, informazioni utilizzate nella fase di *Esecuzione della Query*, la cui implementazione verrá descritta nel dettaglio essendo uno degli argomenti di questa tesi.
Sempre all'interno di questo package é contenuta la classe **QueryManager**, che,

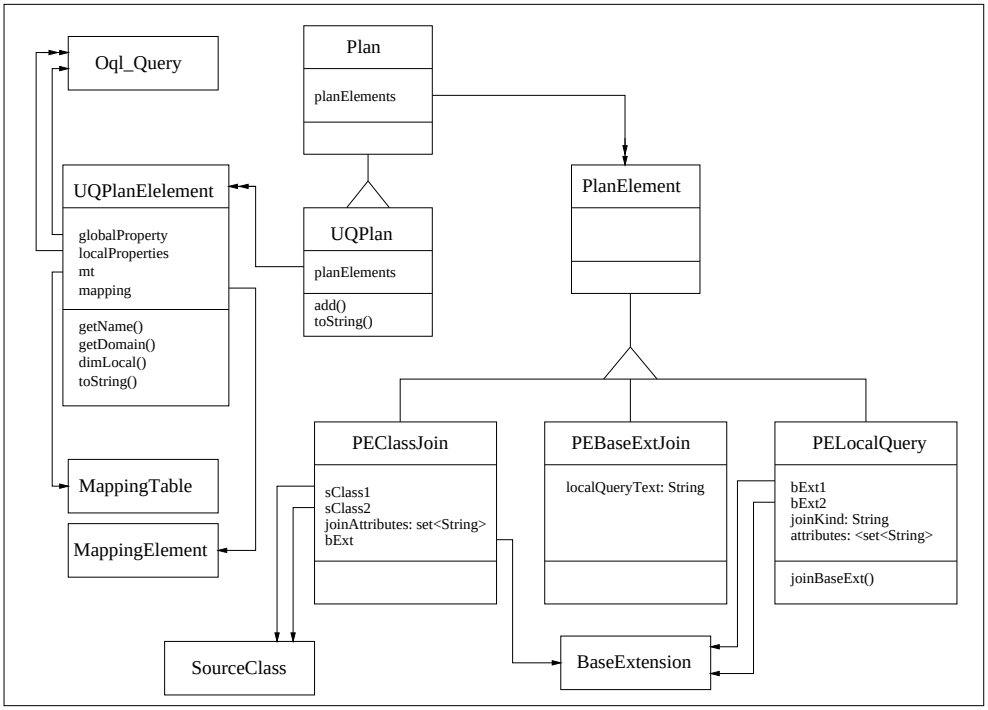


Figura 4.2: Modello ad oggetti della classe Plan

avendo a disposizione uno schema globale, consente la creazione e la gestione delle interrogazioni poste dall'utente.

### 4.1.2 Package oql

Il package *oql* raccoglie le classi utilizzate per fornire un'immagine java della query OQL posta dall'utente, immagine che verrá utilizzata per tutte le elaborazioni successive del Query Manager.
La grammatica del linguaggio OQL prevede un insieme di espressioni che possono essere composte ricorsivamente, per ottenere l'interrogazione finale. Il package *oql* é stato organizzato in una gerarchia le cui classi, rappresentanti i diversi campi della query, ereditano dalla generica classe **Oql_Query**, in questo modo la ricorsione della grammatica OQL viene implementata sfruttando il meccanismo dell'ereditarietá di java.

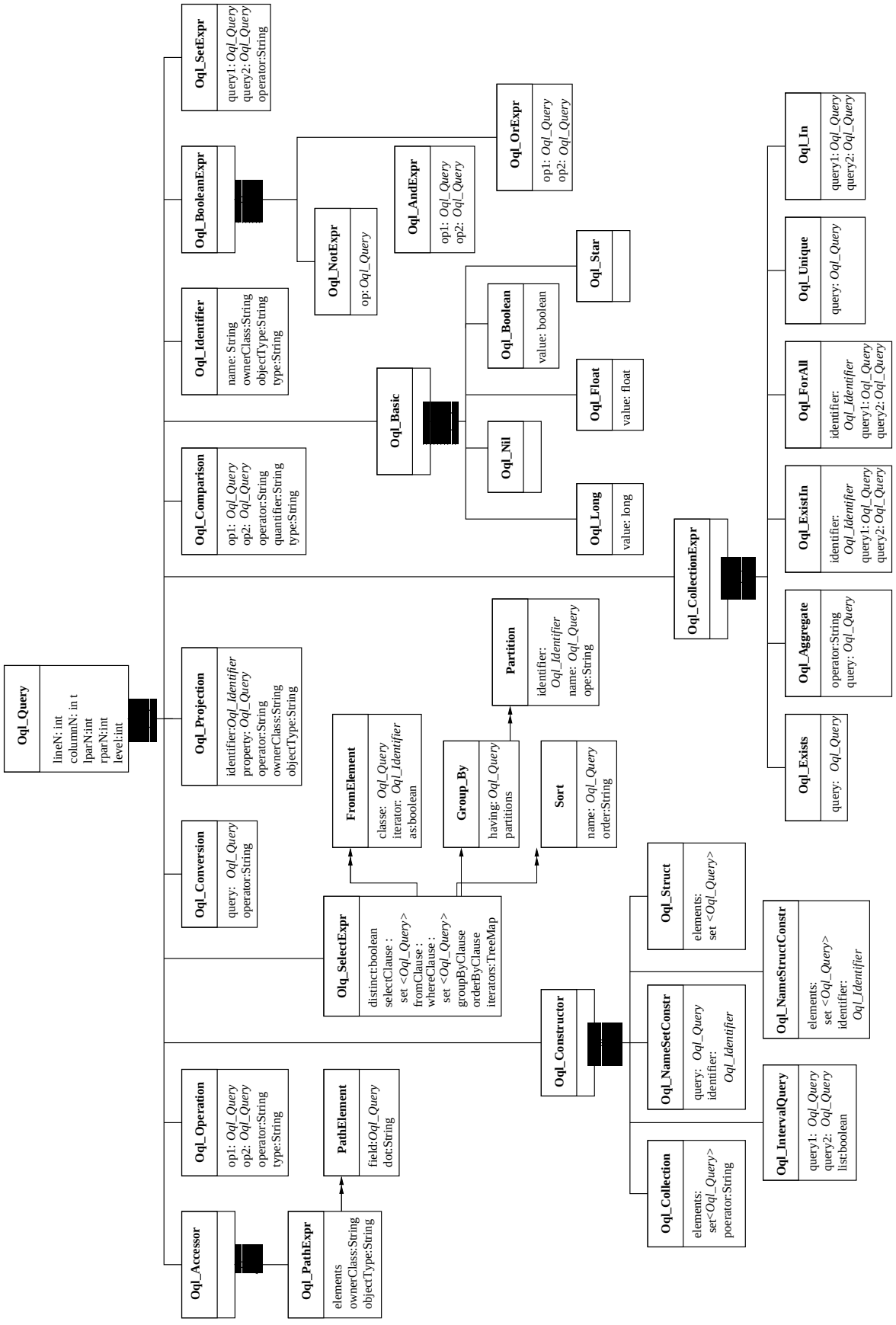


Figura 4.3: Gerarchia di classi per la rappresentazione delle query oql

### 4.1.3 Package globalschema

Questo package contiene le classi che descrivono uno schema globale: **GlobalClass** rappresenta la singola classe globale, mentre **MappingTable**, **BaseExtension** e **ExtensionalHierarchy** ne rappresentano rispettivamente la conoscenza intensionale ed estesionale descritta in 2. Il software implementato in questa tesi

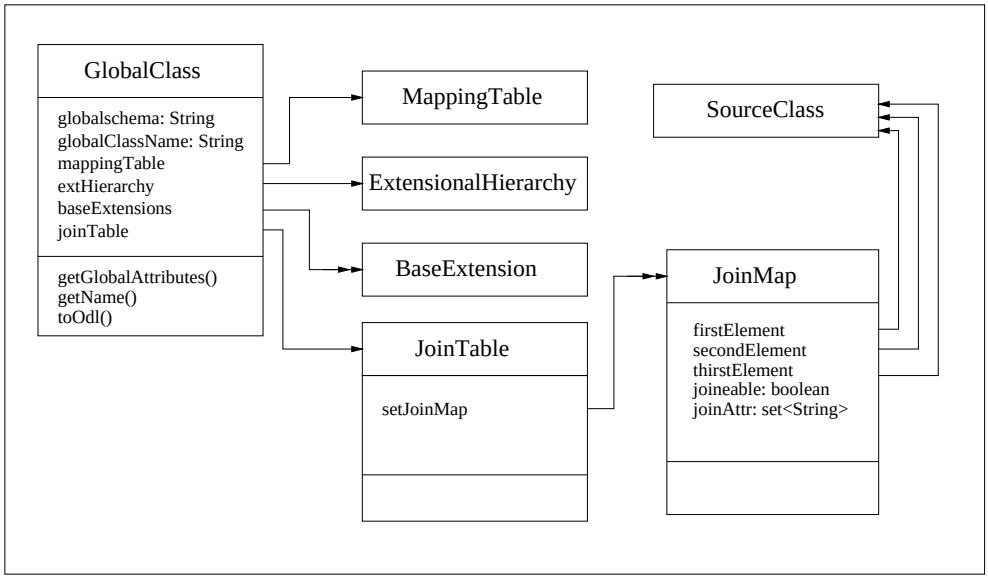


Figura 4.4: Modello ad oggetti della classe GlobalClass

utilizzerá entrambi questi tipi di conoscenza, in particolare le informazioni contenute all'interno della MappingTable per quanto rigurda la gestione della clausola where. La classe **JoinMap** riveste un'importanza fondamentale per la fase di *Esecuzione della Query*, in essa infatti sono contenute le *regole di join*, cioe' le informazioni riguardanti la possibilitá di fusione di due classi locali implementate da **SourceClass**, eventualmente ricorrendo ad una terza classe di collegamento. In figura 4.1.3 é rappresentata la classe GlobalClass con le rispettive informazioni, é riportata anche la classe **JoinTable** che al momento non é ancora stata completata, ma che conterrá il vettore di elementi JoinMap riguardanti le classi locali presenti nella MappingTable corrispondente. Allo stato attuale le *regole di join* sono informazioni definite a livello di singola BaseExtesion.

### 4.1.4 Package utility

Questo package contiene classi che svolgono funzioni di utilitá generale per il Query Manager. Fra le altre vi sono classi che si occupano della gestione dei

messaggi dovuti ad errori o eccezioni durante la fase di elaborazione della query e le classi **Parser** e **OqlAnalizer** che realizzano il parser per il linguaggio OQL e producono l'immagine java della query.

## 4.2 Moduli del Query Manager

Come abbiamo visto in 3, la fase di *Query Processing* puó essere scomposta in tre sottofasi che il Query Manager deve eseguire in sequenza:

1. *Acquisizione della query*;
2. *Definizione del Query Plan*;
3. *Esecuzione della Query*.

Ora verranno presentati i moduli software che eseguono queste tre fasi, fornendo una breve descrizione relativa a quelli implementati in [8] e [9], per poi approfondire l'analisi del software realizzato nel corso di questa tesi.

### 4.2.1 Acquisizione della Query

In Figura 4.5 é rappresentata la fase di Acquisizione della Query e si puó notare che , come giá accennato in 3, all'interno di MOMIS le fasi di parsing e validazione sono invertite rispetto alla teoria dei Mediatori, questo per sfruttare al meglio il software esistente.

- La fase di validazione e ottimizzazione viene svolta dal *Query Optimizer* di ODB_Tools, questo modulo C genera in uscita una struttura dati, sempre in C, che viene usata per produrre un file di testo contenente l'interrogazione ottimizzata.
- A questo punto il modulo java *Query Parser and Validator* riceve in ingresso il file, effettua il parsing della query e produce in memoria centrale una struttura dati che ne rappresenta il contenuto. L'immagine java della query, creata sfruttando le classi del package *oql*, é il punto di partenza per le elaborazioni successive.
- Rimane l'ultimo modulo, introdotto in questa tesi, il *Where Clause Normalization*, che analizza la clausola where della query per porla in forma normale congiuntiva.

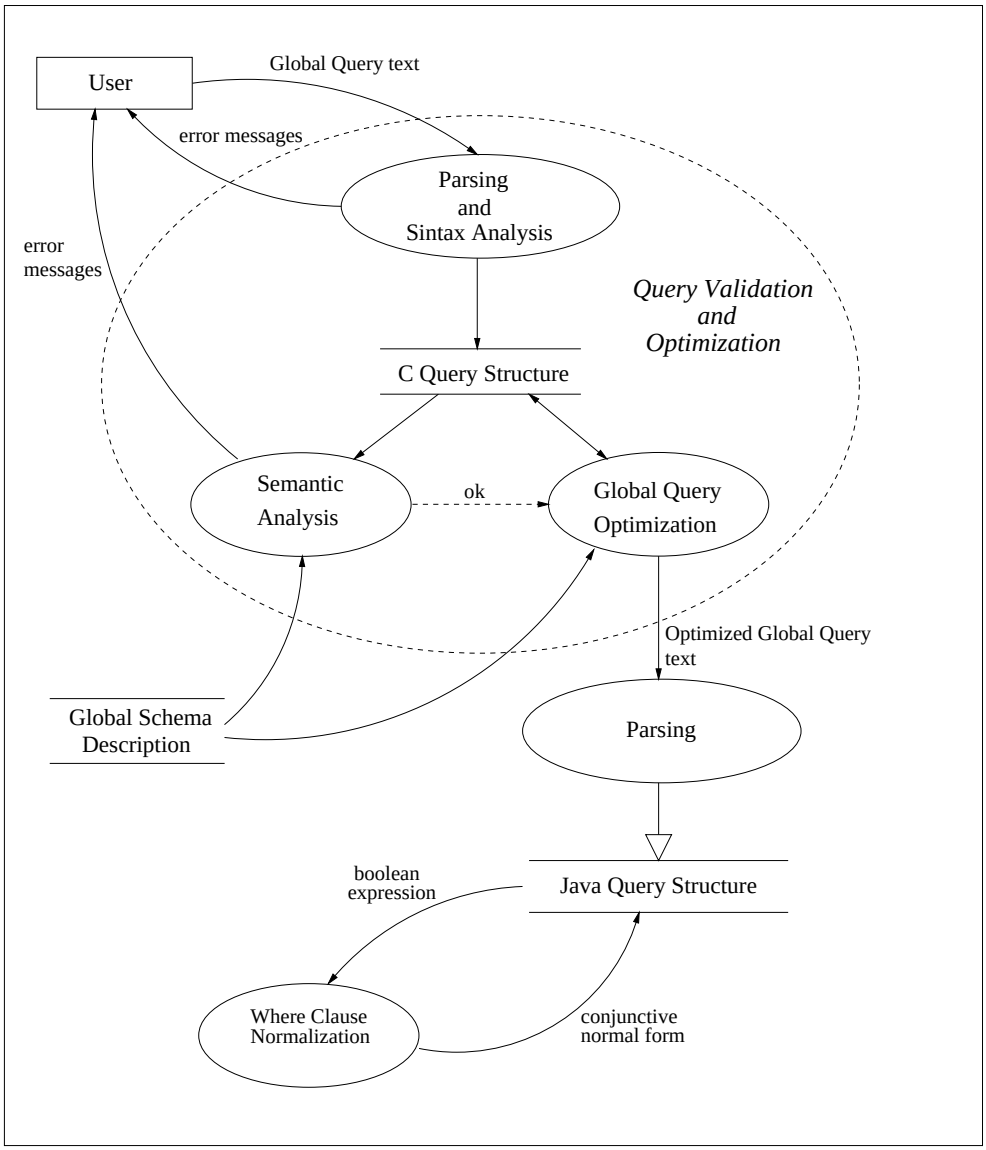


Figura 4.5: Schema di acquisizione di una query

### 4.2.2 Definizione del Query Plan

In Figura 4.6, sono rappresentati i quattro moduli che costituiscono la fase di *Definizione del Query Plan*.

- *Execution Plan Evaluator*
  opera a livello Global Query e fornisce come output il piano di esecuzione, cioé individua le Basic Query da generare e le operazioni per la

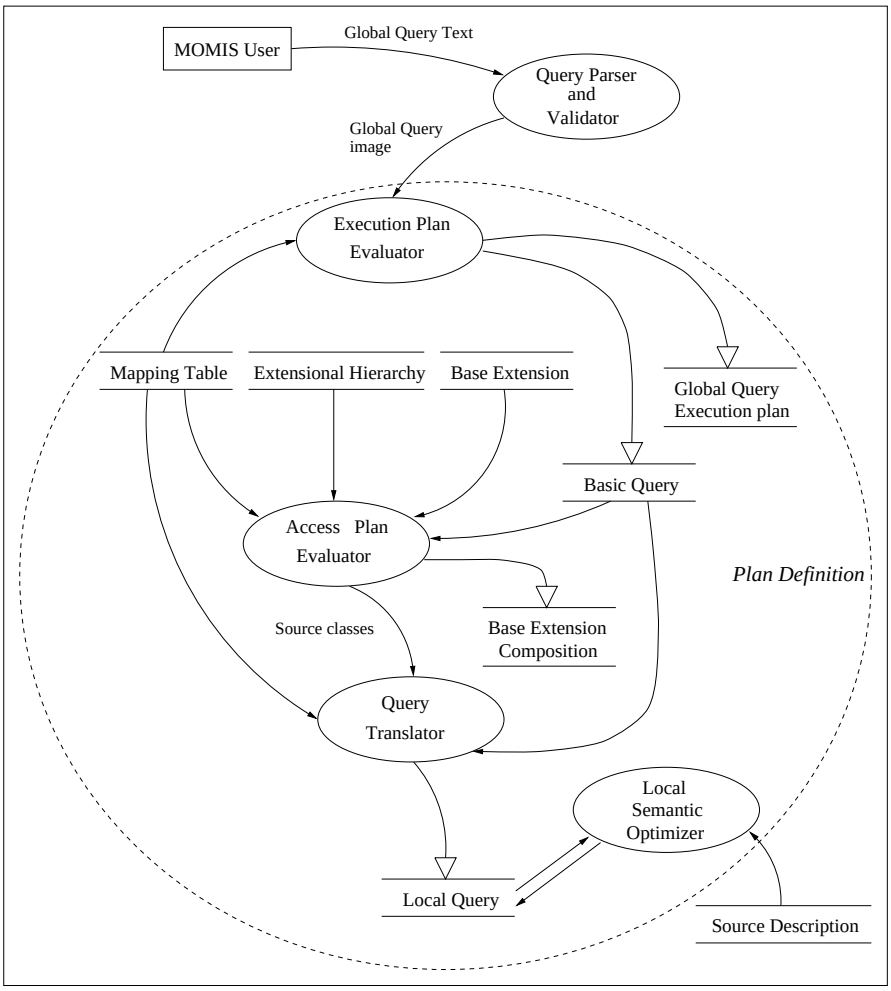


Figura 4.6: Definizione del piano di esecuzione di una Query

ricostruzione dei risultati. In realtá questo modulo non é ancora stato implementato, allo stato attuale sono gestite solo Basic Query, per questo, all'interno del modulo di parsing, viene richiamata la classe **BQChecker** che contolla l'effettiva natura della query e se non é di tipo Basic interrompe l'esecuzione riportando un messaggio di errore.

- *Access Plan Evaluator*
  fornisce il piano di accesso, in pratica definisce per ogni Basic Query le Local Query da inviare alle sorgenti e le informazioni che indicano come ricostruire il risultato.
  In questa tesi é stato aggiunto, rispetto alla versione descritta in [9], un metodo che individua una chiave semantica per la fusione delle Base Ex-

tension sovrapposte. L'individuazione di questa chiave é necessaria per la fase di *Esecuzione della Query*.

- *Query Translator*
  la sua funzione é quella di tradurre la Basic Query nelle varie Local Query da inviare alle sorgenti. La descrizione di questo modulo verrá ripresa nella sezione seguente, per l'aggiunta della gestione della clausola where della query.
- *Local Semantic Optimizer*
  esegue l'ottimizzazione semantica delle Local Query generate in precedenza. Anche in questo caso si tratta di un modulo C di ODB_Tools.

### 4.2.3 Esecuzione della Query

La Figura 4.8 sono raffigurati i moduli che eseguono questa fase, corrispondenti ai tre livelli in cui si articola l'esecuzione della query:

- *Global Query Executor*
  non essendo gestite le Global Query, questo modulo non é ancora stato implementato.
- *Basic Query Executor*
  riguarda l'esecuzione delle Basic Query cioé l'integrazione dei risultati delle Local Query in base ai criteri definiti nel piano di accesso.
- *Local Query Executor*
  elabora i risultati restituiti dai Wrapper per ottenerne una rappresentazione in termini globali.

La descrizione dettagliata del software degli ultimi due moduli verrá fornita piú avanti.

## 4.3 Software per la gestione della clausola where

In questa sezione é riportata la descrizione del software relativo alla gestione della clausola where, durante l'acquisizione ed elaborazione della query.

### 4.3.1 Generazione dell'immagine java della query

Come abbiamo giá visto, la query posta dall'utente é sottoposta inizialmente ad un processo di validazione sintattica e semantica, seguito dalla fase di ottimizzazione gestita dal modulo *"Query Optimizer"* di ODB-Tools.

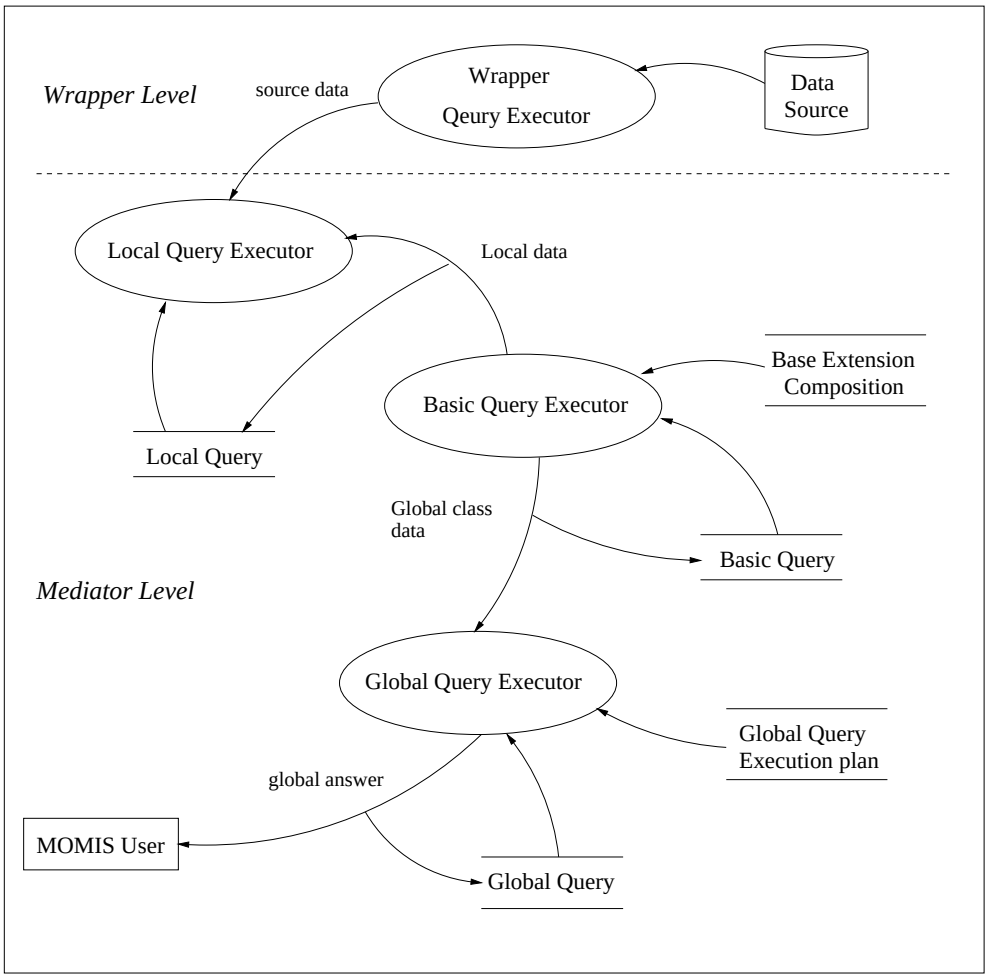


Figura 4.7: Esecuzione del piano

La struttura cosi' generata viene analizzata una seconda volta dalla classe **OqlAnalizer**, la quale impiega la gerarchia di classi del package **oql** per creare un'immagine Java della query, come descritto nel seguito.
Una generica query di tipo Select viene rappresentata nel suo complesso tramite un'istanza della classe **Oql_SelectExpr** e per ogni campo dell'interrogazione viene prodotta la corrispondente struttura Java, specializzazione della classe **Oql_Query**.

I campi che seguono la parola chiave *select* individuano gli attributi di proiezione, questi elementi possono assumere due forme differenti:

- *attributi semplici*
  Questi oggetti sono rappresentati da istanze della classe **Oql_Identifier**;

- *pathexpression*, cioé navigazioni implicite
  Queste strutture vengono rappresentate tramite gli oggetti della classe **Oql_PathExpression**.

La clausola *from* individua la classe globale su cui é posta la query e l'eventuale iteratore; la struttura che rappresenta questo campo della query é un'istanza della classe **FromElement**.

La clausola *where* prevede un insieme di espressioni che possono essere composte ricorsivamente per mezzo delle classi specializzate da Oql_Query. I casi finora gestiti per questa clausola sono:

- *espressioni booleane*
  Sono tre la classi java che rappresentano le corrispondenti espressioni booleane: **Oql_AndExpr**, **Oql_OrExpr**, **Oql_NotExpr**;

- *espressioni di confronto*
  La classe java che le rappresenta é **Oql_Comparison**, i cui operandi possono essere attributi, espressi in forma semplice o come pathexpression, tipi base e operazioni aritmetiche rappresentate dalla classe **Oql_Operation**.

### 4.3.2 Normalizzazione della clausola where

L'ultimo passo della fase di *Acquisizione della Query* consiste proprio nella *normalizzazione della clausola where*, la cui descrizione é riportata in 3.1.3.
É stato inserito all'interno della classe Oql_Query, il metodo **normalForm()**, il quale restituisce sempre un oggetto Oql_Query che rappresenta la trasformazione della struttura iniziale in funzione dei teoremi dell'algebra di commutazione descritti in 3.1.3.
Sfruttando la proprietá di *overriding* del linguaggio java, ogni classe **Oql_BooleanExpr** implementa il metodo normalForm() in base alle trasformazioni richieste, mentre per una generica classe Oql_Query, questo metodo restituisce l'oggetto chiamante senza effettuare elaborazioni. All'interno della classe Oql_Comparison viene gestita la negazione di un predicato di confronto attraverso il metodo **notComparison()**, che ne restituisce la condizione complementare.
All'interno della classe **GlobalQuery**, viene richiamato il metodo di normalizzazione dalla generica struttura Oql_Query, che contiene inizialmente la clausola where. La trasformazione delle espressioni innestate avviene ricorsivamente e il risultato é un prodotto di somme che rappresenta appunto la **forma normale congiuntiva** desiderata.

### 4.3.3 Generazione della Basic Query Assembler

In 3.2.3 é descritta a livello teorico la scomposizione della clausola where e la creazione della Basic Query Assembler.
L'implementazione di questo processo é avvenuto tramite il metodo **createassembler()**, inserito nella classe **BasicQuery** del package *queryman*.
Questo metodo riceve in ingresso diversi parametri:

- la *clausola where* della Basic Query, in forma normale congiuntiva;
- il *vettore ottimo di classi locali* a cui rivolgere le Local Query;
- la *Mapping Table* contenente le informazioni intensionali;
- la struttura contenente la *query iniziale*;

  Questo metodo riempie due strutture aggiunte fra le proprietá della classe BasicQuery:

- il vettore dei *predicati* da tradurre;
- la struttura **Oql_SelectExpr** rappresentante la *Basic Query Assembler*;

Il metodo analizza i fattori in and della clausola where normalizzata: se il predicato é mappato da almeno una delle classi locali passate come parametro lo inserisce nel vettore di *predicati* da tradurre, in caso contrario lo aggiunge alla clausola where della *Basic Query Assembler*.
Abbiamo visto che, per poter eseguire correttamente la fase di integrazione dei risultati, la struttura contenente la Basic Query iniziale deve essere aggiornata con gli attributi aggiuntivi presenti nella clausola where della Basic Query Assembler, in modo da poter recuperare dalle sorgenti locali le informazioni necessarie per l'esecuzione finale.

Per effettuare il controllo degli attributi contenuti all'interno di un generico oggetto Oql_Query, é stato creato il metodo **getAttributes()**.
Questo metodo, implementato in ognuna delle classi specializzate da Oql_Query, restituisce un vettore contenente i nomi degli attributi presenti all'interno dell'oggetto; inoltre, ricevedo in ingresso la Mapping Table e il nome della classe locale, il metodo é in grado di riconoscere eventuali iteratori associati alla query o alla classe globale e quindi considera solo i nomi effettivi degli attributi.

### 4.3.4 Traduzione della query

Nella fase di generazione delle Local Query, la Basic Query deve essere *"riscritta"* in funzione degli schemi locali, viene quindi trasformata ogni struttura dell'interrogazione di partenza.

Anche in questo caso sfruttando i meccanismi di specializzazione e overriding dei metodi presenti nel linguaggio Java, ogni classe specializzata da *Oql_Query* implementa la propria versione del metodo **translateQuery()** che restituisce le strutture dati rappresentanti la traduzione locale dell'oggetto stesso. Questo metodo restituisce infatti un'istanza della classe **TransOutput** in cui vengono riportati gli elementi trasformati da inserire nella query locale. Gli elementi necessari per la trasformazione sono:

- la *Mapping Table* contente le informazioni sul mapping fra attributi globali e locali;
- la *classe locale* considerata, che individua una specifica riga della Mapping Table;
- la *struttura della Local Query*, contenente il risultato delle trasformazioni giá effettuate.

Ogni elemento della Mapping Table é un'istanza della classe **MappingElement** che a sua volta é specializzata nelle sottoclassi rappresentanti i diversi tipi di mapping consentiti. Ognuna di queste classi implementa una propria versione del metodo **toQuery()**, che restituisce la versione locale dell'attributo.

É importante che il Query Manager gestisca, oltre alla traduzione dei singoli attributi, anche quella di interi fattori booleani in cui l'attributo e' inserito.
L'implementazione del metodo translateQuery() all'interno della classe **Oql_Comparison** é finalizzata all'ottenimento di una corretta traduzione dei predicati. I due operandi di un'espressione di confronto sono tradotti ricorsivamente e col risultato di queste traduzioni viene creato il nuovo predicato. Per gestire i casi in cui uno dei due elementi é un'istanza della classe **Oql_Basic**, é stata introdotta una nuova versione del metodo translateQuery() che richiama **toComparison()** specializzato per ogni tipologia di MappingElement, in modo da fornire una traduzione dell'operando base in funzione del tipo di mapping del secondo elemento del predicato. Consideriamo ad esempio il seguente predicato di confronto:

```
name = 'Maria Rossi'
```

e la relativa traduzione per la classe locale `SU.School_Member`. In questo caso il MappingElement é un elemento della classe **AndMapping** che mappa l'attributo globale *name* con i due attributi locali *first_name, last_name*. L'elemento restituito dalla traduzione é:

```
(first_name = 'Maria') and (last_name = 'Rossi')
```

La traduzione dello stesso predicato in funzione della classe locale `SCS.Student` deve fornire:

```
name = 'Maria Rossi'
```

in quanto il MappingElement corrispondente é un'istanza della classe **SimpleMapping**.

Per ogni Local Query da generare, dopo aver effettuato la traduzione degli attributi della `<select-clause>`, viene chiamato il metodo **translateWhereElement()** che analizza uno ad uno gli elementi contenuti nel vettore di *predicati* della Basic Query corrispondente. Nel caso in cui tutti gli attributi presenti in questi fattori siano mappati dalla classe locale, viene richiamato il corrispondente metodo di traduzione translateQuery() che restituisce il predicato trasformato da aggiungere in *and* alla clausola where della Local Query considerata.

## 4.4 Implementazione della fase di esecuzione della query

Come abbiamo giá visto in 4.8, durante la fase di *Esecuzione della Query*, il Query Manager utilizza le informazioni contenute nel *plan* per integrare i risultati restituiti dalle sorgenti, fornendo una risposta corretta e quanto piú possibile completa e minima alla query posta dall'utente.

Le operazioni che il Query Manager deve eseguire in questa fase sono costituite da query sql da porre sui dati restituiti dai Wrapper sotto forma di tuple, organizzate in formato tabellare.
Il sistema MOMIS sfrutta un DBMS[1] relazionale, DB2, per facilitare il compito del Query Manager, consentendo di mantenere i dati restituiti dalle elaborazioni intermedie in tabelle su un database "interno" e di porre su di esse gli statement sql necessari.
Nel seguito diamo una breve descrizione di JDBC, lo strumento java che consente una facile interazione di applicazioni e applet java con i database.

### 4.4.1 Driver JDBC e package sql

L'accesso in via remota ad un database ne consente l'uso condiviso a piú utenti, ognuno dei quali necessita di un *client* di database. I client necessitano dei driver per mandare istruzioni sql al *server* e per ricevere da questo i risultati.

[1]Un DataBase Management System é un *server* di database, cioé un programma software che gestisce i database organizzandoli e gestendone un accesso condiviso.

Il package *java.sql* e i *driver* JDBC [40] semplificano notevolmente l'accesso e l'interazione con i database ad applicazioni ed applet java, consentendo un approccio indipendente dal server.
Vediamo alcune delle classi di questo package che forniscono gli strumenti necessari per aprire una connessione, eseguire istruzioni sql e lavorare con gli insiemi ottenuti come risultato.
La classe *DriverManager* consente di effettuare una connessione utilizzando il metodo *getConnection()*, ottenendo un oggetto in grado di implementare i metodi per interagire col database utilizzando la connessione stabilita. Il metodo si presenta nella forma: *getConnection(String url)*, dove l'argomento String deve specificare la URL del database, costituita da *jdbc:subprotocol:subname*.
L'interfaccia *Statement()* definisce i metodi per l'esecuzione degli statement sql, tra cui *executeQuery(), executeUpdate()* ed *execute()*.
I risultati di una query posta sul database sono restituiti come una tabella di dati, a cui é possibile accedere attraverso le interfacce *ResultSet* e *ResultSetMetaData*.

### 4.4.2 Uso di DB2 all'interno di MOMIS

L'uso di un DBMS interno al sistema MOMIS, consente la gestione di un database, chiamato "momis", su cui effettuare le elaborazioni dei risultati intermedi durante la fase di *Esecuzione della Query*.
Le varie istanze del **QueryManager** sono dei *client*, che accedono in via remota al database utilizzando i *driver JDBC*, per inviare istruzioni sql al *server* DB2 e ricevere da questo i risultati.
Un problema legato all'accesso condiviso di piú utenti, riguarda l'identificazione univoca delle tabelle: ogni istanza della classe QueryManager utilizza delle tabelle in cui inserire i dati temporanei, é quindi necessario che il nome di queste tabelle sia univoco all'interno del database. Per fare ció ad ogni <table-name> viene aggiunto un prefisso che rappresenta l'iteratore univoco associato alla Global Query, iteratore ottenuto col metodo **hashCode()** della classe **Object** di java ed ereditato dalle sottoquery.

All'interno del package *queryman*, sono state create alcune classi che richiamano i metodi visti in precedenza.

**Classe ConnectionJDBC**

Il metodo costruttore di questa classe inizializza la variabile statica *_connection* con un'oggetto che implementa l'interfaccia *Connection*.
L'URL utilizzato come argomento di *getConnection()* é: `"jdbc:db2:momis"`, mentre il nome del driver é : `"COM.ibm.db2.jdbc.app.DB2Driver"`.
Vediamo nel dettaglio i metodi messi a disposizione da questa classe che

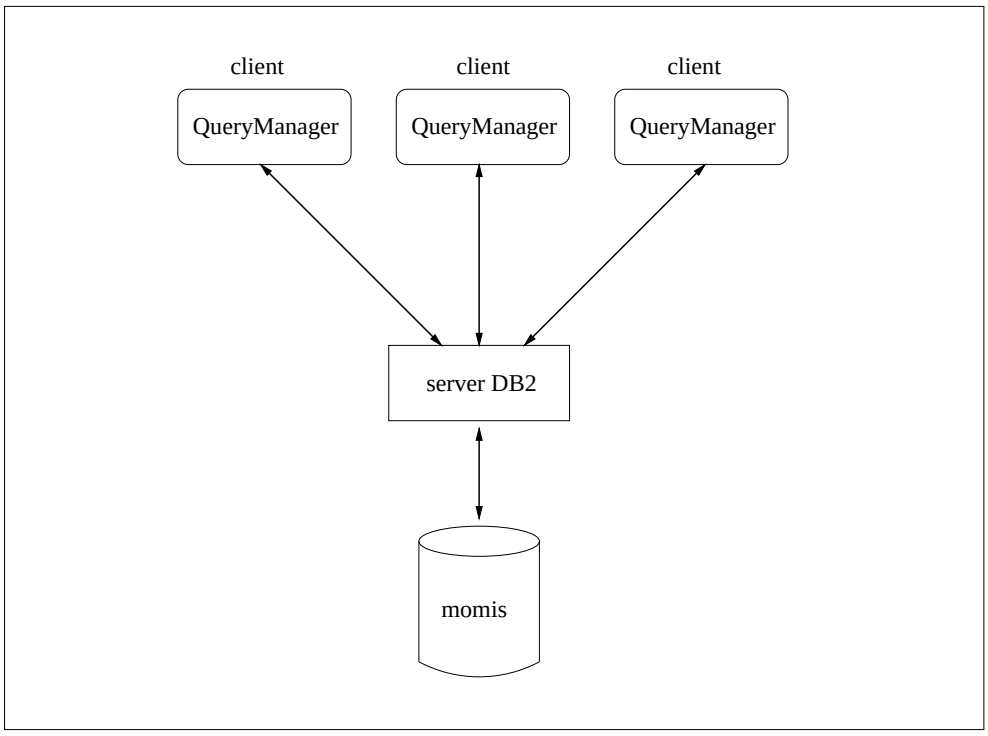


Figura 4.8: Query Manager e server DB2

consentono di eseguire generici statement sql e di visualizzare i risultati:
METODI

- *getStatement()*: crea semplicemente un oggetto Statement richiamando il metodo createStatement() dell'interfaccia Connection.
- *closeStatement()*: chiude un oggetto Statement passato come parametro.
- *executeStatement()*: esegue il generico statement sql passato come parametro sotto forma di stringa . Questo metodo viene utilizzato dal Query Manager per eseguire le CREATE E DROP TABLE.
- *executeQuery()*: esistono due versioni di questo metodo, la prima esegue la query passata come unico parametro e ne visualizza il risultato, la seconda riceve come parametro anche il nome della tabella in cui inserire direttamente il risultato dell'interrogazione. In pratica lo statement eseguito é:

```
INSERT INTO <table-name>
   SELECT <select-clause>
```

```
FROM <from-list>
WHERE <where-clause>
```

Questa seconda forma é molto utile al Query Manager, in quanto le fasi di elaborazioni intermedie prevedono di mantenere i dati in memoria per le elaborazioni successive.

- *dropTables()*: questo metodo elimina le tabelle il cui nome é contenuto nel vettore passato come parametro. In pratica per ogni elemento viene creata un'istanza della classe **DropTable** che vedremo fra poco.
- *display()*: visualizza i dati contenuti nell'oggetto ResultSet passato come parametro.
- *closeConnection()*: chiude la connessione.

**Classe Create**

Implementa l'istruzione di creazione di una tabella nel linguaggio SQL:

```
CREATE TABLE <table-name>
 ( <column1-name>,<domain1-name>,
   <column2-name>,<domain2-name>,
   ...
   <columnN-name>,<domainN-name>)
```

PROPRIETÁ

- *tableName*: indica il nome della tabella;
- *columns*: vettore contenente i nomi delle colonne della tabella;
- *domains*: vettore contenente i domini delle colonne;
- *like*: nome dell'eventuale tabella da cui copiare la struttura, se questo campo non é una stringa nulla, viene eseguito lo statement:

```
CREATE TABLE <table-name> LIKE <liketable-name>
```

COSTRUTTORE

- *Create*: esistono tre versioni del metodo costruttore, una che prevede come unico argomento il nome della tabella, il secondo che necessita anche dei vettori columns e domains da inizializzare ed infine l'ultima versione che prevede il nome della tabella di like.

METODI

- *toString()*: restituisce il contenuto dell'espressione SQL sotto forma di una stringa, che puó essere passata ai metodi della classe ConnectionJDBC per l'esecuzione.

**Classe Insert**

Implementa gli statement di inserimento:

```
INSERT INTO <table-name> VALUES ('valore11',...,'valore1N'),
                                ('valore21',...,'valore2N'),
                                  ...
                                ('valoreN1',...,'valoreNN')
```

PROPRIETÁ

- *tableName*: indica il nome della tabella su cui effettuare l'inserimento;
- *values*: vettore i cui elementi rappresentano i valori da inserire ;
- *domColumns*: vettore che contiene i domini delle colonne della tabella su cui effettuare l'inserimento. Questo campo é necessario in quanto DB2 accetta l'inserimento di stringhe e singoli caratteri fra apici, a differenza degli altri tipi base. Quindi il Query Manager deve essere in grado di individuare il dominio della colonna prima di procedere all'inserimento.

COSTRUTTORI

- *Insert()*: esistono tre versioni del metodo costruttore che creano rispettivamente un'istanza vuota, un'istanza inizializzando il nome della tabella e infine un'istanza costruita a partire dall'oggetto di tipo *Create* passato come parametro. Quest'ultimo caso permette di avere a disposizione i domini delle colonne della tabella considerata.

METODI

- *toString()*: anche in questo caso viene creata la stringa contenente il testo dello statement di inserimento.

**Classe DropTable**

Implementa l'istruzione SQL che consente l'eliminazione di una tabella:

```
DROP TABLE <table-name>
```

Ricordiamo infatti che la distruzione delle tabelle contenenti i dati temporanei é necessaria, in quanto il numero di tabelle che possono essere contenute all'interno del database é limitato dal software che costituisce il server e dallo spazio fisico di memorizzazione disponibile.
Per questo motivo il Query Manager si deve occupare, terminata una fase di elaborazione intermedia, dell'eliminazione delle tabelle giá utilizzate.
PROPRIETÁ

- *tableName*: che indica il nome della tabella da eliminare.

COSTRUTTORI

- *DropTable()*: crea un'istanza della classe inizializzando il campo *tableName* col nome passato come parametro.

METODI

- *toString()*: restituisce l'istruzione in una stringa.

Le classi appena descritte forniscono gli strumenti di ausilio al Query Manager per la fase di *Esecuzione della Query* descritta nelle sezioni successive.
In sostanza quando viene creata un'istanza della classe **QueryManager**, si crea un'oggetto connectionJDBC che apre una connessione inizializzando i *driver JDBC*. Il metodo **runQuery()** richiede come argomento la query posta dall'utente sotto forma di stringa e crea una **GlobalQuery** passando come parametri la connessione, il testo della query e il **GlobalSchemaProxy**, che permette di risalire allo schema globale su cui é posta la query. Il metodo restituisce un oggetto di tipo **MomisResultSet** che rappresenta il risultato finale della Global Query.
Vediamo nel dettaglio l'implementazione della fase di esecuzione svolta a livello Basic Query e Local Query.

### 4.4.3 Esecuzione delle Local Query

Ogni **LocalQuery** generata deve essere inviata alla sorgente corrispondente e il risultato ottenuto va memorizzato in una relazione temporanea.
L'obiettivo fondamentale di questo passo é la traduzione dei valori restituiti in termini locali, in funzione dello schema globale su cui é posta la query. Per effettuare questa trasformazione il Query Manager dovrá seguire le informazioni contenute in **UQPlan** con un procedimento inverso rispetto a quello seguito nella fase di *Definizione del Query Plan*.

L'implementazione di queste attivitá avviene mediante il metodo **executeQuery()** che esegue i seguenti passi:

1. richiama, utilizzando il *GlobalSchemaProxy*, il Wrapper che corrisponde alla sorgente considerata;

2. richiama il metodo *runQuery()* del Wrapper passandogli come parametro il testo della Local Query;

3. ricevuti i risultati dalla fonte locale, sotto forma di un oggetto **MomisResultSet**, deve fornirne una rappresentazione globale. Il metodo **transformResult()** riceve come parametro l'oggetto *MomisResultSet* e ne effettua la traduzione sfruttando le informazioni contenute negli **UQPlanElement**.
   Le proprietá di questa classe rappresentano la *globalProperty*, vale a dire l'attributo globale corrispondente, il vettore di *localProperties*, la *MappingTable* ed il *MappingElement*. I metodi in essa implementati restituiscono il nome dell'attributo globale, il suo dominio e il numero di attributi locali corrispondenti.
   Queste informazioni vengono utilizzate da *transformResult()* per creare sul database una tabella in grado di contenere i risultati dell'elaborazione: il nome della tabella é formato dell' iteratore univoco associato alla Global Query e dal nome completo della classe locale, mentre la struttura corrisponde agli attributi globali con i rispettivi domini.
   A questo punto vengono esaminate le righe dell'oggetto MomisResultSet e, per ogni attributo globale, vengono recuperati i corrispondenti valori locali sotto forma di stringhe, passate come parametri al metodo **toResult()** specializzato per ogni tipologia di mapping. Il metodo restituisce una stringa che rappresenta la versione globale dei risultati, inseriti tramite un'istanza della classe *insert*;

4. comunica alla Basic Query il termine dell'elaborazione aggiungendo il nome della tabella che contiene i dati temporanei al campo di tipo **Data**;

### 4.4.4 Esecuzione delle Basic Query

Come abbiamo visto in 3.3.2 l'esecuzione di una Basic Query avviene in tre passi successivi, di cui analizziamo l'implementazione che, anche in questo caso, avviene tramite il metodo **executeQuery()**.

**Ricostruzione di ogni base extension**

Le indicazioni su come fondere i risultati restituiti dalle Local Query sono contenute negli elementi **PEClassJoin** del *plan*. Le istanze di questa classe indicano quali sono gli attributi da utilizzare per effettuare il join fra due classi locali in una certa base extension. Vediamo le azioni eseguite dal Query Manager in questa fase:

1. individua le base extension da ricostruire, ad ognuna associa un *nome* univoco, costituito dall'iteratore della GlobalQuery e dal numero di riferimento della base extension ed infine crea sul database la tabella corrispondente;
2. per ogni base extension, sfruttando le informazioni contenute nei corrispondenti elementi PEClassJoin, genera un'interrogazione SQL, cioé una struttura **Oql_SelecExpr** contenente nella clausola FROM le classi locali da fondere (e quindi i nomi delle tabelle create nella fase precedente), e nella clausola WHERE i predicati di join.
   Il medodo **addJoin()** della classe Oql_SelectExpr aggiorna la query aggiungendo i predicati di join costruiti avendo a disposizione due elementi presenti nella clausola FROM, eventualmente con i corrispondenti iteratori, e dal vettore di attributi necessari;
3. esegue la query SQL sul database e il risultato restituito, rappresentante i dati della ricostruzione della base extension, viene inserito nella tabella corrispondente. Questa operazione é fatta automaticamente dall'apposito metodo *executequery()* della classe ConnectionJDBC visto in precedenza;
4. elimina le tabelle contenenti i dati ottenuti dall'elaborazione delle Local Query, in quanto non servono alle fasi successive. Il Query Manager non fa altro che invocare il metodo *dropTables()* della classe ConnectionJDBC, passandogli come parametro il vettore di elementi del campo *data*.

**Fusione delle base extension**

In questa fase vengono sfruttate le indicazioni contenute negli elementi **PEBaseExtJoin**, che contengono il tipo di fusione da effettuare per ogni coppia di base extension.
All'interno di questa classe é stato aggiunto il vettore *attributes*, contenente gli attributi da utilizzare per la fusione, ed il metodo **joinBaseExt()** per determinarli. Il procedimento eseguito per individuare gli identificatori comuni, é quello descritto in 3.3.2, ricordando di aggiungere eventuali attributi mancanti. Vediamo i passi svolti dal Query Manager:

1. scorre gli elementi **PEBaseExtJoin** per individuare le base extension da fondere in outer join e quelle da unire semplicemente. In questa fase in realtá il Query Manager si occupa solo della prima tipologia di fusione, creando la query SQL in funzione delle base extension coinvolte e dei predicati di join da costruire;
2. per ogni outer join da effetture crea una struttura **Oql_SelectFullJoin**,

specializzazione della classe Oql_SelectExpr, che descrive la seguente interrogazione SQL:

```
SELECT <selec-clause>
FROM a FULL JOIN b on (a.attributes = b.attributes)
WHERE <where-clause>
```

3. il risultato dell'interrogazione viene inserito nella tabella corrispondente.

**Basic Query Assembler**

Per ottimizzare l'implementazione del software, la Basic Query Assembler non viene eseguita sui risultati della fusione delle base extension, infatti per poter effettuare l'UNIONE :

```
<subquery> UNION <subquery> UNION ... UNION <subquery>
```

é necessario che le due `<subquery>` abbiano lo stesso schema. Poiché la `<select-clause>` della Basic Query Assembler, come abbiamo visto in 3.3.2, contiene le informazioni effettivamente richieste dall'utente, la sua struttura viene utilizzata per costruire le `subquery` su cui effettuare l'UNIONE. A questo livello il Query Manager ha i seguenti compiti:

1. genera, partendo dalla struttura della Basic Query Assembler, la subquery da inviare sia alle tabelle contenenti i dati delle base extension non sovrapposte sia a quelle contenenti i risultati della fusione in outer join;
2. esegue la query finale costituita dalle `UNION` delle `subquery` generate al passo precedente, ottenendo il risultato della fase di esecuzione della Basic Query;
3. elimina tutte le tabelle temporanee utilizzate in questa fase.

## 4.5 Ipotesi di sviluppi futuri

Come piú volte ricordato, allo stato attuale non sono gestite le Global Query, quindi i prossimi sforzi nell'ambito dello sviluppo del software dovrebbero essere orientati alle Global Query.
Questo comporta la realizzazione dei due moduli citati in 4.2:

- *Execution Plan Evaluator*: preposto alla decomposizione da Global a Basic Query ed alla definizione delle informazioni rigurdanti la ricomposizione dei risultati;

- *Global Query Executor*: preposto alla fase di integrazione che, partendo dai risultati restituiti dalle Basic Query, genera la risposta finale alla query posta dall'utente.

Un altro ambito di sviluppo riguarda la gestione di dati semistrutturati [41, 42]. Nelle sorgenti semistrutturate non é definito a priori nessuno schema: i dati contenuti in queste sorgenti si possono definire auto-descrittivi, in quanto le informazioni associate allo schema si possono estrarre direttamente dai dati stessi.
In previsione di questi sviluppi futuri, é stata definita la classe **UnionQuery**, in grado di gestire tutte le possibili rappresentazioni associate ad uno stesso oggetto, classe di cui al momento manca una effettiva implementazione.

# Conclusioni

Come abbiamo visto nel Capitolo 1, il sistema MOMIS si prefigge l'obiettivo di realizzare un mediatore in grado di integrare sorgenti eterogenee di dati, generando una vista globale che l'utente puó interrogare senza possedere un'effettiva conoscenza delle diverse sorgenti.

Questa tesi si é concentrata sul componente Query Manager, proseguendo il progetto presentato in [8] e esteso in [9]. In particolare é stata implementata la gestione della clausola where, sia nella fase iniziale di acquisizione, sia nella fase di generazione delle Local Query. Il Query Manager pone la clausola where, inizialmente rappresentata da un'espressione booleana innestata, in forma normale congiuntiva e successivamente la analizza ricorsivamente per individuare i predicati irrisolti creando la Basic Query Assembler. Segue la fase di traduzione delle Local Query e la ricomposizione del risultato finale.

Per riuscire nell'intento si sono studiate e individuate le varie sottofasi in cui é articolato il Query Processing (Capitolo 3), si é analizzato il software esistente e lo si é integrato con le nuove funzionalitá implementate (Capitolo 4).
Approssimativamente sono state prodotte 2800 linee di codice commentate, implementate utilizzando la versione 1.2 del Java Development Kit della Sun (jdk1.2) disponibile sul Web presso http://java.sun.com. Il software relativo al Query Manager é presente nella directory /export/home/progetti.comuni/Momis/prototype/server del server Sparc20 presso il diparimento di Scienze dell'Ingegneria.
I prossimi sviluppi nella progettazione del Query Manager dovranno essere rivolti alla gestione delle Global Query, anche se al momento sono gestibili le Basic Query, che costituiscono la tipologia di interrogazioni rivolte piú di frequente ad un Mediatore.

# 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 organizzazione 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 delle 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 autonomamente, 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 University
  extent School_Members
  keys first_name, last_name )
{ attribute string  first_name;
  attribute string  last_name;
  attribute string  faculty;
  attribute integer year; }

interface Department
( source relational University
  extent Departments

interface Section
( source relational University
  extent Sections
```

```
  key dept_code )
{ attribute string dept_name;
  attribute integer dept_code;
  attribute integer budget;
  attribute string dept_area; };
```

```
  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
  extent Locations
  keys city, street, county, number)
{ attribute string city;
  attribute string street;
```

```
interface Course
( source object Computer_Science
  extent Courses
  key course_name )
{ attribute string course_name;
  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

# Grammatica OQL

La grammatica OQL viene descritta usando la notazione BNF e utilizzando la seguente simbologia:

- `symbol` indica una espressione OQL che non viene tradotta ma che è necessario specificare in quanto è prevista dalla sintassi.

- **symbol** indica un elemento terminale del linguaggio.

- *symbol_name* indica che deve essere specificato un identificatore il cui significato semantico è indicato dalla prima parte del nome.

- *symbol_literal* indica che deve essere specificato un simbolo di tipo literal. Ad es. “una stringa” viene indicato come *string_literal*

⟨Query⟩ : : = **(** ⟨Query⟩ **)**|
⟨SelectExpr⟩
⟨SelectPreamble⟩ : : = **select distinct** |
**select**
⟨SelectExpr⟩ : : = ⟨SelectPreamble⟩ ⟨ProjectionList⟩
⟨FromClause⟩
⟨WhereClause⟩
⟨ProjectionList⟩ : : = ⟨ProjectionAttributes⟩ |
*
⟨ProjectionAttributes⟩ : : = ⟨Attribute⟩ |
⟨ProjectionAttributes⟩ **,** ⟨Attribute⟩
⟨Attribute⟩ : : = ⟨Projection⟩ |
⟨Property⟩
⟨Projection⟩ : : = **(** ⟨Projection⟩ **)** |
⟨Identifier⟩ **,** ⟨Property⟩ |
⟨Property⟩ **as** ⟨Identifier⟩ |
⟨Property⟩ : : = **(** ⟨Property⟩ **)** |
⟨Basic⟩ |
⟨Identifier⟩ |
⟨Accesor⟩
⟨Basic⟩ : : = **nil** |
**true** |
**false** |
⟨FloatLiteral⟩ |
⟨IntegerLiteral⟩ |
⟨StringLiteral⟩
⟨StringLiteral⟩ : : = " ⟨String⟩ "
⟨IntegerLiteral⟩ : : = ⟨UnsignedLong⟩ |
⟨Sign⟩ ⟨UnsignedLong⟩
⟨FloatLiteral⟩ : : = **.** ⟨UnsignedLong⟩ |
⟨Sign⟩ **.** ⟨UnsignedLong⟩ |
⟨IntegerLiteral⟩ **.** ⟨UnsignedLong⟩
⟨Sign⟩ : : = + | −
⟨Digit⟩ : : = **0** | **1** | . . . | **9**
⟨Char⟩ : : = **a** | **b** | . . . | **z** | **A** | **B** | . . . | **Z**
⟨UnsignedLong⟩ : : = ⟨Digit⟩ |
⟨Digit⟩ ⟨UnsignedLong⟩
⟨String⟩ : : = ⟨Char⟩ | ⟨Digit⟩ | _ |
⟨Char⟩ ⟨String⟩ |
⟨Digit⟩ ⟨String⟩ |
_ ⟨String⟩
⟨Identifier⟩ : : = ⟨Char⟩ |
⟨Char⟩ ⟨String⟩
⟨Accessor⟩ : : = ⟨Identifier⟩ ⟨Path⟩ ⟨Accessor⟩ |
⟨Identifier⟩ ⟨Path⟩ ⟨Identifier⟩
⟨Path⟩ : : = . | − >

⟨FromClause⟩ ::= **from** ⟨VariableDeclaration⟩
⟨VariableDeclaration⟩ ::= ⟨Identifier⟩ |
⟨Identifier⟩ **as** ⟨Identifier⟩ |
⟨Identifier⟩ ⟨Identifier⟩
⟨WhereClause⟩ ::= $\epsilon$ |
**where** ⟨Predicates⟩
⟨Predicates⟩ ::= **(** ⟨Predicates⟩ **)** |
⟨Comparison⟩ |
⟨BooleanExpr⟩ |
⟨CollectionExpr⟩
⟨Comparison⟩ ::= ⟨Operand⟩ ⟨ComparisonOperator⟩
⟨Quantifier⟩ ⟨Operand⟩ |
⟨Property⟩ **like** ⟨StringLiteral⟩
⟨ComparisonOperator⟩ ::= $>$ | $>=$ | $<$ | $<=$ | $=$ | $!=$
⟨Quantifier⟩ ::= $\epsilon$ |
**some** |
**any** |
**all**
⟨Operand⟩ ::= ⟨Basic⟩ |
⟨SimpleExpr⟩ |
⟨Property⟩
⟨SimpleExpr⟩ ::= ⟨Property⟩ + ⟨Property⟩ |
⟨Property⟩ - ⟨Property⟩ |
⟨Property⟩ / ⟨Property⟩ |
⟨Property⟩ * ⟨Property⟩ |
- ⟨Property⟩ |
+ ⟨Property⟩ |
⟨Property⟩ **mod** ⟨Property⟩ |
**abs(** ⟨Property⟩ **)** |
⟨Property⟩ || ⟨Property⟩
⟨BooleanExpr⟩ ::= **not** ⟨Predicates⟩ |
⟨Predicates⟩ **and** ⟨Predicates⟩ |
⟨Predicates⟩ **or** ⟨Predicates⟩
⟨CollectionExpr⟩ ::= **for all** ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
**exists** ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
**exists(** ⟨Accessor⟩ **)** |
**unique(** ⟨Accessor⟩ **)** |
⟨Property⟩ **in** ⟨Condition⟩ |
**count(** ⟨Property⟩ **)** |
**sum(** ⟨Property⟩ **)** |
**min(** ⟨Property⟩ **)** |
**max(** ⟨Property⟩ **)** |
**avg(** ⟨Property⟩ **)**

# Appendice D

# Restrizione dell’ OQL per le BasicQuery

Le Basic Query costituiscono di fatto una restizione del linguaggio OQL. Di seguito viene riportata la descrizione in forma BNF di tale restrizione.

⟨Query⟩ : : = **(** ⟨Query⟩ **)**|
⟨SelectExpr⟩
⟨SelectPreamble⟩ : : = **select distinct** |
**select**
⟨SelectExpr⟩ : : = ⟨SelectPreamble⟩ ⟨ProjectionList⟩
⟨FromClause⟩
⟨WhereClause⟩
⟨ProjectionList⟩ : : = ⟨ProjectionAttributes⟩ |
*
⟨ProjectionAttributes⟩ : : = ⟨Attribute⟩ |
⟨ProjectionAttributes⟩ **,** ⟨Attribute⟩
⟨Attribute⟩ : : = ⟨Projection⟩ |
⟨Property⟩
⟨Projection⟩ : : = **(** ⟨Projection⟩ **)** |
⟨Identifier⟩ **,** ⟨Property⟩ |
⟨Property⟩ **as** ⟨Identifier⟩ |
⟨Property⟩ : : = **(** ⟨Property⟩ **)** |
⟨Basic⟩ |
⟨Identifier⟩ |
⟨Accesor⟩
⟨Basic⟩ : : = **nil** |
**true** |
**false** |
⟨FloatLiteral⟩ |
⟨IntegerLiteral⟩ |
⟨StringLiteral⟩
⟨StringLiteral⟩ : : = " ⟨String⟩ "
⟨IntegerLiteral⟩ : : = ⟨UnsignedLong⟩ |
⟨Sign⟩ ⟨UnsignedLong⟩
⟨FloatLiteral⟩ : : = **.** ⟨UnsignedLong⟩ |
⟨Sign⟩ **.** ⟨UnsignedLong⟩ |
⟨IntegerLiteral⟩ **.** ⟨UnsignedLong⟩
⟨Sign⟩ : : = + | −
⟨Digit⟩ : : = **0** | **1** | . . . | **9**
⟨Char⟩ : : = **a** | **b** | . . . | **z** | **A** | **B** | . . . | **Z**
⟨UnsignedLong⟩ : : = ⟨Digit⟩ |
⟨Digit⟩ ⟨UnsignedLong⟩
⟨String⟩ : : = ⟨Char⟩ | ⟨Digit⟩ | _ |
⟨Char⟩ ⟨String⟩ |
⟨Digit⟩ ⟨String⟩ |
_ ⟨String⟩
⟨Identifier⟩ : : = ⟨Char⟩ |
⟨Char⟩ ⟨String⟩
⟨Accessor⟩ : : = ⟨Identifier⟩ ⟨Path⟩ ⟨Accessor⟩ |
⟨Identifier⟩ ⟨Path⟩ ⟨Identifier⟩
⟨Path⟩ : : = . | − >

⟨FromClause⟩ : : = **from** ⟨VariableDeclaration⟩
⟨VariableDeclaration⟩ : : = ⟨Identifier⟩ |
⟨Identifier⟩ **as** ⟨Identifier⟩ |
⟨Identifier⟩ ⟨Identifier⟩
⟨WhereClause⟩ : : = $\epsilon$ |
**where** ⟨Predicates⟩
⟨Predicates⟩ : : = **(** ⟨Predicates⟩ **)** |
⟨Comparison⟩ |
⟨BooleanExpr⟩ |
⟨CollectionExpr⟩
⟨Comparison⟩ : : = ⟨Operand⟩ ⟨ComparisonOperator⟩ ⟨Quantifier⟩
⟨Operand⟩ |
⟨Property⟩ **like** ⟨StringLiteral⟩
⟨ComparisonOperator⟩ : : = $>$ | $>=$ | $<$ | $<=$ | $=$ | **!** $=$
⟨Quantifier⟩ : : = $\epsilon$ |
**some** |
**any** |
**all**
⟨Operand⟩ : : = ⟨Basic⟩ |
⟨SimpleExpr⟩ |
⟨Property⟩
⟨SimpleExpr⟩ : : = ⟨Property⟩ + ⟨Property⟩ |
⟨Property⟩ - ⟨Property⟩ |
⟨Property⟩ / ⟨Property⟩ |
⟨Property⟩ * ⟨Property⟩ |
- ⟨Property⟩ |
+ ⟨Property⟩ |
⟨Property⟩ **mod** ⟨Property⟩ |
**abs(** ⟨Property⟩ **)** |
⟨Property⟩ || ⟨Property⟩
⟨BooleanExpr⟩ : : = **not** ⟨Predicates⟩ |
⟨Predicates⟩ **and** ⟨Predicates⟩ |
⟨Predicates⟩ **or** ⟨Predicates⟩
⟨CollectionExpr⟩ : : = **for all** ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
**exists** ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
**exists(** ⟨Accessor⟩ **)** |
**unique(** ⟨Accessor⟩ **)** |
⟨Property⟩ **in** ⟨Condition⟩ |
**count(** ⟨Property⟩ **)** |
**sum(** ⟨Property⟩ **)** |
**min(** ⟨Property⟩ **)** |
**max(** ⟨Property⟩ **)** |
**avg(** ⟨Property⟩ **)**

# Appendice E

# La classe Java BasicQuery

Eseguendo la classe java QueryManager è possibile utilizzare il Query Manager di MOMIS interrogando lo schema globale fornito dall'esempio citato in questa tesi (Capitolo 2 ed Appendice B).

A titolo esemplificativo viene riportato qui di seguito il codice relativo ad una classe Java particolarmente significativa per la fase di esecuzione delle Basic Query implementata in questa tesi: la classe **BaseExtension.java**

Come si può notare dal listato riportato piú sotto, 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’‘
```

```
package queryman;

import java.util.Vector;
import java.util.TreeMap;
import java.util.HashSet;
import java.util.Iterator;
import globalschema.*;
import oql.*;
import GlobalSchemaProxy;

/** questa specializzazione della classe <b>Query</b> rappresenta le
 *  <i>Basic Query</i> ottenute dalla <i>Global Query</i> mediante la
 *  generazione del piano.
 */
public class BasicQuery extends Query
{
```

```
//
//PROPERTIES
//
// rispetto alla classe <b>Query</b> dovrebbe essere ridefinito il
//  tipo di piano individuando la struttura dati che meglio si adatta
//  alla rappresentazione del piano di accesso alle classi nelle
//  sorgenti. Tale piano indica l'ordine ed il tipo di join da eseguire
//  sui set di dati ritornati dalle query alle sorgenti per ricostruire
//  l'immagine cercata della classe globale.
//
/** e' il testo della query.*/
public String text = "";

/** indica il nome della classe globale interrogata */
private String className;

/** struttura contenente la query assemblatrice da eseguire nella fase
 *  finale, sul risultato delle query locali
 */
private Oql_SelectExpr queryAssembler ;

/** vettore contenente i predicati della clausola where normalizzata da
 *  tradurre in forma locale.
 */
public Vector predicati = new Vector(0);

//
// CONSTRUCTORS
//
/** ricevendo come parametro una struttura dati rappresentante la
 *  <i>basic query</i> crea un'istanza della classe caratterizzata
 *  dall'avere un piano di accesso ed un insieme di subquery che
 *  rappresentano le interrogazioni in cui la query viene tradotta
 *  e che dovranno essere eseguite sulle sorgenti.
 *  La query passata deve essere una Basic Query, pertanto il
 *  costruttore
 *  effettua, per prima cosa, un controllo volto ad accertare questa
 *  condizione. Nel caso in cui si sia certi di aver passato una
 *  Basic Query questo controllo puo' essere comunque bypassato,
 *  riducendo i tempi di esecuzione.
 *
 *  @param  bQu e' la struttura contenente la query.
 *  @param  check e' un boolean che permette di evitare il controllo
```

```
 *          sul tipo di query passata <br>
 *          - true   il controllo viene eseguito <br>
 *          - false  il controllo viene saltato <br>
 *  @exception -Exception indica il verificarsi di errori nel processo
 *             di trasformazione o la presenza di una query che non
 *             e‘ di tipo <i>Basic</i>
 */
public BasicQuery(Oql_Query bQu, boolean check,ConnectionJDBC con,
                                    GlobalSchemaProxy gsp,String it)
throws Exception
  {
    MappingTable mt = null;
    connection = con;
    proxy = gsp;
    iterQuery = it;
    FromElement fr = null;
    Oql_SelectExpr bQ = null;
    ExtensionalHierarchy eh = null;
    Vector qAttributes = new Vector(0);

    if ( check )
      {
        // si controlla se la query rientra nell’insieme delle
        // BasicQuery.
        //
     if (! bQu.isBasic() )
     throw new Exception(bqErr("La query non e‘ di tipo Basic"));
      }
    // si inizializza il campo text e subQueries
    //
    bQ = (Oql_SelectExpr)bQu;
    text = bQ.toString();
    data = new Data();
    subQueries = new TreeMap();
    // viene creata la struttura della queryAssembler,
    // con le clusole SELECT e FROM uguali
    // a quelle della BasicQuery corrispondente
    queryAssembler = new Oql_SelectExpr(bQ.selectClause.clone(),
                               bQ.fromClause.clone(),bQ.distinct);
    //
    // si individua la classe target della query
    //
    fr = (FromElement)(bQ.getClasses().get(0));
```

```
      className = fr.getClassName();
      //    iteratorName = fr.getIteratorName();
      if (! proxy.isGlobalClass(className) )
        throw new Exception(bqErr("classe globale inesistente"));
      else
        {
          mt = proxy.getMappingTable(className);

          eh = proxy.getExtensionalHierarchy(className);

          if (eh == null) evaluatePlan(bQ,mt);
          // non sfrutta la conoscenza estensionale
          else
          {
              qAttributes = getQueryAttributes(bQ,mt);
              int i;
              System.out.println("VERIFICA ATTRIBUTI DELLA QUERY");
              for (i=0;i<qAttributes.size();i++)
                 System.out.println(qAttributes.get(i));
              evaluatePlan(bQ,mt,eh,qAttributes);
            }
        }
    }

  //
  // METHODS
  //
  /** metodo per la definizione del piano di accesso.<br>
   *  Questa versione del metodo NON sfrutta la conoscenza
   *  estensionale.
   *  In questa fase occorre individuare le sorgenti interessanti per
   *  l'interrogazione ed indicare un piano per accedere alle stesse.
   *  Per svolgere tali funzioni in modo corretto, completo ed
   *  efficiente occorre disporre delle informazioni estensionali.
   *
   *  @param basQ e' la struttura dati contenente la <i>Basic Query</i>
   *  @param mapT e' la struttura dati relativa alla classe globale
   *              coinvolta nell'interrogazione
   *
   */
  private void evaluatePlan(Oql_SelectExpr basQ, MappingTable  mapT)
  throws Exception
    {
```

```
    TreeMap classes;
    classes = classLocalization(basQ,mapT);
    queryTransformation(basQ,mapT,classes);
  }

/** metodo per la definizione del piano di accesso.<br>
 *  Questa versione del metodo sfrutta la conoscenza estensionale
 *  In questa fase occorre individuare le sorgenti interessanti per
 *  l'interrogazione ed indicare un piano per accedere alle stesse.
 *  Per svolgere tali funzioni in modo corretto, completo ed
 *  efficiente occorre disporre delle informazioni estensionali.
 *
 *  @param basQ e' la struttura dati contenente la <i>Basic Query</i>
 *  @param mapT e' la struttura dati relativa alla classe globale
 *              coinvolta nell'interrogazione
 *  @param extH e' la struttura dati contenente la gerarchia
 *              estensionale relativa alla classe globale coinvolta
 *  @param queryA e' un vettore di stringhe contenente i nomi di tutti
 *         gli attributi globali presenti nelle clausole WHERE e SELECT
 *         della <i>Basic Query</i> in esame
 */
private void evaluatePlan(Oql_SelectExpr basQ, MappingTable  mapT,
                          ExtensionalHierarchy extH, Vector queryA)
throws Exception
  {
    TreeMap classes;
    classes = classLocalization(basQ,mapT,extH,queryA);
    queryTransformation(basQ,mapT,classes);
  }

/** metodo per la determinazione delle classi locali coinvolte
 *  nella query.
 *  Questa versione del metodo NON sfrutta la conoscenza
 *  estensionale. Se si disponesse delle informazioni
 *  estensionali sarebbe possibile individuare le classi
 *  effettivamente coinvolte ed il piano necessario
 *  per ricomporre i risultati (insieme di join tra risultati
 *  forniti dalle sorgenti). Non disponendo di tali informazioni
 *  non si e' in grado di definire un piano di accesso
 *  (e' lasciato null) e ci si limita a prendere in esame tutte
 *  le classi locali presenti nella MappingTable.
 *
 *  @param  bq e' la struttura dati che rappresenta la Basic Query
```

```
 *  @param  mt e‘ la struttura dati che rappresenta la <i>mapping
 *          table</i>
 *  @return restituisce una matrice contenente per ogni classe
 *          locale coinvolta nella query, il nome e l’insieme di
 *          attributi di join che devono essere aggiunti per la
 *          ricostruzione della risposta.
 *
 */
private TreeMap classLocalization(Oql_SelectExpr bq, MappingTable mt)
throws Exception
  {
    int i = 0;
    TreeMap tm = new TreeMap();
    Vector classNames = new Vector(0);
    //
    // non viene definito nessun piano
    //
    plan = null;
    //
    // si recuperano tutti i nomi delle classi locali nella mapping
    // table
    //
    classNames = mt.getClasses();
    String pippo ="";
    for( i=0; i<classNames.size(); i++)
      {
        pippo = (String)classNames.get(i);
        tm.put(classNames.get(i),null);
      }
    return tm;
  }

/** metodo per la determinazione delle classi locali coinvolte
 *  nella query.
 *  Questa versione del metodo sfrutta la conoscenza estensionale
 *  Disponendo delle informazioni estensionali e’ possibile
 *  individuare
 *  le classi effettivamente coinvolte ed il piano necessario per
 *  ricomporre i risultati (insieme di join tra risultati forniti
 *  dalle sorgenti)
 *
 *  @param  bq e‘ la struttura dati che rappresenta la Basic Query
 *  @param  mt e‘ la struttura dati che rappresenta la <i>mapping
```

```
 *          table</i>
 *  @param  eh e' la struttura dati che rappresenta la gerarchia
 *               estensionale
 *  @param  qa e' un vettore di stringhe contenente i nomi degli
 *          attributi globali presenti nella Basic Query
 *  @return restituisce una matrice contenente per ogni classe
 *          locale coinvolta nella query, il nome e l'insieme di
 *          attributi di join che devono essere aggiunti per la
 *          ricostruzione della risposta.
 *
 */
private TreeMap classLocalization(Oql_SelectExpr bq, MappingTable
                        mt, ExtensionalHierarchy eh, Vector qa)
throws Exception
  { // vettore contenente gli attributi della clausola select
    Vector selectattr = bq.getSelectAttributes("", mt);
    int i,y,h,j = 0;
    TreeMap tm = new TreeMap();
    Vector classNames, ext, extOpt, extDel, optClasses, planEle;
    BaseExtension be1,be2,be;
    boolean toBeDeleted[],found;
    PEBaseExtJoin beJoin;
    JoinMap jmap;
    PEClassJoin pe;
    classNames = new Vector(0);
    Vector classes = new Vector(0);
    ext = eh.findExtension(qa);
    planEle = new Vector(0);
    System.out.println("");
    System.out.println("Base Extension all'inizio:");
    for ( i=0; i<ext.size(); i++)
      { be = (BaseExtension)ext.get(i);
        System.out.println(be.baseExtNumber);
      }
     extOpt = new Vector(0);
    if (ext.size() > 1)
      { // se ci sono piu' base extension vediamo se se ne possono
        // eliminare alcune sfruttando il concetto di Dominanza
        extDel = new Vector(0);
        toBeDeleted = new boolean[ext.size()];
        for (i=0;i<ext.size();i++)
           toBeDeleted[i] = false;
        for (i=0;i<(ext.size()-1);i++)
```

```
                    for (y=(i+1);y<(ext.size());y++)
        {
            be1 = (BaseExtension)ext.get(i);
            be2 = (BaseExtension)ext.get(y);
            if (be1.localClasses.size() > be2.localClasses.size())
            {
         //be1 e' sempre la base extension con meno classi locali
                    be = be1;
                    be1 = be2;
                    be2 = be;
                    h = i;
             }
              else h = y;
              if (be1.dominates(be2)) toBeDeleted[h] = true;
              // se e' dominata si puo' eliminare
              // System.out.print("B.E. da cancellare: "
              // + be2.baseExtNumber + ": " + toBeDeleted[h]);
            }
        for (i=0;i<ext.size();i++)
         // eliminiamo le base extension dominate
               if (!toBeDeleted[i]) extOpt.add(ext.get(i));
               else extDel.add(ext.get(i));
System.out.println("Base Extension rimaste dopo l'ottimizzazione:");
      for ( i=0; i<extOpt.size(); i++)
        { be = (BaseExtension)extOpt.get(i);
          System.out.println(be.baseExtNumber);
        }
     // si riempie il plan con le indicazioni su come combinare
     // le base extension
      for (i=0;i<(extOpt.size()-1);i++)
                    for (y=(i+1);y<(extOpt.size());y++)
                {
                  be1 = (BaseExtension)extOpt.get(i);
                  be2 = (BaseExtension)extOpt.get(y);
                  found = false;
                  for (j=0;j<extDel.size();j++)
                  {
                   be = (BaseExtension)extDel.get(j);
                   if ((be1.dominates(be)) && (be2.dominates(be)))
                    {
                        found = true;
                        break;
                     }
```

```
            }
//  se due base extension ne dominano una terza occorre
// un outer join
       if (found)
       { beJoin = new PEBaseExtJoin(be1,be2,"outerjoin");
        // trova gli attributi per la fusione e determina
        // quelli eventualmente da aggiungere
        beJoin.joinBaseExt(qa,selectattr,bq);
        }
        // altrimenti e' sufficente un'unione dei risultati
           else beJoin = new PEBaseExtJoin(be1,be2,"union");
           planEle.add(beJoin);
        }

// si recuperano le classi locali dalle base extension rimaste
   be = (BaseExtension)extOpt.get(0);
   classNames = be.classOptimization(qa);
   for (i=0;i<extOpt.size();i++)
     {
       be = (BaseExtension)extOpt.get(i);
       optClasses = new Vector(be.classOptimization(qa));
       classes.add(optClasses);
       // si costruisce il vettore delle classi
       // ottimizzate senza duplicazioni di classi
       for (j=0;j<optClasses.size();j++)
           if (! classNames.contains(optClasses.get(j)))
             classNames.addElement(optClasses.get(j));
    }
   //
   // analizza la clausola WHERE
   //
   if ( bq.whereClause != null)
   {
     // riempie la query assembler e i predicati da tradurre
     // ed aggiunge gli eventuali attributi da recuperare dalle
     // classi locali
     createassembler((Oql_Query)bq.whereClause,classNames,mt,bq,
                                                    selectattr);
   }

    // si riempie il plan con le indicazioni su come ricostruire
    // le varie base extension
    for (i=0;i<extOpt.size();i++)
```

```
          {
            be = (BaseExtension)extOpt.get(i);
            // considera gli attributi della select aggiornata
            planEle.addAll(baseExtRebuilding(be,(Vector)
                          classes.get(i),  selectattr));
          }

    }
      else // c'e' una sola base extension
        {
          be = (BaseExtension)ext.get(0);
          classNames = be.classOptimization(qa);
          extOpt.add(be);
          //
          // analizza la clausola WHERE
          //
          if ( bq.whereClause != null)
          {
            // riempie la query assembler e i predicati da tradurre
            // ed aggiunge gli eventuali attributi da recuperare
            // dalle classi locali
            createassembler((Oql_Query)bq.whereClause,classNames,
                                          mt,Vector selectattr);
          }
        // selectattr contiene gli attributi della query aggiornata
        planEle.addAll(baseExtRebuilding(be,classNames,selectattr));
        }
      SourceClass pippo;
      System.out.println(" ");
      System.out.println("Classi locali da interrogare:");

       for( i=0; i<classNames.size(); i++)
             {
              pippo = (SourceClass)classNames.get(i);
              System.out.println(pippo.name);
              Vector attributesToAdd = new Vector(0);
              attributesToAdd.addAll(pippo.addAttributes
                         (extOpt,classNames, selectattr));
              if (attributesToAdd.size()>0)
               {
                tm.put(pippo.name,attributesToAdd);

          // aggiungiamo gli attributi per il join al piano
```

```
            for (j=0;j<planEle.size();j++)
            {
            if ((PlanElement)planEle.get(j) instanceof PEClassJoin)
             {
              pe = (PEClassJoin)planEle.get(j);
              if (((pe.sClass1 == pippo) || (pe.sClass2 == pippo))
                              && (pe.joinAttributes.size() == 0))
              {
               jmap = pe.bExt.getJoinMap(pe.sClass1,pe.sClass2);
               pe.joinAttributes.addAll(jmap.joinAttr);
               }
             }
             }
           }
          else tm.put(pippo.name,new Vector(0));
         }
 plan.planElements = planEle;
 // stampa del piano su std output
 PlanElement pluto;
 PEBaseExtJoin plutoBE;
 PEClassJoin plutoC;
 System.out.println(" ");
System.out.println("PLAN:Ricostruzione Risposta e Base Extensions");
 for(i=0; i<plan.planElements.size(); i++)
   { pluto = (PlanElement)plan.planElements.get(i);
     if (pluto instanceof PEBaseExtJoin)
      {plutoBE = (PEBaseExtJoin)pluto;
       System.out.println("Base Extensions
           "+plutoBE.bExt1.baseExtNumber
       +", "+plutoBE.bExt2.baseExtNumber+": "+plutoBE.joinKind);
       System.out.println("attributi per la fusione:");
        for (j=0;j<plutoBE.attributes.size();j++)
            System.out.println(plutoBE.attributes.get(j)); }
      else if (pluto instanceof PEClassJoin)
       { plutoC = (PEClassJoin)pluto;
         System.out.println("In base Extension "+
                 plutoC.bExt.baseExtNumber+" classes "+
               plutoC.sClass1.name+" and "+plutoC.sClass2.name);
         System.out.println("joineable by");
         for (j=0;j<plutoC.joinAttributes.size();j++)
            System.out.println(" "+plutoC.joinAttributes.get(j));
       }
    }
```

```
      System.out.println(" ");
      System.out.println("I PREDICATI DA TRADURRE SONO:");
      for(i=0;i<predicati.size();i++)
         System.out.println(predicati.get(i).toString());
      System.out.println(" ");
      System.out.println("QUERY ASSEMBLER: ");
      System.out.println(" ");
      System.out.println(queryAssembler.toString());
      System.out.println(" ");
      queryAssembler.fromClause.clear();
      if(queryAssembler.whereClause != null)
        { // la basic query di partenza e' stata aggiornata
         System.out.println("");
         System.out.println("");
         System.out.print("CLAUSOLA SELECT AGGIORNATA :\n select ");
         for(i=0;i<selectattr.size() - 1;i++)
            System.out.print(selectattr.get(i) + ", ");
         System.out.println(selectattr.get(i));
         System.out.println("");
         }
      if (tm.size() >0)
      {
      String key;
      System.out.println("ATTRIBUTI DA AGGIUNGERE:");
      Iterator iter = tm.keySet().iterator();
          while(iter.hasNext())
          { key = (String)iter.next();
            System.out.print(key + ": ");
            for(i=0;i<((Vector)tm.get(key)).size();i++)
            System.out.print(((Vector)tm.get(key)).get(i) + " ");
            System.out.println("");
          }
       }
      //
      return tm;
    }

  /** metodo per la traduzione dell'interrogazione in query locali.<br>
   *  Viene popolato il vettore contenente le subqueries.
   *
   *  @param  bq e' la struttura dati rappresentante la Basic Query
   *  @param  mt e' la mapping table
   *  @param  localDes e' una matrice formata da coppie che riportano
```

```
 *          il nome delle classi locali coinvolte nella query e
 *          gli attributi per i
 *          join necessari nella fase di ricostruzione della risposta
 *
 */
private void queryTransformation(Oql_SelectExpr bq, MappingTable mt,
                                      TreeMap localDes)
throws Exception
  { Vector v = new Vector(0);
    Vector joinAttr = new Vector(0);
    Vector selectattr = new Vector(0);
    String queryName = "", ss = "LQ";
    String key = "";
    int count = 1;
    int i;
    //   passa alle LocalQuery la BasicQuery con la clausola
    //  di selezione aggiornata,cioe' comprendente anche gli
    //attributi aggiuntivi della clausola where della
    // queryAssembler
     System.out.println("");
     System.out.println("");
     System.out.println("TRADUZIONE DELLE CLASSI LOCALI:");
     System.out.println("");
     Iterator iter = localDes.keySet().iterator();
     while(iter.hasNext())
      {
        queryName = "LQ" + count++;
        key = (String)iter.next();
        joinAttr = (Vector)(localDes.get(key));
   subQueries.put(queryName,new LocalQuery(key,joinAttr,mt,bq,this));

      }
     // inserisce l'elemento che indica la fine della generazione
     // delle local query
     executeQuery();
  }

/** routine per l'esecuzione della <i>Query</i>.<br>
 *  Produce in uscita una rappresentazione dei dati generati
 *  dalla Basic
 *  Query ed ottenuti applicando il piano <i>plan</i> alle
 *  sottoquery <i>subQueries</i>.
 */
```

```
public void executeQuery()
throws Exception
  { int i,j = 0;
    LocalClass lc1,lc2;
    //  BaseExtension be;
    Oql_SelectExpr select = null;
    Oql_SelectFullJoin selectFull = null;
    Vector vet = new Vector(0);
    Vector classes = new Vector(0);
    Vector baseTables = new Vector(0);
    Iterator iter;
    Oql_Query pred;
    Create cr,newCreate;
    FromElement from1,from2,ele1,ele2;
    // la struttura query associa ad ogni base extension la query
    // da effettuare per la ricostruzione dei risultati delle
    // singole base extension
    TreeMap query = new TreeMap();
    // struttura contenente le query degli outerjoin
    // fra base extension
    TreeMap outer = new TreeMap();
   // struttura che contiene le query assembler finali su cui
   // fare l'unione
    Vector unionQuery = new Vector(0);

    // il primo passo e' il ripopolamento delle base extension
    // vengono cercati gli elementi PEClassJoin
     for(i=0;i<plan.planElements.size();i++)
     { if (plan.planElements.get(i) instanceof PEClassJoin)
       { PEClassJoin pe = (PEClassJoin)plan.planElements.get(i);
         String  be = iterBaseExt((BaseExtension)pe.bExt);
         if (!(query.containsKey(be)))
         { Oql_SelectExpr qu = new Oql_SelectExpr();
           query.put(be,qu);
           select = qu;
         }
         // in ogni caso riempie le strutture
         // viene aggiornato il vettore delle classi locali e la
         // clausola FROM della struttura Oql_SelectExpr se le classi
         // non sono gia' state considerate.
       else
       select = (Oql_SelectExpr)query.get(be);
```

```
  lc1 = (LocalClass)pe.sClass1;
  lc2 = (LocalClass)pe.sClass2;
  from1 = select.getFromElement(iterQuery + lc1.getClassName());
  from2 = select.getFromElement(iterQuery + lc2.getClassName());
  // se non sono presenti le classi locali vengono aggiunte
  if (from1 == null)
   {
    from1 = new FromElement(iterQuery + lc1.getClassName(),
                                      lc1.getClassName(),true);

     select.addClass(from1);
    }
    if (from2 == null)
    { from2 = new FromElement(iterQuery + lc2.getClassName(),
                                      lc2.getClassName(),true);
      select.addClass(from2);
    }
    // aggiorna i predicati di join della clausola where
    pred = select.addJoin(from1,from2,pe.joinAttributes,proxy);
    select.addSelPred(pred);
   }
  }
// utilizza la connessione per eseguire la query
// e crea la tabella corrispondente sul database
System.out.println("");
System.out.println("RICOSTRUZIONE DELLE BASE EXTENSION:");
System.out.println("");
iter = query.keySet().iterator();
   while(iter.hasNext())
   { String key = (String)iter.next();
     System.out.println("");
     System.out.println("base extension: " + key);
     select = (Oql_SelectExpr)query.get(key);
     // crea la create da associare alla query
     newCreate = new Create(key);
     // aggiorna la clausola select e la Create
     classes = select.getClasses();
     for(j=0; j<classes.size();j++)
     { FromElement fr = (FromElement)classes.get(j);
       Vector vett = new Vector(0);
       cr = data.getCreate(fr);
       for(i=0;i<cr.columns.size();i++)
       { String attr = (String)cr.columns.get(i);
```

```
            if (!(newCreate.columns.contains(attr)))
            { newCreate.columns.add(attr);
              newCreate.domains.add(cr.domains.get(i));
              vett.add(attr);
            }
          }
          select.completeQuery(vett,fr,proxy);
        }
        System.out.println(select.toString());
        System.out.println(newCreate.toString());
        connection.executeStatement(newCreate.toString());
        baseTables.add(newCreate.tableName);
    connection.executequery(select.toString(),newCreate.tableName);
    // elimina la select e inserisce la Create corrispondente
        // query.remove(key);
        query.put(key,newCreate);
        }

   // vengono eliminate le tabelle delle classi locali
      System.out.println("");
 System.out.println("TABELLE DELLE CLASSI LOCALI DA ELIMINARE:");
 connection.dropTables(data.getName());
// vengono fuse fra loro le base extension seguendo le indicazioni
 // contenute negli elementi PEBaseExtJoin
  for(i=0;i<plan.planElements.size();i++)
  { if (plan.planElements.get(i) instanceof PEBaseExtJoin)
    { PEBaseExtJoin pe2 = (PEBaseExtJoin)plan.planElements.get(i);
      if (pe2.joinKind.equals("outerjoin"))
     {
       String be1 = iterBaseExt(pe2.bExt1);
       String be2 = iterBaseExt(pe2.bExt2);
    ele1 = new FromElement(be1,"base"+ pe2.bExt1.baseExtNumber,true);
     ele2 = new FromElement(be2,"base"+pe2.bExt2.baseExtNumber,true);
       // verifico l'esistenza della query
       if (!(outer.containsKey(be1)))
            // se non esiste gia' la query
        { selectFull = new Oql_SelectFullJoin();
        // basta riportare nella select tutti gli attributi di una
        // delle base extension presenti.
          selectFull.fromClause.add(ele1);
          selectFull.fromClause.add(ele2);
          Vector star = new Vector(0);
          String s = "*";
```

```
            star.add(s);
          ((Oql_SelectExpr)selectFull).completeQuery(star,ele1,proxy);
            outer.put(be1,selectFull);
            outer.put(be2,selectFull);
            String name = be1 + "outer" ;
            newCreate = new Create(name,be1);
            baseTables.add(name);
            connection.executeStatement(newCreate.toString());
          }
          else
          // altrimenti aggiorna la query
         { selectFull = (Oql_SelectFullJoin)outer.get(be1);
           if (!(outer.containsKey(be2)))
           {selectFull.fromClause.add(ele2);
            outer.put(be2,selectFull);
          }
         }
          // elimina le base extension da query
          query.remove(be1);
          query.remove(be2);

          // in select ho la query da aggiornare
          // aggiorna i predicati di join della clausola where
          pred = selectFull.addJoin(ele1,ele2,pe2.attributes,proxy);
          selectFull.fullPred.add(pred);
         }
       }
}
// elimina i duplicati
HashSet varset = new HashSet(outer.values());
Object[] arr = varset.toArray();
// utilizza la connessione per eseguire le outerjoin query
// e crea la tabella corrispondente sul database
System.out.println("");
System.out.println("OUTER JOIN:");
System.out.println("");
  for(i=0;i<arr.length;i++)
  { selectFull = (Oql_SelectFullJoin)arr[i];
    String name = ((FromElement)selectFull.fromClause.get(0)).
                   getClassName() + "outer";
    // crea anche la query assembler corrispondente
    Oql_SelectExpr unio = new Oql_SelectExpr(queryAssembler);
    FromElement ele = new FromElement(name,"",false);
```

```
          unio.fromClause.add(ele);
          unionQuery.add(unio);
          connection.executequery(selectFull.toString(),name);
          System.out.println(name);
          System.out.println(selectFull.toString());
        }
      // crea le union query da eseguire partendo dalla struttura
      // della
      // query assembler per le base extension escluse dagli
      // outerjoin, cioe' quelle contenute in query:
        iter = query.keySet().iterator();
        while(iter.hasNext())
        { String key = (String)iter.next();
          // crea anche la query assembler corrispondente
          Oql_SelectExpr unio = new Oql_SelectExpr(queryAssembler);
          FromElement ele = new FromElement(key,"",false);
          unio.fromClause.add(ele);
          unionQuery.add(unio);
        }
    // esegue la query finale costituita dall'unione delle
    // query assembler
        String s = "";
        for(i=0;i<unionQuery.size() - 1;i++)
        {
          select = (Oql_SelectExpr)unionQuery.get(i);
          s = s + select.toString() + " union ";
         }
        // aggiunge l'ultimo elemento
         select = (Oql_SelectExpr)unionQuery.get(i);
         s = s + select.toString();
         System.out.println("");
         System.out.println("UNION QUERY FINALE:\n\n" + s);
         System.out.println("");
          // invia la query finale
         //  connection.executequery(s,iterQuery + className);
         connection.executequery(s);
         // elimina le tabelle delle base extension
         connection.dropTables(baseTables);

         // manca l'implementazione della Global Query

         // aggiunge un risultato alla Global Query di partenza
```

```
        // se tutte le Basic query hanno restituito i risultati
        // inizia l'esecuzione della Global query


  }

 /** restituisce in una stringa il testo della query */
public String toString()
  {
      return text;
  }

/** compone un messaggio di errore aggiungendo la stringa passata
 *  comeparametro.
 */
 public String bqErr(String ss)
  {
    String s = "Basic Query Plan Generation : \n";
    s = s + ss;
    return s;
  }



/** Questo metodo fornisce l'insieme degli attributi presenti nelle
 *  clausole di Selezione della <i>BasicQuery</i> passata
 *  come parametro, gestisce anche la clausola where
 *
 *  @param basQ e' la struttura dati contenente la <i>BasicQuery</i>
 *  @param mt e' la MappingTable da usare per l'individuazione degli
 *         attributi
 *  @return restituisce un vettore di stringhe contenenti i nomi
 *          degli attributi globali presenti nelle clausole SELECT
 *          e WHERE
 *
 */
public Vector getQueryAttributes(Oql_SelectExpr basQ,MappingTable mt)
throws Exception
 {

     Vector queryAttributes = new Vector(0);
     Vector selectAttributes = new Vector(0);
```

```
      selectAttributes = basQ.getSelectAttributes("", mt);
      if (basQ.whereClause != null)
      { queryAttributes = ((Oql_Query)basQ.whereClause).
                                              getAttributes("",mt);
        selectAttributes.removeAll(queryAttributes);
        queryAttributes.addAll(selectAttributes);
        return queryAttributes;
       }
       else
       return selectAttributes;
   }


 /** Questo metodo fornisce un vettore di elementi di tipo PEClassJoin
  *  che contiene i join necessari per ricostruire una determinata
  *  Base Extension
  *  passata come parametro. Da notare che i join tra classi che
  *  necessitano
  *  di attributi aggiuntivi non presentano nessun attributo;
  *  questi saranno aggiunti in PLAN successivamente
  *
  *  @param be e' la Base Extension in questione
  *  @param classes e' un vettore di elementi di tipo SourceClass
  *         contenente l'insieme ottimizzato di classi
  *  @param attributes e' un vettore di stringhe contenente gli
  *         attributi presenti nella select della query
  *  @return restituisce un vettore di elementi di tipo PEClassJoin
  *          contenente
  *          le indicazioni per ricostruire la base extension passata
  *
  */
  private Vector baseExtRebuilding(BaseExtension be,Vector classes,
                                           Vector attributes)
  {
    SourceClass c1, c2;
    JoinMap jm;
    PEClassJoin cJoin;
    Vector pE;
    int i,y;

    pE = new Vector(0);
    for (i=0;i<(classes.size()-1);i++)
      for (y=(i+1);y<(classes.size());y++)
```

```
    {
            c1 = (SourceClass)classes.get(i);
            c2 = (SourceClass)classes.get(y);
            jm = be.getJoinMap(c1,c2);
            if (jm.joineable)
              {
                cJoin = new PEClassJoin(be,c1,c2);
                //  aggiorna il vettore di PEClassJoin
                if (attributes.containsAll(jm.joinAttr))
                cJoin.joinAttributes.addAll(jm.joinAttr);
                pE.add(cJoin);
              }
     }
  return pE;
 }


/** questo metodo aggiunge alla clausola where della query
 *  assembler le clausole che non sono comprese in nessuna
 *  delle classi locali passate nel vettore classes e inserisce
 *  nel vettore predicati le altre clausole.Gli attributi delle
 *  clausole non interamente mappate sono aggiunti alla SELECT
 *  della basic query di partenza
 *  e al vettore passato come ultimo parametro.
 *  @param where e' la clausola Oql_Query
 *  @param classes sono le classi locali da interrogare
 *  @param mt e' la mapping table
 *  @param bq e' la struttura che contiene la basic query iniziale
 *  @param selattr rappresenta gli attributi di selezione
 */

 private  void createassembler(Oql_Query where, Vector classes,
               MappingTable mt,Oql_SelectExpr bq, Vector selattr)
 {
   Vector v,attr ;
   boolean ok;
   SourceClass sc;
   int i;
   // analizza la forma normale congiuntiva ricorsivamente
   if (where instanceof Oql_AndExpr)
 { createassembler(((Oql_AndExpr)where).op1,classes,mt,bq,selattr);
   createassembler(((Oql_AndExpr)where).op2,classes,mt,bq,selattr);
```

```
    }
      else // predicato da analizzare
      {
        v = where.getAttributes("",mt);
        ok = true;
        for(i=0;i<classes.size();i++)
        { sc = (SourceClass)classes.get(i);
          attr = queryAssembler.mappingNonNull(sc.name,mt);
          // se la classe locale contiene tutti gli attributi
          if (attr.containsAll(v))
           { ok = false;
             break;
            }
         }
      if (ok == true) // se nessuna classe contiene la clausola
      { // aggiunge il predicato alla query assembler
        queryAssembler.addSelPred(where);
        v.removeAll(selattr);
        // aggiorna la clausola select della query di partenza e
        // vettore degli attributi di selezione
        selil attr.addAll(v);
        bq.completeQuery(v);
       }
      else // aggiorna i predicati da tradurre
       predicati.add(where);
     }
    }



    /** fornisce l'iteratore da associare alla base extension
     *  passata come parametro.
     *  @param be base extension
     *  @return restituisce la stringa costituita da:
     *
     *          iterQuery + "b" + baseExtNumber
     *
     */
public String iterBaseExt(BaseExtension be)
    { String s = "";
      s = iterQuery + "base" + be.baseExtNumber;
      return s;
     }
```

```
 /** compone un messagio di errore aggiungendo la stringa
  *  passata come parametro.
  */
 public String gqaErr(String ss)
  {
    String s = "\n\n -- Get Query Attributes Method -- \n";
    s = s + ss;
    return s;
  }


}
```

# Bibliografia


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

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

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

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

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

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

[7] S. Castano, V. De Antonellis, and S. De Capitani Di Vimercat. Global viewing of heterogeneous data sources. Technical Report 98-08, Dip. Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 1998.

[8] 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.

[9] Massimiliano Franceschi. Query Managerdi MOMIS: utilizzo della Conoscenza Estensionale. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1998-1999.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.*

[16] 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.*

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

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

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

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

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

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

[39] I. Schmitt and C. Türker. An Incremental Approach to Schema Integration by Refining Extensional Relationships. In G. Gardarin, J. French, N. Pissinou, K. Makki, and L. Bougamin, editors, *Proc. of the 7th ACM CIKM Int. Conf. on Information and Knowledge Management, November 3–7, 1998, Bethesda, Maryland, USA*, pages 322–330, New York, 1998. ACM Press.

[40] Driver Java JDBC.
Available at http://www.javasoft.com/products/jdbc

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

[42] 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*.