UNIVERSITA' DEGLI STUDI DI MODENA

E REGGIO EMILIA

Facoltà di Ingegneria
Sede di Modena

Corso di Laurea Specialistica in Ingegneria Informatica

PROGETTO E REALIZZAZIONE DEL PACKAGE MOMIS*Query Manager* SU ORACLE EXPRESS

Relatore:
Chiar.ma Prof. Sonia Bergamaschi

Correlatore:
Ing. Alberto Corni

Candidato:
Sculco Luca

Accademico 2007/2008

PAROLE CHIAVE:
Porting
SQLServer
Oracle
Java Package

Indice

Introduzione

Il diffondersi dell'utilizzo del Web, ha portato alla nascita dell'esigenza di poter reperire contenuti informativi provenienti da diverse risorse. Ciò, ha fatto emergere un crescente interesse verso lo sviluppo di sistemi di integrazione di risorse eterogenee. L'integrazione Intelligente delle Informazioni (*I3*) rappresenta la soluzione a tale problema. Il suo scopo è quello di ottenere in maniera automatica, una selezione ragionata dei dati provenienti dalle varie sorgenti e di conseguenza proporne una fusione intelligente. Un sistema che tenta di concretizzare tale obiettivo è MOMIS (*Mediator EnvirOment for Multiple Information Sources*), il quale rappresenta un progetto di sistema I3, ideato per l'integrazione di sorgenti di dati testuali, strutturati e semi-strutturati.

Le problematiche legate alla realizzazione di tale sistema sono molteplici.

Il problema semantico si intuisce facilmente, se si considera la possibilità che diverse persone possano fornire descrizioni, anche molto diverse tra loro, della stessa porzione di mondo. Anche se si possiede un insieme di conoscenze comuni (per esempio un'ontologia), non è possibile affermare che tali concetti saranno rappresentati, nelle diverse sorgenti, attraverso gli stessi vocaboli. Di conseguenza, nel processo d'integrazione, uno fra i tanti obbiettivi dovrà essere quello di risolvere le differenze semantiche fra le diverse rappresentazioni dei dati.

L'obiettivo finale è quello di fornire un Global Virtual Schema (GS), una concettualizzazione (ontologia) che descrive un insieme di sorgenti dati distribuite e eterogenee, e consente all'utente di porre queries e ricevere un unica risposta, in modo trasparente alle sorgenti coinvolte.

Il vantaggio dell'integrazione virtuale dei dati è che questi rimangono nelle sorgenti.

Quando si deve risolvere la query dell'utente se ne occuperà il modulo Query Manager di MOMIS, il quale interfacciandosi con un database di supporto creerà delle tabelle temporanee per l'inserimento dei dati che successivamente elaborati costituiranno il risultato.

Fino ad adesso al sistema MOMIS era consentito solamente l'utilizzo del DBMS di Microsoft SQLServer e l'obiettivo di questa tesi è di poter permettere la connessione con altri DBMS, aumentando la portabilità del software, in particolare non essere più vincolato ad una piattaforma Windows.

Il contenuto della tesi è così strutturato:

• **Capitolo 1: Il Sistema MOMIS**. Si descriverà in maniera sintetica l'architettura del sistema MOMIS e dei suoi componenti fondamentali.

• **Capitolo 2: Il modulo Query Manager**. In questo capitolo sono descritte le classi principali e i loro collegamenti, seguendo l'esempio di esecuzione di una query globale.

• **Capitolo 3: Il Porting**. E' trattato passo per passo il lavoro di porting effettuato durante questa tesi che ci ha permesso di includere Oracle tra i DBMS utilizzabili da MOMIS.

• **Capitolo 4: Differenze tra SQLServer e Oracle**. Sono annotate le differenze implementative che sono state riscontrate tra i 2 DBMS e come sono state risolte.

• **Capitolo 5: Le modifiche di MOMIS**. Vengono descritte le classi in cui sono state applicate delle modifiche, con attenzione particolare verso quelle necessarie per il funzionamento di MOMIS con ORACLE.

Capitolo 1

1 Il Sistema MOMIS

Al giorno d'oggi, un problema cui devono far fronte numerose imprese ed organizzazioni, è quello della dispersione del loro patrimonio informativo. Si pensi ai numerosissimi metodi di immagazzinamento di informazioni presenti sul mercato o utilizzabili gratuitamente: DBMS, pagine HTML, pagine XML ecc…

Nel caso in cui un utente voglia reperire informazioni da sorgenti diverse, fatto che accade sempre più frequentemente ogni giorno, si trova di fronte a problemi di non facile soluzione: le sorgenti di conoscenza, infatti, sfrutteranno tecnologie differenti, difficilmente uniformabili, senza contare le possibili contraddizioni ed inconsistenze fra i dati ottenuti da diverse fonti. Un grande aiuto per quanto concerne il problema dello sfruttamento di tecnologie differenti proviene dagli standard esistenti, (come l'ODBC, CORBA ecc…), che risolvono il problema della comunicazione fra moduli diversi. Ciò che rimane irrisolta, è la questione della modellazione delle informazioni: i modelli dei dati, possono differenziarsi gli uno dagli altri, a tal punto da fornire ognuno una propria struttura logica di rappresentazione dei dati da immagazzinare. Tutto ciò crea un'eterogeneità semantica non risolvibile dagli attuali standard. Da quanto descritto in precedenza, si evincono le difficoltà che sorgono nel creare un sistema di integrazione e mediazione delle informazioni eterogenee che sia affidabile, flessibile, modulare (in modo da consentire il riuso delle diverse parti all'evolvere delle tecnologie), e capace di interagire con altri sistemi esistenti. Nel seguito si descriverà una proposta di ARPA (***A**dvanced **R**eserch **P**roject **A**gency*) per un'architettura di integrazione di informazioni, flessibile e riusabile. L'approccio descritto dall'ARPA in [**1**], è stato seguito anche nel progetto MOMIS.

1.1 L'Integrazione Intelligente delle Informazioni

Come viene citato in [**2**], l'integrazione delle informazioni (I_2) si distingue da quella dei dati e dei database, in quanto non cerca di collegare semplicemente alcune sorgenti, ma risultati opportunamente selezionati da esse. Lo scopo dell'integrazione dell'informazione è, quindi, quello di ottenere una selezione ragionata dei dati prelevati dalle varie sorgenti, e produrre una fusione intelligente ed una seguente sintesi degli stessi. Proprio a questo scopo, è stato sviluppato dall'ARPA, un progetto di ricerca atto a fornire un'architettura di riferimento, che realizzi l'integrazione di risorse eterogenee in maniera automatica; il nome di questo progetto è appunto I_3 (**I**ntegrazione **I**ntelligente dell'**I**nformazione). I risultati ottenuti in questo ambito sono molto importanti poiché danno una concreta indicazione su come costruire un sistema di mediazione che sia riusabile, e le cui parti non comportino costi eccessivi di sviluppo. L'integrazione delle informazioni, inoltre, ne aumenta il valore, ma non è semplicemente gestire gli aggiornamenti, le eliminazioni e le sostituzioni fra le varie sorgenti, le loro ontologie e semantiche. L'ARPA ritiene, che una grossa mano a tal proposito, possa essere data dall'utilizzo dell'Intelligenza Artificiale che, essendo in grado di dedurre dagli schemi delle sorgenti informazioni utili, può essere considerata uno strumento prezioso ed in grado di fornire soluzioni flessibili e riusabili. Secondo il programma I_3 è opportuno costruire architetture modulari, in grado di abbassare i costi di sviluppo e mantenimento, eseguite seguendo uno standard che ponga le basi dei servizi necessari all'integrazione. Il paradigma impiegato nel progetto I_3 per la suddivisione dei servizi e delle risorse fra i vari moduli, si basa su due partizionamenti fondamentali:

- Il *partizionamento orizzontale* che fornisce tre sezioni: database, sorgenti, basi di conoscenza intermedie, ed applicazioni utente.
- Il *partizionamento verticale* che distingue i domini in cui raggruppare le sorgenti.

I domini non sono strettamente interconnessi fra loro all'interno di un certo livello, ma si scambiano dati e informazioni. Per facilitare la flessibilità e migliorare le prestazioni del sistema, la fase importantissima della combinazione delle informazioni avviene a livello d'utente. Nel seguito verrà illustrata l'architettura di riferimento per i sistemi I_3.

1.1.1 L'architettura dei sistemi I3

L'architettura del programma I3 definita dall'ARPA, deriva la sua forma dal tentativo di far fronte a problemi complessi come quelli citati nei paragrafi precedenti, e che vengono qui brevemente riproposti in un elenco :

1. *Eterogeneità delle sorgenti*: differenze fra i tipi di dato, schemi logici, interfacce per accedere ai dati…
2. *Evoluzione delle sorgenti di dati*: si possono aggiungere nuove fonti o modificare o eliminare quelle vecchie.
3. *Dimensioni delle fonti*: bisogna far fronte all'aumento dei dati di una singola sorgente (ed all'aumento delle sorgenti) ed all'aumento dei tempi di risposta che ne derivano.
4. *Semantica nascosta*: bisogna dedurre regole, dai differenti schemi, per elaborare ed interpretare i dati da integrare.
5. *Necessità di sistemi modulari e riusabili:* questo punto è fondamentale per ridurre i tempi ed i costi di sviluppo delle varie applicazioni, e far fronte ai mutamenti tecnologici, che inevitabilmente si susseguono nel tempo.

Una volta compresi i problemi fondamentali, si può passare all'analisi dell'architettura di riferimento dei sistemi I3 proposta da ARPA. L'architettura del progetto I3 si propone di evidenziare (separando in più moduli), i vari servizi che devono essere svolti ai fini dell'integrazione intelligente d'informazioni. I servizi evidenziati a questo proposito sono cinque:

- *Servizi di coordinamento*
- *Servizi di amministrazione*
- *Servizi di integrazione e trasformazione semantica*
- *Servizi di wrapping*
- *Servizi ausiliari*

Fra tutti quelli elencati in precedenza, i servizi principali sono quelli di coordinamento il cui scopo è, appunto, quello di coordinare le operazioni attuate dai vari servizi sia in fase di progettazione dei vari link di integrazione fra le sorgenti, sia in fase di esecuzione (in tempo reale) su specifiche richieste dagli utenti.
Di seguito si descriveranno brevemente i servizi precedentemente elencati.

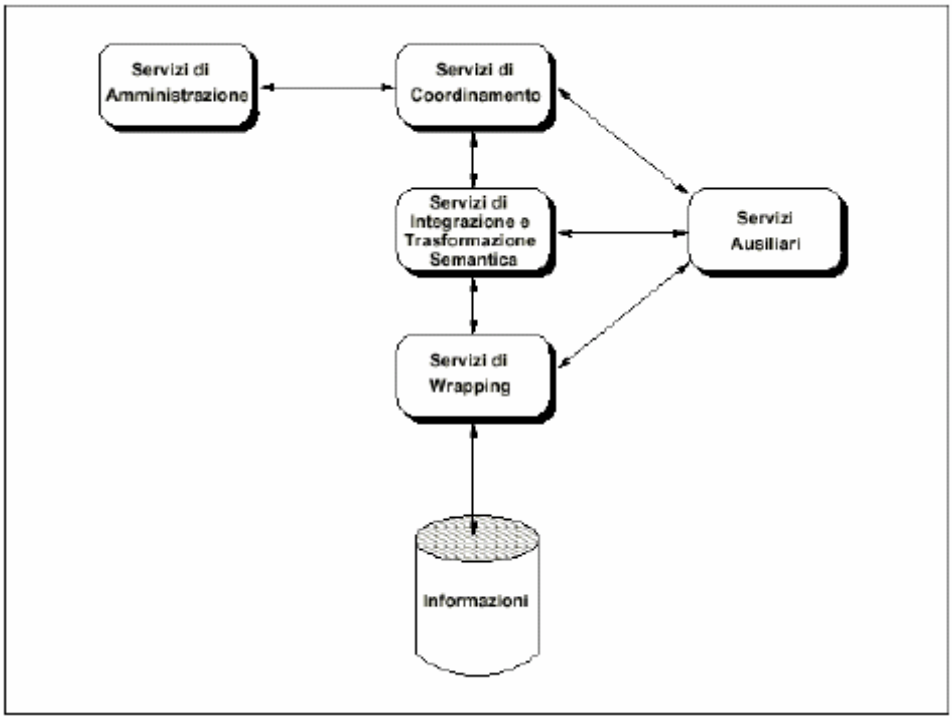

Figura 1.1: Diagramma dei servizi I3

Servizi di coordinamento

I servizi di coordinamento svolgono i lavori di supporto, sia in fase di progettazione di nuove configurazioni, che a tempo di esecuzione delle richieste dell'utente. Questi servizi sono di alto livello e, oltre ad individuare quali sorgenti possono essere utili per soddisfare una data richiesta, presentano all'utente finale l'intero sistema, diviso fra i suoi vari moduli, come un blocco unico ed omogeneo. Grazie ai servizi di coordinamento, quindi, le divisioni interne di un sistema I^3, sono trasparenti all'utente. I principali moduli appartenenti a questa sezione sono:

- *Broker*: il suo compito è quello di reperire gli strumenti in grado di trattare le richieste dell'utente. Il broker si occupa di contattare un modulo alla volta.
- *Interative query formulation*: si tratta di un modulo di ausilio nella espressione di una query che ha come oggetto lo schema intergrato. In particolare, è di aiuto se si ha già espresso una query che non ha prodotto risultati interessanti.
- *Primitive di costruzione delle configurazioni***:** servono a scegliere quali servizi e quali strumenti possono essere utilizzati per la costruzione di una configurazione e come collegarli fra loro.

Servizi di Integrazione semantica

I servizi d'integrazione semantica, hanno come input una o più sorgenti di dati tradotte dai servizi di *Wrapping,* e, come output, la "vista" integrata o trasformata di queste informazioni. Essi vengono indicati spesso come servizi di mediazione. I principali sono:

- *Servizi d'integrazione degli schemi*: creano il vocabolario e le ontologie condivise dalle sorgenti; integrano gli schemi con in una vista globale, mantengono il *mapping* tra gli schemi globali e le sorgenti;
- *Servizi d'integrazione di informazione:* aggregano, riassumono ed estraggono le risposte di più sotto-query per fornire un'unica risposta alla query originale;
- *Servizi di Supporto al processo d'integrazione*: sono utilizzati quando la query deve essere scomposta in più sotto-query da inviare a fonti differenti, con la necessità di integrare, poi, i loro risultati.

Servizi di Wrapping

I servizi di *wrapping* fungono da interfaccia tra il sistema integratore e le singole sorgenti, in particolare, rendendo omogenee le informazioni. Si comportano come dei traduttori dai sistemi locali ai servizi di alto livello dell'integratore. Il loro obiettivo è, quindi, quello di standardizzare il processo di *wrapping* delle sorgenti, permettendo la creazione di una libreria di fonti accessibili; inoltre, il processo di realizzazione di un *wrapper* dovrebbe essere standardizzato, in modo da poter essere riutilizzato per altre fonti.

Servizi Ausiliari

I servizi ausiliari aumentano le funzionalità degli altri servizi e sono utilizzati prevalentemente dai moduli che agiscono direttamente sulle informazioni; essi vanno dai semplici servizi di monitoraggio del sistema, ai servizi di propagazione degli aggiornamenti e di ottimizzazione.

Il mediatore

Il Mediatore è un modulo intermedio che si pone tra l'utente e le sorgenti d'informazione. Secondo al definizione di Wiederhold in [**2**] " un Mediatore è un modulo software che sfrutta la conoscenza su un certo livello superiore. Dovrebbe essere piccolo e semplice, così da poter essere amministrato da uno o al più, da pochi esperti."

I compiti del Mediatore sono:

- Assicurare un servizio stabile, anche quando cambiano le risorse;
- Amministrare e risolvere le eterogeneità delle diverse fonti;
- Integrare le informazioni ricavate da più risorse;
- Presentare all'utente le informazioni attraverso un modello scelto dall'utente stesso.

L'approccio architetturale adottato, è quello classico, che consta principalmente di tre livelli:

1. *utente*: attraverso un'interfaccia grafica l'utente pone delle query su uno schema globale e riceve risposta, come se stesse interrogando un'unica sorgente d'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 interrogatore) comune a tutte le fonti;
3. *wrapper*: ogni wrapper gestisce una sorgente, ed ha una duplice funzione: da un lato converte le richieste del Mediatore in una forma comprensibile dalla sorgente, dall'altro traduce informazioni estratte dalla sorgente nel modulo usato dal mediatore.

Esistono due approcci fondamentali all'architettura precedentemente descritta:

- Approccio strutturale caratterizzato dall'uso di *self-describing model* per rappresentare gli oggetti da integrare, limitando così 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, bensì è l'oggetto stesso che, attraverso delle etichette, si auto-descrive, specificando tutte le volte, per ogni suo singolo campo, il significato associato.
- Approccio semantico: è l'approccio utilizzato in MOMIS, ed è caratterizzato dal fatto che 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 i metadati, e infine, deve essere possibile un'integrazione (parziale o totale) delle sorgenti di dati. In questo modo il Mediatore può individuare i concetti comuni a più sorgenti e relazioni che li legano.

1.1.2 Problemi da affrontare

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

1. problemi ontologici

2. problemi semantici

Problemi ontologici

Per ontologia si intende, in questo ambito, "l'insieme dei termini e delle relazioni usate in un dominio, che denotano concetti ed oggetti.". Con ontologia, quindi, ci si riferisce a quell'insieme di termini che, in un particolare dominio applicativo, denotano in modo univoco, una particolare conoscenza e, fra i quali, non esiste ambiguità poiché sono condivisi dall'intera comunità di utenti del dominio applicativo stesso. I livelli di ontologia e le problematiche ad esse associate sono le seguenti:

1. *top-level ontology*: descrive concetti molto generali (spazio, tempo, …), che sono quindi indipendenti da un particolare dominio di appartenenza; si considera ragionevole, almeno in teoria, che anche comunità separate di utenti condividano la stessa top-level ontology;
2. *domain e task ontology:* descrivono rispettivamente il vocabolario relativo a un generico dominio, o quello relativo a un generico obiettivo, dando una specializzazione dei termini introdotti nella top-level ontology;
3. *application ontology:* descrive concetti che dipendono sia da un particolare dominio, sia da un particolare obiettivo.

Problemi semantici

Pur ipotizzando che anche sorgenti diverse condividano una visione similare del problema da modellare, e quindi, un insieme di concetti comuni, niente ci assicura che diversi sistemi usino esattamente gli stessi vocaboli per rappresentare questi concetti, ne tanto meno le stesse

strutture dati. Poiché le diverse strutture dati sono state progettate e modellate da persone differenti, è molto improbabile che queste persone condividano la stessa "concettualizzazione" del mondo esterno, ovvero non esiste nella realtà una semantica univoca cui chiunque possa riferirsi.
Ragion per cui, c'è un'incertezza di interpretazione insita nell'ambiguità del linguaggio; Bates in [Bates,86] scrive: *"the probability of two person using the same term in describing the same thing is less than 20%*".
Qualche esempio: una persona P1 disegna una fonte d'informazione DB1 e un'altra persona P2, disegna la stessa fonte DB2; sarà molto probabile che le due basi di dati presentino diverse semantiche: le coppie sposate potranno essere rappresentate in DB1 usando oggetti della classe COPPIA, con attributi MARITO e MOGLIE, mentre in DB2 potrebbe esserci una classe PERSONA con un attributo SPOSATO_A.
Come riportato in [**3**], la causa principale delle differenze semantiche, si può identificare nelle diverse concettualizzazioni del mondo esterno che le persone distinte possono avere, ma questa non è l'unica. Le differenze nei sistemi DBMS, possono portare all'uso di differenti modelli per la rappresentazione della porzione del mondo in questione; partendo così dalla stessa concettualizzazione, determinate relazioni fra concetti, avranno strutture diverse a seconda che siano state realizzate, ad esempio, attraverso un modello relazionale o un modello ad oggetti.
L'obiettivo dell'integratore, che ricordiamo è di fornire un accesso integrato ad un insieme di sorgenti, si traduce nel non facile compito di identificare i concetti comuni all'interno delle sorgenti, e risolvere le differenze semantiche che possono essere presenti. Possiamo classificare queste incoerenze semantiche in tre gruppi principali:

- *Eterogeneità tra le classi di oggetti*: benché due classi in due differenti sorgenti rappresentano 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;
- *Eterogeneità tra le strutture delle classi*: comprendono le differenze nei criteri di specializzazione, nelle strutture per realizzare un'aggregazione, ed anche le discrepanze schematiche;
- *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.

1.2 L'architettura di MOMIS

MOMIS acronimo di ***M**ediator **E**nvir**O**ment for **M**ultiple **I**nformation **S**ources*, è il progetto di un sistema I^3, ideato per l'integrazione di sorgenti di dati testuali, strutturati e semi-strutturati. MOMIS nasce all'interno del progetto MURST 40% INTERDATA, come collaborazione fra le unità operative dell'Università di Milano e dell'Università di Modena e Reggio Emilia. MOMIS è stato progettato per fornire un accesso integrato ad informazioni eterogenee, memorizzate sia all'interno di un *database* di tipo tradizionale (e.g. relazionali, *objectoriented*) o *file system* sia in sorgenti di tipo semi-strutturato come quelle descritte in XML. Seguendo l'architettura di riferimento in [**1**] si possono distinguere i componenti disposti su tre livelli (figura 1.2):

• **Livello dati.** Qui si trovano i *Wrapper.* Posti al di sopra di ciascuna sorgente, sono i moduli che rappresentano l'interfaccia fra il Mediatore le sorgenti di dati locali. La loro funzione è duplice:

- In fase d'integrazione forniscono la descrizione dell'informazione in essi contenute. Questa descrizione è fornita attraverso il linguaggio $ODLI^3$.
- In fase di *query processing* traducono la query ricevuta dal Mediatore(espressa quindi nel linguaggio comune d'interrogazione $OQLI^3$, definito a partire dal linguaggio OQL) in un'interrogazione comprensibile dalla sorgente stessa. Devono, inoltre, esportare i dati ricevuti come risposta all'interrogazione, presentandoli al mediatore, attraverso il modello comune di dati utilizzato dal sistema.

• **Livello Mediatore.** Il Mediatore rappresenta il cuore del sistema ed è essenzialmente composto da due sotto-moduli:

- ***Global Schema Builder*** (GSB): è il modulo che integra gli schemi locali, il quale partendo dalle descrizioni delle sorgenti espresse, attraverso il linguaggio $ODLI^3$ genera un unico schema globale da presentare all'utente. L'interfaccia grafica di GSB, cioè il *tool* d'ausilio al progettista, è *SI-Designer.*
- ***Query Manager*** (QM)*:* è il modulo di gestione delle interrogazioni. In particolare, genera la *query* in linguaggio $ODLI^3$ da inviare ai *wrapper,* partendo dalla singola *query* formulata dall'utente sullo schema globale. Servendosi delle tecniche di

Description Logics di ODB-Tools, il QM genera automaticamente la traduzione della query sottomessa nelle corrispondenti sub-query da sottoporre ai wrapper (query e sotto-query sono espresse in linguaggio OQLI3).

- ***SI-Designer****:* è la GUI (Graphic User Interface) che guida l'utente attraverso le varie fasi dell'integrazione, dall'acquisizione delle sorgenti, fino alla messa a punto del Common Thesaurus. SI-Designer risulta a sua volta composto da quattro moduli:
  - SIM (*Source Integrator Module*): estrae le relazioni inter-schema sulla base della struttura delle classi ODLI3 e delle sorgenti relazionali usando ODBTools. Inoltre effettua la "validazione semantica" delle relazioni e ne inferisce delle nuove sfruttando sempre ODB-Tools
  - SLIM (S*ource Lexical Intergrator Module*): estre le relazioni inter-schema tra nomi di attributi e classi ODLI3, sfruttando il database lessicale WordNet.
  - TUNIM (*Tuning of the Mapping Table*): questo modulo gestisce la fase di creazione dello schema globale.

La GUI di SI-Designer, è una sequenza di finestre, ognuna delle quali relativa ad una fase del processo d'integrazione, e mette a disposizione l'interfaccia per interagire con i moduli SIM, SLIM ed ARTEMIS.

• **Livello Utente.** Il progettista interagisce con il Global Schema Builder e crea la vista integrata delle sorgenti; l'utente formula le interrogazioni sullo schema globale, passandole come input al Query Manager, che interrogherà le sorgenti e fornirà all'utente la risposta cercata.

Nella figura 1.2 compaiono inoltre, altri tre tool che accompagnano il Mediatore nella fase di integrazione e sono:

• *ODB-Tolls Engine:* un tool basato sulle *Description Logics* [**4,5**] che compie la validazione di schemi e l'ottimizzazione di query [**6,7, 8**].

• *ARTEMIS-Tool Enviroment*: tool basato sulle tecniche di *clustering affinity-based* che compie l'analisi ed il clustering delle classi ODLI3 [**9**].

• *WordNet*: un database lessicale, disponibile gratuitamente presso il sito http://www.cogsci.princeton.edu/wn.

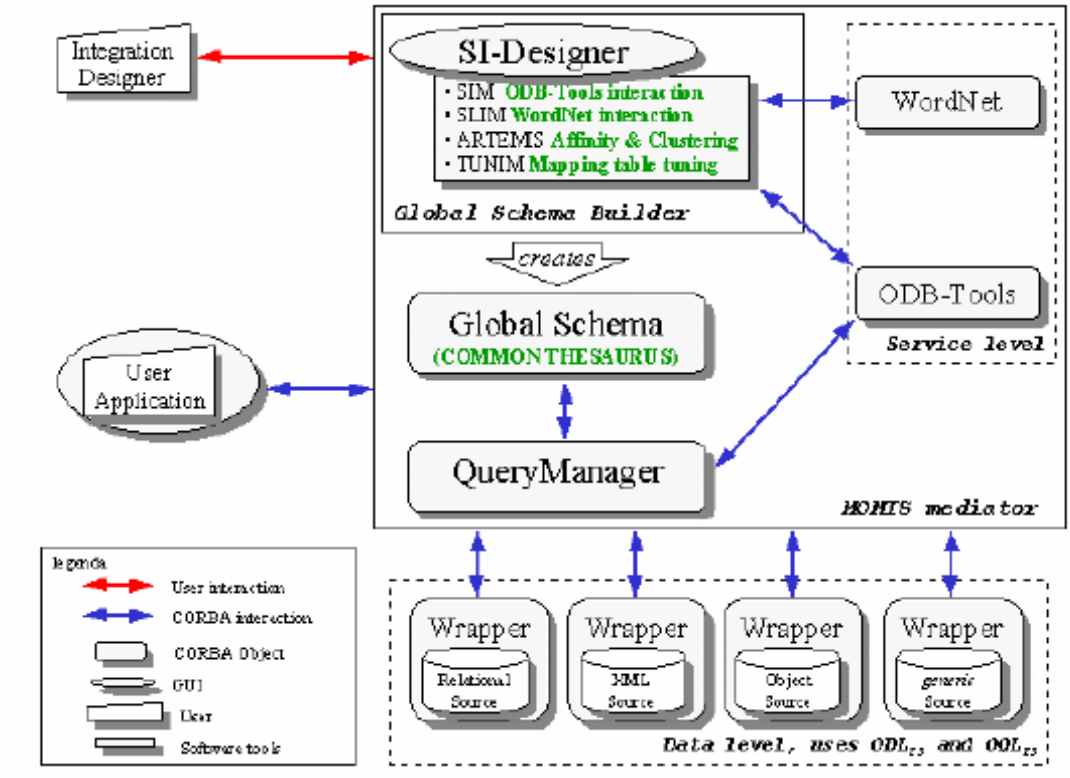

Figura 1.2: Architettura generale del sistema MOMIS

Lo scopo principale a cui ci si è proposti con MOMIS, è la realizzazione di un sistema di mediazione che, a differenza di molti altri progetti, contribuisca a realizzare, oltre alla fase di query processing, una reale integrazione delle sorgenti.

1.2.1 Il processo di Integrazione

L'integrazione delle sorgenti informative strutturate e semi-strutturate, è compiuta in modo semi-automatico, utilizzando degli schemi locali in linguaggio ODLI3, e combinando tecniche di *Description Logic* e di *clustering*. Come mostrato in figura 1.3, le attività compiute sono le seguenti:

1. *Generazione del Thesaurus Comune*, con il supporto di ODB-Tool e di WordNet. In questa fase è identificato un Thesaurus comune di relazioni terminologiche. Tali relazioni, esprimono la conoscenza inter-schema su sorgenti diverse e corrispondono alle asserzioni intenzionali utilizzate in [**10**]. Le relazioni terminologiche, sono derivate in modo semi-automatico a partire dalle descrizioni degli schemi in ODLI3, attraverso l'analisi strutturale (utilizzando ODB-Tools e le tecniche di *Description Logics*) e di contesto (attraverso l'uso di WordNet) delle classi coinvolte.

2. *Generazione dei cluster di classi ODLI3* con il supporto dell'ambiente ARTEMISTool. Le relazioni terminologiche contenute nel Thesaurus, sono utilizzate per valutare il livello di affinità tra le classi ODLI3 in modo da identificare le informazioni che devono essere integrate a livello globale. A tal fine ARTEMIS, calcola i coefficienti che misurano il livello di affinità tra le classi, basandosi sia sui nomi delle stesse sia sugli attributi. Le classi con maggiore affinità sono raggruppate utilizzando tecniche di clustering [**11**].

3. *Costruzione dello Schema Globale*. I cluster delle classi ODLI3 affini, sono analizzati per costruire lo schema globale del Mediatore. Per ciascun cluster si definisce una classe globale che rappresenta tutte le classi locali riferite al cluster ed è caratterizzata dall'unione ragionata dei loro attributi e da una *mapping-table*. L'insieme delle classi globali definite, costituisce lo schema globale del Mediatore che sarà usato per porre le query alla sorgenti locali integrate.

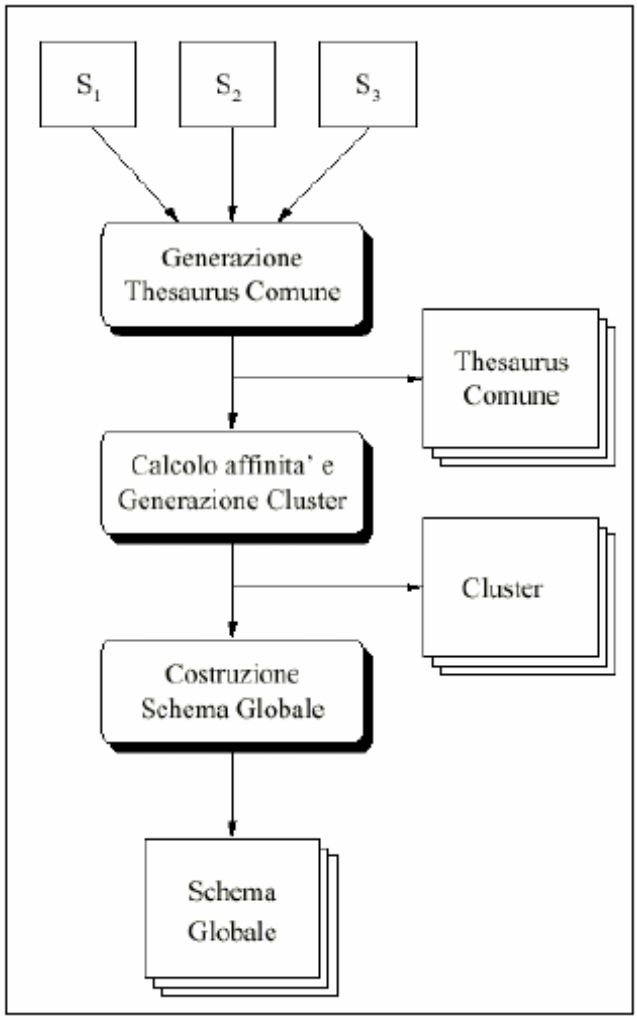

Figura 1.3: Fasi del processo d'integrazione.

1.2.2 Query Processing e Ottimizzazione

Quando un'utente pone una query sullo schema globale, MOMIS la analizza e produce un insieme di sotto-query che saranno inviate a ciascuna sorgente informativa coinvolta. Il processo consiste di due attività principali:

1. *Ottimizzazione Semantica.* L'ottimizzazione semantica è basata sull'inferenza logica a partire dalla conoscenza contenuta nei vincoli d'integrità dello schema globale. La stessa procedura di ottimizzazione semantica, si realizza in termini locali, su ogni sotto-query tradotta dal Mediatore nella formulazione del piano d'accesso: in tal caso ci si basa sui vincoli d'integrità presenti sui singoli schemi locali.

2. *Formulazione del piano d'accesso*. Il mediatore utilizza una "mappa" (generata nella costruzione dello schema globale) che definisce l'associazione tra le classi globali e le classi locali. La query globale è espressa in termini degli schemi locali, tenendo in considerazione anche l'eventuale conoscenza di regole inter-schema definite sull'estensioni delle classi locali.

Il Mediatore agisce sulla query, sfruttando la tecnica di ottimizzazione semantica da ODBTools, in modo da ridurre il costo del piano d'accesso, e, dopo aver ottenuto la query ottimizzata, genera l'insieme di sotto-query relative alle sorgenti coinvolte.

1.2.3 Il linguaggio ODLI3

Il linguaggio ODL (Object Definition Language) per la specifica di schemi ad oggetti proposto dal gruppo di standardizzazione ODGM-93 [**12**] è universalmente riconosciuto come standard. Le sue caratteristiche peculiari, al pari di altri linguaggi basati i sul paradigma ad oggetti, possono essere così riassunte:

- Definizioni di tipi-classe e tipi valore;
- Definizione fra intenzione ed estensione di una classe di oggetti.
- Definizione di attributi semplici e complessi.
- Definizione di attributi atomici e collezioni.
- Definizione di relazioni binarie con relazioni inverse.
- Dichiarazione delle segnature dei metodi.

Con l'estensione di ODL al linguaggio ODLI3 sono stati raggiunti i seguenti obiettivi:

- Per ogni classe il wrapper, può indicare nome e tipo di sorgente di appartenenza.
- Per le classi appartenenti alle sorgenti relazionali, è possibile definire le chiavi candidate ed eventuali *foreign key.*
- Attraverso l'uso del costrutto "*union*" ogni classe può avere più strutture alternative, mentre il costrutto "*optional*" consente di indicare la natura opzionale di un attributo. Queste caratteristiche, sono in accordo con la strategia utilizzata per la descrizione di dati semi-strutturati.

• Il linguaggio supporta la definizione di grandezze locali e di grandezze globali.
• Il linguaggio supporta la dichiarazione di regole di *mapping* fra grandezze locali e di grandezze globali.
• È data la possibilità di definire regole di integrità sia sugli schemi locali che globali;
• Il linguaggio supporta la definizione di relazioni terminologiche di sinonimia, ipernimia, iponimia, e associazione;
• Il linguaggio può essere automaticamente tradotto nella logica descrittiva OLCD usata da ODB-Tools, e quindi utilizzarne le capacità nei controlli di consistenza e nell'ottimizzazione semantica delle interrogazioni. [13]

1.3 ODB-Tools Engine

Uno degli aspetti più innovativi di MOMIS è rappresentato dall'impiego di logiche descrittive e tecniche di intelligenza artificiale sia in fase di costruzione della vista globale sia dell'ottimizzazione semantica delle interrogazioni.

Questi comportamenti intelligenti sono stati introdotti utilizzando ODB-Tools che è uno strumento software, sviluppato presso il Dipartimento di Ingegneria dell'Università di Modena [7, 8], per la validazione di schemi e l'ottimizzazione semantica di interrogazioni per le Basi di Dati Orientate agli Oggetti (OODB).

Gli algoritmi operanti in ODB-Tools sono basati su tecniche di inferenza che sfruttano il calcolo della *sussunzione* e la nozione di *espansione semantica* di interrogazioni per la trasformazione delle query al fine di ottenere tempi di risposta inferiori.

Il primo concetto è stato introdotto nell'area dell'Intelligenza Artificiale, più precisamente nell'ambito delle Logiche Descrittive, il secondo nell'ambito delle Basi di Dati. Questi concetti sono stati studiati e formalizzati in **OLCD** (*Object Languages with Complements allowing Descriptive cycles*), una logica descrittiva per basi di dati. Come interfaccia verso l'utente esterno è stata scelta la proposta ODMG-93 [12], utilizzando il linguaggio ODL (Object Definition Language) per la definizione degli schemi ed il linguaggio OQL (Object Query Language) per la scrittura di query. Entrambi i linguaggi sono stati estesi per la piena compatibilità al formalismo **OLCD**(**O**bject **L**ogics for **C**onstraint **D**escription).

Capitolo 2

2 Il Query Manager

Il MOMIS **Query Manager** è il modulo software che è in esecuzione durante tutta la fase di elaborazione della query globale inserita dall'utente. E' composto da diversi moduli che consentono di eseguire query sulla **Global Virtual View** (GVV), genera automaticamente un Query Plan per l'esecuzione della query globale, esegue l'insieme delle query locali sulle sorgenti, effettua la fusione delle risposte locali in una risposta unificata consistente e concisa, e ritorna la risposta globale all'utente.

L'utente può essere una persona che interagisce con la **Graphical User Interface** (GUI) per comporre e visualizzare i risultati, oppure un'applicazione software che interagisce con il Query Manager mandando queries e ricevendo risposte attraverso la rete.

2.1 Architettura

L'architettura del **Query Manager** è organizzata in diversi blocchi funzionali che coincidono coincidono con i moduli software implementati. Il **Query Manager** è scritto in Java. Il software è suddiviso in 10 package ed è composto da più di 100 classi.

La figura 2.1 mostra l'architettura:

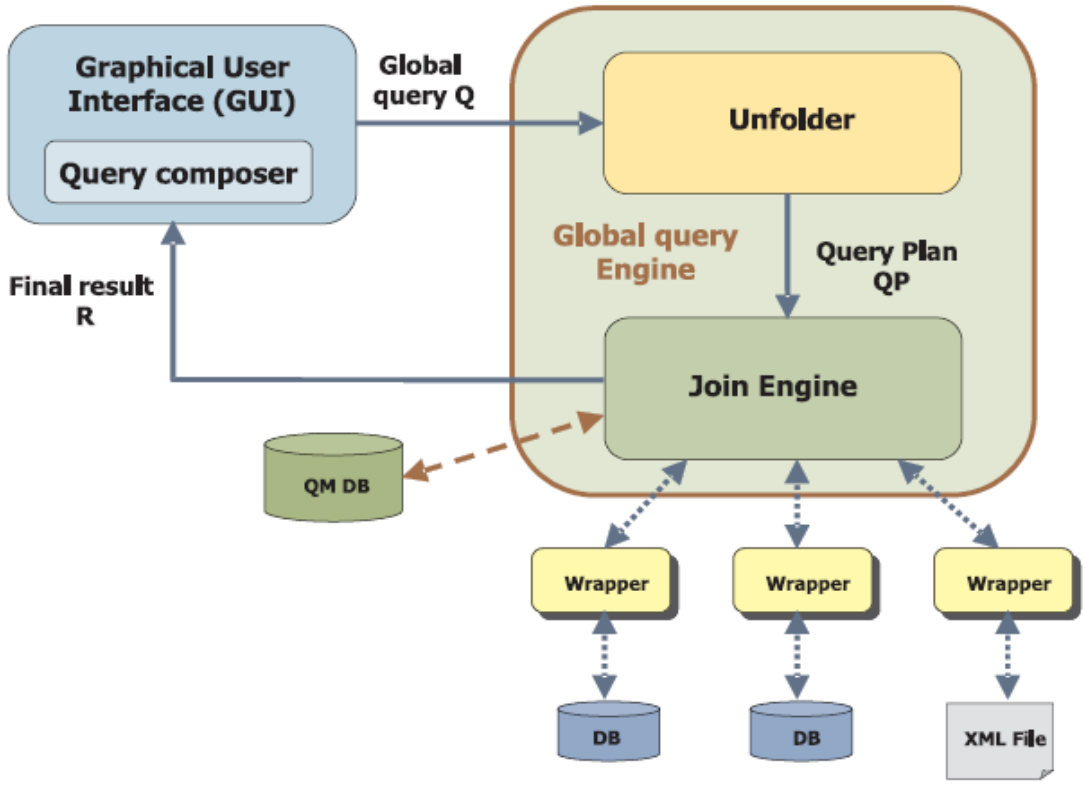

Figura 2.1: Architettura del Query Manager

- **Graphical User Interface (Query Composer):** permette all'utente di comporre la query sulle classi globali della GVV attraverso un'interfaccia grafica presentando la GVV in una rappresentazione ad albero. La query composta può quindi essere eseguita e il risultato è visualizzato in una tabella.
- **Unfolder:** riceve la globale query e mediante il mapping tra gli attributi genera il **Query Plan** QP. Il QP è composto dall'insieme delle local queries da eseguire sulle sorgenti, dalla

mapping query e dalla final query, consiste nell'insieme delle operazioni da compiere per rispondere alla query globale.

• **Join Engine:** riceve come input il QP e esegue le query che contiene per ottenere il risultato finale. Il Join Engine, prima esegue le local queries sulle sorgenti simultaneamente, poi unisce i risultati parziale eseguendo la mapping query, e infine ottiene il risultato globale applicando la final query. Ogni query locale è mandata al rispettivo wrapper per la traduzione nello specifico linguaggio della sorgente, così che la query può essere eseguita nella sorgente dati. Un database relazionale funge da supporto per la fusione dei dati parziali, che sono memorizzati in tabelle temporanee.

L'architettura mostrata in figura 2.1 considera l'esecuzione di una global query espressa su una sola singola classe globale. Nel caso in cui la global query che deve essere eseguita dal Query Manager è espressa su più di una classe globale, l'architettura è mostrata in figura 2.2:

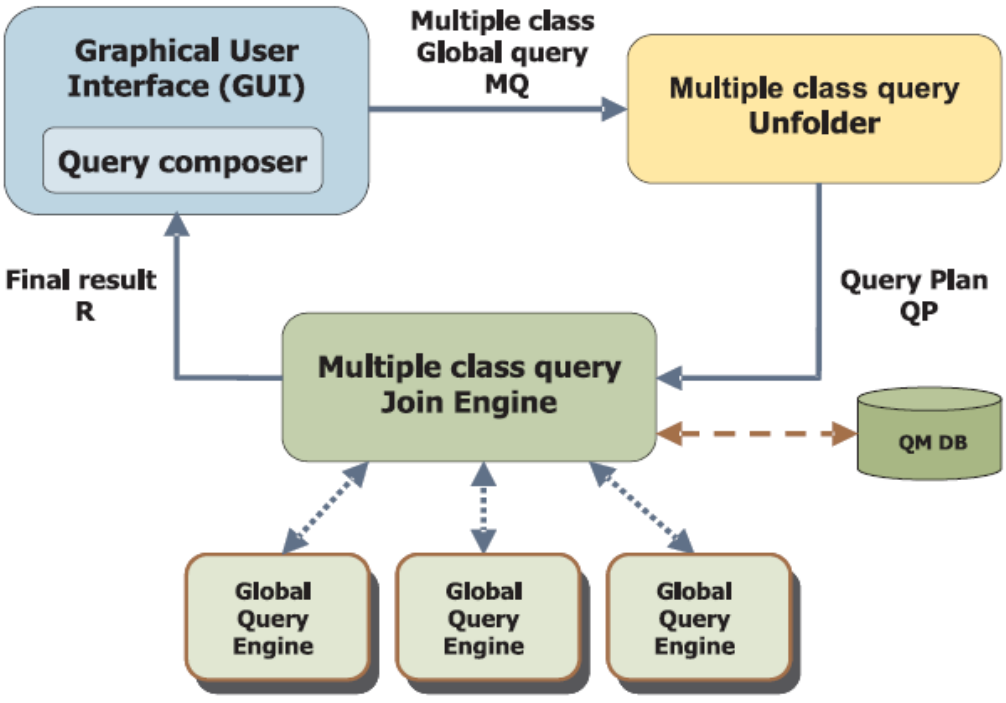

Figura 2.2: Architettura del Query Manager per query su più classi globali

- **Graphical User Interface (Query Composer):** le query espresse su più classi globali possono essere composte dall'utente con l'aiuto del modulo **Query Composer**. Più di una classe globale possono essere unite semplicemente scegliendole dalla finestra “Referenced Classes” di quella attualmente selezionata, senza il bisogno di specificare alcuna condizione di join.
- **Multiple class query Unfolder:** riceve la query e considerando le clausole di join presenti genera un **Query Plan**. Il QP è composto dall'insieme delle global query, ognuna delle quali coinvolge una sola classe globale e dalla join query.
- **Multiple class query Join Engine:** riceve come input il QP ed esegue le global queries ottenendo i risultati finali. Il **Join Engine** prima esegue l'insieme delle global queries simultaneamente, quindi unisce i risultati parziali eseguendo la join query per ottenere i risultati finali. Ogni global query è mandata al **Global Query Engine** per essere eseguita sulle sorgenti dati. Come mostrato in figura 2.2 ogni **Global Query Engine** è composto da un modulo **Unfolder** e da un **Join Engine**. Un database relazionale funge da supporto per la fusione dei dati parziali, che sono memorizzati in tabelle temporanee. [14]

2.2 I package e le classi

L'esecuzione del Query Manager coinvolge un elevato numero classi, perciò nel class diagram mostriamo solo il collegamento tra quelle principali.

package it.unimo.dbgroup.momis.SIDesigner.qmInterface: QmInterfacePanel

Questo pannello contiene l'interfaccia per usare il Query Manager, ovvero la Graphical User Interface (GUI).

package it.hln.momis.queryManager.joinEngine.function: DatabaseHandler

Gerarchia che crea l'handler per il DBMS di supporto e mette a disposizione i metodi per consentirne l'utilizzo.

package it.hln.momis.queryManager.gui: QryMngPanelGui

Classe in cui è definito il pannello della GUI.

package it.hln.momis.queryManager.joinEngine: JoinEngine

Classe che si occupa di eseguire le query contenute nel Query Plan. Gestisce il database interno, i Wrapper e tutte le operazioni di reperimento dati e di manipolazione dei dati ottenuti.

package it.hln.momis.oql: Oql_SelectExpr, Oql_Query

La prima rappresenta la query globale con le sue clausole: select, from sono obbligatorie, where, order by, group by sono opzionali.

La seconda è la super classe dalla quale tutte le queries OQL sono derivate.

package it.hln.momis.oql.query: Oql_QueryPlan, Oql_ExpandedQueryPlan

La prima si occupa di definire del Query Plan, la seconda gestisce la risoluzione di una query globale multiple class.

package it.hln.momis.oql.parser: Oql_Parser

Si occupa del parse della query e ne ricava le clausole con le quali crea delle istanze di Oql_SelectExpr.

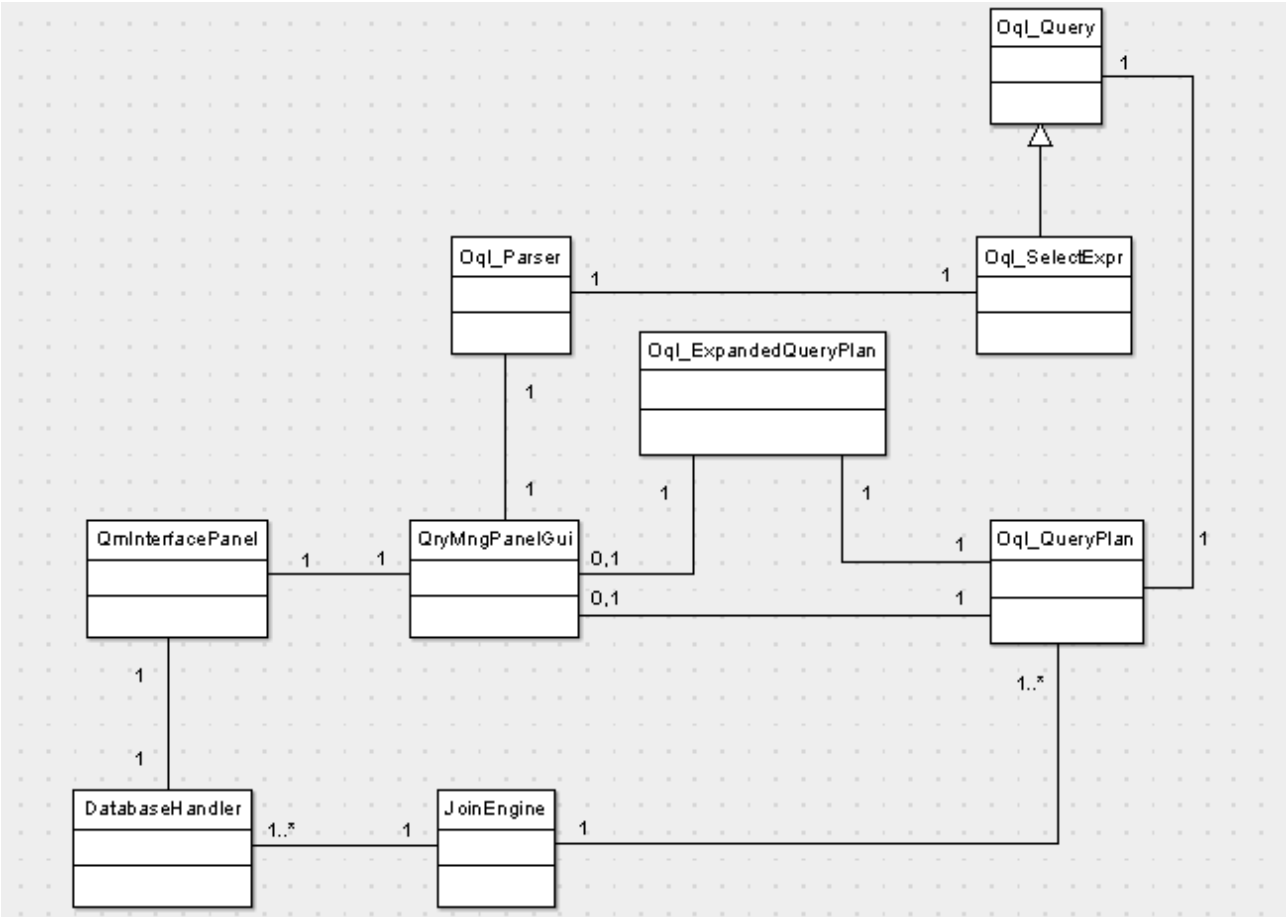

Figura 2.3: Class Diagram delle principali classi del Query Manager

Entreremo nel dettaglio di come funziona il Query Manager e in particolare di come viene gestita la query globale, prendendo come esempio la seguente situazione, dove i tre database PRONTOCOMUNE, FIBRE2FASHION e USAWEAR sono le sorgenti.

CLASSI GLOBALI	CLASSI LOCALI		
GVV	**PRONTOCOMUNE**	**FIBRE2FASHION**	**USAWEAR**
Address	**Indirizzo**		
Code	Codice		
Region	Regione		
Street	Via		
Town	Comune		
ZipCode	CAP		
Category	**Categoria**	**Category**	
CategoryCode	CodiceCategoria	CategoryCode	
Description	Descrizione	Description	
SubCategory		SubCategory	
SubCategory		**SubCategory**	
Description		Description	
SubCategory		SubCategoryCode	
Company	**Azienda**	**Company**	**Company**
Address	Indirizzo	Address	Address
Category	Categoria	Category	
Description		AboutUs	Description
E-mail	E-mail	E-mail	E-mail

Dopo aver avviato il software MOMIS, per interrogare la GVV dobbiamo accedere alla tabella che riguarda il Query Manager ed è da questo punto che inizia la nostra descrizione.

Fase 1: Inizializzazione

L'aver selezionato la tabella “Query Manager” fa si che la classe **QmInterfacePanel** crei un oggetto **QryMngPanelGui** passandogli lo schema della GVV e il driver per connettersi al DBMS di supporto che useremo per le tabelle temporanee.

E' nella classe **QryMngPanelGui** che viene inizializzato il **JoinEngine**, oggetto che effettua la connessione al database e grazie allo schema della GVV crea prima le tabelle globali

```
Join_Eng_Address

Join_Eng_Category

Join_Eng_SubCategory

Join_Eng_Company
```

e poi quelle locali

```
Join_Eng_Address_prontocomune_Indirizzo

Join_Eng_Category_prontocomune_Categoria

Join_Eng_Category_fibre2fashion_Category

Join_Eng_SubCategory_fibre2fashion_SubCategory

Join_Eng_Company_prontocomune_Azienda

Join_Eng_Company_fibre2fashion_Company

Join_Eng_Company_usawear_Company
```

Ritornato il controllo al **QryMngPanelGui**, non è chiaro il perché della creazione di un'altro **JoinEngine** che ricrea le stesse tabelle, ma con clientID Join_Eng001 al posto di Join_Eng.

Fase 2: Inserimento query globale

Il controllo viene passato all'utente che inserisce la query globale e seguendo il nostro esempio questa sarà:

```
Select *
from Category as C, SubCategory as D
where C.SubCategory = D.SubCategoryCode and C.Description like '%import%'
order by C.CategoryCode,C.Description
```

E' il **QryMngPanelGui** che si accorge che è stata inserita una query, poiché è stato premuto il pulsante “exec....”.

Viene eseguito il parse della query, ovvero nella classe **Oql_Parser** viene creato un oggetto **Oql_SelectExpr** che contiene la query e la sua decomposizione nelle clausole from, where, order by e group by.

L'oggetto ritornato al **QryMngPanelGui** contiene quindi nella clausola from le classi globali su cui dobbiamo eseguire la query: se è solo una viene subito generato il Query Plan, altrimenti come nel nostro esempio, la query globale deve essere riscritta in più query globali dove però è coinvolta un'**unica** classe globale.

1

```
SELECT C.CategoryCode , C.Description , C.SubCategory  FROM Category as C
WHERE (C.Description like  '%import%' )
```

2

```
SELECT D.Description , D.SubCategoryCode  FROM SubCategory as D
```

Quindi viene creato un oggetto **Oql_ExpandedQueryPlan** a cui è passato il vettore contenente queste query che sono state anche loro parserizzate.

Nella classe **Oql_ExpandedQueryPlan** vengono creati tanti JoinEngine quante sono le classi globali + 1, quindi secondo il nostro esempio 3, usando come clientID la posizione della query nel

vettore + 1; vengono quindi create le tabelle mostrate nella fase 1 usando come clientID

```
Join_Eng001

Join_Eng002

Join_Eng003
```

anche questo passaggio è risultato poco chiaro.

A questo punto vengono creati i Query Plan per ogni query globale.

Fase 3: calcolo del Query Plan

Come detto nelle fase 2, le queries che otteniamo sono degli oggetti Oql_SelectExpr che hanno come classe padre **Oql_Query** che contiene i metodi getLocalQueries, getGlobalQuery e getFinalQuery i quali ritornano rispettivamente le queries locali, la mapping query e la query finale.

Tutte queste queries sono ottenute confrontando i mapping tra gli attributi delle classi locali e globali e soprattutto considerando le condizioni della clausola where. Facendo riferimento al nostro esempio, otteniamo il Query Plan sulla prima query, ovvero sulla classe globale **CATEGORY**, che contiene:

LOCAL QUERIES:

globalSource.Category, fibre2fashion.Category

```
SELECT Category.SubCategory, Category.CategoryCode, Category.Description

FROM Category

WHERE (Description) LIKE ('%import%')
```

globalSource.Category, prontocomune.Categoria

```
SELECT Categoria.CodiceCategoria, Categoria.Descrizione

FROM Categoria

WHERE (Descrizione) LIKE ('%import%')
```

MAPPING QUERY:

```
select "Join_Eng001_Category_fibre2fashion_Category"."SubCategory" AS
SubCategory_1, "Join_Eng001_Category_fibre2fashion_Category"."CategoryCode" AS
CategoryCode_1, "Join_Eng001_Category_prontocomune_Categoria"."CodiceCategoria"
AS CategoryCode_2, "Join_Eng001_Category_fibre2fashion_Category"."Description"
AS Description_1, "Join_Eng001_Category_prontocomune_Categoria"."Descrizione" AS
Description_2

from "Join_Eng001_Category_fibre2fashion_Category" full outer join
"Join_Eng001_Category_prontocomune_Categoria" on
((("Join_Eng001_Category_prontocomune_Categoria"."CodiceCategoria") =
("Join_Eng001_Category_fibre2fashion_Category"."CategoryCode")))
```

FINAL QUERY:

```
SELECT CategoryCode, Description, SubCategory FROM "Join_Eng001_Category"
```

Il Query Plan sulla seconda query contenente la classe globale **SUBCATEGORY**:

LOCAL QUERIES:

globalSource.SubCategory, fibre2fashion.SubCategory

```
SELECT SubCategory.SubCategoryCode, SubCategory.Description FROM SubCategory
```

MAPPING QUERY:

```
select "Join_Eng002_SubCategory_fibre2fashion_SubCategory"."SubCategoryCode" AS
SubCategoryCode_1,
"Join_Eng002_SubCategory_fibre2fashion_SubCategory"."Description" AS
Description_1 from "Join_Eng002_SubCategory_fibre2fashion_SubCategory"
```

FINAL QUERY:

```
SELECT Description, SubCategoryCode FROM "Join_Eng002_SubCategory"
```

Fase 4: esecuzione Query Plan

L'esecuzione del Query Plan avviene subito dopo che è stato calcolato, perciò non avviene il calcolo di tutti come mostrato nella fase 3. E' il **JoinEngine** che si occupa di eseguire le local queries, di inserire i risultati nelle tabelle tamporanee, di unirli eseguendo la mapping query e di creare una vista contenente la final query.

```
CREATE VIEW Join_Eng001_0_ResultView_TMP AS SELECT CategoryCode, Description,
SubCategory FROM "Join_Eng001_Category"
```

```
CREATE VIEW Join_Eng002_0_ResultView_TMP AS SELECT Description, SubCategoryCode
FROM "Join_Eng002_SubCategory"
```

Fase 5: Fusione dei risultati

Questa fase è presente solo per le query dell'utente su più classi globali, come nel nostro esempio. Viene calcolata una nuova query finale che unisce le viste create alla fase 4, e in cui sono inserite le clausole prima trascurate, come quella di order by e le condizioni di where non ancora soddisfatte.

```
CREATE VIEW Final_View AS
SELECT TOP 100 PERCENT C.CategoryCode AS CategoryCode_Category , C.Description
AS Description_Category , C.SubCategory AS SubCategory_Category , D.Description
AS Description_SubCategory , D.SubCategoryCode AS SubCategoryCode_SubCategory
FROM Join_Eng001_0_ResultView as C , Join_Eng002_0_ResultView as D
WHERE (C.SubCategory = D.SubCategoryCode )
ORDER BY C.CategoryCode ASC , C.Description ASC
```

All'utente vengono mostrati i risultati ottenuti dalla query:

```
Select * from Final_View
```

Il controllo torna a **QryMngPanelGui** che conclude l'esecuzione.

Con lo schema temporale in figura 2.4, cerchiamo di riassumere come il controllo dell'esecuzione è passato tra le classi principali.

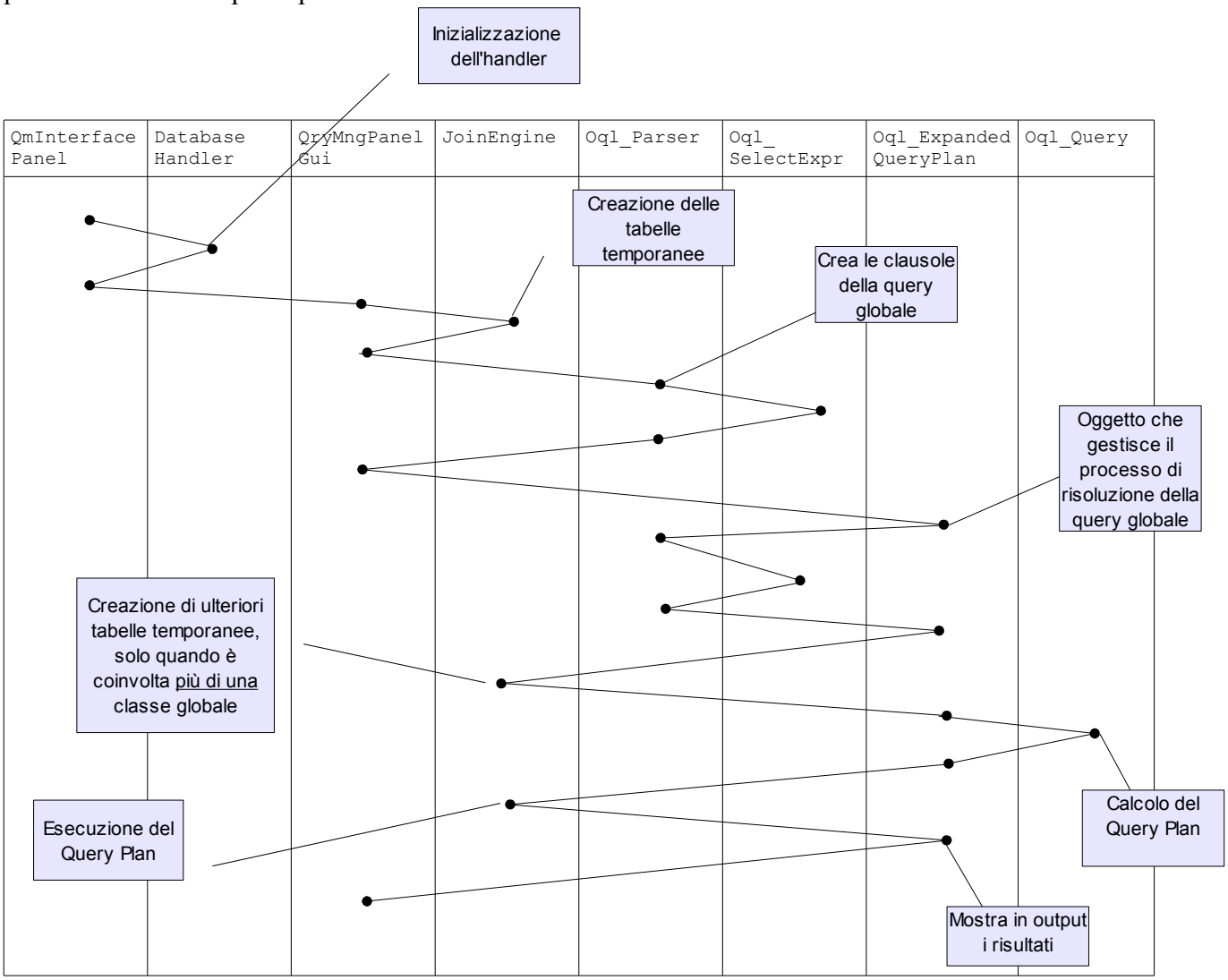

Figura 2.4: diagramma temporale dell'esecuzione del Query Manger

Capitolo 3

3 Il Porting

Il DBMS di supporto utilizzato da MOMIS, sul quale il Query Manager crea le tabelle temporanee globali e locali, è stato fin dall'inizio SQLServer di Microsoft. Essendo uno tra i DBMS migliori e completi la scelta è motivata dal fatto di ottenere così un software competitivo, ma per poterlo utilizzare è necessaria una licenza e un sistema operativo Windows. Visto che uno degli obbiettivi futuri è far di MOMIS un software open source, l'utilizzo di un DBMS open che sostituisca SQLServer è un'operazione fondamentale e prioritaria.
All'inizio di questa tesi ci si era preposti di affiancare a SQLServer, un altro DBMS, però open source, PostgreSQL, che aveva vinto il ballottaggio con MySQL, in quanto MySQL non mette a disposizione il full outer join, operatore indispensabile per il corretto funzionamento del Query Manager. In questo modo l'utente avrà la possibilità di scegliere il DBMS a seconda del contesto in cui verrà usato MOMIS.
Oracle, come SQLServer, è un DBMS commerciale, che però ha il grande vantaggio di essere multi-piattaforma, e di non dover essere installato solo su macchine Windows.

3.1 L'idea dei Driver

Questa nuova possibilità di poter scegliere il DBMS, fa si che tutti gli statement SQL, che il Query Manager esegue sul database di supporto, non dovranno più essere scritti solo nella versione per SQLServer, ma dovranno essere generati in conformità con il *dialetto* SQL del DBMS scelto. Quindi serviranno delle classi che fungano da driver, ovvero classi che contengano l'insieme di procedure necessarie per generare i corretti statement SQL e che consentano al Query Manager di funzionare sempre allo stesso modo senza preoccuparsi del DBMS scelto.

Per poter fare ciò, è stata creata una gerarchia di classi **DatabaseHandler**, che ha come classe padre astratta **DatabaseHandler_base** in cui sono definiti tutti i metodi usati nella gerarchia e ognuno di questi metodi è specializzato ed è in grado di generare la sintassi SQL richiesta dal relativo DBMS. Quindi a seconda del DBMS scelto, dovrà essere inizializzato il giusto driver (o handler).
All'interno di queste classi, oltre ai metodi fondamentali (creare tabelle, indici, viste,…..) saranno presenti anche i metodi che servono per gestire altre particolarità del DBMS come ad esempio la lunghezza massima dei nomi.
La gerarchia che otterremo avrà quindi tante classi figlie quanti sono i DBMS che Momis potrà utilizzare, al momento due, MsSqlServer e PosgreSQL.

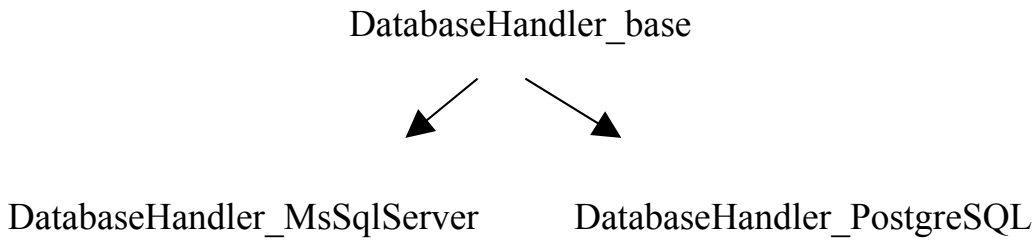

figura 3.1

3.2 PostgreSQL

Il lavoro di porting è iniziato con una prima fase di analisi del codice per trovare in quali punti il Query Manager (QM) si collega al DBMS, e in particolare nella ricerca degli statement SQL di CREATE e INSERT delle tabelle globali e locali. Una volta trovati, seguendo la gerarchia di figura 1, li abbiamo inseriti nella classe **DatabaseHandler_base**, che quindi conterrà tutti i metodi che fino ad adesso hanno consentito al QM di funzionare con SQLServer; seguendo questo approccio nella classe **DatabaseHandler_MsSqlServer** i metodi già presenti in Momis non saranno specializzati, mentre nella classe **DatabaseHandler_PostgresSQL** si, ovviamente solo se richiedono una sintassi diversa.
I metodi che aggiungeremo per gestire le particolarità di PostgreSQL saranno introdotti nella classe padre e specializzati in quelle figlie a seconda delle richieste del DBMS.

Dopo aver riscritto i metodi necessari per la creazione e l'utilizzo delle tabelle globali e locali secondo la sintassi di PostgreSQL, abbiamo provato ad eseguire una query di esempio. Il risultato non è stato molto incoraggiante, infatti un errore ci ha fermati:

ERROR: FULL JOIN is only supported with merge-joinable join conditions

L'operazione di full outer join, che avviene tra le tabelle locali, è implementata in PostgreSQL con un merge join, e quindi viene eseguita su tabelle che devono essere prima ordinate e poi confrontate record per record con un operatore di uguaglianza.
Nel nostro caso la presenza dell'**OR** nella clausula -on- del full outer join non permette l'ordinamento dei due flussi prima del match e questo fa si che il full outer join non possa essere eseguito.
Un esempio non funzionante:

```
select "Join_Eng_Company_fibre_Company"."Tel" AS Phone_1
   , "Join_Eng_Company_pronto_Azienda"."Telefono" AS Phone_2
   , "Join_Eng_Company_usa_Company"."Phone" AS Phone_3
from "Join_Eng_Company_pronto_Azienda"
        full outer join "Join_Eng_Company_fibre_Company"
                on ((("Join_Eng_Company_fibre_Company"."Name") =
                    ("Join_Eng_Company_pronto_Azienda"."Nome")))
        full outer join "Join_Eng_Company_usa_Company"
             on ((("Join_Eng_Company_usa_Company"."CompanyName") =
                     ("Join_Eng_Company_pronto_Azienda"."Nome"))
                OR (("Join_Eng_Company_usa_Company"."CompanyName") =
                            ("Join_Eng_Company_fibre_Company"."Name")))
```

Il problema si può risolvere riscrivendo la query come l'unione tra il left join e il right join, il che però non è efficiente e se ci sono sottoquery il risultato può non essere corretto se l'output non è costante.
Visto questo problema con PostgreSQL si è pensato di cambiare DBMS, il nuovo obbiettivo è il porting del QM su Oracle Express.

Per informazioni sul full outer join in PostgreSQL:
http://qaix.com/postgresql-database-development/500-442-error-full-join-is-only-supported-with-merge-joinable-join-conditions-read.shtml

3.3 Oracle Express

Oracle Express (XE) è la versione Free del più diffuso RDBMS commerciale: Oracle. Si tratta di una versione (Edition per essere precisi) con alcuni limiti, ma per il resto analoga e perfettamente compatibile con le altre versioni commerciali di Oracle (http://www.oracle.com/index.html).
Come detto, a differenza di PostgreSQL, non è opensource, ma **free** e può essere installato su diverse piattaforme tra cui Windows e Linux; questo cambia l'obbiettivo di rendere Momis open source, però è sempre un passo avanti verso la portabilità di Momis su DBMS diversi da SQLServer.
Dall'handler per PostgreSQL deriviamo quello per Oracle Express, copiando la classe **DatabaseHandler_PostgreSQL** in **DatabaseHandler_OracleExpress** e riscrivendo i metodi che erano al suo interno con la sintassi per il nuovo DBMS.

3.3.1 Problemi e soluzioni

In questa sezione sono riportati i problemi che sono emersi e risolti durante la creazione dell'handler per Oracle.
(1) Identificativi con più di 30 caratteri: in Oracle i nomi di tabelle e attributi non possono essere piú lunghi di 30 caratteri
(2) Uso delle virgolette: se sono usate nella definizione dei nomi di tabella e attributi hanno un comportamento diverso da SQLServer
(3) Prerared Statement: Oracle preferisce usare i Prepared Statement piuttosto che i resutlSet per inserire i dati nelle tabelle
(4) Tipi di dato NUMBER: tutti i valori numerici sono creati come tipo NUMERIC, questo non va bene per gli interi che devono essere INTEGER
(5) Sostituzione clientID: a seconda del JoinEngine che si sta usando, bisogna sostituire nel nome di tabella la stringa #ID->ID_RIF# con il suo clientID

Identificativi con più di 30 caratteri

Dopo l'esecuzione della prima query, il **primo problema** riscontrato è che Oracle non accetta nomi identificativi di oggetti del database, con più di 30 caratteri; visto che le tabelle temporanee locali sono costruite secondo la formula

clientID + nome classe globale + nome sorgente + nome classe locale

è molto probabile che queste superino il limite; quindi per proseguire abbiamo fatto in modo che tutti gli oggetti coinvolti nell'esecuzione della query siano regolari, posticipando la creazione di una funzione di accorciamento dei nomi.

Uso delle virgolette

Risolto per il momento il problema precedente con l'utilizzo di nomi che non superano i 30 caratteri, abbiamo proseguito con l'esecuzione della nostra query e il **secondo problema** riscontrato è che l'uso delle virgolette nella definizione delle tabelle temporanee hanno un comportamento diverso da SQLServer.

Se in SQLServer si crea una tabella "Nome", i suoi identificativi possono essere "Nome" o Nome. Se invece è creata una tabella Nome, il suo identificativo è Nome o nome, quindi le virgolette hanno una funzione di case sensitive.

Anche in Oracle hanno questa funzione, ma se definiamo una tabella o una colonna con le virgolette, quando vogliamo accedergli non possiamo ometterle, fanno parte del nome.

Infatti se creiamo una tabella "Nome" il suo **unico** identificativo è "Nome", e se proviamo ad accedergli con Nome, viene dato errore: tabella inesistente.

Visto che MOMIS non considerava questo, perché in SQLServer non da problemi, adesso con Oracle bisogna che ogni volta che si accede ad una tabella temporanea i suoi identificativi siano compresi delle virgolette, perché al momento della creazione sono usate.

Nel metodo INSERT, poiché viene invocato solo in un punto (nella classe **WrapperThread**) il problema è stato risolto concatenando ai nomi le virgolette.

Per tutti gli altri casi in cui i nomi di tabella o colonna devono essere usati per una query sulle tabelle temporanee, viene chiamata la funzione dell'handler **getName()** che si occupa racchiudere la stringa passata tra virgolette solo se il database di supporto è Oracle.

Prerared Statement

Risolto il problema di formattazione, ci siamo trovati di fronte ad un **problema (terzo)** che riguarda i resultSet che ritornano dall'esecuzione delle query SQL, in particolare con il cursore che naviga la tabella del resultSet. Anche se creiamo una connessione aggiornabile e uno statement che genere un ResultSet FORWARD_ONLY e CONCUR_UPDATABLE, quando proviamo a muovere il cursore per scorrere il ResultSet, un errore ci ferma:

operazione non valida nel result set di sola lettura: moveToInsertRow

Non trovando un motivo a questo errore, abbiamo comunque risolto il problema riscrivendo il codice adottando una soluzione in generale più appropriata riguardo alla situazione.
Abbiamo infatti usato un PreparedStatement (PS), con un'istruzione di insert che aggiunge direttamente la tupla alla tabella:

```
String sql = "insert into "+table+" ("+attrNames+") values ("+attrNamesQuestions+")";
PreparedStatement ps = this.connection_d.getConn().prepareStatement(sql);
```

dove **attrNames** è la lista delle colonne e **attrNamesQuestions** è la lista dei ? ai quali in seguito saranno sostituiti i valori; con questo codice non abbiamo più il problema del cursore, perché inseriamo la tupla corrente direttamente nella tabella e non tramite il ResultSet.

Tipi di dato NUMBER

Fatto ciò l'esecuzione della query riesce a terminare e MOMIS ci ritorna il risultato, resta solo da verificare se corretto.
Infatti abbiamo ancora un **difetto (quarto)** che riguarda i valori interi che sono inseriti nelle tabelle, perchè Oracle per definire tutti i valori numerici usa solo il tipo di dato NUMBER[(….)]; il problema è che NUMBER è un tipo di dato SQL NUMERIC in cui a seconda di cosa è inserito tra parentesi viene considerato intero, float, decimal.
Quando nell'esecuzione della query è necessario sapere quale è il tipo di colonna per settare il valore dei ? nel PS, se abbiamo NUMBER, che è un tipo SQL NUMERIC, verrebbe usato il metodo ps.setDouble() anche per valori interi (vedi tabella al paragrafo 5.3).
Anche questo problema è stato risolto modificando il codice del Query Manager, aggiustando i valori da passare ad un metodo che non erano quelli maggiormente adatti.

Sistemato questo, finalmente Oracle esegue la query correttamente e riporta i giusti risultati. Dobbiamo ancora però fare in modo che tutto funzioni anche con oggetti di database con identificativi più lunghi di 30 caratteri.

3.3.2 Funzione getShortName

Metodo che sarà specializzato solo nella classe DatabaseHandler_OracleExpress e verrà richiamato ogni volta che il QM dovrà dare il nome ad una tabella, usato soprattutto per le tabelle locali.
Visto che i nomi di colonna non vengono "allungati" da Momis e le uniche viste create hanno sempre nome minore di 30 caratteri, l'utente che usa Oracle deve saperedi questa sua particolarità e quindi fare in modo che le uniche possibili eccezioni riguardino i nomi di tabella.

getShortName() accetta come parametro una stringa che è il nome originale della tabella (longName) e ritorna il suo shortName

Se longName è maggiore di 29 caratteri viene troncato a 25 e gli viene aggiunto un numero di 4 cifre che lo rende univoco, infatti questo numero viene incrementato ogni volta che viene usato.
Questa corrispondenza **longName → shortName** viene inserita in una hashTable (_tableLongName) e anche il viceversa **shortName → longName** è inserito in un'altra hashTable (_tableShortName).
Se il longName che viene passato a getShortName() è già presente in _tableLongName, allora è subito ritornato lo shortName.
Sono da considerare anche i nomi di tabella che non subiscono modifiche sono inseriti nelle due hashTable, questo per poter gestire anche i casi in cui esista già una tabella con uno shortName che è uguale ad un longName che non deve essere accorciato.
Facendo tutti i controlli utilizzando le 2 hashTable appositamente create, avremo che il Query Manager avrà a che fare con nomi di tabella tutti univoci e potrà utilizzarle senza problemi, tutto questo in modo trasparente all'utente.
Il metodo getShortName, come anche quelli descritti successivamente, sono stati testati usando una test unit chiamata: **DatabaseHandler_OracleExpress_JUTest**, vedi il paragrafo 5.1.1.

Sostituzione #ID→ID_RIF#

Inserita questa funzione, pensavamo di aver concluso il porting, ma ancora una volta qualcosa non permetteva il corretto funzionamento.

Cercando dove si verificava il nuovo **problema (quinto)**, siamo arrivati alla funzione replaceAll() nella classe **Oql_QueryPlan**.

Questa funzione compiva sostituzioni di stringhe nelle query che poi venivano eseguite, per ottenere i giusti nomi di tabella da usare

1° sostituzione	"#ID→ID_RIF#_*nomeTabella*" → "Join_Eng_*nomeTabella*"
2° sostituzione	"Join_Eng_*nomeTabella*" → "Join_Eng001_*nomeTabella*"

Per quanto riguarda la 1° sostituzione nessun problema (getShortName() non accorcia i nomi di tabella che iniziano con #ID→ID_RIF# perché non verranno create e quindi non saranno nemmeno presenti nelle 2 hashTable).

Quando invece avviene la 2° sostituzione, se un nome di tabella è stato accorciato la sostituzione ci darà un nuovo nome di tabella che sicuramente non esiste, per cui prima di effettuarla dobbiamo ricavare il nome originale (con **getLongName**, paragrafo 3.3.3) e poi sostituire; è questo il punto in cui cui replaceAll() non va più bene se viene usato Oracle.

Dobbiamo quindi definire una nuova funzione che prende il posto di replaceAll(), che permetta il corretto funzionamento, e l'abbiamo chiamata **getRealQuery**.

Esempio di come avviene questo passaggio:

1° sostituzione

"#ID→ID_RIF#_Company_fibre_Category" → "Join_Eng_Company_fibre_Category"

Visto che la tabella "Join_Eng_Company_fibre_Category" supera i 30 caratteri, quando è stata creata è il suo nome è stato accorciato a "Join_Eng_Company_fibre_Ca0001".

2° sostituzione

"Join_Eng_Company_fibre_Ca0001" → "Join_Eng001_Company_fibre_Ca0001"

Errore, la tabella "Join_Eng001_Company_fibre_Ca0001" non esiste!

La sostituzione deve avvenire sui nomi originali, quindi prima dobbiamo usare getLongName() e poi sostituire. Correttamente sarà:

2° sostituzione

"Join_Eng_Company_fibre_Category" → "Join_Eng001_Company_fibre_Category"

Anche in questo caso "Join_Eng001_Company_fibre_Category" supera i 30 caratteri e la tabella sarà stata creata con "Join_Eng001_Company_fibre0002".

3.3.3 Funzione getLongName

Funzione duale a getShortName(), viene usata solo nella funzione getRealQuery() per ottenere il nome originale di un nome che è stato accorciato.

getLongName() accetta una stringa che sarà lo shortName
di una tabella e ritorna il suo nome originale (longName)

Molto più semplice da implementare perché dovrà solo cercare in _tableShortName la chiave shortName e ritornare il suo valore (longName).

3.3.4 Funzione getRealQuery

E' stata inserita nella classe dove prima era usata replaceAll(), ovvero in **Oql_QueryPlan**.

getRealQuery() accetta 3 parametri: la query da eseguire,
la stringa da sostituire e la stringa sostituta;
ritorna la corretta query da eseguire

Una volta ottenuto dalla query in ingresso, con una substring, il nome della tabella (ovvero i nomi racchiusi tra virgolette), prima di eseguire la sostituzione, usiamo la funzione **getLongName()** (create apposta per questo passaggio) che ci ritorna il nome originale della tabella (se siamo nella 1° sostituzione ci ritorna lo stesso nome, poiché non è stato inserito nelle hashTable) e su questo

eseguiamo la sostituzione. In questo modo siamo sicuri che il nome ottenuto è di una tabella che è già stata creata nella fase di inizializzazione compiuta dal Query Manager, e che quindi, il seguente uso della funzione **getShortName(),** ci ritorni il giusto nome della tabella da usare, che è già presente nelle hashTable.

Messo a posto questa ultima parte, il porting su Oracle Express, può considerarsi completato.

3.4 Considerazioni Finali

Il principale problema affrontato è stato la gestione dei naming, a causa della diversa interpretazione che hanno Oracle e SQLServer delle virgolette negli identificativi di tabelle e colonne.
Gli identificativi che usa il Query Manager sono gli stessi con cui sono state definite le sorgenti; perciò se in origine le tabelle che abbiamo integrato possedevano tutti gli identificativi senza virgolette, che sono necessarie solo quando è presente uno spazio, anche durante l'esecuzione del QM non sono presenti.
Però una scelta implementativa di MOMIS è di creare tutte le tabelle temporanee con gli identificativi racchiusi tra virgolette e, visto il problema di Oracle, non sarà possibile accedergli se prima non saranno aggiunte.
Nei metodi CREATE TABLE e INSERT sono state concatenate ai nomi.
Invece “aggiustare” gli identificativi nelle queries non è stato così immediato, poiché a causa della suddivisione in clausole, sono numerosi i punti in cui bisogna settarli correttamente.
Nella costruzione delle local queries non ci sono problemi, visto che gli identificatori sono uguali a quelli delle sorgenti, mentre per le queries sulle tabelle temporanee (mapping e final query) è stata usata la funzione getName() definita nell'handler e specializzata per Oracle in modo che aggiunga le virgolette.
Non sono però stati considerati i casi in cui gli identificativi delle sorgenti siano definiti in modo errato, ad esempio:
-category code
-name “surname”
perchè il DBMS dovrebbe correggere la sintassi (“category code”) o non permetterne la creazione

(name “surname”). Resta comunque da verificare che se sia possibile avere questi identificativi, il QM sia in grado di trattarli correttamente.
A parte questo problema con le virgolette, gli altri sono stati piuttosto brevi da risolvere, poiché limitati a singole classi e risolti con nuove funzioni o correzione del codice.
Adesso che si possiede una gerarchia di driver per l'utilizzo dei database di supporto, in futuro di potrà effettuare il porting su altri DBMS più facilmente, anche se si potranno incotrare diversità sia da Oracle che da SQLServer che dovranno essere risolte con la definizione di nuove funzioni.

Capitolo 4

4 Differenze tra SQLServer e Oracle

Discutiamo le diverse modalità di implementazione e funzionamento tra i due DBMS che abbiamo incontrato durante questo studio e cosa abbiamo dovuto modificare per rendere il software funzionante.

4.1 Nomi identificativi

Nella classe **JoinEngineNaming**, che implementa l'interfaccia JoinEngineID, sono presenti i metodi per dare il nome alle tabelle, getGlobalTable() e getLocalTable().
Con SQLServer questi erano semplicemente:

```
public String getGlobalTable(String globalClass) {
      return "\""+clientID+"_"+globalClass+"\"";
}


public String getLocalTable(String globalClass,String source,String localClass)
{
      return
"\""+clientID+"_"+globalClass+"_"+source+"_"+localClass.replace('.','_')+"\"";
}
```

In Oracle è però presente un vincolo sulla lunghezza dei nomi di tabella e colonna, ovvero che il loro identificativo non può superare i 30 caratteri; quindi abbiamo dovuto riscrivere questi due metodi, aggiungendo la funzione **getShortName()** descritta al paragrafo 3.3.2.

```
public String getGlobalTable(String globalClass) {

      DatabaseHandler_base dbHandler = DatabaseHandler_base.getDbHandler();
      String s = clientID+"_"+globalClass;
      s = dbHandler.getShortName(s);
      s = "\""+ s +"\"";
      return s; }
```

```
public String getLocalTable(String globalClass,String source,String localClass)
{
      DatabaseHandler_base dbHandler = DatabaseHandler_base.getDbHandler();
      String s = clientID+"_"+gc+"_"+src+"_"+lc.replace('.','_');
      s = dbHandler.getShortName(s);
      s = "\""+ s +"\"";
      return s;

}
```

Inoltre come si può notare sono queste due le funzioni che aggiungono le virgolette ai nomi di tabella e che sono invocate, oltre che al momento della creazione, quando servono le tabelle locali per comporre la Global Query e le tabelle globali per la Final Query.

4.2 Tipi di dati

Abbiamo già parlato nel paragrafo 3.3.1 della differente dichiarazione dei tipi di dato numerici da parte dei due DBMS, adesso mostriamo in tabelle la corrispondenza tra tutti i tipi che possono essere usati.

Tipi di dati usati da Oracle per creare le tabelle locali:

ORACLE → SQLSERVER	
number	float, numeric([1,38]), numeric([0,38],[1,38])
varchar2	varchar
date	datetime
char	char
nvarchar2	nvarchar

Tipi di dati usati da SQLServer per creare le tabelle locali:

SQLSERVER → ORACLE	
int	number(38)
float	number(p,s)
nvarchar	nvarchar2
char	char
ntext	nvarchar2
datetime	date

4.3 Alias di tabella

In Oracle per dare un alias ad una tabella nella clausola From non si può inserire **AS**, ma bisogna inserire l'alias semplicemente dopo uno spazio, mentre per quanto riguarda le colonne si possono usare entrambi i modi.

Select Name [as] n from Company → **Ok**
Select C.Name from Company as C → **Errore!**

La presenza di questi **AS** non ci da problemi durante l'esecuzione del programma, perché in tutte le query locali, globali e finali, non sono usati, ma si usa il vero nome di tabella.
Solo alla fine, quando viene scritta la Final_View che è la copia della query inserita dall'utente, dobbiamo fare in modo che se sono presenti degli AS nella clausola from vengano tolti se il driver è quello di Oracle.

Oracle:
```
public String getFromClause(String sql) {
      sql = sql.replaceAll(" as ", " ");
      sql = sql.replaceAll(" AS ", " ");
      return sql;
}
```

SQLServer:
```
public String getFromClause(String s) {
      return s;
}
```

4.4 Viste ordinate

In SQLServer non è possibile inserire la clausola Order by nella creazione di una vista, senza che sia specificata la clausola TOP N PERCENT all'interno della query, per prendere l'N% dei dati ordinati; quindi quando Momis deve lo statement SQL della Final_View per ottenere il result set finale bisogna aggiungere questa clausola nella Select.
Oracle invece non necessita di tale modifica, poiché non ha questo problema.
Perciò, mentre nel software precedente la clausola Select della Final_View conteneva sempre TOP 100 PERCENT, che veniva aggiunta nella classe **Oql_SelectExpr**, adesso a seconda del driver utilizzato, ci pensa la funzione getOrderBy(), della classe **DatabaseHandler_base** ad inserirla o no.

SQLServer:
```
public String getLimitTo(String sql) {
      sql = sql + "TOP 100 PERCENT ";
      return sql;
}
```

Oracle:
```
public String getLimitTo(String sql) {
      return sql;
}
```

Capitolo 5

5 Le modifiche in MOMIS

Durante l'operazione di porting del Query Manager su Oracle Express, è stato necessario analizzare molto attentamente il funzionamento di numerose classi ed i relativi metodi.
Questa analisi ci ha permesso di trovare le parti di codice in cui effettuare le nostre modifiche, ma allo stesso tempo ci ha fatto scoprire alcuni passaggi implementati in maniera non proprio corretta, che quindi è stato deciso di migliorare.

5.1 Le nuove classi

Sono state inserite nel package **it.hln.momis.queryManager.joinEngine.function**, 3 nuove classi, 1 padre astratta (**DatabaseHandler_base**) e 2 figlie (**DatabaseHandler_MsSqlServer** e **DatabaseHandler_OracleExpress**) che rappresentano i driver dei possibili DBMS che Momis può usare (fig. 1). L'aggiunta di queste nuove classi, nelle quali sono stati spostati tutti i metodi che permettono al Query Manager di eseguire operazioni sul database, ha implicato la cancellazione di 2 classi del package **it.hln.momis.queryManager.joinEngine**, **CreateIndex** e **Insert**, perché le loro funzioni sono state inglobate nelle nuove.
Come già detto nel cap. 3, abbiamo così raggiunto l'obbiettivo di permettere al Query Manager l'utilizzo di vari DBMS, e in particolare, grazie alla gerarchia introdotta, se in futuro si eseguirà il porting su altri DBMS, sarà sufficiente aggiungere la rispettiva classe figlia con i relativi metodi implementati correttamente.
Nella seguente tabella riassumiamo i metodi della classe astratta **DatabaseHandler_base** che vengono specializzati nei figli a seconda del DBMS che gestiscono.

NOME	PARAMETRI	RETURN	NUOVO	DOVE INVOCATO	COSA FA	
					SqlServer	**Oracle**
createTable	String tableName Vector columns Vector domains Vector domains_precision	String	No	JoinEngine	compone lo statement SQL per creare una tabella	
createIndex	String attr String tableName	String	No	JoinEngine	compone lo statement SQL per creare un indice sugli attributi di una tabella	
Insert	String tableName Vector colName Vector values Vector domains	String	No	WrapperThread	compone lo statement SQL per inserire i dati in tabella	
getShortName	String tableName	String	Si	Oql_QueryPlan JoinEngineID		accorcia tableName se supera i 30 caratteri
getLongName	String tableName	String	Si	Oql_QueryPlan		dato shortName ritorna il suo nome originale
getDbHandler (static)	String dbms	DatabaseHandler_base	Si	QmInterfacePanel	inizializza l'handler, l'oggetto che rappresenta il nostro dbms e ne utilizza i metodi	
getDbHandler (static)		DatabaseHandler_base	Si	in tutti i punti in cui ci serve l'handler	restituisce l'handler	
getName	String nome	String	Si	Oql_ExpandedQueryPlan		aggiunge le virgolette al nome
getLimitTo	String selectClause	String	Si	Oql_SelectExpr	inserisce TOP 100 PERCENT nella clausola Select	
getWithoutAs	String query	String	Si	Oql_ExpandedQueryPlan		togli AS e as dalla query

5.1.1 JUnit

Il testing è la fase del Corso di Sviluppo (CdS) che serve a rilevare la presenza di errori ed è importante testare il maggior numero di funzioni il più spesso possibile.
Se si segue una metodologia di sviluppo Test Driven Development (TDD), che si può riassumere con il motto “scrivere i test prima del codice”, questa fase è fondamentale è fa si che il codice abbia un'elevata qualità.
Durante lo studio effettuato, non è stata seguita questa metodologia, anche se tutti i nuovi metodi che sono stati aggiunti al software MOMIS, sono stati anche inseriti nella classe **DatabaseHandler_OracleExpress_JUTest,** dove è verificato il corretto funzionamento.
In questi metodi si creano gli oggetti necessari per il funzionamento del test e si confronta ciò che ritorna dal metodo che si sta testando con quello che ci si aspetta se tutto viene eseguito correttamente.
Il test può avere tre differenti esiti:

-successo (pass): i due valori confrontati sono uguali, quello che otteniamo dal metodo è proprio quello che ci aspettavamo

-fallimento (failure): il test è stato eseguito correttamente, ma i due valori non sono uguali; dobbiamo quindi rivedere il codice dell'oggetto del test

-errore (error): qualcosa è andato storto nell'oggetto del test o nel test stesso (ad es. NullPointerExcepiton); dobbiamo riguardare l'oggetto del test o il test stesso

L'obiettivo della metodologia di sviluppo TDD è di ridurre la complessità del codice, in particolare si ottengono i seguenti vantaggi:

-semplifica le modifiche: se vengono fatte modifiche successiva del codice di un modulo e questo produce un fallimento del test, si può individuare facilmente la modifica responsabile. Unit test già predisposti semplificano la vita al programmatore nel controllare che una porzione di codice sta ancora funzionanado correttamente.

-semplifica l'integrazione: lo unit testing semplifica l'integrazione di moduli diversi perché limita i malfunzionamenti dovuti a problemi nella interazione tra i moduli e non nei moduli stessi, rendendo i test di integrazione più semplici.

-supporta la documentazione: lo unit testing fornisce una documentazione "viva" del codice. I test

case incorporano le caratteristiche critiche per il successo di un'unità di codice. Tali caratteristiche indicano l'uso appropriato dell'unità e i comportamenti errati che devono essere identificati nel suo funzionamento. Pertanto lo unit testing documenta tali carattestiche, sebbene in molti ambienti questi non possono costituire la sola documentazione necessaria. In compenso, la tradizionale documentazione diventa spesso obsoleta a cause di successive modifiche del codice non documentate.

5.2 Il prepared statement

Nella classe **FunctionOdli3JDBC**, nel metodo exegue(), per inserire una tupla in tabella, si usava il metodo updateObject del **ResultSet rs_d**.
Visto che con Oracle questo non funzionava, è stato sostituito da codice che usa un **PreparedStatement ps**.

Codice precedente:

```
.....
while (gaResolver.next()){
     rs_d.moveToInsertRow();
     for (int i=0; i<nCol; i++) {
       obj = gaResolver.getValue(i);
       rs_d.updateObject(gaNames.get(i).toString(), obj);
     }
     count++;
     rs_d.insertRow();
  }
  gaResolver.close();
.....
```

Codice attuale:

```
.....
while (gaResolver.next()) {
for (int i=0; i<nCol; i++) {
   attrNames = attrNames + comma + "\"" + gaNames.get(i) + "\"";
   attrNamesQuestions = attrNamesQuestions + comma + "?";
   comma = ", ";
}
String s = "insert into "+table+" ("+attrNames+") values
      attrNamesQuestions+")";
PreparedStatement ps = this.connection_d.getConn().prepareStatement(s);
for (int i=1; i<=nCol; i++) {
      int j=i;
      j--;
      Type type = gaResolver.getColumnType(j);
```

```
            Object o = gaResolver.getValue(j);
            if (type instanceof StringType) {
               ps.setString(i,o.toString());
            } else if (type instanceof IntegerType) {
               ps.setInt(i,((Integer)o).intValue());
            } else if (type instanceof FloatingType) {
               ps.setDuoble(i,((Double)o).doubleValue());
                  ............
            } else  {
             String m = "Unknown type " + type.getClass().getName();
             throw new Exception(m);
            }
      }
      ps.executeUpdate();
      count++;
}
gaResolver.close();
      ...........
```

5.3 I tipi di colonna

Nella classe **FunctionOdli3JDBC**, nel metodo exegue(), viene creato un oggetto **QMResultSet_JDBCResultSet** che contiene informazioni sul ResultSet.
Il costruttore **QMResultSet_JDBCResultSet** effettua la conversione dei tipi di colonna, in modo da ottenere i tipi Odli3 accettati da Momis.

classe **Odli3_jdbc_conversion**

Tipi Sql	Tipi Odli3
varchar	StringType
longvarchar	StringType
char	CharType
integer	IntegerType
smallint	IntegerType
bigint	IntegerType
tinyint	IntegerType
double	FloatingType
decimal	FloatingType
float	FloatingType
real	FloatingType
numeric	FloatingType
time	DateType
timestamp	DateType
date	DateType

*i tipi SQL non presenti in questa tabella non possono essere usati per definire i dati delle sorgenti se sono tabelle relazionali, perché Momis non ne permette la conversione

figura 2

L'oggetto ritornato però conteneva le informazioni sbagliate, perchè veniva effettuata la conversione dai tipi che il DBMS aveva utilizzato per creare le colonne (e come abbiamo spiegato al cap 3, Oracle usa il tipo NUMERIC anche per gli interi) e non dai tipi con i quali Momis aveva definito le tabelle globali e locali, che sono gli stessi delle sorgenti.
Quindi anziché usare un **ResultSetMetaData** per ottenere i tipi di colonna con cui il DBMS le ha create, è sufficiente che il costruttore **QMResultSet_JDBCResultSet** accetti un altro parametro che è il vettore contenente i tipi di colonna con i quali noi abbiamo definito la tabella.

classe **QMResultSet_JDBCResultSet**

Codice precedente:

```
      ...........
public QMResultSet_JDBCResultSet(ResultSet resultSet) throws Exception {
      _resultSet = resultSet;
      _converters = new Vector();
      _names      = new Vector();
      _tables     = new Vector();
      ResultSetMetaData mtdt = resultSet.getMetaData();
      int istop = mtdt.getColumnCount();
      for (int i=1; i <= istop; i++ ){
         int columnType = mtdt.getColumnType(i);
         CoJdbcOdli3 converter = Odli3_jdbc_conversion.getConverter(columnType);
         if (converter == null){
            String m = "Conversion NOT defined for JdbcType type: "+columnType;
            throw(new Exception(m));
         }
         _converters.add(converter);
         _names.add(mtdt.getColumnName(i));
         _tables.add(mtdt.getTableName(i));
      }
   }
      ...........
```

Codice attuale:

```
      ...........
public QMResultSet_JDBCResultSet(ResultSet resultSet, Attribute attributes[])
                        throws Exception {
   _resultSet = resultSet;
   _converters = new Vector();
   _names      = new Vector();
   _tables     = new Vector();

   for (int i=1; i <= attributes.length; i++ ){
      Attribute attribute =(Attribute)attributes[i-1];
      CoJdbcOdli3 con = Odli3_jdbc_conversion.getConverter(attribute.getType());
         if (con == null){
            String m = "Conversion NOT defined for JdbcType type: "
                     + attribute.getType();
            throw(new Exception(m));
         }
         _converters.add(con);
      ...........

   }
}
      ...........
```

Nella classe **FunctionOdli3JDBC**, dobbiamo quindi creare il vettore **attributes[]** contenente i tipi di colonna della tabella locale per passarlo al nuovo costruttore.

Codice precedente:

```
      ..........
rs_s = this.connection_s.executeQuery(this.globalQuery);
QMResultSet localColumns = new QMResultSet_JDBCResultSet(rs_s);
      ..........
```

Codice attuale:

```
      ..........
//contiene le coppia <tipo di colonna, numero di colonna>
private AbstractMap localAttributes = null;
      ..........

Attribute localAttributesVector[] = new
      Attribute[this.localAttributes.keySet().size()];

for (Object o: this.localAttributes.keySet()){
  Attribute a = (Attribute)o;
  Integer pos = (Integer)this.localAttributes.get(a);
  localAttributesVector[pos.intValue()] = a;

}

rs_s = this.connection_s.executeQuery(this.globalQuery);
QMResultSet localColumns = new QMResultSet_JDBCResultSet(rs_s,
      localAttributesVector);
      ..........
```

5.4 Il singleton

La creazione della gerarchia DatabaseHandler ha imposto la necessità di avere un'unica istanza di DatabaseHandler_base all'interno del programma e di accedere sempre a quella.
Utilizzare un singleton è la scelta più appropriata e la sua implementazione avviene nella classe padre astratta **DatabaseHandler_base**, dove con il metodo **getDbHandler (String dbms)** viene inizializzata un'istanza del DBMS scelto (la strnga dbms è il nome completo della classe) solo se è la prima volta che viene invocato, se no ritorna l'istanza già esistente con il metodo **getDbHandler()**.
Il metodo statico per inizializzare l'istanza è chiamato nella classe **QmInterfacePanel**, cioè subito all'inizio del programma quando vengono lette le properties dal file di configurazione siDesigner.
Una volta inizializzato ogni volta che abbiamo bisogno di questa istanza è sufficiente invocare il metodo statico **getDbHandler()** per ottenerla, sapendo che è la stessa finchè è in esecuzione la virtual machine che esegue MOMIS.

```
private static DatabaseHandler_base _dbHandler = null;


public static DatabaseHandler_base getDbHandler(String dbms) throws Exception {
   if (_dbHandler == null) {
         System.err.println("Initializing dbHandler to: " + dbms);

 //- db handler initialization
      {
        Class c = null;
        try {
        c = Class.forName(dbms);
        } catch (Exception e) {

String m = "Not found Query manager database handler class ["+dbms+"]";
throw(new Exception(m));
          }
          try {
              _dbHandler = (DatabaseHandler_base)c.newInstance();
          } catch (Exception e) {

String m = "Query manager database handler class ["+dbms+"] does not hinerit
from DatabaseHandler_base";
throw(new Exception(m));
            }
       }

  //- esiste già l'istanza
   } else {
     if (!dbHandlerClassName.equals(_dbHandler.getClass().getName())) {

  System.err.println("Warning dbHandlerClass Differes, new: " + dbms
```

```
  + " old: " + _dbHandler.getClass().getName());
      }
   }
      return(_dbHandler);
}


public static DatabaseHandler_base getDbHandler() {
      return(_dbHandler);
}
```

5.5 La funzione getRealQuery

Descritto il suo funzionamento e il perché della sua necessità al paragrafo 3.3.4, qui sotto mostriamo il codice.

```
public String getRealQuery(String query, String idRif, String id) {
      int pos=0;
      String s = null;

      while (query.indexOf(idRif)>0){
            pos=query.indexOf(idRif);

            while (query.charAt(pos) != '"') pos--;
            int start = pos+1;

            while (query.charAt(pos+1) != '"') pos++;
            int end = pos+1;

            //prendo i nomi tra " ", ovvero le tabelle
            s = query.substring(start, end);

            //nome originale di s
            s = DatabaseHandler_base.getDbHandler().getLongName(s);

            //sostituisco idRif→id
            s = s.replace(idRif, id);

            //eventuale nome accorciato di s
            s = DatabaseHandler_base.getDbHandler().getShortName(s);

            //ricompongo la query con le giuste tabelle da usare
            query = query.substring(0, start)+ s + query.substring(end);

      }

      return query;
}
```

Riferimenti

[1] R. Hull and R. King et al. *Arpa I3 reference architecture.* 1995. Reperibile presso: http://www.isse.gmu.edu/I3_Arch/index.html/.

[2] Gio Wiederhold et al. *Intergrating artificial intelligence and database technology.* Journal of Intelligent Integration System, 2/3 Giugno 1996.

[3] F.Saltor and E. Rodriguez .*On intelligent access to heterogeneous information*. In Proceeding of the 4th KRDB Workshop, Atene, Grecia, Agosto 1997.

[4] D. Beneventano, S. Bergamaschi, S.Lodi, e C. Sartori. *Consistency checking in complex object database schemata with integrity constraints*. Technical Report 103, CIOC , Bologna, Italia , 1994.

[5] S. Bergamaschi e B.Nebel. *Acquisition and validation of complex object database schemata supporting multiple inheritance.* Applied Intelligence: The International Journal of Artificial Intelligence, Neural Networks and Complex Problem Solving Technologies, 4:185-203, 1994.

[6] D.Beneventano, S.Bergamaschi, C.Sartori e M.Vincini. *ODB-tools: a description logics based tool for schema validation and semantic query ottimization in object oriented databases*. In Proc. Of Int. Conference of the Italian Association for Artificial Intelligence (AI*IA, 97), Roma, 1997.

[7] D.Beneventano, S.Bergamaschi, C.Sartori, e M.Vincini. *Odb-qoptimizer: a tool for semantic query optimization in oodb*. In Proc. Of Int. Conf. On Data Engineering ICDE'97, Birningham, UK, April 1997.

[8] D.Beneventano, S.Bergamaschi, C.Sartori, e M.Vincini. *A description logics based tools for schema validation and semantic query optimization in oodb*. In Proc. Of Int. Conf. On Data Engineering ICDE'97, Birningham, UK, April 1997.

[9] S.Castano e V.De Antonellis. *Deriving global conceptual views from multiplie information sources*. In preProc. Of ER'97 Preconference Symposium on Conceptual Modeling, Historical Perspective and future Directions, 1997.

[10] T. Catarci e M. Lenzerini. *Rapresenting and using interschema knowledge in cooperative information systems.* Journal of Intelligent and Cooperative Information Systems, 2(4)375-398, 1993.

[11] B. Everitt. *Computer-Aided Database Design: the DATAID Project*. Heinemann Educational Books Ltd, Social Science Research Council, 1974.

[12] R.G.G. Cattel, editor. *The object Database Standard: ODMG93*. Morgan Kaufman Publisher, San Francisco, CA, 1997.

[13] S. Sorrentino. *Metodi di disambiguazione del testo ed estensioni di WordNet nel sistema Momis*. Reperibile presso: http://www.dbgroup.unimo.it/tesi/sorrentinoserena.pdf

[14] M. Orsini. *Query Management in Data Integration System: the MOMIS approach*. Reperibile presso: http://www.dbgroup.unimo.it/~orsini/thesis_Mirko_Orsini.pdf