FONDAMENTI DI BASI DI DATI

Antonio Albano, Giorgio Ghelli, Renzo Orsini

12 febbraio 2019

INDICE

Prefazione

Dopo molti anni dalla pubblicazione della prima edizione del volume *Fondamenti di Basi di Dati* presso l'Editore Zanichelli, aggiornato con una seconda versione uscita nel 2005, è tempo di un'ulteriore ammodernamento, che coincide con la sua diffusione con un canale diverso da quello della carta stampata: il web, attraverso il sito http://fondamentidibasididati.it.

Riteniamo che questo materiale possa essere utile non solo per i classici corsi di *Basi di Dati*, fondamentali per le lauree in *Informatica* o *Ingegneria Informatica*, ma, data l'attenzione cha sta oggi avendo l'informatica in sempre piuù ampi settori della formazione e dell'istruzione ad ogni livello, in molti altri corsi di laurea e momenti formativi, in forma anche parziale, come abbiamo potuto sperimentare di persona durante questi ultimi anni.

Il passaggio alla nuova modalità di distribuzione, permettendo di mantenere aggiornati i contenuti, ci ha richiesto di modificare la struttura del sito di supporto al libro, che non avrà più la parte di errata corrige e di approfondimenti, ma conterrà materiale aggiuntivo utile per lo studio, come le soluzioni agli esercizi, i collegamenti ai software gratuiti, gli appunti del corso, e gli esempi scaricabili.

Organizzazione del testo

Il libro inizia presentando le ragioni che motivano la tecnologia delle basi di dati, ed i concetti principali che caratterizzano le basi di dati ed i sistemi per la loro gestione.

In maggior dettaglio, il Capitolo 2 si sofferma sulle nozioni fondamentali di modello informatico finalizzato al trattamento delle informazioni di interesse dei sistemi informativi delle organizzazioni e sui meccanismi d'astrazione per costruire modelli informatici. Il modello di riferimento è il modello ad oggetti, motivato non solo dalla sua naturalezza per la progettazione di basi di dati, ma anche per essere il modello dei dati dell'attuale tecnologia relazionale ad oggetti per basi di dati. Il formalismo grafico adottato si ispira all'*Unified Modeling Language*, UML, ormai uno standard dell'ingegneria del software.

Il Capitolo 3 presenta una panoramica del problema della progettazione di basi di dati, si sofferma sulle fasi dell'analisi dei requisiti e della progettazione concettua-

le usando il modello ad oggetti e il formalismo grafico proposto nel Capitolo 2, e descrive un metodo di lavoro per queste fasi.

I Capitoli 4 e 5 sono dedicati alla presentazione rigorosa del modello relazionale dei dati e ad un'introduzione alla teoria della normalizzazione. La scelta di dedicare questo spazio all'argomento è giustificata dal ruolo fondamentale che svolge il modello relazionale per la comprensione della tecnologia delle basi di dati e per la formazione degli addetti.

I Capitoli 6, 7 e 8 trattano il linguaggio relazionale SQL da tre punti di vista, che corrispondono ad altrettante categorie di utenti: (1) gli utenti interessati ad usare interattivamente basi di dati relazionali, (2) i responsabili di basi di dati interessati a definire e gestire basi di dati, (3) i programmatori delle applicazioni.

Il Capitolo 9 presenta una panoramica delle principali tecniche per la realizzazione dei sistemi relazionali. Vengono presentate in particolare la gestione delle interrogazioni e dei metodi di accesso e la gestione dell'atomicità e della concorrenza in sistemi centralizzati.

Ringraziamenti

L'organizzazione del materiale di questo testo è il risultato di molti anni di insegnamento deii corsi di *Basi di Dati* per le lauree in *Scienze dell'Informazione* e in *Informatica*.

Molte persone hanno contribuito con i loro suggerimenti e critiche costruttive a migliorare la qualità del materiale. In particolare si ringraziano i numerosi studenti che nel passato hanno usato le precedenti versioni del testo e Gualtiero Leoni per la sua collaborazione.

Si ringrazia infine l'Editore Zanichelli per il supporto che ci ha dato in questi anni, e, adesso che il libro è uscito dal suo catalogo, per il permesso di diffondere la nuova versione attraverso il web.

A. A.
G. G.
R. O.

Capitolo 8

SQL PER PROGRAMMARE LE APPLICAZIONI

Uno dei principali obiettivi dei proponenti dell'SQL fu che il linguaggio dovesse essere utilizzabile direttamente da utenti occasionali per interrogare basi di dati, senza dover ricorrere a programmi sviluppati da esperti. Per questa ragione fu subito proposta anche una versione grafica del linguaggio per nascondere la sintassi SQL, chiamata *Query By Example* (*QBE*). L'obiettivo però è stato raggiunto solo in parte, perché ancora oggi l'uso interattivo di basi di dati con il linguaggio SQL è limitato a persone con competenze tecniche, nonostante i progressi fatti con le interfacce grafiche di ambienti interattivi tipo Microsoft Access per Windows, versione moderna del *QBE*. Nella maggioranza dei casi occorre invece sviluppare applicazioni interattive in modo che gli utenti del sistema informatico possano usare i servizi offerti senza nessuna competenza specifica. Per questo è necessario disporre di opportuni linguaggi di programmazione che consentano sia l'accesso a basi di dati che la realizzazione delle interfacce grafiche per il dialogo con gli utenti.

Un altro motivo per cui tali linguaggi sono necessari è la necessità di scrivere applicazioni che usano basi di dati per una grande varietà di compiti diversi: applicazioni gestionali, analisi complesse, applicazioni web ecc. In tutti questi casi la tendenza attuale è di usare SQL come componente dedicata all'accesso ai dati all'interno di un linguaggio di programmazione adatto al compito specifico.

La differenza fra le varie soluzioni è data soprattutto dalla modalità di integrazione fra SQL e il linguaggio di programmazione. In generale le soluzioni più comuni possono essere raggruppate nelle seguenti categorie:

1. Linguaggi di programmazione generale, come C, C++, Java, Visual Basic, la cui sintassi viene estesa con costrutti SQL per operare sulla base di dati (si dice anche che *ospitano* l'SQL).
2. Linguaggi tradizionali con i quali l'uso della base di dati avviene attraverso la chiamata di funzioni di una opportuna libreria (*Application Programming Interface, API*).
3. Linguaggi cosiddetti della *quarta generazione* (4^{th} *Generation Languages, 4GL*), come Informix 4GL, Oracle PL/SQL, e Sybase Transact/SQL. Questi sono linguaggi di programmazione di tipo generale costruiti "attorno" ad SQL, nel senso che lo estendono con costrutti di controllo ed altri tipi di dati, ma si basano sul modello dei dati relazionali.

8.1 Linguaggi che ospitano l'SQL

Un approccio classico allo sviluppo di applicazioni per basi di dati relazionali prevede l'immersione dei costrutti di SQL in un linguaggio di programmazione tradizionale (come COBOL, C, Basic, Java, C++) con una sintassi "aggiuntiva", senza alterare nè la sintassi corrente nè il sistema dei tipi.[1] Poiché in questi linguaggi non è previsto il tipo relazione e il tipo ennupla, vengono usati opportuni accorgimenti per scambiare dati fra la "parte" SQL e la parte tradizionale del linguaggio.

I vantaggi di questo approccio sono diversi:

- il costo ridotto di addestramento dei programmatori, che continueranno ad usare un linguaggio già conosciuto per la parte di applicazione che non tratta direttamente la gestione dei dati persistenti;
- la semplicità della sintassi di estensione, che rende i programmi più comprensibili rispetto ad approcci basati su chiamate di funzione;
- la possibilità di usare meccanismi di controllo dei tipi per validare durante la compilazione la correttezza dell'uso dei comandi SQL;
- la possibilità di ottimizzare le interrogazioni durante la compilazione del programma;
- la possibilità di attivare meccanismi di sicurezza basati sull'analisi del codice SQL.

Lo svantaggio principale, invece, è il fenomeno detto di *impedence mismatch*: la differenza fra i tipi di dati del linguaggio e quelli relazionali obbliga a curare la conversione dei dati fra i due diversi modelli. Ad esempio, per trattare il risultato di una interrogazione SQL (una relazione, cioè un insieme) occorre usare costrutti iterativi propri del linguaggio, operando su un solo elemento alla volta.

In questa sezione, si mostra come vengono immersi i comandi SQL (*Embedded SQL*) nel linguaggio Java. Il linguaggio risultante, detto SQLJ, presenta una serie di vantaggi:

- è un linguaggio standard, definito dall'organismo internazionale di standard ISO, quindi i programmi scritti con questo linguaggio sono utilizzabili su DBMS diversi, al contrario di altre soluzioni basate su linguaggio ospite;
- il traduttore dal linguaggio esteso al linguaggio di base, Java, è a sua volta scritto in Java, così come il sistema di supporto per l'esecuzione (*SQLJ runtime*), quindi, sfruttando la portabilità di Java, questa è una soluzione disponibile su moltissimi sistemi operativi;
- vengono usati gli aspetti "object-oriented" di Java per semplificare la comunicazione fra le due componenti del linguaggio.

In ogni caso, qualunque sia il linguaggio ospite che si utilizza, un'approccio al problema dell'uso di basi di dati relazionali all'interno di un linguaggio di programmazione deve risolvere i seguenti aspetti fondamentali:

1. In inglese vengono detti *SQL embedded languages*, cioè linguaggi con SQL immerso.

1. come indicare la connessione alla base di dati e fornire le credenziali di utente;
2. come specificare i comandi SQL, associando ad eventuali loro parametri dei valori calcolati dal programma;
3. come accedere ai risultati di un comando SQL, in particolare ad una relazione restituita da un'interrogazione;
4. come gestire le condizioni anormali di esecuzione di un comando, ed eventualmente come gestire questo aspetto in relazione alle proprietà di atomicità di una transazione.

Questi punti verranno esaminati nel seguito per il linguaggio SQLJ, ma si tenga presente che le soluzioni adottate sono simili a quelle offerte da altri linguaggi.

8.1.1 Connessione alla base di dati

Il concetto di *connessione*, o *contesto di connessione* (*connection context*), indica in generale il contesto di lavoro a cui dei comandi SQL fanno riferimento. Un contesto specifica la base di dati a cui si fa riferimento, con quale nome di utente si accede, e qual'è il tipo dei dati utilizzati. Un contesto è un oggetto di una classe (o tipo oggetto) Java particolare, detta *classe di contesto*, che viene dichiarata e istanziata come nel seguente esempio:[2]

```
Class.forName(DatabaseDriver);
#sql context ClasseContesto;
ClasseContesto contesto = new ClasseContesto(url, utente, password);
```

La prima riga serve per caricare nel sistema il *driver* di accesso alla base di dati, cioè la libreria specifica al DBMS che si vuole utilizzare. Ad esempio, se volessimo accedere ad un sistema Oracle con il driver JDBC della stessa azienda, dovremmo dare come parametro "oracle.jdbc.driver.OracleDriver".

La seconda riga è scritta con la sintassi SQLJ estesa (ogni comando SQLJ deve iniziare con il simbolo #sql), e indica che nel programma viene definita una nuova classe di contesto, con campi e metodi predefiniti, chiamata in questo esempio ClasseContesto.

Infine nell'ultima riga si crea una nuova istanza di questa classe, un oggetto di nome contesto, che verrà usato in tutte le operazioni successive. Il costruttore prende come parametri tre stringhe: la prima è il riferimento alla base di dati, la seconda è il nome dell'utente e la terza è la parola chiave dell'utente specificato.[3]

2. Vi sono in realtà diversi modi per creare una classe di contesto e una sua istanza. Nell'esempio viene mostrato il modo più semplice.
3. Ad esempio il riferimento alla base di dati Oracle di nome acmedb sul server db.acme.com sarà: "jdbc:oracle:thin:@db.acme.com:1521:acmedb" .

8.1.2 Comandi SQL

Una volta aperta una connessione, è possibile usare i comandi SQL premettendovi il simbolo #sql e il nome del contesto all'interno del quale devono essere eseguiti. Ad esempio, se nella base di dati esiste una tabella Province con attributi Nome, Sigla, stringhe di caratteri, e NumeroComuni un numero intero, potremmo creare una nuova Provincia con la riga seguente:

```
#sql [contesto] INSERT INTO Province VALUES ("Milano", "MI", 166 );
```

Normalmente, però, i dati in un comando SQL non sono costanti, ma valori di variabili del programma che vanno usate aggiungendo il prefisso ":". Se volessimo così effettuare un'inserzione usando dati letti da terminale, potremmo scrivere:

```
String provincia, sigla;
int numeroComuni;
... lettura dei valori corretti nelle tre variabili sopra definite ...
#sql [contesto] INSERT INTO Province VALUES (:provincia, :sigla, : numeroComuni);
```

L'uso delle variabili consente il passaggio di dati non solo dal programma alla base di dati, come nell'esempio precedente, ma anche nella direzione opposta, dalla base di dati al programma, usando la versione del comando **SELECT** esteso con la clausola **INTO** per assegnare a delle variabili il valore degli attributi dell'*unica* ennupla del risultato di una **SELECT**, come nel seguente esempio:

```
int numeroProvince;
#sql [contesto] SELECT COUNT(*) INTO :numeroProvince FROM Province
System.out.println("In Italia ci sono" + numeroProvince + " province.");
```

Questa forma del comando **SELECT** non si può usare se il risultato è un insieme di ennuple, ma per accedere ad esse una per volta si utilizza il meccanismo dei *cursori* (chiamati in SQLJ, *iteratori*, o *result set iterators*).

8.1.3 I cursori

Un cursore è un oggetto che rappresenta un insieme di ennuple, con metodi che permettono di estrarre dall'insieme un elemento alla volta. Ad esempio, se vogliamo stampare tutte le province con più di 60 comuni, potremmo scrivere:

```
#sql iterator IteratoreProvince (String nome, int comuni);
IteratoreProvince province;
#sql [contesto] province = {SELECT    Nome, NumeroComuni
                            FROM      Province
                            WHERE     NumeroComuni > 60};
while (province.next()) {
    System.out.println(province.nome() + " " + province.comuni());
}
province.close();
```

La prima riga, analogamente alla dichiarazione di una classe di contesto, è una dichiarazione di una classe di iteratore di nome IteratoreProvince. Gli oggetti istanza di questa classe verranno usati per scorrere insiemi di ennuple con due campi, una stringa e un intero.

La seconda riga dichiara una variabile di tipo iteratore, che viene inizializzata nella terza riga al risultato di un comando **SELECT**. Il risultato del comando è quindi un oggetto iteratore che ha il tipo compatibile con IteratoreProvince: infatti la **SELECT** effettua la proiezione su due attributi, il primo stringa e il secondo intero.

Un oggetto iteratore ha associato in ogni istante un'ennupla del risultato, che chiameremo *riga corrente*, oppure un valore nullo (all'inizio e alla fine dopo la scansione di tutte le ennuple). Il metodo booleano **NEXT** ha un duplice scopo: ritorna vero o falso per indicare se ci sono ancora ennuple da scorrere, e fa diventare corrente l'ennupla successiva (o la prima, quando viene chiamato la prima volta). Quindi è sufficiente usarlo all'interno della condizione del comando **WHILE** per scorrere in sequenza tutte le ennuple del risultato e terminare il ciclo quando non ce ne sono più. Il corpo del ciclo, invece, mostra come viene usata l'ennupla corrente: attraverso dei metodi, con i nomi uguali a quelli dati nella dichiarazione della classe di iteratore, si selezionano i campi associati. Così il metodo nome restituisce il valore dell'attributo Nome dell'ennupla corrente, mentre il metodo comuni restituisce il valore dell'attributo NumeroComuni. Alla fine della scansione l'iteratore viene chiuso, rilasciando così le risorse impegnate per la gestione del risultato.

8.1.4 Transazioni

Come abbiamo già visto nel Capitolo 1, una transazione è un programma che non può avere un effetto parziale: la transazione termina correttamente, oppure, se si verifica un errore, ne vengono disfatti tutti gli effetti ed è come se la transazione non fosse mai iniziata. Come si concretizza questo concetto all'interno di un programma per basi di dati?

I vari linguaggi rispondono in maniera diversa a questa domanda. SQLJ, in particolare, stabilisce la seguente regola iniziale: *ogni singolo comando SQL è considerato una transazione a sè stante* (regola di *autocommit on*). Dato che non sempre questa regola è soddisfacente, ad esempio perchè vogliamo fare una serie di letture e scritture che potrebbero essere completamente annullate in seguito al verificarsi di qualche evento particolare, è possibile imporre la regola di *autocommit off*: una transazione inizia con il primo comando SQL, prosegue con i comandi successivi, e termina solo quando si dà un comando esplicito di terminazione (con successo **COMMIT** o con fallimento **ROLLBACK**).

Se non viene specificato altrimenti, viene seguita la regola *autocommit on*; si può specificare quale delle due regole seguire con un quarto parametro al costruttore di contesto: se questo è true si vuole il comportamento standard, se invece è false si vuole disabilitare l'*autocommit*. In questo secondo caso, una transazione inizia con il primo comando SQL e viene terminata dando il comando SQLJ:

```
#sql [contesto] COMMIT;
```

se vogliamo far terminare la transazione con successo e quindi salvare le modifiche, oppure

#sql [contesto] **ROLLBACK**;

se vogliamo far fallire la transazione annullando tutte le modifiche effettuate dall'inizio.

Si noti che la gestione del fallimento delle transazioni è indipendente dal verificarsi di errori SQL all'interno del programma. Infatti, quando un comando SQL provoca un errore, questo viene riflesso nel programma Java sotto forma di una eccezione di classe SQLException. Sta al programmatore gestire l'eccezione e decidere come comportarsi rispetto alla transazione corrente (in caso di *autocommit off*, dato che nell'altro caso la transazione consiste solo dell'operazione SQL che ha provocato errore e che quindi non viene eseguita).

Spetta quindi al programmatore la decisione della regola da seguire: nei casi più semplici è sufficiente lasciare quella default (*autocommit on*), mentre nei casi più complessi sarà necessario impostare *autocommit off* e gestire esplicitamente l'inizio e la fine delle transazioni. Alla fine di questo capitolo, nella Sezione 8.4, questo argomento verrà discusso in dettaglio e verranno mostrati degli esempi.

Esempio 8.1 Usando la base di dati del capitolo precedente, si mostra un programma per (a) la stampa dell'ammontare di un certo ordine, (b) l'aggiornamento della zona di un agente e (c) la stampa delle coppie (clienti, ammontare degli ordini) ordinate in modo decrescente secondo l'ammontare.

```
import java.sql.*;
import sqlj.runtime.*;

public class Esempio {

    public static void main(String [] args) {
        try {
            // Connessione alla base di dati
            Class.forName("oracle.jdbc.driver.OracleDriver");
            #sql context ClasseContesto;
            ClasseContesto contesto =
                new ClasseContesto("jdbc:oracle:thin:@localhost:1521:mydb",
                        "utente1","password1");

            // Ricerca dell'ammontare di un certo ordine
            // Lettura dei parametri
            String numeroAgente = leggi("Numero agente: ");
            String numeroCliente = leggi("Numero cliente: ");
            String numeroOrdine = leggi("Numero ordine: ");
            int ammontare;
```

```
        #sql [contesto] SELECT   Ammontare INTO  :ammontare
                        FROM     Ordini
                        WHERE    CodiceAgente = :numeroAgente
                        AND      CodiceCliente = :numeroCliente
                        AND      NumOrdine = :numeroOrdine;

        // Stampa risultato
        System.out.println("L'ammontare e': " + ammontare);

        // Aggiornamento della zona di un agente
        numeroAgente = leggi("Numero agente: ");
        String zona = leggi("Zona: ");

        #sql [contesto] UPDATE   Agenti SET Zona = :zona
                        WHERE    CodiceAgente = :numeroAgente;

        // Stampa le coppie (numero cliente, ammontare ordini)
        // ordinate in modo decrescente secondo l'ammontare
        #sql iterator IteratoriClientiOrdini(String codCliente, int ammontare);
        IteratoriClientiOrdini risultato;
        #sql [contesto] risultato = SELECT      CodiceCliente, SUM(Ammontare)
                                    FROM        Ordini
                                    GROUP BY    CodiceCliente
                                    ORDER BY    SUM(Ammontare) DESC;

        //Stampa intestazione tabella
        System.out.println("Cliente Uscite");
        while (risultato.next()) {
            System.out.println(risultato.codCliente() + " " + risultato.ammontare());
        }
        risultato.close();

    } catch ( SQLException e ) {
        System.err.println("SQLException " + e);
    } finally {
        try {
            contesto.close(); // disconnessione dal DBMS
        } catch ( SQLException e ) {
            System.err.println("SQLException " + e);
        }
    }
}
```

```
    static BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    static String leggi(String messaggio) throws IOException {
        System.out.println(messaggio);
        return reader.readLine();
    }
}
```

8.2 Linguaggi con interfaccia API

L'uso della base di dati con i comandi SQL immersi nel linguaggio ospite richiede la presenza di un compilatore apposito, o, come avviene più frequentemente, di un precompilatore che trasforma il programma nel linguaggio esteso in un programma nel linguaggio base con opportune chiamate al sistema a run-time del DBMS.

Ci sono tuttavia casi in cui non è disponibile il precompilatore, oppure, per motivi di efficienza e di flessibilità, ci si vuole interfacciare direttamente dal programma con il sistema di gestione di basi di dati.

In tali casi viene fornita una libreria di funzioni *API* da richiamare nel linguaggio tradizionale utilizzato passando opportuni parametri, che possono essere, ad esempio, comandi SQL sotto forma di stringhe di caratteri, oppure informazioni di controllo o per lo scambio dei dati.

La possibilità di passare al DBMS i comandi SQL sotto forma di stringhe, se da un lato comporta lo svantaggio che eventuali errori nei comandi possono essere individuati solo durante l'esecuzione del programma, e non durante la sua compilazione, dall'altro permette l'utilizzo del cosiddetto *Dynamic SQL*. Con questo termine si intendono programmi in cui i comandi SQL non sono noti durante la scrittura del programma, ma vengono calcolati durante la sua esecuzione, ad esempio leggendoli come stringhe da terminale, o generandoli con informazioni ricavate dall'accesso al catalogo della base di dati.

In Figura 8.1 viene mostrata la differenza fra i due approcci: quello del linguaggio ospite (linee continue) e delle chiamate dirette delle librerie messe a disposizione dal DBMS, nel caso del linguaggio Java (linee tratteggiate).

Le principali soluzioni adottate, per mantenere la portabilità del codice, prevedono l'utilizzo di librerie che, sebbene siano diverse per i vari DBMS e ambienti di esecuzione, hanno un'interfaccia standard, che viene usata all'interno dei programmi applicativi sempre nello stesso modo. Il programma, una volta compilato rispetto all'interfaccia, viene quindi eseguito insieme alla libreria opportuna per il suo ambiente di esecuzione e per il DBMS a cui deve accedere. Un vantaggio ulteriore di questo approccio è che queste librerie prevedono in generale che il DBMS risieda su un elaboratore diverso da quello in cui si esegue il programma, e si occupano di gesti-

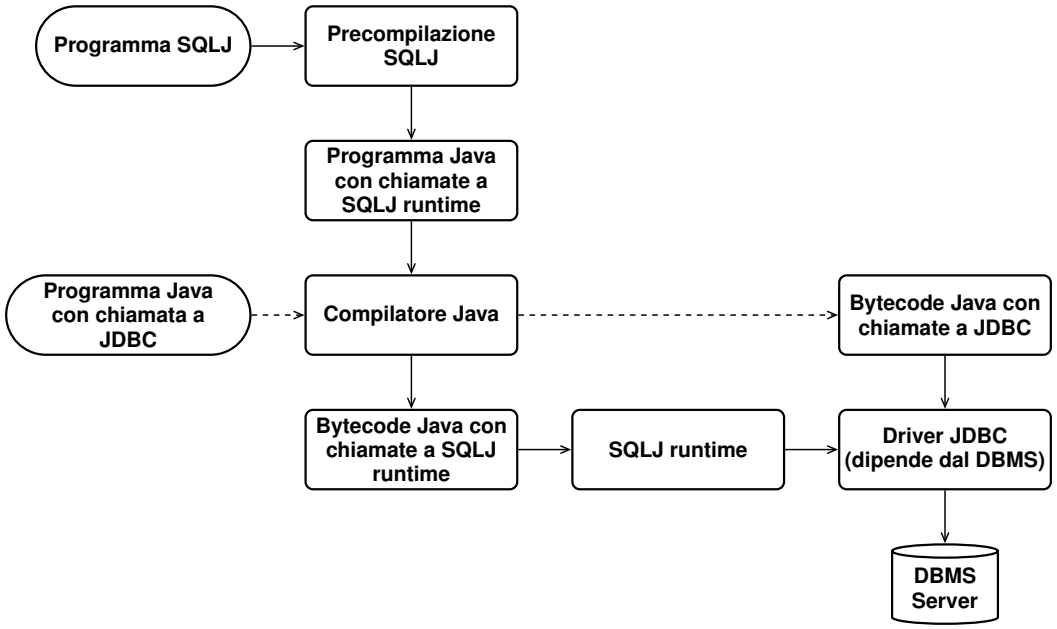

Figura 8.1: Sviluppo di applicazioni con linguaggio ospite e con chiamate ad API del DBMS

re in maniera trasparente tutte le problematiche di comunicazione con un approccio *client-server*.

8.2.1 L'API ODBC

L'API ODBC (*Open Data Base Connectivity*) è, attualmente, uno standard molto diffuso per l'utilizzo di DBMS relazionali. ODBC è stato definito dalla Microsoft, e specifica un DDL ed un DML relazionali basati sullo standard SQL CLI (*Call Level Interface*) proposto dal comitato X/Open.

Uno strumento che implementi l'API ODBC è composto principalmente da un insieme di *driver* o librerie ODBC. Un *driver* ODBC per il DBMS X è una libreria che traduce le chiamate generiche ODBC in chiamate allo specifico sistema X e gliele invia, appoggiandosi ad un sistema di comunicazione (Figura 8.2).

ODBC permette di accedere ad un sottoinsieme limitato di SQL, quello implementato da tutti i sistemi relazionali, ma permette anche di usare estensioni non standard presenti su di uno specifico sistema, anche se questo riduce la portabilità dell'applicazione. Sono disponibili anche *driver* ODBC per sistemi di archiviazione; questi *driver* interpretano in proprio le istruzioni SQL.

Mentre ODBC è un'API utilizzabile per lo sviluppo di applicazioni in C o in altri linguaggi di programmazione, ne esistono anche delle versioni integrate in ambienti di sviluppo di vario tipo, come Visual Basic di Microsoft, che ne permettono anche l'uso all'interno di strumenti di produttività individuale. In particolare, tutti gli stru-

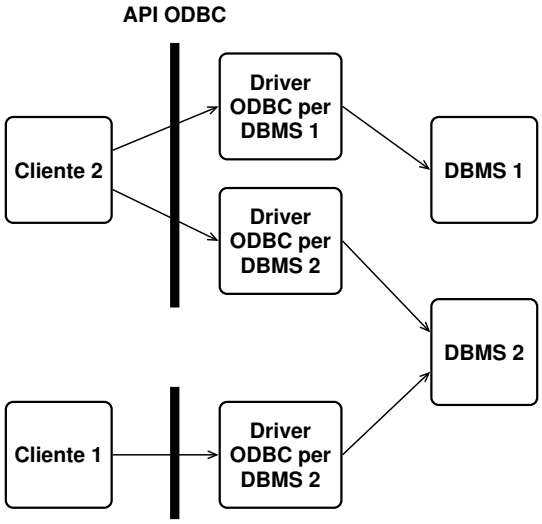

Figura 8.2: Uso dei *driver* ODBC

menti più diffusi per l'implementazione di interfacce hanno la capacità di sfruttare *driver* ODBC.

Il seguente esempio mostra un semplice programma Visual Basic che può essere richiamato all'interno di un foglio elettronico Excel, per inserire i risultati di una interrogazione SQL in una colonna del foglio di calcolo con nome Risultato:

```
Procedure TrovaNomiStudenti()

' Si apre la connessione alla base di dati
connessione = SQLOpen("DSN=MioDatabase;UID=ut1;PWD=pw1")

' Si esegue l'interrogazione (senza recuperare i dati)
SQLExecQuery connessione; "SELECT Nome FROM Studenti"

' Si dichiara che i risultati saranno restituiti a partire
' dalla prima cella del foglio di calcolo
SQLBind connessione; 1
FogliDiLavoro("Risultati").Celle(1; 1)

' Si recuperano effettivamente i dati
SQLRetrieve connessione

' Si chiude la connessione
SQLClose connessione
End Procedure
```

8.2.2 L'API JDBC

L'API JDBC (*Java Data Base Connectivity*) può essere definita come la versione Java di ODBC, e ha implementazioni analoghe a quelle di ODBC. L'interesse per questo tipo di API è duplice:

- grazie alla portabilità dei programmi Java, con JDBC si ha il vantaggio di poter sviluppare applicazioni indipendenti non solo dal tipo di sistema che gestisce la base di dati, ma anche dal tipo di piattaforma su cui l'applicazione viene eseguita;
- JDBC è un'interfaccia per un linguaggio ad oggetti, Java, progettato per evitare quelle caratteristiche del C e C++ che rendono la programmazione rischiosa.

JDBC è lo strumento utilizzato per la realizzazione del linguaggio SQLJ: tutti i comandi SQL di tale linguaggio vengono trasformati dal precompilatore in chiamate alle primitive JDBC. In effetti, dato che i compilatori SQLJ sono ancora scarsamente diffusi, la programmazione di basi dati relazionali dal linguaggio Java è fatta molto spesso con JDBC.

La logica con cui dal programma si utilizza il DBMS è molto simile a quella di SQLJ. Le differenze principali sono nel fatto che non è possibile nessun controllo dei tipi, mentre altre differenze dipendono dalle classi diverse che sono usate per accedere ai dati:

- **Connection** per stabilire il collegamento con una base di dati: una connessione è l'equivalente di un contesto in SQLJ;
- **Statement** per costruire il comando SQL da inviare al sistema tramite una *connection* , e
- **ResultSet** per ricevere il risultato (equivalente al *result set iterator* di SQLJ).

Vediamo un semplice esempio d'uso di JDBC.

```
import java.sql.*;
class StampaNomiStudenti {
    public static void main(String argv[]) {
        try {

            // URL della base di dati da usare
            String url = "jdbc:odbc:MioDatabase";

            // si apre la connessione con la base di dati
            Connection con = DriverManager.getConnection(url, "utente1", "password1");

            // si esegue un comando SELECT
            Statement stmt = con.createStatement();
            ResultSet risultato = stmt.executeQuery("SELECT Nome FROM Studenti");
```

```
            // scansione del risultato usando il metodo next
            System.out.println("Nomi trovati:");
            while (risultato.next()) {
                String nome = risultato.getString("Nome");

                // Stampa del nome trovato
                System.out.println(" Nome = " + nome);
            }

            risultato.close();
            stmt.close();
            con.close();
        }
        ...
    }
}
```

Si noti la differenza dell'uso del ResultSet rispetto a SQLJ: dato che a tempo di compilazione non si conosce la struttura della base di dati, non si possono usare metodi specifici per accedere ai campi dell'ennupla corrente. Si usano invece metodi generici (getString, getInteger ecc.) che prendono come parametro o il nome dell'attributo, oppure il suo numero d'ordine.

La gestione delle transazioni è analoga a quella di SQLJ, attraverso il meccanismo di *autocommit* (che può essere esplicitamente manipolato con il metodo setAutocommit(Boolean) di una connessione). Si può anche richiedere l'esecuzione della transazione con un certo livello di isolamento con un opportuno valore del parametro del metodo setTransactionIsolation.

Per passare valori dal programma ai comandi SQL, dato che sono solamente stringhe, si può procedere in due modi diversi:

- usando la concatenazione di stringhe, come nell'esempio seguente, in cui viene usato il valore della variabile matricola per indicare la matricola dello studente da ricercare:

  ```
  stmt.executeQuery("SELECT Nome FROM Studenti WHERE Matricola = " + matricola);
  ```

- usando la classe PreparedStatement, che le cui istanze corrispondono a comandi SQL all'interno dei quali vi sono dei parametri indicati da "?" che si possono sostituire con valori qualunque, in maniera ordinata, come nel seguente frammento di programma:

  ```
  PreparedStatement pstmt =
        con.prepareStatement("SELECT Nome FROM Studenti WHERE Matricola = ?");
  pstmt.setInt(1, matricola);
  risultato = pstmt.executeQuery();
  ```

Si noti che il secondo metodo è preferibile per evitare che caratteri particolari all'interno dei parametri, come le virgolette, interferiscano con le operazioni di concatenamento di stringhe o con la sintassi del comando SQL.

Infine, è importante sottolineare come l'interfaccia JDBC permetta anche lo sviluppo di applicazioni con SQL dinamico (*Dynamic SQL*), cioè in cui i comandi SQL che devono essere eseguiti sono calcolati a tempo di esecuzione.

Esempio 8.2 Si consideri ad esempio un programma che, consultando il catalogo, propone all'utente la lista delle tabelle della base di dati chiedendogli quale di queste vuole interrogare. Una volta che l'utente ha fatto la sua scelta, il programma potrebbe proporre l'elenco degli attributi, e permettere di all'utente di dare i valori di alcuni attributi per restringere la ricerca, e di indicare gli attributi di cui vuole la visualizzazione. In base a queste informazioni il programma potrebbe sintetizzare la query sotto forma di stringa, con la sicurezza che sia comunque corretta perché generata utilizzando le informazioni del catalogo dei dati.

Un programma come quello descritto nell'esempio precedente può essere facilmente scritto con JDBC utilizzandone le funzionalità che permettono di interrogare, in maniera indipendente dal particolare DBMS usato, il catalogo dei dati (i metadati). Esiste infatti il metodo getMetaData di un oggetto Connection che restituisce un oggetto istanza della classe predefinita DataBaseMetaData, con moltissimi metodi per conoscere tutti gli elementi del catalogo dei dati (tabelle, attributi, chiavi, chiavi esterne, procedure memorizzate ecc.). Inoltre, per visualizzare il risultato di una interrogazione espressa con una stringa calcolata a tempo di esecuzione, si può usare il metodo getMetaData dell'istanza di Result Set restituita. Questo metodo produce un oggetto di tipo ResultSetMetaData, che descrive completamente il risultato dell'interrogazione (numero di attributi, loro tipo, lunghezza ecc.).

8.3 Linguaggi integrati

Per ovviare al problema dell'*impedence mismatch* presente nell'uso di SQL con un linguaggio ospite, si integrano i costrutti di un linguaggio relazionale e di un linguaggio di programmazione in un unico linguaggio. Il primo esempio di linguaggio progettato con questo obiettivo è stato il Pascal/R, in cui il sistema dei tipi del Pascal è esteso con un nuovo tipo di dato, la relazione, e di costrutti per agire su relazioni [Schmidt, 1977].

Una direzione completamente diversa è stata invece presa da molti costruttori di sistemi relazionali commerciali, con i cosiddetti linguaggi della quarta generazione (4GL): l'idea è quella di costruire un linguaggio di programmazione estendendo l'SQL con costrutti di controllo di tipo generale, spesso derivati da linguaggi tipo BASIC.

L'esempio che presenteremo in questa sezione è il PL/SQL, linguaggio del sistema ORACLE.

Le caratteristiche principali del PL/SQL sono le seguenti:

- Si possono definire variabili o costanti dei tipi dei domini relazionali, sia esplicitamente (come in **DECLARE** x **CHAR**(2); che dichiara x di tipo stringa di due caratteri), che uguali al tipo di una colonna (come in **DECLARE** x Clienti.CognomeENome%**TYPE**; che dichiara x dello stesso tipo della colonna CognomeENome della tabella Clienti). Si possono, inoltre, dichiarare variabili di un tipo ennupla, come in **DECLARE** Cliente Clienti%**ROWTYPE**.
- Si possono definire cursori, equivalenti agli iteratori SQLJ, per operare in maniera iterativa su un insieme di ennuple restituite da una **SELECT**. Un cursore è associato ad una espressione SQL, come ad esempio in **DECLARE CURSOR** c1 **IS SELECT** Zona **FROM** Agenti, e quindi può essere usato in due modi: all'interno di un costrutto **FOR**, per scandire tutte le ennuple del risultato dell'espressione associata (come in **FOR** z **IN** c1 **LOOP** ... **END**), oppure con dei comandi per eseguire il ciclo in maniera esplicita (**OPEN**, **FETCH**, **CLOSE**).
- I comandi possibili sono l'assegnamento, tutti comandi SQL (per cui il linguaggio è in effetti un soprainsieme dell'SQL), e i tradizionali costrutti di controllo (**IF**, **WHILE**, **LOOP** ecc.).
- Esiste un meccanismo di gestione delle eccezioni, sia generate da operazioni illegali, che esplicitamente dall'utente. Il verificarsi di un'eccezione provoca l'annullamento delle modifiche alla base di dati.
- Si possono definire funzioni e procedure con parametri (**PROCEDURE**) in maniera usuale.
- Si possono definire moduli (**PACKAGE**) che racchiudono procedure, funzioni, cursori e variabili collegate. Un modulo ha una parte pubblica, che elenca le entità esportabili dal modulo, ed una privata, che contiene la realizzazione delle procedure pubbliche ed eventuali variabili e procedure locali al modulo.
- Sia le singole procedure e funzioni che i moduli possono essere *memorizzati* nella base di dati, assegnando loro un nome (**CREATE PROCEDURE**, **CREATE PACKAGE**). Il sistema memorizza procedure e moduli in forma compilata, ma mantiene anche il codice sorgente e tiene traccia delle dipendenze in esso presenti, per ricompilarlo nel caso di modifiche dei dati usati (ad esempio, se viene modificata la definizione di una tabella).
- Si possono definire dei *trigger* (**CREATE TRIGGER**), come discusso nel capitolo precedente.

Un programma in PL/SQL viene scritto sotto forma di blocco anonimo (anonymous block), e può essere presentato, interattivamente oppure in un archivio di testo, ad uno dei vari tool di ORACLE per l'esecuzione. Usando i comandi **CREATE PROCEDURE** o **CREATE PACKAGE** si può memorizzare una procedura o un modulo nella base di dati. Procedure e componenti di moduli memorizzati possono quindi essere richiamate ed usate da altri programmi, anche remoti.

Esempio 8.3 Riferendoci allo schema già presentato dei clienti, agenti e venditori, si vuole: (*a*) stampare l'ammontare di un certo ordine; (*b*) aggiornare la zona di

un agente e (*c*) stampare le coppie (clienti, ammontare degli ordini) ordinate in modo decrescente secondo l'ammontare.

```
DECLARE
    NumCliente Clienti.CodiceCliente%TYPE;
    NumOrdine Ordini.NumFattura%TYPE;
    NumAgente Agenti.CodiceAgente%TYPE;
    NomeZona Agenti.Zona%TYPE;
    VAmmontare Ordini.Ammontare%TYPE;
    TotaleAmmontare INTEGER;
BEGIN
/* Ricerca dell'ammontare di un certo ordine */
PRINT "Scrivi CodiceAgente, CodiceCliente, NumOrdine";
READ NumAgente, NumCliente, NumOrdine;

SELECT Ammontare INTO VAmmontare
FROM   Ordini
WHERE  CodiceAgente=NumAgente AND CodiceCliente=NumCliente
AND    NumOrdine=NumOrdine;

PRINT VAmmontare;
/* Aggiornamento della zona di un agente */
PRINT "Scrivi CodiceAgente, Zona";
READ NumAgente, NomeZona;

UPDATE Agenti
SET    Zona = NomeZona
WHERE  CodiceAgente = NumAgente;

/* Creazione di un cursore per i clienti e l'ammontare */
DECLARE
    c CURSOR IS
        SELECT    CodiceCliente, SUM(Ammontare) AS Totale
        FROM      Ordini
        GROUP BY  CodiceCliente
        ORDER BY  SUM(Ammontare) DESC;
/* Uso del cursore; */
/* coppia e' implicitamente di tipo c%ROWTYPE */
FOR coppia IN c LOOP
    PRINT coppia.CodiceCliente, coppia.Totale
END LOOP;          END
```

8.4 La programmazione di transazioni

Come specificato nel Capitolo 1, una transazione è un programma che il DBMS esegue garantendone atomicità e serializzabilità.

L'atomicità viene garantita (concettualmente) facendo sì che, quando una transazione fallisce, tutti i suoi effetti sulla base di dati siano disfatti.

La serializzabilità è garantita in genere con il meccanismo del bloccaggio dei dati (*record and table locking*). Prima di leggere, o modificare, un dato una transazione lo deve bloccare in lettura o, rispettivamente, in scrittura. Quando una transazione T1 cerca di ottenere un blocco in scrittura su di un dato già bloccato da T2, T1 viene messa in attesa, finché T2 non termina e quindi rilascia il blocco. Con questa tecnica si garantisce la serializzabilità delle transazioni ed il loro isolamento, ovvero il fatto che una transazione non veda mai le modifiche fatte da un'altra transazione non ancora terminata. Le richieste di blocco sono fatte automaticamente dal sistema senza intervento del programmatore.

Se si adotta l'approccio di considerare ogni comando SQL come una transazione (*autocommit on*), possiamo non essere in grado di disfare un'operazione che abbiamo già fatta e che ha portato la base di dati in uno stato non consistente. È quindi necessario, a volte, gestire esplicitamente le transazioni (*autocommit off*).

La prima possibilità da considerare in questi casi, è quella di far sì che l'intero programma diventi una transazione. Questo approccio, però, funziona bene in casi semplici, ma in generale è necessario poter prevedere altri comportamenti:

- Quando il programma scopre una condizione anomala, che impedisce il completamento di un suo compito, è spesso opportuno avere la possibilità di disfare solo una parte delle operazioni fatte, cercando di aggirare l'anomalia usando del codice alternativo.
- Quando un programma può richiedere un tempo lungo per terminare le proprie operazioni, ad esempio perché interagisce con l'utente, può essere opportuno spezzare il programma in più transazioni, in modo da poter rilasciare i blocchi, per permettere l'esecuzione di altre transazioni che necessitano degli stessi dati.

I comandi **COMMIT** e **ROLLBACK**. permettono di ottenere questi comportamenti, spezzando un programma in più transazioni. Vediamo come questo può essere ottenuto nel caso di SQLJ, assumendo di operare quindi attraverso un contesto con *autocommit off*.

Una transazione viene considerata iniziata dal sistema quando un programma esegue un'operazione su una tabella (ad esempio, **SELECT**, **UPDATE**, **INSERT**, **DELETE**).[4]

La transazione quindi prosegue finché:

1. viene eseguito il comando #sql [contesto] **COMMIT**; che comporta la terminazione normale della transazione e quindi il rilascio dei blocchi sui dati usati, che

4. Di solito i comandi che modificano lo schema (CREATE, DROP e ALTER) sono eseguiti in modo atomico e i loro effetti non possono essere annullati.

diventano così disponibili ad altre transazioni;
2. viene eseguito il comando #sql [contesto] **ROLLBACK**; (*abort transaction*) che comporta la terminazione prematura della transazione, e quindi (*a*) il disfacimento di tutte le modifiche fatte dalla transazione (per garantire la proprietà dell'atomicità) e (*b*) il rilascio dei blocchi sui dati usati;
3. il programma termina senza errori e quindi la transazione termina normalmente;
4. il programma termina con fallimento e provoca la terminazione prematura della transazione.

Nell'esempio precedente, quindi, il programma andrebbe riscritto per portare le parti di codice che interagiscono con l'utente al di fuori della transazione.

Per ottenere questo risultato è sufficiente organizzare il programma come due transazioni: la prima che comincia dopo la prima interazione con l'utente, consistente di un unico **SELECT**, e la seconda, che comprende il comando **UPDATE**, e la dichiarazione e l'uso del cursore. La modifica da fare al programma è quindi l'inserzione del comando:

#sql [contesto] **COMMIT**;

dopo il primo **SELECT**, per forzare la terminazione della prima transazione, dato che la seconda termina con la fine del programma.

In realtà, quando un programma esegue transazioni occorre cercare di renderle meno "estese" possibili, in modo da permettere il più alto grado di concorrenza possibile fra programmi diversi. Così, nell'esempio precedente, è preferibile suddividere ancora la seconda transazione in due, dato che si eseguono due operazioni non correlate (l'aggiornamento della zona di un agente e la stampa delle coppie cliente, ammontare ordini).

La struttura finale del programma diventa quindi:

```
Class Esempio;
Dichiarazioni e Inizializzazioni

{ Lettura dal terminale dei dati di un ordine }

{ Prima transazione: ricerca ordine }
#sql [contesto] COMMIT;

{ Stampa risultato prima transazione }
{ Lettura dal terminale dei dati per la seconda transazione }

{ Seconda transazione: aggiornamento zona di un agente }
#sql [contesto] COMMIT;

{ Terza transazione: recupero e stampa dei dati sui clienti e totali ordini }
#sql [contesto] COMMIT;
END programma.
```

Esempio 8.4 Supponiamo che nella base di dati esista anche la tabella:

Magazzino(Prodotto, Quantità, Prezzo)

Si consideri il seguente frammento di programma, che potrebbe servire ad un venditore al momento della richiesta di un ordine da parte di un cliente. Nel programma si legge la quantità di prodotto disponibile in magazzino e il prezzo corrente. Quindi si legge la quantità ordinata, e si crea l'ordine.

```
import java.sql.*;
import sqlj.runtime.*;

public class Esempio2 {

    public static void main(String [] args) {

        try {
            // Connessione alla base di dati
            Class.forName("oracle.jdbc.driver.OracleDriver");
            #sql context ClasseContesto;
            ClasseContesto contesto =
                    new ClasseContesto("jdbc:oracle:thin:@localhost:1521:mydb",
                            "utente1","password1");

            // Ricerca della quantita' di un certo prodotto
            // Lettura dei parametri
            String numProdotto = leggi("Codice prodotto: ");

            // inizio della prima transazione
            int quantita, prezzo;
            #sql [contesto] SELECT   Quantita, Prezzo INTO  :quantita, :prezzo
                            FROM     Magazzino
                            WHERE    Prodotto = :numProdotto;

            //si termina la transazione prima di interagire con l'utente
            #sql [contesto] COMMIT;

            // Stampa risultato
            System.out.println("La quantita e': " + quantita);
            System.out.println("Il prezzo e': " + prezzo);

            int quantitaRichiesta = new Integer(leggi("Quantita ordinata: ")).intValue();
            int prezzoProposto = prezzo;
```

```
// inizio della seconda transazione
#sql [contesto] SELECT   Quantita, Prezzo INTO  :quantita, :prezzo
                FROM     Magazzino
                WHERE    Prodotto = :numProdotto;

if (quantita >= quantitaRichiesta && prezzo == prezzoProposto) {
    #sql [contesto] UPDATE   Magazzino
                    SET      Quantita = :quantitaRichiesta
                    WHERE    Prodotto = :numProdotto;
    ... soddisfacimento dell'ordine ...
} else { // se la quantita' e' diventata insufficiente o il prezzo e' cambiato
            #sql [contesto] ROLLBACK;
            System.out.println("Richiesta non evasa per cambiamento" +
                " quantita' oppure prezzo");
...
```

Questo esempio mostra come la necessità di dividere un programma in più transazioni per aumentare il grado di concorrenza comporti alcune complicazioni.

Infatti la seconda transazione, quella che effettua l'ordine vero e proprio, invece di aggiornare direttamente il valore della quantità del prodotto, richiede una *nuova* lettura per controllare l'effettiva disponibilità della merce e il suo prezzo. Il controllo è dovuto al fatto che fra la chiusura della prima transazione e l'inizio della seconda, può essere stata eseguita un'altra transazione che ha modificato la quantità (ad esempio per fare un altro ordine), o il prezzo (ad esempio per aggiornare i prezzi dei prodotti). Il controllo quindi è indispensabile per evitare di vendere la stessa merce due volte o ad un prezzo diverso da quello stabilito. A seconda dell'esito del controllo, può essere necessario provocare il fallimento della transazione.

8.4.1 Ripetizione esplicita delle transazioni

Un'altra situazione da prevedere nella programmazione di transazioni è la ripetizione della transazione quando questa viene interrotta dal sistema per sbloccare una condizione di *stallo* (*deadlock*), che si presenta quando due o più transazioni sono bloccate in attesa l'una dei dati dell'altra, e viceversa.

In questo caso, il DBMS scopre lo stallo e interrompe, secondo una propria strategia, una delle transazioni in modo che le altre possano proseguire.

Il fatto che una transazione venga interrotta per sbloccare uno stallo viene segnalato al programma con (**SQLCODE** = **DEADABORT**), e quindi si può decidere di far ripartire la transazione.

Esempio 8.5 Supponiamo di dover fare un aggiornamento che coinvolge molte ennuple, come cambiare il supervisore di tutti gli agenti di una certa zona. Si può usare il seguente frammento di codice, che prova ripetutamente, fino ad un massimo di quattro volte, l'operazione in caso di interruzione della transazione per uno stallo.

```
int tentativi = 0;
boolean riuscito = false;

while ((tentativi < 4) && ! riuscito) {
    try {
        #sql [contesto] UPDATE   Agenti
                        SET      Supervisore = :nuovoSupervisore
                        WHERE    Zona = :nomeZona;
        riuscito = true;
    } catch (SQLException e) {
        #sql [contesto] ROLLBACK;
        tentativi = tentativi + 1;
    }
}
if (! riuscito) {
    System.out.println("Aggiornamento non eseguito per troppi stalli!");
    ...
}
```

In generale, se si decide di far ripartire una transazione interrotta dal sistema, bisogna ricordarsi che mentre i suoi effetti sulla base di dati vengono disfatti automaticamente, i valori di variabili del programma, eventualmente significativi per la corretta esecuzione della transazione, rimangono inalterati perché non sono sotto il controllo del sistema e quindi vanno inizializzati dal programma prima di far ripartire la transazione.

8.4.2 Transazioni con livelli diversi di isolamento

Con l'aumentare del numero di transazioni eseguite concorrentemente in modo serializzabile si può ridurre l'effettivo grado di concorrenza del sistema a causa del fatto che aumenta la probabilità di avere transazioni in attesa di dati bloccati da altre o interrotte per il verificarsi di situazioni di stallo. Per questa ragione i sistemi commerciali prevedono la possibilità di programmare transazioni rinunciando alla proprietà della serializzabilità e quindi di isolamento delle transazioni.

Nella proposta dell'SQL-92, con il comando **SET TRANSACTION** si può scegliere uno dei seguenti livelli di isolamento per consentire gradi di concorrenza decrescenti:

SET TRANSACTION ISOLATION LEVEL [**READ UNCOMMITTED** |
READ COMMITTED |
REPEATABLE READ |
SERIALIZABLE]

Il primo livello di isolamento, *read uncommitted*, detto anche *dirty read* o *degree of isolation 0*, consente transazioni che fanno solo operazioni di lettura (quelle di modifica sono proibite) che vengono eseguite dal sistema senza bloccare in lettura i dati. La conseguenza di questo modo di operare è che una transazione può leggere dati modificati da un'altra non ancora terminata, che vengono detti *sporchi* perché potrebbero non essere più nella base di dati se la transazione che li ha cambiati venisse abortita.

Il secondo livello di isolamento, *read committed*, detto anche *cursor stability* o *degree of isolation 1*, prevede che i blocchi in lettura vengano rilasciati subito, mentre quelli in scrittura vengono rilasciati alla terminazione della transazione. In questo modo, quando una transazione T modifica un dato, quel dato non può essere letto da altri fino a che T non abbia effettuato un commit o un rollback. La conseguenza di questo modo di operare è che una transazione può fare *letture non ripetibili*, ovvero letture successive degli stessi dati possono dare risultati diversi perché i dati sono stati modificati da altre transazioni terminate nell'intervallo tra la prima e la seconda lettura.

Il terzo livello di isolamento, *repeatable read*, detto anche *degree of isolation 2*, prevede che i blocchi in lettura e scrittura siano assegnati solo su ennuple di tabelle e vengano rilasciati alla terminazione della transazione. Questa soluzione evita il problema delle letture non ripetibili, ma non è ancora il tipo di isolamento che occorre per avere transazioni serializzabili in quanto consente ad altre transazioni di fare inserzioni di ennuple nella stessa tabella, creando il fenomeno cosiddetto dei *fantasmi* (*phantoms*). Ad esempio, supponiamo che esista una tabella di riepilogo sulle vendite dei prodotti:

Vendite(Prodotto,TotaleAmmontare)

che venga mantenuta aggiornata dalla transazione che inserisce un nuovo ordine per un prodotto.

Supponiamo che vengano eseguite due transazioni T_1 e T_2 con il livello di isolamento *repeatable read*: T_1 inserisce un ordine per il prodotto 200 e quindi aggiorna la riga corrispondente della tabella Vendite; T_2 legge gli ordini del prodotto 200, calcola il totale e controlla che sia quello della tabella Vendite.

Allora può capitare che le due transazioni vengano eseguite come segue:

T_1	T_2
	Legge gli ordini di un certo prodotto e calcola il totale
Inserisce un nuovo ordine per lo stesso prodotto e aggiorna la tabella Vendite	
	Controlla la somma

in cui T_2 segnala che il valore della tabella Vendite è scorretto, anche se non è vero (dato che è stato inserito un nuovo ordine e aggiornata la somma). Questo è possibile proprio perché T_2 blocca solo le righe degli ordini che legge, e non tutta la tabella, pertanto T_1 può effettuare l'inserzione e la modifica di Vendite.

Per evitare il problema occorre invece che l'insieme delle righe di Ordini lette da T_2 non venga modificato da T_1, ad esempio bloccando tutta la tabella, come fanno quei sistemi che garantiscono il quarto livello di isolamento, *serializable*, detto anche *degree of isolation 3*.

In alcuni sistemi il blocco della tabella può essere anche richiesto esplicitamente dalla transazione con il comando:

LOCK TABLE *Tabella* **IN** [**SHARE** | **EXCLUSIVE**] **MODE**

per semplificare la gestione dei blocchi da parte del sistema o per operare correttamente anche con un livello di isolamento diverso da quello *serializable*. Il comando **LOCK** non è previsto da SQL-92.

La programmazione di transazioni con i primi tre livelli di isolamento richiede in generale un maggior impegno per garantire la corretta evoluzione della base di dati, perché non è possibile astrarre dal fatto che la transazione è eseguita insieme ad altre. In tabella sono riassunti i fenomeni che si possono presentare operando con i diversi livelli di isolamento.

Livello	Letture sporche	Letture non ripetibili	Dati fantasmi
READ UNCOMMITTED	Possibile	Possibile	Possibile
READ COMMITTED	Non possibile	Possibile	Possibile
REPEATABLE READ	Non possibile	Non possibile	Possibile
SERIALIZABLE	Non possibile	Non possibile	Non possibile

8.5 Conclusioni

Sono state presentate le caratteristiche di tre tipi di strumenti per lo sviluppo di applicazioni che usano basi di dati: linguaggi che ospitano SQL, linguaggi che usano interfacce API e linguaggi integrati della quarta generazione.

L'attenzione è stata posta su due aspetti principali: come scambiare informazioni con una base di dati da un programma e come programmare le transazioni. C'è un altro aspetto importante che non è stato discusso per ragioni di spazio: come programmare la parte dell'applicazione dedicata alle interazioni con gli utenti con opportune interfacce grafiche. Ogni sistema offre linguaggi dotati dei meccanismi per farlo ed esistono anche strumenti di ditte indipendenti finalizzati a questo scopo.

Esercizi

1. Si consideri il seguente frammento di codice SQLJ:

```
#sql [contesto]
        SELECT  Ammontare INTO :ammontare
        FROM    Ordini
        WHERE   CodiceAgente = :numAgente
```

   L'elaborazione del codice procede in tre fasi: (a) precompilazione dei frammenti SQL in Java, (b) compilazione del programma Java risultante, (c) esecuzione. Durante l'esecuzione, il controllo si alterna tra macchina astratta Java e il DBMS. Esemplifichiamo ora alcuni motivi per cui tale codice potrebbe essere scorretto. Immaginando che ogni errore sia scoperto prima possibile, specificare chi dei quattro attori (precompilatore, compilatore, macchina astratta Java e DBMS) segnala ciascun errore.

   a) Sintassi SQL: il programmatore potrebbe scrivere WEHRE anziché **WHERE**.
   b) Nomi: il programmatore potrebbe avere sbagliato a scrivere il nome della relazione, oppure quello dell'attributo Ammontare, oppure quello della variabile :ammontare.
   c) Tipi: il tipo di Ammontare e quello di :ammontare potrebbero essere incompatibili.
   d) Variazione dello schema: quando il programma viene eseguito la relazione Ordini potrebbe essere stata cancellata, oppure il tipo dell'attributo Ammontare potrebbe essere cambiato.
   e) Univocità: il valore di :NumAgente potrebbe non essere associato ad alcun agente, oppure essere associato a più agenti.

2. Si consideri la versione API del frammento di codice dell'esempio precedente.

```
PreparedStatement pstmt =
    con.prepareStatement(
        "SELECT Ammontare INTO :ammontare
         FROM Ordini WHERE CodiceAgente = ?");
pstmt.setString(1, codAgente);
risultato = pstmt.executeQuery();
```

   In questo caso l'elaborazione di tale frammento procede come segue: compilazione del programma Java, esecuzione da parte della macchina astratta Java che interagisce con il DBMS. Anche in questo caso si cerchi di individuare in quale momento verrebbe scoperto ciascuno degli errori sopra elencati.

3. Specificare vantaggi e svantaggi della programmazione di applicazioni usando un linguaggio integrato anziché un'API.

4. Si considerino le seguenti applicazioni in ambito bancario. Indicare il livello di isolamento più opportuno per ciascuna di esse, spiegando la risposta.
   a) Per ogni cliente della banca, contare le operazioni effettuate negli ultimi 500 giorni, ed aggiungere il nome del cliente ad un elenco se le operazioni sono più di trecento. L'elenco servirà a scopi di marketing.
   b) Effettuare un trasferimento fondi, sottraendo un ammontare da un conto per aggiungerlo ad un altro.
   c) Gestire un prelievo allo sportello come segue: il cassiere legge sul terminale il saldo corrente del cliente; se questo supera la cifra richiesta dal cliente, il cassiere comunica al sistema la cifra ed effettua il pagamento (specificare quali di queste operazioni sarebbero racchiuse nella transazione).

Note bibliografiche

Per approfondire il problema della programmazione delle applicazioni in SQL si veda [van der Lans, 2001] e [Kifer et al., 2005]. Molte informazioni su JDBC e SQLJ sono disponibili sulla rete, in particolare ai siti http://java.sun.com/products/jdbc/ e http://www.sqlj.org.

BIBLIOGRAFIA

Abiteboul, S. and Bidoit, N. (1984). An algebra for non normalized relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*. 135

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Database Foundations*. Addison-Wesley, Reading, Massachusetts. 27, 136, 178

Albano, A. (2001). *Costruire sistemi per basi di dati*. Addison-Wesley, Milano. 220, 252, 279

Albano, A., Antognoni, G., and Ghelli, G. (2000). View operations on objects with roles for a statically typed database language. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):548–567. 40, 48, 49

Albano, A., Cardelli, L., and Orsini, R. (1985). Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. 40, 48, 49

Armstrong, W. (1974). Dependency structures of database relationships. In *Proceedings of the IFIP Congress*, pages 580–583. 143

Atzeni, P. and Antonellis, V. D. (1993). *Relational Database Theory*. Morgan Kaufmann Publishers, San Mateo, California. 136, 156, 178

Atzeni, P., Ceri, S., Paraboschi, S., and Torlone, R. (2002). *Basi di dati. Modelli e linguaggi di interrogazione*. McGraw-Hill, Milano. 27, 99

Batini, C., Ceri, S., and Navathe, S. (1992). *Conceptual Database Design. An Entity-Relationship Approach*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California. 72, 99

Batini, C., Lenzerini, M., and Navathe, S. (1987). A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):2–18. 90

Beeri, C. and Bernstein, P. (1979). Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59. 147

Bernstein, P. (1976). Synthesizing third normal form relations from functional

dependencies. *ACM Transactions on Database Systems*, 1(4):277–298. 169

Ceri, S., editor (1983). *Methodology and Tools for Database Design*. North-Holland, Amsterdam. 99

Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Data Bases*. Springer-Verlag, Berlin. 136

Codd, E. (1970). A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387. 136, 141

Connolly, T. and Begg, C. (2000). *Database Solutions. A step-by-step guide to building databases*. Addison-Wesley, Reading, Massachusetts. 99

Diederich, J. and Milton, J. (1988). New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems*, 13(3):339–365. 152

Elmasri, R. and Navathe, S. (2001). *Sistemi di basi di dati. Fondamenti. Prima edizione italiana*. Addison-Wesley, Milano. 27

Fraternali, P. and Tanca, L. (1995). A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20(4):414–471. 216

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274. 73

Kifer, M., Bernstein, A., and Lewis, P. M. (2005). *Database Systems*. Addison-Wesley, Reading, Massachusetts, second edition. 27, 202, 250

Lucchesi, C. and Osborn, S. (1978). Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2):270–280. 148

Maciaszek, L. A. (2002). *Sviluppo di sistemi informativi con UML*. Addison-Wesley, Milano. 99

Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland. 136, 152, 153, 178

Mannila, H. and Räihä, K. (1992). *The Design of Relational Databases*. Addison-Wesley, Reading, Massachusetts. 178

Marco, T. D. (1979). *Structured Analysis and System Specification*. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 72

Ramakrishnan, R. and Gehrke, J. (2003). *Sistemi di basi di dati*. McGraw-Hill, Milano. 27

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall International, Inc., London. 72, 73

Schmidt, J. (1977). Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261. 239

Shasha, D. and Bonnet, P. (2002). *Database Tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers, San Mateo, California. 226

Silberschatz, H. F., Korth, H. F., and Sudarshan, S. (2002). *Database System Concepts*. McGraw-Hill, New York, 4th edition. 27, 136

Teorey, T. (1999). *Database Modeling and Design. The E-R Approach*. Morgan Kaufmann Publishers, San Mateo, California, third edition. 99

Tsou, D. and Fischer, P. (1982). Decomposition of a relation scheme into Boyce-Codd

Normal Form. *ACM SIGACT News*, 14(3):23–29. 165

Ullman, J. (1983). *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, second edition. 158

Ullman, J. D. and Widom, J. (2001). *A First Course in Database System*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, second edition. 27, 136, 152

van der Lans, R. (2001). *Introduzione a SQL. Seconda edizione italiana*. Addison-Wesley, Milano. 202, 250

Widom, J. and Ceri, S., editors (1996). *Active Database Systems: Trigger and Rules for Advanced Database Programming*. Morgan Kaufmann Publishers, San Mateo, California. 226

Yourdon, E. (1989). *Modern Structured Analysis*. Yourdon Press, Englewood Cliffs, New Jersey. 72

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R., Subrahmanian, V., and Zicari, R., editors (1997). *Introduction to Advanced Database Systems*. Morgan Kaufmann Publishers, San Mateo, California. 226

INDICE ANALITICO