Programmazione delle basi di dati attraverso JDBC

Dispense di Basi di dati

Queste breve note non hanno la pretesa di essere un nuovo libro di testo sulle tecnologie informatiche. Vogliono invece costituire *un quaderno di appunti* agile per i corsi di TSI per gli allievi Ingegneri Gestionali e Basi di Dati per gli allievi ingegneri Informatici dell'Università di Napoli Federico II. Per questa sua natura e specificità, questa dispensa non è definitiva ed è soggetta a ulteriori revisioni individuate dalla data di compilazione del documento.
Draft del: 4 giugno 2006

Contents

1 Inroduzione

Nella pratica gli utenti finali accedono al contenuto di una base di dati, non direttamente attraverso l'uso del linguaggio SQL, ma attraverso programmi applicativi che realizzano apposite interfacce di accesso alla base di dati stessa. A tale proposito, basta pensare alle interfacce web che consentono in maniera del tutto "grafica" di prenotare un aereo, di acquistare un prodotto o di visualizzare gli ultimi movimenti sul proprio conto bancario.

Negli esempi illustrati, la classica interazione SQL con la base di dati viene nascosta all'utente e racchiusa all'interno del programma, ed è quindi compito del programmatore dell'applicazione implementare le varie funzionalità previste per l'accesso ai dati. In tale ottica fondamentale importanza assumono quindi i meccanismi con cui le applicazioni accedono ai dati.

In questo capitolo verranno esaminate quelle che sono le tecniche basate sulle **API JDBC** per l'accesso a basi di dati da programmi applicativi.

2 Cenni agli approcci per la programmazione di basi di dati

Nel corso degli anni molte sono state le tecniche sviluppate per includere interazioni con le basi di dati all'interno di programmi applicativi; gli approcci principali possono essere considerati quelli descritti di seguito.

—*SQL-embedded*: in tale approccio le istruzioni SQL per l'interazione con la base di dati sono direttamente incapsulate nel linguaggio di programmazione ospite (e.g., C/C++, PASCAL, COBOL, etc...) e identificate attraverso dei prefissi speciali (e.g., EXEC SQL). Un **precompilatore** o **preprocessore** esamina prima di tutto il codice sorgente del programma per identificare le istruzioni di interazione con la base di dati ed estrarle per l'elaborazione con il DBMS. Queste vengono sostituite nel programma da chiamate di funzione al codice generato dal DBMS. La comunicazione dei risultati elaborativi del DBMS all'applicazione avviene a mezzo di apposite **variabili condivise** (e.g., SQLCODE, SQLSTATE), mentre le variabili del programma possono essere usate come parametri nelle istruzioni SQL . L'esempio 0**??** mostra un esempio di programma C con SQL incapsulato che trova il nome e cognome di uno studente, data la matricola (la clausola INTO specifica le variabili del programma in cui verranno memorizzati i valori degli attributi della selezione, mentre la variabile condivisa SQLCODE è una variabile intera che vale zero se l'istruzione SQL viene eseguita correttamente). Le operazioni di selezione che producono una sola tupla e le operazioni di aggiornamento possono essere immerse senza problemi, di contro, le *select* che ritornano un insieme di tuple o record vanno gestite con l'introduzione di apposite strutture dati, dette **cursori**, che agendo

come dei puntatori, permettono di accedere iterativamente e alle singole tuple del risultato.

—*SQL/CLI (Call Level Interface)*: in tale approccio delle apposite libreria di funzioni sono rese disponibili al linguaggio ospiste per l'interfacciamento alla base di dati. Vi sono funzioni per la connessione al database, per l'esecuzione di interrogazione, per l'aggiornamento dei dati, e così via. I comandi effettivi di interrogazione ed aggiornamento delle base di dati e, qualsiasi altra informazione necessaria, vengono inclusi come parametri nelle chiamate a funzione. Questo approccio fornisce delle apposite `API` (Application Programming Interface) per l'accesso ad una base di dati da programmi applicativi.

—*Linguaggi di programmazione per basi di dati*: in tale approccio, vengono definiti appositi linguaggi di programmazione "compatibili" con il modello logico della base di dati e con il linguaggio di interrogazione SQL. Le istruzioni classiche SQL di interazione col database sono arricchite ed estese dalle classiche istruzioni tipiche di un linguaggio di programmazione (definzione di tipi e variabili, strutture di controllo, statement e costrutti iterativi, definizione di procedure e funzioni, etc..), al fine di ottenere un vero e proprio linguaggio di programmazione completo per le basi di dati. Un esempio di linguaggio di questo tipo è il `PL/SQL` dell'Oracle che integra la potenza e la flessibilità di SQL con i costrutti tipici di un linguaggio procedurale.

I primi due approcci sono sicuramente i più comuni, in quanto molte applicazioni scritte in linguaggi di tipo general-purpose richiedono l'accesso ad una base di dati. In tali approcci, le tecniche di accesso alla base di dati devono essere in grado di colmare le differenze esistenti tra il modello della base di dati ed il modello del linguaggio di programmazione (i.e., *conflitto di impedenza o impedance mismatch*). Basti pensare alla necessità di effettuare un mapping tra i tipi di dato del linguaggio e quelli previsti dallo standard SQL, oppure all'esigenza, come già accennato, di convertire la struttura dei dati del risultato di un interrogazione (insieme di tuple o record) in una struttura dati appropriata del linguaggio di programmazione (i.e, uso dei cursori).

Il terzo approccio è invece più diffuso per applicazioni che hanno un'interazione molto forte con la base di dati e non presenta problemi di conflitti di impedenza, in quanto è il linguaggio stessa a colmora il gap tra la tecnologia dei database e le caratteristiche di un linguaggio procedurale.

Algorithm 1 SQL Embedded

```
EXEC SQL BEGIN DECLARE_SECTION;
varchar nome_stud [50];
varchar cognome_stud [50];
varchar matr_stud [10];
int SQLCODE;
EXEC SQL END DECLARE_SECTION;
printf("Immettere matricola: ");
scanf("%s",&matr_stud);
EXEC SQL;
SELECT NOME, COGNOME into :nome_stud, :cognome_stud FROM STUDENTI
WHERE MATRICOLA=:matr_stud;
if (SQLCODE==0) printf("Studente: %s %s", nome_stud,cognome_stud);
else printf("Matricola inesistente");
```

3 Accesso mediante SQL/CLI e lo standard ODBC

Con l'evoluzione dei database sono cambiati i meccanismi con cui le applicazioni accedono ai dati. Il linguaggio SQL-embedded risulta un approccio di programmazione delle basi di dati **statico** in quanto il testo dell'interrogazione è scritto all'interno del programma e non può essere cambiato senza ricompilare o rielaborare il codice sorgente. Inoltre poichè ogni applicazione per interfacciarsi ad un DBMS fa riferimento a delle proprie librerie (di solito scritte `C`), se quest'ultima deve utilizzare per qualche ragione un nuovo database, diverso dal precedente, occorre riscrivere tutto il codice di gestione dei dati.

Per rimediare alle problematiche sopra esposte è stato creato uno standard a "livello di chiamata" per l'interfacciamento alle basi di dati detto appunto `SQL/CLI` proposto da X/Open.

L'uso delle chiamate a funzione risulta essere un approccio **dinamico** per la programmazione delle basi di dati: per l'accesso alla base di dati viene usata una libreria di funzioni che da un lato fornisce maggiore flessibilità e non richiede la presenza di alcun preprocessore, dall'altro però, comporta che la verifica della sintassi e altri controlli sui comandi SQL avvenga solo al momento dell'esecuzione del programma. Con `SQL/CLI` viene defininta la sequenza standard di chiamate a funzione che un'applicazione deve eseguire per accedere in modo corretto alle informazioni presenti nella base di dati (vedi figura 1).

Basandosi sullo standard `SQL/CLI`, ogni DBMS poteva quindi mettere a disposizione apposite interfacce di programmazione (API) per i principali linguaggi (`C/C++`,

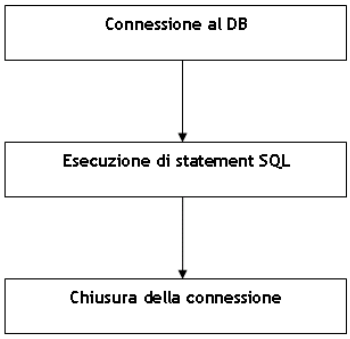

Fig. 1. Sequenza chiamate SQL/CLI

VB, etc..), che permettevano alle applicazioni scritte in quel linguaggio di interrogare ed aggiornare il contenuto delle base di dati mediante semplici **statement SQL**.

Col passare degli anni nasce poi l'esigenza di standardizzare l'interfaccia attraverso la quale un'applicazione poteva accedere ad una qualsiasi base di dati relazionale (e.g., Oracle, Access, DB2, MySQL, etc..), garantendone l'interoperabilità . A tale proposito nel 1991 la Microsoft propone uno standard per la comunicazione con database remoti noto col nome di ODBC (Open Database Connectivity).

Tramite l'interfaccia ODBC (scritta in C), applicazioni eterogenee, a mezzo dei classici comandi SQL (il linguaggio supportato era inizialmente un SQL "ristretto" caratterizzato da un insieme minimo di istruzioni), possono accedere direttamente a dati remoti , presenti su quelli che comunemente sono definiti **server** di basi di dati.

L'architettura dei sistemi di basi di dati più evoluti si basa infatti sul classico paradigma di comunicazione "client-server, in cui sono presenti uno o più processi **client** che richiedono un servizio, e, un processo server che eroga tale servizio. Nel caso di un Database Management System, il servizio offerto è quello di accesso ai dati presenti all'interno della base di dati, mentre il linguaggio di formulazione delle richieste da parte dei client è l'SQL (vedi figura 2).

Nell'architettura ODBC il collegamento tra un'applicazione client e il server della base di dati, richiede l'uso di un *driver*, ovvero di una libreria che viene collegata dinamicamente all'applicazione e da essa invocata quando si vuole accedere alla base di dati. Ogni driver maschera le differenza di interazione legate non solo al DBMS, ma anche al sistema operativo e ai protocolli di rete utilizzati. In altri termini i driver astraggono dai protocolli specifici dei produttori dei DBMS, fornendo un'interfaccia di programmazione delle applicazioni comune ai client del database (i produttori di

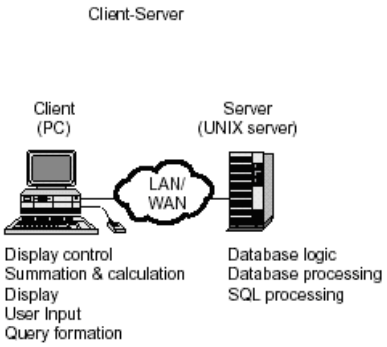

Fig. 2. Architettura client-server

DBMS devono fornire diversi driver a seconda del protocollo di comunicazione scelto e del sistema operativo della macchina server). Usando nel proprio client le chiamate (API) all'interfaccia ODBC si rende il proprio programma capace di accedere quindi a più server di database, senza dovere conoscere le interfacce proprietarie di ogni singolo database.

L'accesso ad una base di dati remota mediante ODBC richiede la cooperazione di quattro componenti (vedi figura 3):

— *l'applicazione*: richiama le funzioni SQL per eseguire interrogazione e acquisire risultati. Tramite l'uso dei driver, all'applicazione sono trasparenti: il protocollo di comunicazione, il tipo di DBMS e il sistema operativo installato sul server dove risiede il DBMS.

—*il driver manager*: è responsbile di caricare i driver di connessione su richiesta dell'applicazione.

—*i driver*: sono responsabili di eseguire le funzioni ODBC e, pertanto, sono in grado di eseguire le interrogazioni SQL traducendole nel dialetto del DBMS locale e di restituire i risultati alle applicazioni tramite meccanismi di buffering.

—*la sorgente dati o data source*: è la fonte delle informazioni, ovvero il sistema remoto che esegue le funzioni trasmesse dal client.

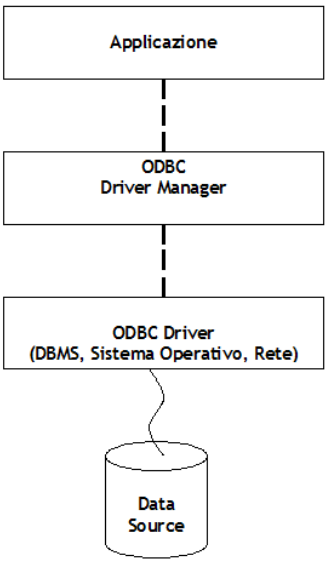

Fig. 3. Architettura ODBC

In ODBC le interrogazioni SQL possono essere specificate in modo statico (e compilate) oppure essere incluse in stringhe che vengono generate ed eseguite dinamicamente.

4 JDBC

4.1 Introduzione

JDBC è un interfaccia di programmazione (API) ad oggetti basata sullo standard CLI e definita dalla Sun Micorsystems (non è un acronimo di JAVA Database Connectivity ma è un semplice marchio registrato) per l'accesso ad un database in maniera indipendente dalla piattaforma.

JDBC consiste di un insieme di classi ed intefacce scritte in JAVA (l'intero insieme delle API JDBC è contenuto nei package java.sql e javax.sql) e costituisce quella che si può dire la "controparte JAVA" di ODBC, fornendo un insieme di oggetti e metodi per l'interazione con una base di dati.

A differenza di ODBC che è maggiormente diffuso per l'interfacciamento di applicazioni eterogenee a database, JDBC risulta molto usato per garantire l'interazione di programmi JAVA con basi di dati.

In particolare, esso fornisce ai programmatori di basi dati i seguenti servizi per l'accesso a dati resisenti in database relazionali:

—Connessione al database;

—Invio di statement SQL;

—Processing dei risultati;

—Gestione delle Transazioni.

JDBC è divenuto oramai uno "standard de facto" per lo sviluppo di applicazioni JAVA "DB-oriented" e fa parte del pacchetto software JDK dalla versione 1.1. Con l'avvento di JAVA 2 è stato poi introdotta la versione 2.0, mentre attualmente nella versione 1.5 della java virtual machine è stata inclusa la sua versione 3.0 che presenta funzionalità estese e migliorate rispetto alle versioni precedenti.

4.2 Architettura

L'architettura base di JDBC, rappresentata in figura , è costituita dalla serie di strati o livelli descritti di seguito.

—*JAVA application*: a questo livello troviamo l'applicazione JAVA che richiede l'accesso ad una base di dati.

—*JDBC Driver Manager*: rappresenta il livello di gestione di JDBC e opera tra l'utente e i driver. Tiene traccia dei driver disponibili e gestisce la connessione tra un DB ed il driver appropriato

—*JDBC Driver*: rappresentano lo strato base dell'architettura proosta, quello più a contatto con i database. A questo livello troviamo una serie di **driver** che realizzano la vera comunicazione fisica con il database. Essi permettono di creare la connessione con la sorgente dati, inviare istruzione di interrogazione o di aggiornamento alla sorgente dati e di processare i risultati.

—*DBMS*: a questo livello troviamo il DBMS relazionale a cui l'applicazione si deve interfacciare.

Come si può osservare dalla figura esistono 4 differenti tipi di driver: due di questi si poggiano su un modello **three-tier**, mentre glia ltri due su un modello **two-tier**.

Nel primo caso affinché l'applicazione possa interagire col database occorre che le chiamate JDBC siano convertite in chiamate API native (caso **JDBC/Native bridge**) o in chiamate ODBC (caso **JDBC/ODBC bridge**). Tale soluzione ,in cui un utente può utilizzare API di alto livello che vengono poi convertite dal driver in chiamate di basso

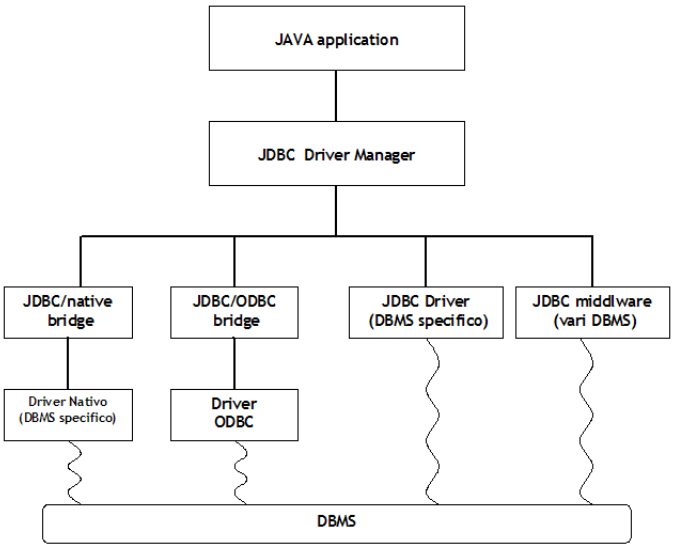

Fig. 4. Architettura JDBC

livello (JDBC funge da ponte per la connessione), non risulta realmente portabile in quanto richiede la presenza di componenti nativi.

Nel secondo caso l'applicazione interagisce direttamente con il database mediante un opportuno protocollo di rete standard come TCP/IP (caso **JDBC middleware**), oppure mediante un protocollo di rete proprietario del database (caso **JDBC driver**). In tal caso la soluzione proposta si basa su un ambienete completamente JAVA.

4.3 Interfacce e classi JDBC

Di seguito viene riportata una breve descrizione delle interfacce (entità simili alle classi, ma che non contengono le implementazioni dei metodi) e delle classi principali che costituiscono il package java.sql.

—*Interfaccia Driver*: rappresenta il punto di partenza per ottenere una connessione ad un DBMS. I produttoti di driver JDBC devono implementare un'interfaccia Driver (mediante un'opportuna classe) affinché un'applicazione possa interfacciarsi con un tipo particolare di DBMS. Avendo a disposizione un oggetto Driver è possbile ottenere la connessione al database.

—*Classe DriverManager*: è la classe che contiente tutti i driver di database registrati e permetti di associarli ai database stessi. Tramite la suddetta classe è possibile registrare un nuovo driver, deregistrare uno esistente e ottenere l'enumerazione di quelli disponibili. Nel momento in cui un oggetto Driver viene istanziato viene automaticamente registrato nella classe DriverManager come disponibile.

—*Interfaccia Connection*: un oggetto di tipo Connection rappresenta una connnessione attiva ad un database.

—*Interfaccia Statement*: un oggetto di tipo Statement viene utilizzato per inviare query semplici SQL, che non fanno uso di parametri, verso il DBMS (una query può comprendere: *update*, *select*, *create*, *insert* o *delete*).

—*Interfaccia PreparedStatement*: un oggetto di tipo PreparedStatement viene utilizzato per inviare query che presentano parametri di input.

—*Interfaccia CallableStatement*: un oggetto di tipo CallableStatement viene usato per costruire query parametriche con parametri di input e output. Consente di eseguire anche **stored procedure** memorizzate sul server.

—*Interfaccia ResultSet*: un oggetto ResultSet rappresenta il risultato di una query di selezione (insieme di record o tuple).

Una volta disponibile un oggetto Driver, affinché un'applicazione Java possa accedere ad una sorgente dati occorre che essa specifichi all'atto della connessione alla classe DriverManager le informazioni necessarie (in un formato prestabilito) per localizzarla.

In JDBC ogni database viene univocamente identificato da una particolare stringa di connessione detta `JDBC URL`. Tale stringa adotta un formalismo simile alla definizione degli URL e serve a definire dove si trova e come accedere al database. In particolare, essa ha una struttura del tipo:

jdbc: ⟨driver⟩ : ⟨database⟩

dove il primo termine indica il protocollo (JDBC) da utilizzare per la gestione della risorsa, il secondo il driver che bisogna adoperare, mentre il terzo specifica la risorsa a cui ci si vuole connettere.

In connessioni di tipo `JDBC-ODBC` il terzo termine coincide con il nome della sorgente dati utente o di sistema (*Data Source Name o DSN*), ed, un possibile JDBC URL potrebbe essere il seguente:

jdbc:odbc:dbsegreteria

dove dbsegreteria rappresenta il nome di un DSN disponibile.

Nel caso di connessioni dirette con driver java puri, il terzo termine invece definisce i parametri di accesso dell'istanza di database. Un esempio più elaborato può essere quello per l'accesso ad un database server Oracle mediante driver di tipo **thin**:

jdbc:oracle:thin:@db.unina.it:1521:dbsegreteria

dove db.unina.it è il nome del server, 1521 è la porta su cui è in ascolto il database e dbsegreteria rappresenta il nome del database vero e proprio, anche noto come `SID`.

5 Programmare un'applicazione con JDBC

La più semplici e comuni operazioni sui database relazionali sono costituite dalle query SQL. I passi principali da compiere per l'esecuzione di una query con l'`API JDBC` sono i seguenti.

(1) *Importazione dei package JDBC*.
(2) *Registrazione dei driver JDBC*.
(3) *Apertura della connessione al database*.
(4) *Creazione dell'oggetto Statement*.
(5) *Esecuzione della query e restituzione dell'oggetto ResultSet*
(6) *Utilizzo dei risultati*.
(7) *Chiusura degli oggetti Statement e ResultSet*.
(8) *Chiusura della connessione*.

5.1 Importazione dei package JDBC

Il primo passo da compiere per la creazione di un'applicazione Java che si connette ad una base di dati mediante è l'importazione dei package JDBC che avviene attraverso l'istruzione:

Algorithm 2 Importazione package

```
import java.sql.*;
```

5.2 Registrazione o caricamento del driver JDBC

Il secondo passo consiste nella registrazione o caricamento del driver che avviene con la seguente istruzione:

```
Class.forName(class_driver);
```

dove class_driver rappresenta il driver che gestisce la nostra base di dati.

Se ad esempio vogliamo conneterci ad una database Oracle, un possibile driver che possiamo caricare è quello mostrato nell'esempio successivo. Il driver caricato sarà d'ora in poi l'interfaccia vera e propria con il database.

Algorithm 3 Caricamento Driver

```
Class.forName(oracle.jdbc.driver.OracleDriver);
```

5.3 Apertura della connessione al database

L'operazione successiva al caricamento dei driver è la connessione al database univocamente individuato dalla stringa di connessione o `JDBC URL`.

Al termine di questa operazione si dispone di un oggetto di tipo Connection che rappresenta la connessione stessa. Per l'apertura della connessione l'applicazione si affida al DriverManager attraverso un'istruzione del tipo:

```
Connection conn =
DriverManager.getConnection(url_connect,User,Password);
```

dove url_connect rappresenta la stringa di connessione `JDBC URL` e User e Password le credenziali d'accesso al database.

Il passo precedente in cui abbiamo caricato la classe del driver nella java virtual machine era quindi necessario affinché il driver manager possa trovare il driver giusto per gestire il database presente al `JDBC URL` specificato. Questa stringa è quindi strettamente legata al tipo di driver e al database e specifica le informazione necessarie per la connessione.

Sempre nel caso di una connessione ad un database Oracle identificato dalla stringa di connessione, già vista precedentemente, avremo la seguente procedura.

Algorithm 4 Apertura Connessione

```
Connection conn=DriverManager.getConnection
("jdbc:oracle:thin:@db.unina.it:1521:dbsegreteria","Vinni","Vinni");
```

La copiia username-password Vinni Vinni rappresenta la credenziale d'accesso al database slezionato.

5.4 Creazione dell'oggetto Statement

Creata la connessione al database si è in possesso di un oggetto che la rappresenta. Da esso è possibile ottenere tre diversi tipi di interfacce, Statement, PreparedStatement, CallableStatement, che effettivamente permettono di sottomottere istruzioni SQL al database.

L'istruzione per creare un oggetto di tipo `Statement` è riportata di seguito.

Algorithm 5 Creazione Statement

```
Statement stmt=conn.createStatement();
```

5.5 Esecuzione della query e restituzione dell'oggetto ResultSet

Una volta creato lo statement è possibile eseguire le query SQL. Se ad esempio vogliamo interrogare la tabella `STUDENTI` e trovare tutti gli studenti iscritti, una possibile istruzione è riportata di seguito.

Algorithm 6 Esecuzione Statement di selezione

```
ResultSet rs=stmt.executeQuery("SELECT * FROM STUDENTI ");
```

Il metodo executeQuery restituisce poi un oggetto di tipo ResultSet contenente i dati recuperati sotto forma di array di record.

In realtà un oggetto di tipo Statement fornisce tre diversi metodi per eseguire una query SQL.

—(StatementObj.)executeQuery(query): per statement che generano un unico oggetto di tipo ResultSet (*select*).

—(StatementObj.)executeUpdate(stmt): per statement di modifica (*insert*, *update*, *delete*) o DDL (*create table* o *drop table*). In questo caso viene restituito un numero intero (contatore di aggiornamento) rappresentante il numero di righe che sono state nserite/aggiornate/cancellate, o il valore 0 per statement di tipo DDL.

—(StatementObj.)execute(stmt): per statement che possono includere più `ResultSet` o contatori di aggiornamento

Ad esempio se vogliamo aggiornare il cognome dello studente avente matricola 041/002059, una serie di possibili istruzioni è riportata di seguito.

Algorithm 7 Esecuzione Statement di aggiornamento

```
int updatedrows=stmt.executeUpdate
("UPDATE STUDENTI SET COGNOME = 'Moscato' WHERE MATRICOLA='041/002059')";
```

5.6 Utilizzo ed elaborazione dei risultati

Come visto il metodo executeQuery ritorna un oggetto di tipo ResultSet che contiene il risultato dell'interrogazione.

Esso in realtà è una sorta di cursore che punta al primo record dei risultati e presenta una serie di metodi che permettono di scorrere l'array dei risultati e di prelevare tutte le infromazioni desiderate dal record corrente.

In particolare, attraverso il suo metodo (ResultSetObj.)next() è possibile scorrere i record del risultato dal primo all'ultimo (tale metodo restituisce true in caso di successo, false se non ci sono più record nel risultatio), mentre attraverso i metodi (ResultSetObj.)getXXX(column name) e (ResultSetObj.)getXXX(column number) è possibile accedere ai singoli campi del record (o attraverso il nome o l'indice del campo) e prelevarne i valori corrispondenti.

Si può notare che esiste una versione di getXXX() per tutti i tipi supportati da Java: e.g., getInt,getFloat, getFloat,getDouble, getString,getBoolean,getTime, getDate,etc...

A tal fine deve essere effettuato un mapping tra i tipi di dato SQL e quelli Java per risolvere il citato conflitto di impedenza. La conversione trai tipi riguarda tre categorie:

—Alcuni tipi di dato SQL hanno i diretti equivalenti in JAVA e possono essere letti direttamente nei tipi Java (e.g., il tipo `INTEGER` SQL è equivalente al tipo int di Java).

—Alcuni tipi di dato SQL possono essere convertiti negli equivalenti tipi in JAVA (e.g., i tipi SQL CHAR e VARCHAR possono essere convertiti nel tipo String di Java).

—Alcuni tipi di dato SQL sono unici e necessitano della creazione di un oggetto speciale Java, relativo ad una data classe per ottenere l'equivalente SQL (esempio: il tipo SQL DATE si converte nell'oggetto Date definito dall'ononima classe Java).

I valori NULL SQL sono convertiti in `null`, 0 o `false` dipendentemente dal tipo di metodo getXXX().

5.7 Chiusura degli oggetti Statement e ResultSet e della connessione

Una volta completate le operazioni sul database è molto importante rilasciare le risorse acquisite. Ciò avviene attraverso la chiusura degli oggetti Statement, ResultSet e Connection con istruzioni del tipo risportato di seguito.

5.8 Osservazioni sull'oggetto PreparedStatement

L'oggetto PreparedStatement viene usato quando una query SQL prende uno o più parametri di input o quando deve essere ripetuta più volte. In questo caso conviene ottimizzare la query prima della sua esecuzione attraverso l'oggetto sopra citato.

Algorithm 8 Chiusura Connessione

```
rs.close();
stmt.close();
conn.close();
```

L'interfaccia PreparedStatement estende l'interfaccia Statement ereditandone tutte le funzionalità con dei metodi aggiuntivi per la gestione dei parametri. Esso risulta correlato ad una data query parametrica specificata all'atto della creazione; i paramentri sostitutivi della query sono indicato con il carattere '?' e vengono numerati a partire da 1. Esecuzione per esecuzione i parametri verranno poi sostituiti dai valori specificati dall'utente.

L'oggetto viene creato con l'istruzione:

```
conn.PreparedStatement(stmt)
```

mentre i parametri sono settati mediante il metodo:

```
(StatementObj.)setXXX(n,value)
```

dove n è l'indice del parametro e value è il valore. Anche qui esiste una versione di setXXX() per tutti i tipi supportati da Java: e.g., setInt,setFloat, setFloat,setDouble, setString,setBoolean, setTime,setDate,etc...

La query precompilata viene poi eseguita mediante i metodi executeQuery, executeUpdate o execute.

Di seguito è riportato un esempio di esecuzione di una query di inserimento parametrica.

Algorithm 9 Uso Prepared Statement

```
PreparedStatement pstmt=conn.prepareStatement
("INSERT INTO STUDENTI VALUES (?,?,?)");
pstmt.setString(1,"010/000010")
pstmt.setString(2,"Diego Armando")
pstmt.setString(3,"Maradona")
rs=pstmt.executeUpdate();
```

5.9 Osservazioni sull'oggetto Callable Statement

Se si vogliono effettuare query in uno o più punti di uno stesso programma, può essere utile, qualora il DBMS utilizzato lo supporti, immagazzinarle direttamente nel database sottoforma di **stored procedure** o **funzioni**.

Le stored procedure sono utilizzate nella programmazione delle basi di dati per effettuare operazioni frequenti e/o pesanti sul database che in generale non ritornino valori se non come parametri. Mentre le funzioni sono usate per effettuare operazioni del medesimo tipo sul database che, invece, ritornano valori o tabelle.

Per invocare le stored procedure o le funzioni, Java mette a disposizione dei programmatori l'oggetto CallableStatement derivato dall'oggetto Statement. Esso è correlato ad una precisa invocazione di funzione o procedura (parametrica), specificata all'atto della sua creazione.

Così come per l'oggetto PreparedStatement, i paramentri di ingresso/uscita sostitutivi della procedura o funzione sono indicati con il carattere '?' e vengono numerati a partire da 1. Esecuzione per esecuzione i parametri verranno poi sostituiti dai valori specificati dall'utente.

Un oggetto CallableStatement si ottiene dall'oggetto Connection tramite il seguente metodo:

```
conn.PrepareCall(String sql)
```

dove la string sql segue una sintassi del tipo:

```
{call nome-procedura(?,?,..)}, {call nome-funzione(?,?,..)}, {?=call
nome-funzione(?,?)}
```

Per impostare i parametri di ingresso, analogamante ai casi precedenti, l'oggetto CallableStatement fornisce una serie di metodi di setXXX(int indice, XXX valore) per tutti i tipi supportati da Java. Per impostare un parametro a NULL è invece disponibile il metodo setNull(int indice, int tipo).

A differenza degli altri casi oltre a specificare i parametri di input, è necessario anche indicare a Java il tipo e valore di eventuali parametri di output ritornati dalle procedure e delle funzioni. Ciò avviene utilizzando il metodo:

```
registerOutParameter(int indice, int tipo)
```

essendo tipo uno dei tipi predefiniti nel package java.sql.Types ed indice l'indice ordinale del parametro.

L'esecuzione della funzione o della procedura avviene invece utilizzando sull'oggetto CallableStatement uno dei due metodi:

```
boolean execute()
ResultSet executeQuery()
```

dove il primo ritorna un valore booleano che indica la procedura o funzione è andata a buon fine, mentre il secondo viene utilizzato per le funzioni che restituiscono un ResultSet.

I risultati di una funzione o procedura sono recuperabili (nel caso di singoli risultati e non di ResultSet) direttamente con metodi di XXX getXXX(int indice) del tipo:

String getString(int indice), int getInt(int indice),...

essendo indice il numero ordinale del parametro da ottenere. E' disponibile il metodo boolean wasNull() per programmare il comportamento dell'applicazione in presenza di valori nulli.

Sono di seguito riportati 2 esempi, uno relativo al richiamo di una procedura PL/SQL inserisci-studente per l'inserimento di un nuovo studente, l'atro relativo al richiamo della funzione PL/SQL conta-studenti-per-cognome per il conteggio del numero di studenti aventi lo stesso cognome .

Algorithm 10 Uso Callable Statement

```
CallableStatament cs=conn.prepareCall("{call inserisci-studente(?,?,?)}");
cs.setString(1,"010/000010");
cs.setString(2,"Diego Armando");
cs.setString(3,"Maradona")
cs.execute();
CallableStatament cs=
conn.prepareCall("{call?=conta-studente-per-cognome(?)}");
cs.registerOutParameter(1,types.INTEGER)
cs.setString(2,"Maradona")
cs.execute();
int numstudenti=cs.getInt(1);
```

6 Gestione degli errori

Così come avviene di norma nelle applicazioni JAVA, un aspetto abbastanza importante da tenere in considerazione risulta essere quello relativo alla **gestione delle eccezioni**.

Tutte le istruzioni JDBC viste finora possono generare eccezioni qualora si riscontrino problemi nella comunicazione con il databsse e, per questo motivo, devono essere inserite all'interno di appositi costrutti di try-cacth.

Le SQLException ed i SQLWarning eventualmente generati contengono informazioni su quello che sta realmente accedendo e possono essere utilizzati per l'individuzione e la diagnostica degli errori.

Ricordiamo che in Java un costrutto di tipo try-cacth ha una sintassi del tipo:

try { ⟨sequenza istruzioni⟩} catch (ExceptionType exc){ ⟨sequenza istruzioni⟩}

dove le istruzioni nel try sono quelle che possono generare eccezioni, ciò comportamenti anomali del programma, mentre le istruzioni nel catch sono quelle che vengono intraprese dal programma a valle dell'intercettamento di un'eccezione.

Ogni qualvolta viene eseguita un'istruzione JDBC, questa può generare un'eccezione di tipo SQLException e qundi deve essere inserita in un apposito costrutto trycacth. La classe SQLException, derivata dalle classi più generali java.lang.Exception e java.lang.Throwable, offre un serie di informazioni relative al tipo di errore verificatosi. In particolare le informazioni contenute nella classe sono le seguenti:

—il tipo di errore verificato sotto forma di stringa descrittiva; tale informazioni può essere utilizzata come exception message e può essere ricavata per mezzo del metodo getmessage();

—una proprietà (SQLState) descrivente l'errore sulla base dello standard X/Open SQLState. Può essere ottenuta con getSQLState;

—un codice di errore spesifico del DBMS, che in genere corrisponde col messaggio di errore fornito dal DBMS stesso; il metodo getErrorCode() permette la sua lettura.

—Un collegamento al successico oggetto di tipo SQLException, eccezione che può essere utilizzata se si sono verificati più errori. Il metodo getNextException() permette di navigare nella catena di eccezioni.

L'esempio seguente mostra un corretto utilizzo delle API JDBC per l'accesso ad un DBMS ORACLE per la lettura della tabella STUDENTI con una gestione completa delle eccezioni (ricordiamo che il metodo System.out.println(String s) ha l'effetto di visualizzare a video il contenuto della stringa s).

7 I metadati

JDB permette quello che in gergo viene chiamato **accesso dinamico al database**, ovvero la possibilità di accedere ad un database e ricavarne, senza sapere nulla a priori, la sua struttura interna (tabelle, relazioni, sinonimi, trigger, etc...).

Ciò è possibile grazie ai dizionari o cataloghi che i moderni DBMS relazionali posseggono. Essi infatti contengono tutte le metainformazioni o metadati circa la struttura interna di un database.

La classe JDBC che consente l'interrogazione del catalogo e l'acquisione, ad esempio, dei nomi e del numero dei campi di una tabella è la classe ResultSetMetadata. Un oggetto di tale classe viene creato a partire da un oggetto di tipo ResultSet (attraverso il metodo ResultSet(Object).getMetadata()) e consente di determinare il numero (attraverso il metodo ResultSetMetadata(Object).getColumnCount()) ed il nome dei campi (attraverso il metodo ResultSetMetadata(Object).getColumnName(Column

Algorithm 11 Applicazione JDBC con gestione degli errori

```
try {
Class.forName(oracle.jdbc.driver.OracleDriver);
Connectio conn = DriverManager.getConnection
(”jdbc:oracle:thin:@db.unina.it:1521:dbsegreteria”,”Vinni”,”Vinni”);
Statement st=conn.createStatement();
ResultSet rs=st.executeQuery(”select * from STUDENTI”);
} catch (ClassNotFoundException cnfe) {
System.out.println(”Classe non trovata” + cnfe.getMessage());}
catch(SQLException sqle) {
System.out.println(”SQL Error”);
while sqle!=null {
System.out.println(”Messaggio SQL: ” + sqle.getMessage());
System.out.println(”SQLState: ” + sqle.getSQLState());
System.out.println(”CODICE ERRORE SQL: ” + sqle.getErrorCode();
sqle.getNextException();}}
```

index)) di un dato insieme di record. L'esempio di seguito mostra di seguito l'utilizzo di tali metodi per la visualizzazione delle colonne di una tabella.

Algorithm 12 Gestione Metadati

```
Statement st=conn.createStatement();
ResultSet rs=st.executeQuery(”select * from STUDENTI”);)
ResultSetMetadata rsmd = rs.getMetadata();
int cols=rsmd.getColumnCount();
for(int i=1;i¡=cols;i++) System.out.println(rsmd.getColumnName(i));
```

Per avere invece informazioni sull'intero database JDBC mette a disposizione la classe DatabaseMetadata ed i suoi metodi (si rimanada a manuali specialistici per l'utilizzo di questa funzione).

8 Cenni alla gestione delle transazioni in JDBC

Il discusso oggetto Connection in JDBC consente anche la gestione delle transazioni. Il comportamento di default previsto è l'**autocommit**, che effettuta il commit ad ogni transazione inviata. Risulta però possibile disabilitare l'autocommit attraverso il metodo:

```
conn.setAutoCommit(true/false)
```

Impostato a `false` l'autocommit, una transazione inizia con la prima istruzione inviata al database e viene terminata se e solo si invoca sull'oggetto Connection o un commit() che serve a concludere una transazione salvando le modifiche, o un rollback() che, invece, termina la transaazione annullando le modifiche.

Altri metodi interessanti sull'oggetto Connection, sempre riguardanti le transazioni, sono quelli per la gestione della concorrenza: getTransactionIsolation(), setTransactionIsolation. Il primo legge il livello di isolamento garantito per le transazioni in concorrenza (lettura/scrittura) sugli stessi dati, mentre il secondo lo imposta per l'applicazione corrente.

I mteodi di isolamento lavorano sul tipo di lock *sola lettura* o *scrittura* che richiede una transazione sui dati, cercando di garantire una sorta di "serializzazione" nell'accesso alle risorse comuni per mantenere a diversi livelli l'integrità e consistenza dei dati.

I 4 livelli isolamento previsti dallo standard ANSI SQL 99 e supportati in JDBC sono:

—*uncommitted read (TRANSACTION_READ_UNCOMMITTED)*: questo livello di isolamento permette alle transazioni di leggere, ma non scrivere, i dati che sono in uso da altre transazioni (il lock in lettura non è esclusivo e non va in conflitto con quello in scrittura, quello in scrittura si), e quindi dati che stanno per essere modificati e potrebbero essere non integri.

—*committed read(TRANSACTION_READ_COMITTED)*: questo livello di isolamento permette alle transazioni di leggere solo i dati che non sono in uso da altre transazioni, altrimento queste si bloccano (solo due transazioni che richiedono un lock in lettura sono eseguite concorrentemente).

—*repeatable read(TRANSACTION_REPEATABLE_READ)*: questo livello di isolamento non solo permette alle transazioni di leggere solo dati integri, ma anche, che se all'interno della stessa transazione questi saranno riletti, si riotterrano gli stessi risultati (cosa che non accadeva nel precedente caso).

—*serializable (TRANSACTION_SERIALIZABLE)*: coincide con il classico **Two-Phase Locking**, ovvero le transazioni in concorrenza su risorse comuni vengono oppurtanamente serializzate (solo due transazioni che richiedono un lock in lettura sono eseguite concorrentemente e transazioni che hanno già rilasciato dei lock, non ne possono acquisirne degli altri) garantendo la piena consistenza dei dati ed evitando qualsiasi forma di anomalia quali: la lettura sporca (presente nel primo caso), la lettura non-ripetuta (presente nel primo e secondo caso) e la lettura fantasma di dati che prima non comparivano e poi compaiono(presente nel primo caso, secondo e terzo caso).

E' chiaro che la scelta di un alto livello di isolamento (es. gli ultimi 2) comporti una diminuzione del grado di parallelismo del database, non permettendo al serve di eseguire le transazioniin maniaera concorrente. La scelta finale è chiaramente dipendente dalla tipologia di applicazione che si deve implementare.

9 Nuove funzionalità di JDBC: i batch update

JDBC nella sua ultima versione supporta una nuova funzionalità nota col nome di *batch update*. Questa nuova possibilità permette di spedire una serie di aggiornamenti al database in blocco (batch) piuttosto che uno dietro l'altro, dando la possibilità al DBMS di effettuare delle opportune ottimizzazioni sulla sequenza di modifiche da apportare.

A tale proposito alle classi Statement, PreparedStatement, CallableStatement sono stati aggiunti dei metodi aggiuntivi (addBatch(), clearBatch(), executeBatch()) per supportare tale tipologie di aggiornamenti , mentre è stata introdotta una nuova eccezione BatcUpdateException, lanciata nel caso di anomalie nel processo di batch update.

L'esempio di seguito mostra una modalità d'uso dei metodi introdotti per la gestione dei batch update. In particolare viene mostrato come gestire in un unico batch una sequenza di inserimenti sulla base di dati.

Algorithm 13 Batch update

```
conn.setAutoCommit(False);
Statement stmt = conn.createStstement();
stmt.addBatch("INSERT INTO ESAME VALUES ('041001','Basi di Dati)");
stmt.addBatch("INSERT INTO STUDENTI (Matricola, Nome, Cognome) VALUES
('0412059','Vincenzo','Moscato')");
stmt.addBatch("INSERT INTO CURRICULA VALUES ('0412059','041001')");
int [] udpateCounts=stmt.executeBatch();
conn.commit;
```

In questi caso è stata disabilitata la modalità di *autocommit* per potere considerare gli inserimenti costituenti il batch un'unica transazione.