Azzolini Riccardo 2019-11-29

Sviluppo di applicazioni per basi di dati

1 Applicazioni e SQL

Per risolvere problemi reali, non è quasi mai sufficiente eseguire singole istruzioni SQL. In pratica, tali istruzioni vengono invece integrate all'interno di un'**applicazione**, necessaria per:

- acquisire e gestire i dati forniti in ingresso;
- gestire la logica applicativa;
- restituire i dati all'utente in formati diversi, spesso non relazionali;
- visualizzare, in maniera anche complessa (grafici, report), le informazioni.

2 Limiti di SQL

Essendo progettato come strumento per la gestione di una base di dati, SQL ha un potere espressivo limitato rispetto a un generico linguaggio di programmazione, quindi non è in grado di esprimere tutte le elaborazioni che può essere necessario applicare ai dati. In particolare, SQL non è:

- *computazionalmente completo*: non permette di esprimere tutte le computazioni teoricamente possibili (ad esempio, non ha i costrutti tipici dei generici linguaggi di programmazione imperativi, quali la selezione e l'iterazione);
- *operazionalmente completo*: non fornisce la possibilità di svolgere alcune operazioni che richiedono l'interazione con il sistema operativo (e, attraverso di esso, con l'hardware), quali, ad esempio, la lettura/scrittura di file e la realizzazione di un'interfaccia utente.

3 Integrazione con un linguaggio di programmazione

Per estendere il potere espressivo di SQL, è possibile combinarlo con un linguaggio di programmazione generico. In questo modo:

- SQL consente l'accesso ottimizzato ai dati;

- il linguaggio di programmazione garantisce la completezza computazionale e operazionale.

Esistono principalmente due approcci all'integrazione:

- integrare un linguaggio di programmazione esistente con SQL (approccio **client-side**);
- **estensioni procedurali** di SQL (approccio **server-side**).

3.1 Estensione di un linguaggio esistente

Con quest'approccio, il codice viene eseguito in un ambiente esterno al DBMS (client-side), sfruttando un linguaggio di programmazione esistente, che prende il nome di **linguaggio ospite**.

L'integrazione tra linguaggio ospite e SQL può avvenire, principalmente, in due modi diversi:

- **Librerie di funzioni**: viene resa disponibile per il linguaggio una libreria di funzioni (API) che definisce l'interfaccia di comunicazione tra il DBMS e il linguaggio ospite, permettendo la connessione alla base di dati e l'esecuzione di comandi SQL.
- **SQL ospitato** (**embedded SQL**): i comandi SQL vengono utilizzati direttamente all'interno di una versione estesa del linguaggio ospite. Poi, un apposito pre-compilatore traduce il programma scritto in questo linguaggio esteso in uno scritto interamente nel linguaggio ospite "normale", sostituendo i comandi SQL con chiamate al DBMS effettuate tramite una libreria di funzioni.

3.2 Estensioni procedurali di SQL

Quest'approccio consiste nell'aggiungere a SQL i costrutti dei linguaggi di programmazione procedurali. Questo linguaggio SQL esteso può essere utilizzato all'interno del DBMS (server-side), per definire funzioni/procedure richiamabili da qualsiasi applicazione.

Lo standard SQL propone l'estensione procedurale SQL/PSM, che però, in pratica, è poco adottata. Invece, ciascun DBMS fornisce una diversa estensione procedurale. Ad esempio:

- Oracle: PL/SQL;
- PostgreSQL: PL/pgSQL;
- Microsoft SQL Server: T(ransact)-SQL;
- DB2: SQL PL.

3.3 Confronto tra i due approcci

L'approccio client-side è caratterizzato da:

- maggiore indipendenza dal DBMS utilizzato;
- minore efficienza, perché la comunicazione tra l'applicazione e il DBMS, necessaria per l'esecuzione di ogni singola operazione, ha un costo.

Esso funziona bene se le operazioni da effettuare sul DBMS sono prevalentemente semplici, ed è la scelta ovvia quando applicazioni esistenti devono essere estese per interagire con una base di dati.

Invece, con l'approccio server-side si hanno:

- dipendenza dal DBMS utilizzato, poiché le estensioni procedurali non sono standard;
- maggiore efficienza, perché è possibile eseguire interazioni più complesse e "dinamiche" con il DBMS direttamente da SQL, evitando di dover comunicare con l'applicazione per ogni operazione;
- minore espressività, perché le estensioni procedurali di SQL non comprendono tutti i costrutti disponibili in un linguaggio generico.

Quest'approccio viene tipicamente adottato da nuove applicazioni che richiedono una forte interazione con il DBMS.

Per sfruttare i vantaggi di entrambi gli approcci, spesso si utilizza una combinazione dei due:

- con le estensioni procedurali di SQL, di definiscono procedure/funzioni relative alle operazioni più frequenti sulla base di dati;
- l'applicazione, sviluppata in un linguaggio tradizionale (esteso con una CLI o con l'SQL ospitato), richiama tali procedure, ed effettua le operazioni che non è possibile/conveniente eseguire a lato server.

4 Impedance mismatch

L'integrazione tra SQL e i linguaggi di programmazione presenta alcune difficoltà, legate alle modalità di rappresentazione ed elaborazione dei dati, che prendono il nome di **impedance mismatch**:

- *differenze nei tipi di dato*: è necessario determinare una corrispondenza tra i tipi di SQL e quelli del linguaggio (ad esempio, `VARCHAR` potrebbe corrispondere al tipo `String` di Java);

- *differenze nelle modalità di elaborazione*, che è *set-oriented* per SQL (che lavora sulle relazioni, definite come insiemi di tuple "semplici"), e *tuple-oriented* per i linguaggi di programmazione (i quali operano su singoli oggetti, caratterizzati da una struttura più complessa).

5 Flusso di esecuzione

Qualunque sia l'approccio scelto, il programma deve eseguire i seguenti passi fondamentali:

1. *connessione alla base di dati*;
2. *esecuzione dei comandi SQL*;
3. *chiusura della connessione*.

In particolare, l'esecuzione di ciascun comando SQL si suddivide ulteriormente in tre passi:

1. *preparazione del comando*: generazione delle strutture dati necessarie per la comunicazione con il DBMS, ed eventuale compilazione e ottimizzazione del comando da parte del DBMS;
2. *esecuzione del comando*: il comando viene eseguito, utilizzando le strutture dati e le informazioni generate nella fase precedente;
3. *manipolazione del risultato*: il risultato del comando, tradotto nelle strutture del linguaggio di programmazione, viene manipolato secondo la logica dell'applicazione.

Il tipo di risultato varia in base al comando eseguito:

- per le interrogazioni, è un insieme di tuple;
- per `INSERT`, `UPDATE`, `DELETE`, e i comandi del `DDL`, è un singolo valore numerico (che indica, ad esempio, in numero di tuple modificate).

Quando un'interrogazione restituisce una sola tupla, conoscendo la struttura della tabella, il numero di informazioni da analizzare è noto a priori, quindi è possibile definire delle variabili da comunicazione in cui ospitare i valori degli attributi di tale tupla. Se, invece, l'interrogazione restituisce più tuple, il risultato ha una dimensione non necessariamente conosciuta in anticipo, e potrebbe anche essere troppo grande per gestirlo nella memoria principale. È allora necessario un meccanismo che permetta di manipolare, una a una, le tuple del risultato.

5.1 Cursori

Un **cursore** è un puntatore a una tupla contenuta nel risultato di un'interrogazione SQL, che permette di accedere a tale tupla per leggere i valori degli attributi.

Un cursore è associato alla valutazione di un'interrogazione (cioè viene creato apposta per eseguire una determinata interrogazione e gestirne il risultato).

Le operazioni che si effettuano sui cursori sono:

- *definizione*: associa un cursore a un'interrogazione;
- *apertura*: esegue l'interrogazione associata al cursore e lo inizializza;
- *posizionamento*: sposta il cursore sulla tupla successiva o precedente a quella attualmente puntata, oppure sulla prima o ultima tupla del risultato;
- *chiusura*: disabilita il cursore e rilascia le risorse necessarie per la gestione del risultato.

6 SQL statico e dinamico

Si parla di **SQL statico** quando le istruzioni SQL da eseguire sono già note durante la scrittura dell'applicazione. Tali istruzioni possono comunque contenere delle variabili, i cui valori possono essere specificati (e sono quindi noti solo) in fase di esecuzione.

Invece, si definisce **SQL dinamico** l'uso di comandi SQL noti solo a tempo di esecuzione, che dipendono dal flusso di esecuzione dell'applicazione, e possono anche essere dati in input dall'utente. Ad esempio, la condizione di una clausola `WHERE` potrebbe variare in base ad alcune opzioni di ricerca, specificate dall'utente.

7 Librerie di funzioni

Un'applicazione può accedere a una base di dati tramite un'interfaccia chiamata **Call Level Interface** (**CLI**), che è una libreria per un determinato linguaggio di programmazione.

La maggior parte dei DBMS offrono delle CLI proprietarie, che però, in quanto tali, non sono compatibili con DBMS diversi. Invece, al fine di garantire l'interoperabilità, esistono anche delle CLI standard, indipendenti dal DBMS: SQL/CLI, prevista dallo standard SQL, ma poco adottata, e gli standard *de facto* **ODBC** e **JDBC**.

Quando si usa una CLI, i comandi SQL sono inviati al DBMS passandoli come argomenti ad apposite funzioni del linguaggio ospite, messe a disposizione dalla libreria.

8 JDBC

JDBC è una CLI standard per interagire con le basi di dati da Java, in modo indipendente dal DBMS. Per ottenere tale indipendenza:

- l'applicazione lavora con un **driver manager**;
- il driver manager comunica con un **driver**, che traduce le chiamate JDBC in chiamate alla CLI proprietaria di uno specifico DBMS.

In questo modo, caricando driver diversi, è possibile interagire con DBMS diversi, senza bisogno di alterare l'applicazione. Inoltre, siccome la comunicazione avviene sempre attraverso le CLI proprietarie, non è necessario che il DBMS supporti direttamente l'interfaccia JDBC.

8.1 Driver manager

Il driver manager è una classe che gestisce la comunicazione tra applicazione e driver, risolvendo alcune problematiche comuni a tutte le applicazioni:

- quale driver caricare (in base alle informazioni, fornite dall'applicazione, relative alla base di dati a essa si vuole connettere);
- caricamento del driver;
- chiamate alle funzioni dei driver.

L'applicazione interagisce solo con il driver manager, e non direttamente con il driver.

8.2 Flusso delle operazioni

Il flusso tipico di un'applicazione che accede a una base di dati mediante JDBC è:

1. caricamento del driver;
2. selezione della sorgente dei dati (una specifica istanza del DBMS, e uno specifico database tra quelli gestiti da tale istanza);
3. invio di comandi SQL;
4. recupero ed elaborazione dei risultati;
5. disconnessione.

Le principali classi di JDBC, rilevanti per queste operazioni, sono:

- `DriverManager`;
- `Connection`;

- `Statement`, con la sotto-classe `PreparedStatement`, che ha a sua volta la sotto-classe `CallableStatement`;
- `ResultSet`;
- `SQLException`.

8.3 Caricamento del driver

Il primo passo di un'applicazione JDBC consiste nel caricare il driver che si intende utilizzare.

Ogni driver corrisponde a una classe Java, che può essere caricata dinamicamente con una chiamata `Class.forName`.[1] Il nome della classe da usare viene indicato nella documentazione relativa al driver.

Ad esempio, per caricare il driver per il DBMS Oracle, si usa la chiamata:

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

8.4 Connessione

Il DBMS a cui connettersi viene specificato mediante una *stringa di locazione*, che ha il formato

```
jdbc:<subprotocol>:<subname>
```

dove:

- `<subprotocol>` identifica il driver da utilizzare;
- `<subname>` identifica la specifica base di dati a cui ci si vuole connettere.

Per creare la connessione, la stringa di locazione viene passata come argomento al metodo `DriverManager.getConnection`, che restituisce un oggetto di classe `Connection`. Ad esempio:

```
Connection con = DriverManager.getConnection(
    "jdbc:oracle:www.bookstore.com:3083",
    userID, password
);
```

Quando non è più necessaria, una connessione può essere chiusa tramite il metodo `close` della classe `Connection`.

[1] Con le versioni recenti di JDBC, il driver può essere caricato automaticamente al momento della connessione al DBMS, quindi quest'operazione non è necessaria.

8.5 Tipi di dato

JDBC definisce un mapping da un insieme di tipi SQL ai tipi Java. Il mapping dei principali tipi è:

Tipo JDBC	Tipo Java
`CHAR`	`String`
`VARCHAR`	`String`
`LONGVARCHAR`	`String`
`NUMERIC`	`java.math.BigDecimal`
`DECIMAL`	`java.math.BigDecimal`
`BIT`	`boolean`
`TINYINT`	`byte`
`SMALLINT`	`short`
`INTEGER`	`int`
`BIGINT`	`long`
`REAL`	`float`
`FLOAT`	`double`
`DOUBLE`	`double`
`BINARY`	`byte[]`
`VARBINARY`	`byte[]`
`LONGVARBINARY`	`byte[]`
`DATE`	`java.sql.Date`
`TIME`	`java.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp`

8.6 Esecuzione di comandi SQL

Un comando SQL viene eseguito mediante la classe `Statement`, oppure una delle sue sotto-classi, a seconda che il comando sia:

- generico (per SQL statico e dinamico);
- *preparato* (solo per SQL statico);
- *callable statement* (invocazione di *stored procedure*, cioè procedure server-side).

Ogni oggetto `Statement` è associato a una connessione, e viene creato tramite un apposito metodo della classe `Connection`.

8.6.1 Comandi generici

Per l'esecuzione di un comando generico, si usa un oggetto di classe `Statement`, che viene creato dal metodo `createStatement` della classe `Connection`:

```
Connection con;
// ...
Statement stmt = con.createStatement();
```

Una volta creato, l'oggetto `Statement` non è ancora associato all'istruzione SQL da eseguire, che viene invece specificata al momento dell'esecuzione (e, per questo, può essere costruita in modo dinamico).

Esistono diversi metodi per l'esecuzione, a seconda del tipo di risultato restituito dal comando:

- `executeQuery`, per l'esecuzione di una query (che restituisce una relazione);
- `executeUpdate`, per l'esecuzione di un'operazione di aggiornamento (che restituisce un valore numerico), compresi i comandi del DDL;
- `execute`, per eseguire un comando la cui tipologia non è nota.

Tutti questi tre metodi di `Statement` ricevono, come argomento, una stringa contenente il comando SQL.

Alcuni esempi sono:

```
stmt.executeUpdate(
    "CREATE TABLE Film (" +
    "    titolo VARCHAR(30)," +
    "    regista VARCHAR(20)," +
    "    anno DECIMAL(4) NOT NULL," +
    "    genere CHAR(15) NOT NULL," +
    "    valutaz NUMERIC(3, 2)," +
    "    PRIMARY KEY (titolo, regista)" +
    ")"
);

stmt.executeUpdate(
    "INSERT INTO Film VALUES" +
    "('la tigre e la neve', 'roberto benigni', 2005, 'commedia', 3.25)"
);

stmt.executeQuery("SELECT * FROM Film");
```

8.6.2 Comandi preparati

Un comando SQL "preparato" viene compilato una sola volta (ad esempio, all'inizio dell'esecuzione dell'applicazione), e poi può essere eseguito più volte. Questo tipo di comando contiene solitamente dei parametri, i cui valori devono essere specificati prima di ogni esecuzione.

I comandi preparati sono utili quando è necessario eseguire ripetutamente una stessa istruzione SQL: compilandola una volta sola, si riduce il tempo di esecuzione. In compenso, essi non sono adatti per il SQL dinamico (perché, se le istruzioni da eseguire cambiano ogni volta, eseguire "in anticipo" la compilazione non dà alcun vantaggio).

Per creare un comando preparato, che è un'istanza della classe `PreparedStatement`, si usa il metodo `prepareStatement` di `Connection`. Esso richiede, come argomento, la stringa corrispondente al comando SQL che dovrà poi essere eseguito.

`PreparedStatement` mette a disposizione metodi diversi per l'esecuzione, analoghi a quelli di `Statement` (`executeQuery`, `executeUpdate` ed `execute`), ma con la differenza che essi non hanno argomenti, in quanto il comando è già stato specificato in fase di creazione.

La stringa passata a `prepareStatement` può contenere dei parametri, indicati dal carattere `?`. In tal caso, prima di eseguire il comando, bisogna associare a esso i valori dei parametri, usando vari metodi `set`, corrispondenti ai diversi tipi Java:

- `setString`,
- `setInt`,
- `setBigDecimal`,
- ecc.

Ognuno di questi metodi ha due argomenti: l'indice del parametro,[2] e il valore da assegnare a esso.

Un esempio di esecuzione di una query con un parametro è:

```
PreparedStatement query = con.prepareStatement(
    "SELECT * FROM Film WHERE genere = ?"
);
query.setString(1, "thriller");
query.executeQuery();
```

[2] I parametri sono numerati a partire da 1, secondo l'ordine in cui compaiono nella stringa del comando SQL.

8.7 Cursori

I risultati delle query vengono restituiti (dai metodi `executeQuery`) mediante oggetti di classe `ResultSet`, che sono cursori:

- Per passare alla tupla successiva, si usa il metodo `next()`: esso restituisce `true` se tale tupla esiste, o `false` quando non ci sono più tuple da analizzare. Inizialmente, il cursore è posizionato *prima* della prima tupla, quindi è necessario chiamare subito `next()` per iniziare ad accedere ai dati.
- I valori che gli attributi assumono nella tupla corrente possono essere recuperati attraverso vari metodi `get`, corrispondenti ai tipi Java (`getInt`, `getString`, ecc.). Per specificare l'attributo da considerare, questi metodi accettano un argomento, che può essere:
  - una stringa, contenente il nome dell'attributo;
  - l'indice dell'attributo nella relazione restituita dalla query (notazione posizionale).

8.7.1 Esempio

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT titolo, regista" +
    "FROM Film" +
    "WHERE genere = 'thriller'"
);

while (rs.next()) { // Iterazione di tutte le tuple
    System.out.println(rs.getString("titolo"));
    System.out.println(rs.getString("regista"));
}
```

8.8 Disconnessione

Quando gli oggetti di classe `Connection`, `Statement`, e `ResultSet` non vengono più utilizzati, è bene chiuderli, mediante i loro metodi `close()`, per rilasciare le risorse associate a essi.

La chiusura di un oggetto `Connection` chiude gli `Statement` associati a esso, e, a sua volta, la chiusura di uno `Statement` chiude i `ResultSet` associati.

8.9 Eccezioni

Quando si verifica un errore nell'interazione con il DBMS, JDBC solleva un'eccezione di classe `java.sql.SQLException`, che contiene varie informazioni relative all'errore, tra cui:

- il codice di errore restituito dal DBMS (DBMS diversi restituiscono codici diversi), che può essere ottenuto con il metodo `getErrorCode()`;
- una descrizione dell'errore, a cui si accede con il metodo `getMessage()`.

9 SQL ospitato

L'SQL ospitato permette l'inserimento dei comandi SQL direttamente all'interno del codice scritto nel linguaggio ospite, senza bisogno di usare funzioni di interfaccia.

Il programma scritto in questo modo viene poi passato a un apposito pre-compilatore (specifico per la combinazione di linguaggio, DBMS, e sistema operativo utilizzati), che sostituisce i comandi SQL con delle chiamate a una libreria di funzioni. Il risultato è un programma scritto completamente nel linguaggio ospite, che può quindi essere compilato con un normale compilatore per tale linguaggio, e infine eseguito.

Per delimitare le istruzioni SQL all'interno del testo del programma, ciascuna di esse è preceduta da un prefisso (spesso `EXEC SQL`) e seguita da un terminatore (solitamente il carattere `;`). Oltre ai comandi SQL veri e propri, si delimitano in questo modo anche altre porzioni di interesse del precompilatore, come ad esempio le dichiarazioni dei dati e la gestione della connessione al DBMS.

All'interno delle istruzioni SQL, dove sintatticamente sono ammesse delle costanti, è possibile usare come parametri le variabili del programma (tipicamente precedute da `:`).

9.1 Esempio

```
#include <stdlib.h>

int main(void) {
    EXEC SQL BEGIN DECLARE SECTION;
    char *nome_dip = "Manutenzione";
    char *citta_dip = "Pisa";
    int numero_dip = 20;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO utente@librobd;
```

```
    if (sqlca.sqlcode != 0) {
        printf("Connessione al DB non riuscita\n");
    } else {
        EXEC SQL
            INSERT INTO Dipartimento
            VALUES (:nome_dip, :citta_dip, :numero_dip);
        EXEC SQL DISCONNECT ALL;
    }
}
```

Nota: `sqlca` è una struttura dati per la comunicazione tra il programma e il DBMS. Il suo campo `sqlcode` contiene il codice di errore dell'ultimo comando SQL eseguito: zero indica il successo, mentre un valore diverso indica un errore/anomalia.

10 Estensioni procedurali di SQL

Le estensioni procedurali di SQL permettono la creazione delle **stored procedure** che sono funzioni/procedure definite all'interno del DBMS:

- fanno parte dello schema logico della base di dati (insieme a relazioni, viste, ecc.);
- possono avere dei parametri di esecuzione, i cui valori vengono specificati quando queste procedure vengono chiamate;
- contengono codice applicativo e istruzioni SQL, fortemente integrati tra loro.

10.1 Esempio: PL/SQL

PL/SQL, l'estensione procedurale di SQL implementata dal DBMS Oracle, è un linguaggio con una struttura a blocchi. Ciascun blocco è composto da tre parti:

- sezione di dichiarazione delle variabili;
- sezione di esecuzione;
- sezione di gestione delle eccezioni.[3]

[3] Gestendo le eccezioni direttamente nel DBMS (se possibile), si ha il vantaggio che l'applicazione non se ne deve occupare.

10.1.1 Creazione di una stored procedure

In Oracle, il comando per la creazione di una stored procedure è:

```
CREATE [OR REPLACE] PROCEDURE <nome> [(<elenco parametri>)]
IS {<istruzione SQL> | <blocco PL/SQL>};
```

La stored procedure può essere composta da una singola istruzione SQL, oppure da un blocco di codice PL/SQL.

10.1.2 Variabili

Le variabili, dichiarbili nell'apposita sezione di ciascun blocco (che è contrassegnata dalla parola chiave `DECLARE`), permettono l'uso dei risultati di un comando SQL per altri comandi. Esse possono avere un qualunque tipo SQL, e, in particolare, è possibile specificare il tipo di una variabile in base a quello di un'altra variabile, o anche in base agli attributi di una relazione:

- Con la sintassi `%TYPE`, si dichiara una variabile che ha lo stesso tipo di un'altra variabile o di un attributo; ad esempio:

  ```
  credito NUMERIC(7, 2);
  debito credito%TYPE;
  -- "debito" ha lo stesso tipo di "credito", cioè NUMERIC(7, 2)

  imp impiegati.nome%TYPE;
  -- "imp" ha lo stesso tipo dell'attributo "nome" della tabella
  -- "impiegati"
  ```

- La sintassi `%ROWTYPE` permette di creare una variabile che rappresenta un record, corrispondente a un'intera tupla di una tabella:

  ```
  dip_rec dipartimenti%ROWTYPE;
  -- "dip_rec" è un record, i cui elementi hanno gli stessi nomi e
  -- tipi degli attributi della tabella "dipartimenti"
  ```

10.1.3 Strutture di controllo

PL/SQL mette a disposizione diverse strutture di controllo:

- selezione (`IF THEN`, `IF THEN ELSE`, `IF THEN ELSIF`);
- vari tipi di cicli (`LOOP`, `WHILE LOOP`, `FOR LOOP`);
- `GOTO`.

10.1.4 Cursori

Per le interrogazioni che restituiscono più di una tupla, PL/SQL consente la dichiarazione di cursori, che (analogamente ai `PreparedStatement` di JDBC) possono anche essere parametrizzati.

Le principali operazioni sui cursori sono:

- dichiarazione:

  ```
  CURSOR <nome cursore> [(<lista parametri>)]
  IS <query>;
  ```

- apertura:

  ```
  OPEN <nome cursore>;
  ```

- avanzamento:

  ```
  FETCH <nome cursore> INTO {<nome record> | <lista variabili>};
  ```

  (con quest'operazione, il cursore viene associato a una variabile booleana `NOTFOUND`, che diventa vera quando non ci sono più tuple da analizzare);

- chiusura:

  ```
  CLOSE <nome cursore>;
  ```

10.1.5 Esempi

- Esecuzione di un'interrogazione che restituisce una singola tupla, e salvataggio dei valori di alcuni attributi in delle variabili:

  ```
  DECLARE
    nome_i impiegati.nome%TYPE;
    stipendio_i impiegati.stipendio%TYPE;
    -- ...
  BEGIN
    -- ...
    SELECT nome, stipendio
    INTO nome_i, stipendio_i
    FROM impiegati
    WHERE nome = 'Rossi';
    -- ...
  END;
  ```

  In alternativa alle variabili separate `nome_i` e `stipendio_i`, si potrebbe salvare l'intera tupla in una variabile record.

- Impiego di una struttura di controllo (selezione), e uso delle variabili in un comando `INSERT`:

```
BEGIN
  IF stipendio > 50000 THEN
    bonus := 1500;
  ELSIF stipendio > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  INSERT INTO stipendi VALUES (id_imp, bonus /* , ... */);
END;
```

- Uso di un cursore, con salvataggio di ciascuna tupla in una variabile record:

```
DECLARE
  imp_rec impiegati%ROWTYPE;
  mio_dip impiegati.dipartimento%TYPE;
  CURSOR c1 IS
    SELECT *
    FROM impiegati
    WHERE dipartimento = mio_dip;
BEGIN
  -- ...
  OPEN c1;
  LOOP
    FETCH c1 INTO imp_rec;
    EXIT WHEN c1%NOTFOUND;
    -- ...
  END LOOP;
  CLOSE c1;
END;
```

11 Conclusioni

Le diverse tecniche per l'integrazione dell'SQL nelle applicazioni hanno caratteristiche diverse. Non esiste un approccio sempre migliore degli altri: la scelta dipende dal tipo di applicazione da realizzare, dalle caratteristiche delle basi di dati, ecc. (ed è anche possibile utilizzare soluzioni miste, per combinare i vantaggi di più approcci).

11.1 JDBC

- *Vantaggi*:
  - portabilità;
  - scrittura di tutto il codice procedurale nello stesso linguaggio (Java).
- *Svantaggi*:
  - performance;
  - minore facilità di programmazione rispetto alle altre soluzioni.

11.2 Embedded SQL

- *Vantaggi*:
  - portabilità (anche se è necessario usare precompilatori diversi per piattaforme diverse);
  - facilità di programmazione rispetto all'utilizzo di API.
- *Svantaggi*:
  - performance;
  - minore facilità di programmazione rispetto alle estensioni procedurali di SQL.

11.3 Estensioni procedurali di SQL

- *Vantaggi*:
  - facilità di programmazione (soprattutto per quanto riguarda la gestione dei dati);
  - performance;
  - gestione ottimizzata delle eccezioni del DBMS;
  - strumenti a supporto della programmazione fortini dal DBMS.
- *Svantaggi*:
  - espressività;
  - portabilità.