BD 2 - Database-Backed Applications

Luca Cosmo

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Introduzione

Molte applicazioni hanno bisogno di interfacciarsi con un database: in questa lezione studieremo il design space delle possibilità esplorate fino ad oggi e ne discuteremo pro e contro.

Ad alto livello, possiamo categorizzare gli approcci principali in tre filoni:

1. Linguaggi integrati: linguaggi che estendono SQL con tradizionali costrutti di programmazione general purpose (es. PL/pgSQL)
2. Linguaggi che ospitano SQL: linguaggi tradizionali, la cui sintassi viene estesa con costrutti SQL inframmezzati al codice
3. Utilizzo di API: linguaggi tradizionali, che si appoggiano a librerie di interfacciamento con SQL

Linguaggi Integrati

Idea: estendere SQL con costrutti di programmazione tradizionali.

```
CREATE FUNCTION cheapest() RETURNS integer AS $$
DECLARE r pc;
BEGIN
SELECT * INTO r FROM pc ORDER BY price;
RETURN r.price;
END; $$ LANGUAGE plpgsql;
```

Vantaggi:

- Stesso livello di astrazione di SQL
- Supporto per controlli statici da parte del type system

Linguaggi Integrati

Svantaggi:

- Costo elevato di apprendimento: nuovo linguaggio
- Tecnologie proprietarie e non standardizzate
- Non adatti allo sviluppo di applicazioni complesse

Pensiamo alla nostra esperienza con PL/pgSQL:

- Sintassi e semantica non standard (es. parametri di output)
- Siamo vincolati all'utilizzo di Postgres come DBMS
- Paradigma di programmazione procedurale e basato su side-effect

Linguaggi che Ospitano SQL

Idea: estendere un linguaggio tradizionale con costrutti SQL.

Vantaggi:

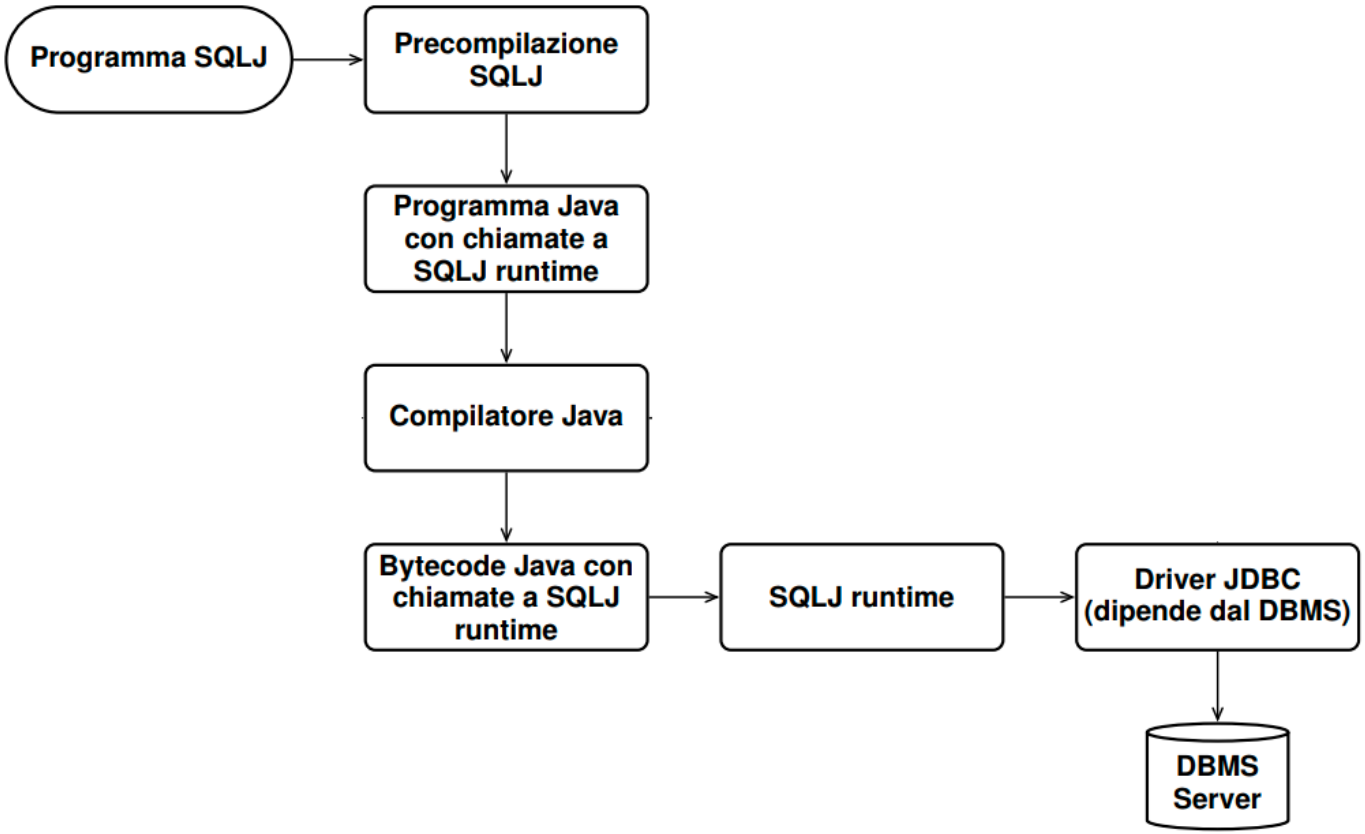
- Costo ridotto di apprendimento
- La sintassi SQL integrata abilita controlli statici (es. typing)

Svantaggi:

- Impedance mismatch: differenza di tipi fra linguaggio ospite e SQL
- E' richiesto un opportuno pre-processore

Esempio: SQLJ (Java + SQL, ormai deprecato)

SQLJ: Processo di Compilazione



SQLJ: Connessioni

Una connessione definisce il contesto di interazione col DBMS, indicando il database acceduto e l'utente che vi si interfaccia (autenticazione).

```
Class.forName(DatabaseDriver); // carica il driver
#sql context MyContext;
MyContext contesto = new MyContext(urlDb, user, pwd);
#sql [contesto] INSERT INTO Province
                              VALUES ('Venezia', 'VE', 44);
```

La scelta del driver ed il formato dell'URL della base di dati dipendono dallo specifico DBMS utilizzato.

SQLJ: Typing di Connessioni

In fase di compilazione è possibile specificare le credenziali di accesso per le varie connessioni, puntandole ad appropriati database di esempio:

```
-user@MyContext=stefano/secret@jdbc:oracle:oci:@
```

Se questo viene fatto, SQLJ può connettersi al database di esempio e sfruttare la sua struttura in fase di compilazione del vostro sorgente.

Attenzione!

E' compito del programmatore garantire che il database di esempio ed i permessi dell'utente siano rappresentativi di quanto avverrà a runtime!

SQLJ: Typing di Connessioni

Quando viene fornito un database di esempio, il pre-processore SQLJ può accedere a varie informazioni, fra cui:

- i nomi delle tabelle
- gli attributi delle tabelle
- i tipi degli attributi

Notate che tutte queste informazioni non sono visibili a Java!

Il pre-processore può essere sfruttato per garantire che il codice Java:

- faccia riferimento solo a tabelle esistenti
- faccia riferimento solo ad attributi esistenti
- soddisfi appropriati controlli di tipo

SQLJ: Comandi SQL

Passaggio disciplinato (tipato) di valori fra codice Java e codice SQL tramite l'uso dell'operatore due punti:

```
String prov = "Venezia";
String sigla = "VE";
int numeroComuni = 44;
#sql [contesto] INSERT INTO Province
                VALUES (:prov, :sigla, :numeroComuni);
```

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

SQLJ: Cursori

Quando il risultato è un insieme di tuple, l'accesso viene effettuato per mezzo di un cursore, simile a quanto abbiamo visto per PL/pgSQL:

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

Quiz Time: SQLJ

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

Chi dei quattro attori coinvolti (pre-processore SQLJ, compilatore Java, runtime Java e DBMS) segnala ciascun errore?

1. Il programmatore scrive WERE anzichè WHERE
2. Il programmatore scrive Codice anzichè CodiceAgente
3. I tipi di Ammontare ed ammontare sono incompatibili
4. La tabella Ordini è stata precedentemente cancellata
5. Il valore di numAgente non identifica alcun agente

API di Interfacciamento a SQL

Idea: non cambiare il linguaggio, appoggiandosi a librerie esterne.

Vantaggi:

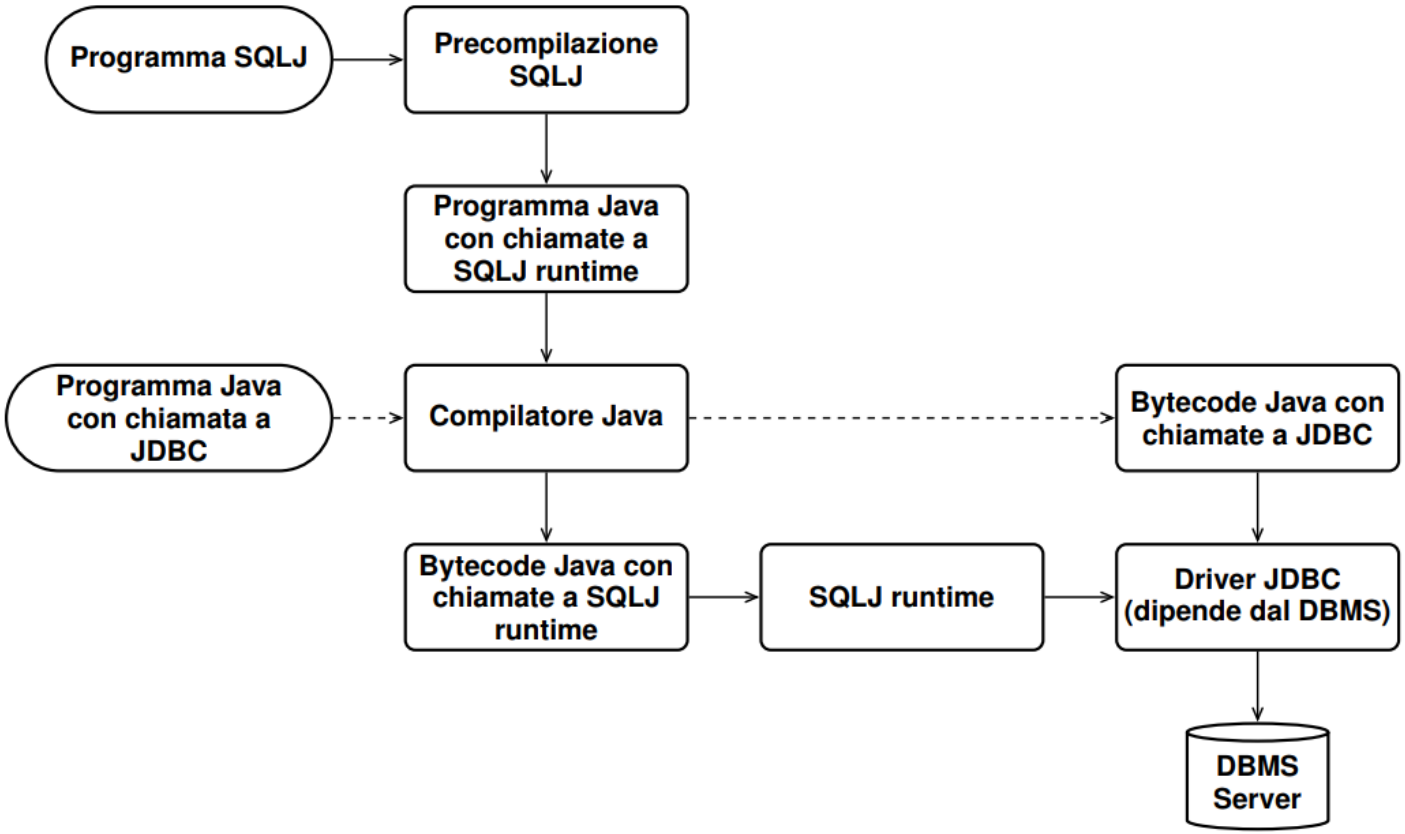
- Costo ridotto di apprendimento
- Non è richiesto alcun pre-processore
- Possibilità di utilizzo di SQL dinamico

Svantaggi:

- Impedance mismatch: differenza di tipo fra linguaggio ospite e SQL
- Assenza di controlli a tempo di compilazione

Esempio: JDBC per Java

JDBC vs SQLJ: Processo di Compilazione



JDBC: Connessioni

Come SQLJ, anche JDBC si appoggia a connessioni per interagire con il database con cui vogliamo interfacciarci:

```
String db = "jdbc:oracle:oci";
String u  = "stefano";
String p  = "secret";
Connection con = DriverManager.getConnection(db, u, p);
```

Essendo JDBC una libreria, non è possibile interfacciarla staticamente ad un database di esempio: le connessioni sono sostanzialmente non tipate!

JDBC: Comandi SQL

Le query da eseguire sono passate alla libreria come stringhe, quindi potenzialmente calcolate a runtime e non soggette ad analisi statica:

```
Statement stmt = con.createStatement();
String val = "VALUES (Venezia, VE, 44)"
String query = "INSERT INTO Province " + val;
stmt.executeQuery(query);
```

Questo può impattare negativamente sulla robustezza del codice!

JDBC: Comandi SQL

Una soluzione più robusta si basa sull'uso di prepared statement, cioè comandi SQL con "buchi" da riempire:

```
String template = "INSERT INTO Province VALUES (?,?,?)";
PreparedStatement pstmt = con.prepareStatement(template);
pstmt.setString(1, "Venezia");
pstmt.setString(2, "VE");
pstmt.setInt(3, 44);
pstmt.executeQuery();
```

Attenzione che JDBC non sa nulla riguardo alla struttura del database! Di conseguenza non avete i controlli di tipo che esegue SQLJ e vi dovete accontentare solo di garanzie parziali.

JDBC: Cursori

L'accesso a dati SQL da Java viene ancora gestita tramite cursori, ma l'accesso ai risultati può produrre errori di tipo a runtime:

```
Statement stmt = con.createStatement();
String q = "SELECT Nome,NumeroProvince FROM Province";
ResultSet res = stmt.executeQuery(q);
while (res.next()) {
    String nome = res.getString("Nome");
    int num = res.getInt("NumeroProvince");
    System.out.println(nome + " " + num);
}
res.close();
```

JDBC: SQL Dinamico

Un punto di forza dell'interfaccia JDBC è che l'uso di stringhe per la definizione di query consente di assemblare query dinamicamente, per esempio scegliendo una tabella da leggere in base all'input dell'utente:

```
Statement stmt = con.createStatement;
String query = "SELECT * FROM " + getUserInput();
ResultSet risultato = stmt.executeQuery(query);
```

SQLJ non consente di usare variabili nella clausola `FROM`, perchè questo impedirebbe il type-checking statico.

La costruzione dinamica di query va usata con moderazione, soprattutto se dipende da input dell'utente!

Quiz Time: JDBC

```
PreparedStatement pstmt = con.prepareStatement(
    "SELECT Ammontare
     FROM Ordini WHERE CodiceAgente = ?");
pstmt.setInt(1, numAgente);
ResultSet risultato = pstmt.executeQuery();
int ammontare = risultato.next().getInt(0);
```

Chi dei tre attori coinvolti (compilatore Java, runtime Java e DBMS) segnala ciascun errore?

1. Il programmatore scrive WERE anzichè WHERE
2. Il programmatore scrive Codice anzichè CodiceAgente
3. I tipi di Ammontare ed ammontare sono incompatibili
4. La tabella Ordini è stata precedentemente cancellata
5. Il valore di numAgente non identifica alcun agente

SQLJ vs JDBC

Problema	**SQLJ**	**JDBC**
WERE anzichè WHERE	Preproc	DBMS[1]
Codice anzichè CodiceAgente	Preproc	DBMS
Tipi di Ammontare ed ammontare incompatibili	Comp	Runtime
La tabella Ordini è stata cancellata	DBMS	DBMS
Il valore di numAgente non identifica un agente	Runtime	Runtime

Attenzione: quando utilizzate JDBC il compilatore ha smesso di aiutarvi!

[1] Solo perchè JDBC non fa parsing delle query!

Object Relational Mapping (ORM)

L'ORM è una tecnica di programmazione che mitiga i problemi associati all'impedance mismatch, pur mantenendo un approccio basato su API.

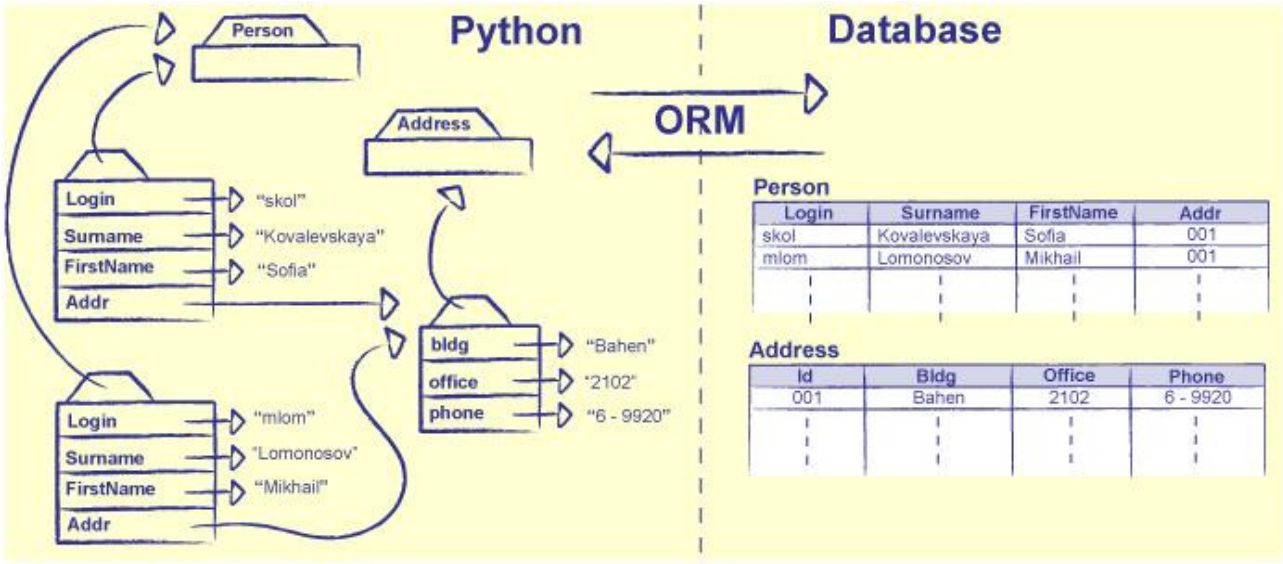
JDBC

```
Statement stmt = con.createStatement();
String query = "SELECT * FROM Students WHERE id = 10"
ResultSet rs = stmt.executeQuery(query);
String name = rs.next().getString("Name");
```

ORM

```
Student stud = db.getStudent(10);
String name = stud.getName();
```

Object Relational Mapping (ORM)



Esempio: trovare tutte le persone che hanno fatto un login da Monaco

- JDBC ritorna un `ResultSet`
- ORM ritorna una `List<Person>`

Perchè questa differenza apparentemente sottile è così importante?

Object Relational Mapping (ORM)

Vantaggi dell'uso di ORM:

- Indipendenza dallo specifico DBMS sottostante
- Non richiede conoscenza approfondita di SQL
- Migliore supporto da parte del compilatore
- Astrazione da dettagli di basso livello (es. sanitizzazione delle query)

Svantaggi dell'uso di ORM:

- Tipicamente più lento rispetto a SQL
- Poco adatto all'esecuzione di query complesse

Cosa Usare?

Al giorno d'oggi possiamo affermare che:

1. i linguaggi integrati hanno un utilizzo di nicchia molto specializzato e non sono impiegati come linguaggi general purpose
2. i linguaggi che ospitano SQL hanno un design molto interessante, ma sono sostanzialmente spariti dal mercato
3. API ed ORM dominano la scelta per la loro praticità: vedremo in seguito come utilizzare SQLAlchemy, che offre varie soluzioni

L'opzione 3 si può considerare ormai uno standard di fatto.

Checkpoint

Concetti Chiave

- Linguaggi integrati, linguaggi che ospitano SQL, API
- Pro e contro delle diverse opzioni, in particolare sui controlli statici
- ORM come soluzione moderna al problema dell'impedance mismatch

Materiale Didattico

Fondamenti di Basi di Dati: Capitolo 8, tranne la Sezione 8.4