## SQL e linguaggi di programmazione

1

## Motivazione

- SQL supporta la definizione e la manipolazione di dati relazionali
- tuttavia, SQL da solo non è sufficiente a programmare un'intera applicazione
  - non è computazionalmente completo
  - non è "system complete"

2

## Completezza computazionale

- Abilità di esprimere in un linguaggio tutte le possibili computazioni
- per garantire l'ottimizzazione delle query, il potere espressivo di SQL è limitato
- Esempio:
  flight(flight_n,dep_time,arr_time,origin,destination)
  - trovare tutte le destinazioni raggiugibili da Singapore, anche con più cambi
    - solo in SQL-99, non in SQL-92
  - calcola il cammino più corto tra due aereoporti
    - non può essere espressa neppure in SQL-99

3

## Completezza a livello di sistema

- Abilità di esprimere in un linguaggio operazioni che richiedono comunicazioni con l'hardware e le periferiche
- SQL non contiene costrutti per scrivere su file, stampare, creare un'interfaccia utente,...

4

## Accoppiamento (coupling)

- La maggior parte delle applicazioni di basi di dati richiedono programmi in cui SQL e un linguaggio di programmazione general-purpose cooperano
- SQL:
  - accesso ottimizzato ai dati
- linguaggio di programmazione:
  - garantisce computational e system completeness

5

## Modalità di accoppiamento

- **Accoppiamento interno**
  - SQL viene esteso con costrutti tipici di linguaggi di programmazione procedurali (procedure, funzioni, variabili, strutture di controllo)
  - il codice viene eseguito all'interno del DBMS
- **accoppiamento esterno**
  - un linguaggio di programmazione esistente (Java, C) viene esteso per permettere l'esecuzione di statement SQL
  - il codice viene eseguito all'esterno del DBMS

6

## Accoppiamento esterno

- **Database connectivity**
  - un insieme di primitive standard può essere utilizzato in programmi scritti utilizzando un linguaggio di programmazione noto
  - interfaccia tra il linguaggio di programmazione e il DBMS per la connessione e l'esecuzione degli statement
  - due interfacce standard: ODBC e JDBC
- **SQL ospitato (embedded SQL)**
  - comandi SQL sono ospitati direttamente in una versione estesa di un linguaggio di programmazione
  - un pre-processore o un compilatore traduce il programma in un programma scritto interamente nel linguaggio ospitante con chiamate remote al sistema di gestione dati
  - tali chiamate vengono implementate utilizzando un modello di connettività standard o proprietario

7

## Struttura di un generico programma

- Preparazione statement
- esecuzione statement
- collezione del risultato

8

## Preparazione statement

- Costruzione delle strutture dati necessarie per la comunicazione con il DBMS
- compilazione ed ottimizzazione dello statement
- due approcci alla preparazione
  - **statement statici**: statement noti a tempo di compilazione (eventualmente con parametri)
    - piano di accesso può essere generato a tempo di compilazione
    - maggiore efficienza
  - **statement dinamici**: statement noti solo a tempo di esecuzione (costruiti a run time)
    - piano di accesso generato a run-time

9

## Esecuzione statement

- Statement di tipo diverso producono diversi risultati
  - INSERT, DELETE, UPDATE, DDL statement restituiscono un valore o una struttura dati che rappresenta
    - il numero di tuple inserite/cancellate/aggiornate
    - dettagli relativi al completamento dell'operazione
  - SELECT statement restituiscono un insieme di tuple

10

## Collezione del risultato

- Se il risultato di uno statement di SELECT è "piccolo" (ad esempio una sola tupla) viene inserito in una tabella o un insieme di variabili in memoria
- se è grande (più di una tupla) o non noto, il risultato viene mantenuto nella base di dati, in una tabella temporanea, e viene utilizzato un **cursore** per caricare in memoria le tuple, una alla volta

11

## Cursore

- Un cursore è un puntatore al risultato di una query, mantenuto in un DBMS
- viene sempre associato all'esecuzione di una query
- le tuple o alcuni attributi delle tuple puntate dal cursore possono essere caricate in strutture dati locali, per poter essere manipolate
- funzioni specifiche permettono al programmatore di manipolare i cursori:
  - **dichiarazione:** un cursore viene associato ad una query
  - **apertura**: la query associata al cursore viene eseguita
  - **posizionamento:** il cursore viene spostato nell'insieme di tuple risultato
  - **chiusura:** il cursore viene chiuso

12

## Altre interazioni per approcci di accoppiamento esterno

- Connessione esplicita con il DBMS e la base di dati
  - identificazione
  - disconnessione dal DBMS
- le applicazioni basate sull'accoppiamento interno non richiedono esplicita connessione e disconnessione in quanto vengono eseguite direttamente dal DBMS

13

## Esempi successivi

- Gli esempi nel seguito si riferiscono alternativamente al solito schema Impiegati/Dipartimenti o allo schema seguente:

```
CREATE TABLE employee
 (name VARCHAR(24) PRIMARY KEY,
  address VARCHAR(36) DEFAULT ' company address'
  department VARCHAR(24) REFERENCES department(name),
  salary NUMERIC);

CREATE TABLE department
 (name VARCHAR(24) PRIMARY KEY,
  location VARCHAR(36),
  budget NUMERIC);
```

14

## SQL procedurale

15

## Introduzione

- La maggior parte dei DBMS supporta un'estensione procedurale di SQL
- i programmi sono in genere organizzati in routine (procedure o funzioni) che vengono poi
  - eseguite direttamente dal DBMS
  - chiamate da applicazioni basate su accoppiamento esterno
- non esiste uno standard per SQL procedurale
  - SQL-99 stabilisce solo come le routine SQL possono essere definite e invocate
  - Oracle: PL/SQL
  - SQL Server: T-SQL vedremo questo

16

## Manipolazione

- Non viene richiesta connessione in quanto i programmi possono solo essere eseguiti direttamente dal DBMS
- ogni programma è composto da due sezioni:
  - **sezione di dichiarazione:** contiene dichiarazioni per tutte le variabili utilizzate nel programma
  - **sezione di esecuzione:** contiene tipici costrutti procedurali (istruzioni) e statement SQL

17

## Sezione di dichiarazione

- In T-SQL è possibile dichiarare variabili per ogni tipo supportato da T-SQL
- Le variabili vengono dichiarate con la clausola DECLARE
- I nomi di variabili devono essere preceduti da @

  **DECLARE @nome_var tipo_var**
- Un comando di DECLARE può dichiarare anche più di una variabile, usando "," come separatore
- Esempio:
  - DECLARE @ImpID int, @ImpSal int

18

## Sezione di esecuzione

- Contiene costrutti procedurali e statement SQL
- la comunicazione tra i costrutti procedurali e gli statement avviene utilizzando le variabili precedentemente dichiarate
- le variabili possono essere usate:
  - clausola WHERE di statement SELECT, DELETE, UPDATE
  - clausola SET di uno statement di UPDATE
  - clausola VALUES di uno statement di INSERT
  - clausola INTO di uno statement SELECT (si veda oltre)

19

## Istruzioni - assegnazione

- Alle variabili è possibile assegnare valori con il comando SET

  SET @ImpID = 1234
- ogni comando SET può inizializzare un'unica variable

20

## Istruzioni - esempio

```
USE <nome_db>
DECLARE @find varchar(30)
SET @find = 'Ro%'
SELECT name, salary
FROM employees
WHERE name LIKE @find
```

21

## Istruzioni - esempio

```
USE <nome_db>
DECLARE @find varchar(30), @Sal integer
SET @find = 'Ro%'
SET @Stip = 1000
SELECT name, Salary
FROM employees
WHERE name LIKE @find and salary = @Stip
```

22

## Istruzioni - Strutture di controllo

- Classici costrutti imperativi per alterare il flusso sequenziale di esecuzione degli statement specificati, tra cui:
  - BEGIN END
    - stesso ruolo {} in Java e in C
  - IF ELSE
    - classico costrutto di scelta (come in Java e C)
  - WHILE
    - come in Java e C
    - Possibilità di utilizzare BREAK e CONTINUE
  - …

23

## Istruzioni - condizioni booleane

- La condizione booleana da inserire nei costrutti IF_ELSE e WHILE è una qualunque espressione che restituisce TRUE o FALSE
- l'espressione può anche coinvolgere statement SQL, che devono essere racchiusi tra parentesi

24

## Istruzioni - Esempio

```
IF (SELECT AVG(Salary) FROM Employee) < 2000
  UPDATE Employee
  SET Salary = Salary * 1.2
  SELECT Name, Salary FROM Employee
```

25

## Istruzioni di Output, commenti

- PRINT <stringa>
- la stringa può essere:
  - una stringa esplicita tra apici
  - una variabile di tipo char o varchar
  - il risultato di una funzione (di tipo char, varchar)
- Per i commenti si utilizzano i caratteri --

26

## Istruzioni - esempio

```
IF (SELECT AVG(Salary) FROM Employee) < 2000
BEGIN
   PRINT 'Gli stipendi sono troppo bassi: verranno
   aggiornati'
   -- aumento gli stipendi del 20 %
  UPDATE Employee
  SET Salary  = Salary * 1.2
  SELECT Name, Salary FROM Employee
END
```

27

## Costanti utili

- Le costanti supportate da T-SQL iniziano con @@
- Ricordiamo solo la seguente:
  - @@ERROR: restituisce il numero corrispondente all'errore generato dall'ultimo statement SQL eseguito, vale 0 se nessun errore è stato generato

28

## Istruzioni - esempio

```
DECLARE    @sales NUMERIC, @sal NUMERIC
[...]
IF @sales > 50000 THEN SET sal = 1500
ELSE
    IF @sales > 35000 THEN SET sal = 1200 ELSE SET sal = 1000
 INSERT INTO employee VALUES (' Tiziana','Dezza',' 132, via','Dellatti',
                  ' research',sal);
```

29

## Query con unico risultato

- Se una query restituisce un'unica tupla come risultato, i valori degli attributi della tupla possono essere inseriti direttamente in variabili utilizzando lo statement SELECT … INTO

```
DECLARE   my_salary NUMERIC
SELECT salary INTO my_salary WHERE name = ' Nanci
   Santi'
```

30

## Query che restituiscono più tuple - Cursori

- Da utilizzare necessariamente quando una query restituisce più di una tupla
- **Dichiarazione:** associa una variabile di tipo cursore ad uno statement di SELECT

```
DECLARE <nome cursore> CURSOR FOR  <select statement>
```

- **Apertura:** esegue la query associata al cursore, le tuple restituite possono essere viste come elementi di un array di record, il cursore prima della prima tupla

```
OPEN <nome cursore>
```

31

## Cursori

- **Avanzamento:** posiziona il cursore sulla tupla successiva

```
FETCH NEXT FROM <nome cursore> INTO <lista variabili>
```

- **Chiusura:** rende il cursore non utilizzabile (è necessario riaprirlo)

```
CLOSE <nome cursore>
```

- **Deallocazione:** il cursore non è più associato alla query specificata durante la sua dichiarazione

```
DEALLOCATE <nome cursore>
```

32

## Cursori

- @@FETCH_STATUS:
  - Variabile di sistema
  - è uguale a 0 se la tupla è stata letta
  - è < 0 se si è verificato qualche problema (ad esempio la tupla non esiste, siamo arrivati alla fine del result set)

33

## Cursori - Esempio

```
DECLARE EmpCursor CURSOR FOR  SELECT Name FROM
   Employee
OPEN EmpCursor
DECLARE @NameEmp VARCHAR(10)
FETCH NEXT FROM EmpCursor INTO @NameEmp
WHILE (@@FETCH_STATUS = 0)
   BEGIN
        PRINT @NameEmp
        FETCH NEXT FROM EmpCursor INTO @NameEmp
   END
CLOSE EmpCursor
DEALLOCATE EmpCursor
```

34

## Organizzazione codice T-SQL

- I programmi scritti in T-SQL possono essere organizzati ed eseguiti in tre modi distinti:
  - inviati all'SQL engine interattivamente (batch)
  - organizzati in procedure (stored procedure)
  - script: sequenza di statement T-SQL memorizzati in un file e quindi eseguiti, utilizzando una funzionalità particolare di SQL Server
    - possono essere eseguiti dalla shell del DOS mediante il comando: osql
      osql -U login -P password -i nome_file_input -o nome_file_risultato
    - se la password non viene specificata, viene comunque richiesta

35

## Batch

- Gruppi di statement  T-SQL inviati interattivamente all'SQL Engine ed eseguiti contemporanemante
- vengono compilati in un singolo piano di esecuzione
  - se si verifica errore di compilazione il piano non viene generato
- se si verifica un errore in esecuzione
  - gli statement seguenti non vengono eseguiti
- per specificare un batch da SQL Query Analyzer:
  - GO
- le variabili dichiarate in un batch sono locali a tale batch

36

## Batch - Esempio

```
CREATE TABLE Impiegati
(Imp# numeric(4) PRIMARY KEY,
Nome VarChar(20),
Mansione VarChar(20),
Data_A Datetime,
Stipendio Numeric(7,2),
Premio_P Numeric(7,2),
Dip# Numeric(2));
GO
SELECT Nome, Dip# FROM Impiegati
WHERE Stipendio>2000 AND
Mansione = ' ingegnere' ;
GO
```

37

## Funzioni e stored procedures

- Per SQL-99, una routine SQL consiste di
  - nome
  - insieme di dichiarazioni di parametri
  - corpo dellòa routine
- il corpo può essere scritto in molti linguaggi, incluse le estensioni procedurali di SQL

38

## Procedure

- Le stored procedure sono un particolare tipo di oggetto per la base di dati, quindi il linguaggio dovrà permettere di:
  - Crearle, tramite il DDL
  - Eseguirle, tramite comandi specifici
- In T-SQL, la creazione di una procedura deve essere l'unica operazione contenuta in un batch

39

## Procedure - Creazione ed esecuzione

```
CREATE PROCEDURE <nome>
  <parametri>
  AS
  <codice>

EXECUTE <nome>
```

40

## Procedure - Esempio

```
CREATE PROCEDURE Salaries
AS
SELECT Salary
FROM Employee

EXECUTE Salaries        oppure
EXEC Salaries
```

41

## Procedure - Parametri

- I parametri sono utilizzati per scambiare valori tra la procedura e l'applicazione o il tool che la richiama
- Tipi di parametri:
  - input
  - output
  - valore di ritorno (se non specificato si assume 0)

42

## Procedure - Parametri di input

- Per ogni parametro di input, è necessario specificare nome e tipo
- Esempio

```
CREATE PROCEDURE EmpSelect  @EmpID INT
AS
SELECT *
FROM Employee
Where Emp# = @EmpID
```

43

## Procedure -Parametri di input

- Al momento dell'esecuzione della procedura, vengono specificati i parametri attuali:
  - EXEC EmpSelect @EmpID = 1234
- nella specifica di una procedura è possibile specificare un valore di default
  - in questo caso, non sarà necessario passare un valore per il parametro

44

## Procedure - Esempio

```
CREATE PROCEDURE EmpSelect  @EmpID INT =
   1234
AS
SELECT * FROM Employee
Where Emp# = @EmpID
GO
EXEC EmpSelect
GO
EXEC EmpSelect @EmpID = 2345
```

45

## Procedure - Parametri di Output

- Per restituire valori all'ambiente chiamante, è possibile utilizzare parametri di output
- un parametro si considera di output se la sua dichiarazione è seguita dalla parola chiave OUTPUT
- nel codice T-SQL è necessario associare un valore al parametro di output
- al momento della chiamata è ovviamente necessario specificare una variabile per il parametro di output

46

## Esempio

```
CREATE PROCEDURE AvgSal @Dept VARCHAR(20), @Avg
   int OUTPUT
AS
SELECT @Avg = avg(salary)
FROM Employee
WHERE Department = @Dept;
GO

DECLARE @AVGex int
EXEC AvgSal @Dept= 1, @Avg = @AVGex OUTPUT
PRINT @AVGex
GO
```

47

## Procedure - Valori di ritorno

- L'istruzione RETURN permette di restituire un valore all'ambiente chiamante
- Per default
  - 0 indica che l'esecuzione è andata a buon fine
  - valore diverso da 0 indica che l'esecuzione ha generato errori

48

## Procedure - Esempio

```
CREATE PROCEDURE AvgSal @Dept VARCHAR(20), @Avg int OUTPUT
AS
SELECT avg(salary) FROM Employee
WHERE Department = @Dept
RETURN @@Error
GO

DECLARE @AVGex int
DECLARE @ReturnStatus INT
EXEC @ReturnStatus = AvgSal @Dept = 1, @Avg = @AVGex OUTPUT
PRINT ' ReturnStatus= '  +  CAST(@ReturnStatus AS CHAR(10))
PRINT @AVGex
GO
```

49

## Funzioni

- Corrispondono al concetto di funzioni presente nei linguaggi di programmazione
- per ogni funzione è necessario specificare:
  - parametri (solo input)
  - tipo valore di ritorno
- è sempre presente l'istruzione RETURNS che definisce il valore restituito all'ambiente chiamante
- le funzioni possoni restituire:
  - un valore (funzioni scalare)
  - una tabella (non le vediamo)
- la chiamata di una funzione può essere utilizzata in qualsiasi espressione del tipo corrispondente al valore di ritorno della funzione

50

## Esempio - dichiarazione

```
CREATE FUNCTION AvgSal (@Dept VARCHAR(20)) RETURNS integer
AS
BEGIN
   DECLARE @Avg integer
   SELECT @Avg = avg(salary)
   FROM Employee
   WHERE Department = @Dept
   RETURN @Avg
END
```

oppure

```
CREATE FUNCTION AvgSal (@Dept VARCHAR(20))
        RETURNS integer
        AS
        BEGIN
          RETURN (      SELECT avg(salary)
                        FROM Employee
                        WHERE Department = @Dept)
        END
```

51

## Esempio - chiamata

- PRINT CAST(dbo.AvgSal(1) AS varchar)
- nella chiamata, il nome della funzione deve sempre essere preceduto dalla login corrispondente al proprietario della funzione (cioè colui che l'ha creata)

52

## SQL dinamico

- Statement dinamici possono essere eseguiti utilizzando una particolare stored procedure, definita dal sistema: sp_executesql
- non lo vediamo

53

## Esempio

```
CREATE PROCEDURE Modify_salary @my_department VARCHAR(20)
AS
BEGIN
   DECLARE   @avg_sal NUMERIC,
             @my_name VARCHAR,
             @my_salary NUMERIC
   DECLARE Name_Salary_cr CURSOR FOR
     SELECT name,salary
     FROM employee
     WHERE department = @my_department;
```

54

## Example

```
OPEN Name_Salary_cr
SELECT AVG(salary) INTO @avg_sal FROM employee
IF (@avg_sal < 1000) THEN
  UPDATE employee SET salary = salary*1,05;
ELSE
  UPDATE employee SET salary = salary*0,95;
END IF
WHILE (@@FETCH_STATUS = 0)
      BEGIN
            FETCH NEXT FROM Name_Salary_cr INTO
                                    @my_name,my_salary
            PRINT '  Name' + my_name
            PRINT '  Salary: ' + my_salary
       END
END
```

55

## Database connectivity

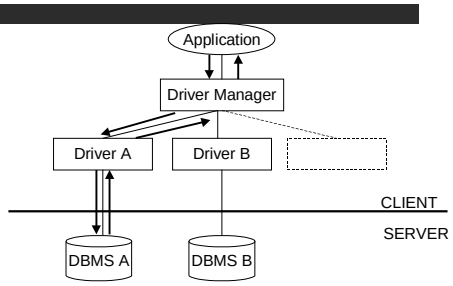
56

## Introduzione

- L'accesso alla base di dati è possibile tramite un'interfaccia chiamata **Call Level Interface** o CLI
- ogni CLI viene implementata come una libreria per il linguaggio che verrà utilizzato
- la maggioranza dei sistemi commerciali offre soluzioni di questo tipo
- SQL-99 specifica come le CLI debbano essere definite
- CLI standard:
  - ODBC per applicazioni C
  - JDBC per applicazioni Java
  - funzionalità più o meno equivalenti
  - JDBC sintatticamente più semplice di ODBC

57

## Architettura di riferimento




58

## Applicazione

- Un'applicazione è un programma che chiama specifiche funzioni API per accedere ai dati gestiti da un DBMS
- Flusso tipico:
  - selezione sorgente dati (DBMS e specifico database) e connessione
  - sottomissione statement SQL per l'esecuzione
  - recupero risultati e processamento errori
  - disconnessione

59

## Driver Manager

- È una libreria che gestisce la comunicazione tra applicazione e driver
- risolve problematiche comuni a tutte le applicazioni
  - quale driver caricare, basandosi sulle informazione fornite dall'applicazione
  - caricamento driver
  - chiamate alle funzioni dei driver
- l'applicazione interagisce solo con il driver manager

60

## Driver

- Sono librerie dinamicamente connesse alle applicazioni che implementano le funzioni API
- ciascuna libreria è specifica per un particolare DBMS
  - driver Oracle è diverso dal driver Informix
- traducono le varie funzioni API nel dialetto SQL utilizzato dal DBMS considerato (o nell'API supportata dal DBMS)
- il driver maschera le differenze di interazione dovute al DBMS usato, il sistema operativo e il protocollo di rete

61

## DBMS

- Il DBMS sostanzialmente rimane inalterato nel suo funzionamento
- riceve sempre e solo richieste nel linguaggio supportato
- esegue lo statement SQL ricevuto dal driver e invia i risultati

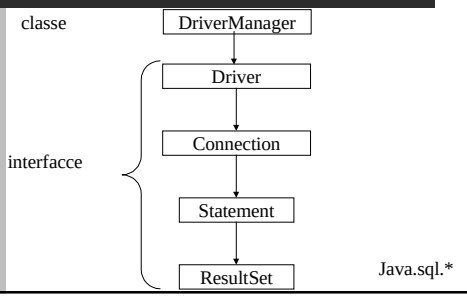
62

## JDBC

- JDBC è una JAVA API che contiene una classe o interfaccia per ogni concetto fondamentale dell'approccio di connettività

63

## Organizzazione




64

## JDBC driver

- Esistono quattro tipi di driver
- i driver si differenziano per come interagiscono con la base di dati
- il driver più semplice è JDBC-ODBC Bridge (driver di tipo 1)
  - viene fornito con l'ambiente Java
  - utilizza ODBC per connettersi alla base di dati

65

## Tipi di dato

- JDBC definisce un insieme di tipi SQL, che vengono poi mappati in tipi Java
- Gli identificatori sono definiti nella classe java.sql.types

66

## Tipi di dato

| JDBC Types Mapped to Java Types | |
|---|---|
| **JDBC Type** | **Java Type** |
| **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 |

67

## Flusso applicativo

- Caricamento driver
- Connessione
- Esecuzione statement
- Disconnessione

- Nel seguito, per semplicità di notazione, negli esempi non sempre inseriremo la gestione delle eccezioni
- La gestione delle eccezioni è però necessaria (si veda avanti)

68

## Caricamento driver

- Il primo passo in un'applicazione JDBC consiste nel caricare il driver che si intende utilizzare
- Ogni driver = classe Java

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- Il nome della classe da usare viene fornito con la documentazione relativa al driver

69

## Caricamento driver: esempio

```
import java.sql.*;

class JdbcTest
{
  public static void main (String args []) {
    Class.forName ("oracle.jdbc.OracleDriver");
  }
}
```

70

## Connessione

- Per connettersi alla base di dati è necessario specificare il DBMS e la base di dati del DBMS al quale ci si vuole connettere
- È possibile connettersi a qualunque database, locale e remoto, specificandone l'URL

71

## Connessione

- In JDBC, l'URL è formato da tre parti:

**jdbc: <subprotocol>: <subname>**

  - **<subprotocol>** identifica il driver o il meccanismo di connessione al database
  - **<subname>** dipende da subprotocol ed identifica lo specifico database

72

## Connessione: esempio su SQL Server

- se si usa JDBC-ODBC driver: jdbc:odbc:subname
- subname è un nome riconosciuto dal sistema come l'identificatore di una sorgente dati accessibile da ODBC
- è possibile utilizzare una semplice applicazione Microsoft per definire questo nome
  - lo vedremo in laboratorio

73

## Connessione

- La connessione avviene chiamando il metodo getConnection della classe DriverManager, che restituisce un oggetto di tipo Connection

```
Connection con =
DriverManager.getConnection("jdbc:odbc:provaDSN",
"myLogin", "myPassword");
```

- Se uno dei driver caricati riconosce l'URL fornito dal metodo, il driver stabilisce la connessione

74

## Connessione: esempio

```
import java.sql.*;
class JdbcTest
{
  static String ARS_URL = "jdbc:odbc:provaDSN";

  public static void main (String args [])
   {
    Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection ARS;
    ARS =DriverManager.getConnection(ARS_URL,
                "rossi",
                "secret");
  } }
```

75

## Manipolazione

- Gli statement SQL sono stringhe che vengono passate a metodi opportuni per la loro creazione e/o esecuzione
- gli statement possono essere
  - **statici**: statement noto a compile time
  - **dinamici:** statement costruito a run-time
  - **preparati**
  - **non preparati**

76

## Manipolazione

- **Statement preparati**
  - lo statement viene compilato e il piano di accesso viene determinato a compile-time
  - maggiore efficienza se lo statement deve essere eseguito varie volte(eventualmente con parametri differenti)
  - statement statico
- **Statement non-preparati**
  - la preparazione e l'esecuzione dello statement sono eseguite in un unico passo a run-time
  - statement statici o dinamici

77

## Creazione statement non preparati

- Un oggetto di tipo Statement viene creato a partire da un oggetto di tipo Connection e permette di inviare comandi SQL al DBMS:

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

- Si noti che l'oggetto statement non è ancora associato all'istruzione SQL da eseguire
  - Tale istruzione verrà specificata al momento dell'esecuzione

78

## Esecuzione statement non preparati

- É necessario distinguere tra statement che rappresentano query e statement di aggiornamento
- Per eseguire una query:

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

- Per eseguire una operazione di aggiornamento, inclusi gli statement DDL:

```
stmt.executeUpdate("INSERT INTO IMPIEGATI VALUES
  'AB34','Gianni', 'Rossi','GT67',1500");
stmt.executeUpdate("CREATE TABLE PROVA (CAMPO1
        NUMBER)");
```

79

## Statement non preparati - esempio

```
Connection con;
Statement stmt = con.createStatement();
stmt.executeQuery("SELECT * FROM employee");
stmt.executeUpdate("INSERT INTO employee
        VALUES (' Stefano','Olivaro',' Piazzale Roma' ,
          ' research',1500)");
```

80

## Statement non preparati - esempio

```
stmt.executeUpdate("CREATE TABLE task
        (employee_name VARCHAR (24),
         department_name VARCHAR(24),
         start_date DATE,
         end_date DATE,
         task_description VARCHAR(128),
         FOREIGN KEY (employee_name, department_name)
         REFERENCES work_for(employee_name, department_name),
         UNIQUE(employee_name, date)))");
```

81

## Creazione prepared Statement

- Un oggetto di tipo PreparedStatement viene creato a partire da un oggetto di tipo Connection e permette di inviare comandi SQL al DBMS:

```
PreparedStatement queryImp = con.prepareStatement("SELECT
  * FROM IMPIEGATI");
```

- La creazione di uno statement preparato richiede la specifica dello statement che dovrà poi essere eseguito

82

## Esecuzione statement preparati

- É necessario distinguere tra statement che rappresentano query e statement di aggiornamento
- Per eseguire una query:

```
queryImp.executeQuery();
```

- Per eseguire una operazione di aggiornamento, inclusi gli statement DDL:

```
queryImp.executeUpdate();
```

83

## Esecuzione statement

- Il terminatore dello statement (es. ';') viene inserito direttamente dal driver prima di sottomettere lo statement al DBMS per l'esecuzione

84

## Statement preparati - esempio

```
PreparedStatement query_pstmt =
        con.prepareStatement("SELECT * FROM employee");
PreparedStatement update_pstmt =
        con.prepareStatement("INSERT INTO employee
                        VALUES (' Stefano Olivaro',
                        ' Piazzale Roma' ,
                        ' research',1500)");

query_pstmt.executeQuery();
update_pstmt.executeUpdate();
```

85

## Uso di parametri in statement preparati

- È possibile specificare che la stringa che rappresenta lo statement SQL da preparare verrà completata con parametri al momento dell'esecuzione
- I parametri sono identificati da '?'

```
PreparedStatement queryImp = con.prepareStatement(
  "SELECT * FROM IMPIEGATI WHERE Nome = ?");
```

- I parametri possono poi essere associati allo statement preparato quando diventano noti

86

## Uso di parametri in statement preparati

- È possibile associare valori ai parametri usando il metodo setXXX, dove XXX rappresenta un tipo Java
  - setString, per stringhe
  - setInt, per interi
  - setBigDecimal, per NUMBER
- Esempio

```
queryImp.setString(1, 'Rossi');
queryImp.executeQuery():
```

87

## Statement preparati con parametri - esempio

```
PreparedStatement query_pstmt =
    con.prepareStatement("SELECT *
                    FROM employee
                    WHERE department = ?");

query_pstmt.setString(1, ' research');

query_pstmt.executeQuery();
```

88

## Elaborazione risultato

- JDBC restituisce i risultati di esecuzione di una query in un result set

```
String query = " SELECT * FROM IMPIEGATI ";
ResultSet rs = stmt.executeQuery(query);
```

- Il result set è costruito solo per query e non per statement di aggiornamento
- In questo caso viene restituito un intero, che rappresenta il numero di tuple modificate (0 in caso di statement DDL)

89

## Elaborazione risultato

- Il metodo next() permette di spostarsi nel result set (cursore):
  - while (rs.next()) {/* get current row */}
- inizialmente il cursore è posizionato prima della prima tupla
- il metodo diventa falso quando non ci sono più tuple da analizzare

90

## Metodi per accedere i valori associati agli attributi

- Il metodo getXXX, di un ResultSet, permette di recuperare il valore associato ad un certo attributo, puntato correntemente dal cursore
- XXX è il tipo Java nel quale il valore deve essere convertito (String, Int, BigDecimal,…)

```
String s = rs.getString("Cognome");
```

- Gli attributi possono anche essere acceduti tramite la notazione posizionali:
  - String s = rs.getString(2);
  - int n = rs.getInt(5);
- Usare getInt per valori numerici, getString per char, varchar

91

## Esempio

```
...
   Statement selImp = ARS.createStatement ();
   String stmt =  "SELECT * FROM Impiegati WHERE Cognome
                                          ='Rossi'";
   ResultSet  impRossi = selImp.executeQuery (stmt);

   while ( impRossi.next() )
   {
     System.out.println (impRossi.getString ("Stipendio"));
   }
  ...
```

92

## Esempio

```
...
    String stmt =
        "SELECT * FROM Impiegati WHERE Cognome = 'Rossi'";
   PreparedStatement prepStmt = ARS.preparedStatement (stmt);
   ResultSet impRossi = prepStmt.executeQuery ();

     while ( impRossi.next() )
   {
     System.out.println (impRossi.getString ("Stipendio"));
   }
...
```

93

## Funzioni e procedure

- É possibile creare, manipolare e chiamare routine SQL da JDBC
- procedure e funzioni possono essere create utilizzando l'appropriato costrutto DDL
- JDBC fornisce metodi per l'esecuzione di procedure e funzioni, anche in presenza di parametri
  - non le vediamo
- é anche possibile eseguire le procedure con gli usuali metodi per l'esecuzione delle query o degli update
  - si deve utilizzare executeUpdate se la procedura può modificare dei dati

94

## Disconnessione

- Per risparmiare risorse, può essere utile chiudere gli oggetti di classe Connection, Statement, ResultSet quando non vengono più utilizzati
- metodo close()
- la chiusura di un oggetto di tipo Connection chiude tutti gli Statement associati mentre la chiusura di uno Statement chiude ResultSet associati

95

## Eccezioni

- La classe java.sql.SQLException estende la classe java.lang.Exception in modo da fornire informazioni ulteriori in caso di errore di accesso al database, tra cui:
  - la stringa SQLState che rappresenta la codifica dell'errore in base allo standard X/Open
    - getSQLState()
  - il codice di errore specifico al DBMS
    - getErrorCode()
  - una descrizione dell'errore
    - getMessage()

96

## SQL dinamico

- I metodi executeQuery() e prepareStatement() vogliono una stringa come argomento
- questa stringa non necessariamente deve essere nota a tempo di compilazione
- questo permette di eseguire facilmente statement SQL dinamici

97

## SQL dinamico: esempio

- Supponiamo che la condizione WHERE di uno statement SQL sia noto solo a tempo di esecuzione

```
String query = “SELECT nome FROM Impiegati”;
if (condition)
    query += “WHERE stipendio > 1000”;
else
    query += “WHERE Nome = ‘Rossi’”;
...
```

98

## Esempio

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

class exampleJDBC
{public static void main (String args [])
 {
 Connection con = null;
 try{
    String my_department = "research";
    Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
    con =DriverManager.getConnection("jdbc:odbc:my_DB",
                                     "my_login",
                                     "my_password");
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("SELECT AVG(salary) FROM employee");
```

99

```
rs.next();
   if (rs.getBigDecimal(1) < 1000)
       st.executeUpdate("UPDATE employee
                         SET salary = salary*1,05");
   else
       st.executeUpdate("UPDATE employee
                         SET salary = salary*0,95");

   PreparedStatement pst =
                con.prepareStatement("SELECT name,salary
                                      FROM employee
                                      WHERE department = ?");
   pst.setString(1,my_department);
   rs = pst.executeQuery();
   while (rs.next())
     System.println("Name: "+rs.getString(1)+
                    "Salary:"+rs.getInt(2));
   con.commit();
   con.close();
   }
```

100

```
catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());}
catch (SQLException e) {while( e!=null)
                {
                System.out.println("SQLState: " + e.getSQLState());
                System.out.println("    Code: " + e.getErrorCode());
                System.out.println(" Message: " + e.getMessage());
                e = e.getNextException();
                }}}}}
```

101

## SQL ospitato

102

## Introduzione

- Soluzione ad accoppiamento esterno
- gli statement SQL vengono direttamente inseriti in programmi scritti in un'estensione di un linguaggio di programmazione, chiamato **linguaggio ospite**
- il linguaggio ospite permette di utilizzare SQL per l'accesso ai dati, senza utilizzare funzioni di interfaccia
- gli statement SQL possono apparire in ogni punto in cui può comparire un'istruzione del linguaggio ospite

103

## Architettura di riferimento

- Gli statement SQL devono essere chiaramente identificati nel testo
  - preceduti da un prefisso e seguiti da un terminatore
- per molti linguaggi (C, Pascal, Fortran, and COBOL):
  - prefisso = EXEC SQL
- SQLj: specifica standard ANSI/ISO per ospitare SQL in programmi JAVA
  - prefisso: #sql
  - terminatore: ';'

104

## Architettura di riferimento

- L'esecuzione dei programmi risultanti avviene in tre passi:
  - pre-compilazione, compilazione, esecuzione
- **precompilazione**: traduce il programma contenente comandi SQL in un programma scritto interamente nel linguaggio ospitante L, utilizzando un'opportuna CLI (standard o proprietaria)
  - viene aggiunta la connessione al DBMS
  - gli statement SQL vengono tradotti in chiamate ad opportune funzioni della CLI
- **compilazione**: poiché il programma ottenuto dopo la pre-compilazione è interamente scritto nel linguaggio ospitante L, per la compilazione si può direttamente utilizzare un compilatore per L

105

## SQLj

- Proposta per ospitare SQL in programmi JAVA
- Partecipanti alla definizione dello standard:
  - Oracle
  - IBM
  - Sybase
  - Informix
  - Sun
  - Compaq
- SQL Server non supporta (per il momento) SQLj
- Nel seguito: esempi su Oracle

106

## Connessione

- La connessione viene stabilita chiamando unmetodo di una classe che dipende dallo specifico DBMS utilizzato

```
oracle.connect("jdbc:odbc:my_DB",
               "my_login",
               "my_password");
```

107

## Manipolazione

- SQLj permette l'utilizzo di statement SQL statici
- gli statement SQL sono preceduti dalla parola chiave #sql, terminati da ; e contenuti tra { e }

```
#sql {SQL_statement};
```

108

## Esempio

```
#sql {INSERT INTO employee VALUES (
    ' Stefano', 'Olivaro',
    ' Piazzale Roma' ,
    ' research', 1500)};
```

109

## Variabili

- Le variabili JAVA possono essere utilizzate negli statement SQL, in ogni contesto in cui possa essere utilizzato un valore del tipo della variabile
- devono essere precedute da : per distinguerle dai nomi degli attributi delle tabelle

```
String department = ' research';
BigDecimal my_budget;
#sql{SELECT budget INTO :my_budget
    FROM department
    WHERE name = :department};
```

110

## Cursori

- In SQLj, un cursore è un'istanza di una classe "iterator"
  - é necessario definire una classe iterator per ogni tipo di risultato da analizzare
  - una classe iterator contiene un attributo per ogni attributo delle tuple da analizzare
- Operazioni sui cursori
  - Dichiarazione cursore = creazione istanza classe iterator
  - apertura cursore = assegnazione all'istanza del risultato di una query
  - chiusura cursore: chiusura invocata sull'istanza
- le istanze della classe iterator supportano un metodo di accesso per ogni attributo

111

## Cursori - esempio

```
#sql iterator EmpIter (String name, Real salary);
EmpIter my_empiter = null;
#sql my_empiter = {SELECT name, salary FROM
    employee};
while (my_empiter.next()) {
   System.out.println("Name: " + my_empiter.name());
   System.out.println("Salary: " + my_empiter.salary());}
my_empiter.close();
```

112

## Procedure e funzioni

- #sql {CALL procedure_call};
- #sql host_variable = {VALUES function_call};

113

## Eccezioni

- Come in JDBC

114

## Esempio

```
import java.sql.*;
import java.io.*;
import java.math.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;
import java.sql.*;

class exampleSQLj
{
 #sql iterator Name_Salary_Iter(String name, int salary);

 public static void main (String args [])
 {
 try{
    BigDecimal avg_sal;
    String my_department = "research";
    oracle.connect("jdbc:odbc:my_DB","my_login", "my_password", false);
```

115

```
#sql{SELECT AVG(salary) INTO :avg_sal FROM employee};
   if (avg_sal > 1000)
       #sql{UPDATE employee SET salary = salary*1,05};
   else
       #sql{UPDATE employee SET salary = salary*0,95};

   Name_Salary_Iter my_iter = null;
   #sql my_iter ={SELECT name,salary
                FROM employee
                WHERE department = :my_department};
   while (my_iter.next())
     System.println("Name: "+my_iter.name()+
                 "Salary:"+my_iter.salary());
   }
```

116

```
catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());}

catch (SQLException e) {
      while( e!=null){
       System.out.println("SQLState: " + e.getSQLState());
       System.out.println("    Code: " + e.getErrorCode());
       System.out.println(" Message: " + e.getMessage());
       e = e.getNextException();}}}}}
```

117