

Atzeni, Ceri, Paraboschi, Torlone
Basi di dati
McGraw-Hill, 1996-2006

Capitolo 6:
SQL nei linguaggi di
programmazione

24/05/2006

SQL e applicazioni

- In applicazioni complesse, l'utente non vuole eseguire comandi SQL, ma programmi, con poche scelte
- SQL non basta, sono necessarie altre funzionalità, per gestire:
 - input (scelte dell'utente e parametri)
 - output (con dati che non sono relazioni o se si vuole una presentazione complessa)
 - per gestire il controllo

Approcci

- Incremento delle funzionalità di SQL
 - Stored procedure
 - Triggers
 - Linguaggi 4GL
- SQL + linguaggi di programmazione

Stored procedure

- Sequenza di istruzioni SQL con parametri
- Memorizzate nella base di dati

```
procedure AssegnaCitta(:Dip varchar(20),  
                      :Citta varchar(20))  
  
update Dipartimento  
set Città = :Citta  
where Nome = :Dip;
```

Invocazione di stored procedure

- Possono essere invocate

- Internamente

```
execute procedure
```

```
AssegnaCitta('Produzione','Milano');
```

- Esternamente

```
...
```

```
$ AssegnaCitta(:NomeDip,:NomeCitta);
```

```
...
```

Estensioni SQL per il controllo

- Esistono diverse estensioni

```
procedure CambiaCittaADip(:NomeDip varchar(20),
                          :NuovaCitta varchar(20))
    if      (      select *
                  from Dipartimento
                  where Nome = :NomeDip ) = NULL
    insert into ErroriDip values (:NomeDip)
    else
        update Dipartimento
        set Città = :NuovaCitta
        where Nome = :NomeDip;
    end if;
end;
```

Linguaggi 4GL

- Ogni sistema adotta, di fatto, una propria estensione
- Diventano veri e propri linguaggi di programmazione proprietari “ad hoc”:
 - PL/SQL,
 - Informix4GL,
 - Delphi,
 - DB2 SQL/PL

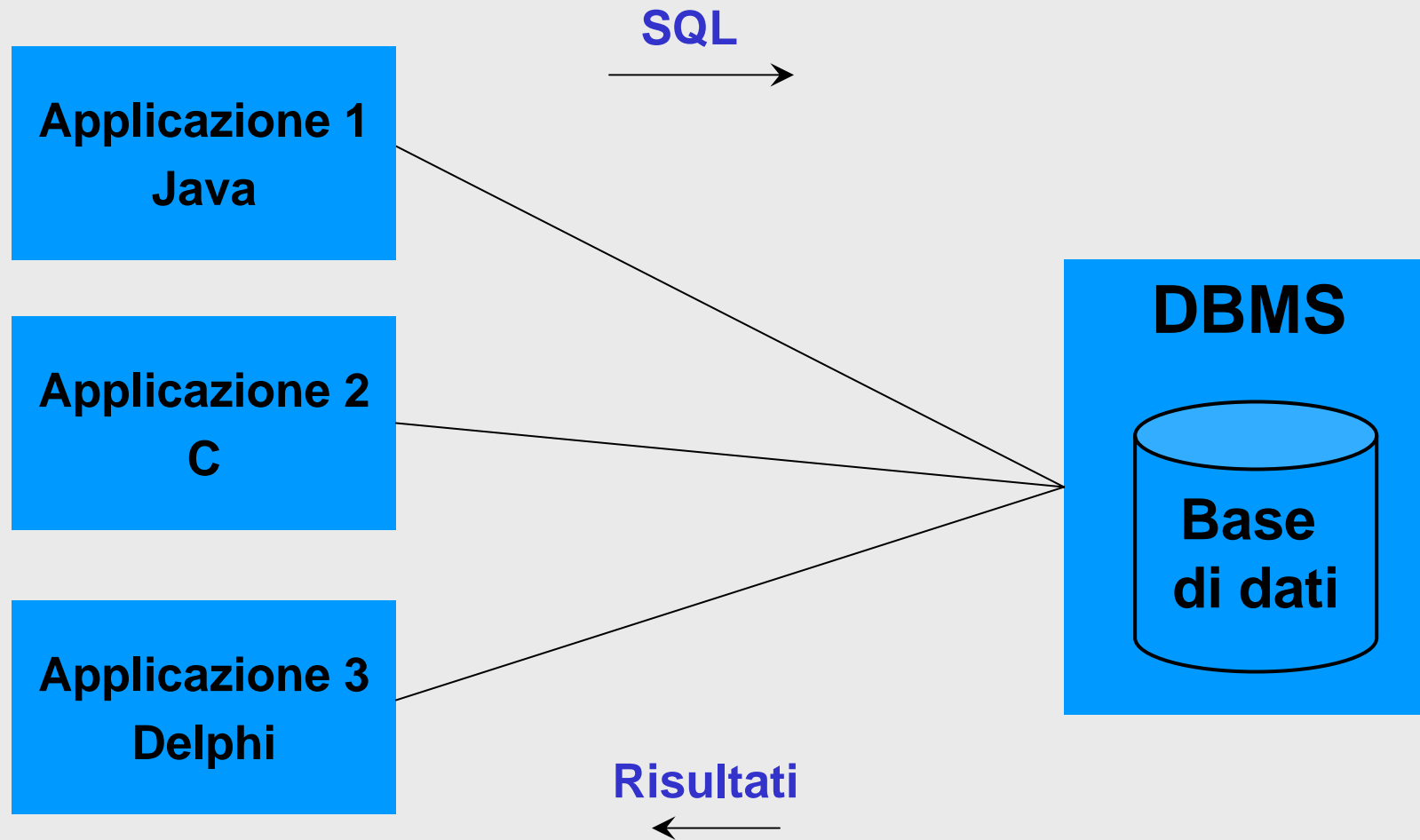
Procedure in Oracle PL/SQL

```
Procedure Debit(ClientAccount char(5),Withdrawal
integer) is
  OldAmount integer;
  NewAmount integer;
  Threshold integer;
begin
  select Amount, Overdraft into OldAmount, Threshold
  from BankAccount
  where AccountNo = ClientAccount
  for update of Amount;
  NewAmount := OldAmount - Withdrawal;
  if NewAmount > Threshold
  then update BankAccount
        set Amount = NewAmount
        where AccountNo = ClientAccount;
  else
    insert into OverDraftExceeded
    values(ClientAccount,Withdrawal,sysdate);
  end if;
end Debit;
```


SQL e linguaggi di programmazione

- Le applicazioni sono scritte in
 - linguaggi di programmazione tradizionali:
 - Cobol, C, Java, Fortran
 - linguaggi “ad hoc”, proprietari e non:
 - vedi lucidi precedenti
- Vediamo solo l’approccio “tradizionale”, perché più generale

Applicazioni ed SQL: architettura



Una difficoltà importante

- Conflitto di impedenza (“disaccoppiamento di impedenza”) fra base di dati e linguaggio
 - linguaggi: operazioni su singole variabili o oggetti
 - SQL: operazioni su relazioni (insiemi di ennuple)

Altre differenze

- Tipi “strutturati” disponibili:
 - linguaggio: dipende dal paradigma
 - SQL: relazioni e ennuple
- Accesso ai dati e correlazione:
 - linguaggio: dipende dal paradigma e dai tipi disponibili; ad esempio scansione di liste o “navigazione” tra oggetti
 - SQL: join (ottimizzabile)
- tipi di base:
 - linguaggi: numeri, stringhe, booleani
 - SQL: CHAR, VARCHAR, DATE, ...

SQL e linguaggi di programmazione: tecniche principali

- SQL immerso (“Embedded SQL”)
 - sviluppata sin dagli anni '70
 - “SQL statico”
- SQL dinamico
- Call Level Interface (CLI)
 - più recente
 - SQL/CLI, ODBC, JDBC

SQL immerso

- le istruzioni SQL sono “immerse” nel programma redatto nel linguaggio “ospite”
- un precompilatore (legato al DBMS) viene usato per analizzare il programma e tradurlo in un programma nel linguaggio ospite (sostituendo le istruzioni SQL con chiamate alle funzioni di una API del DBMS)

SQL immerso, un esempio

```
#include<stdlib.h>
main(){
    exec sql begin declare section;
        char *NomeDip = "Manutenzione";
        char *CittaDip = "Pisa";
        int NumeroDip = 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(:NomeDip, :CittaDip, :NumeroDip);
        exec sql disconnect all;
    }
}
```

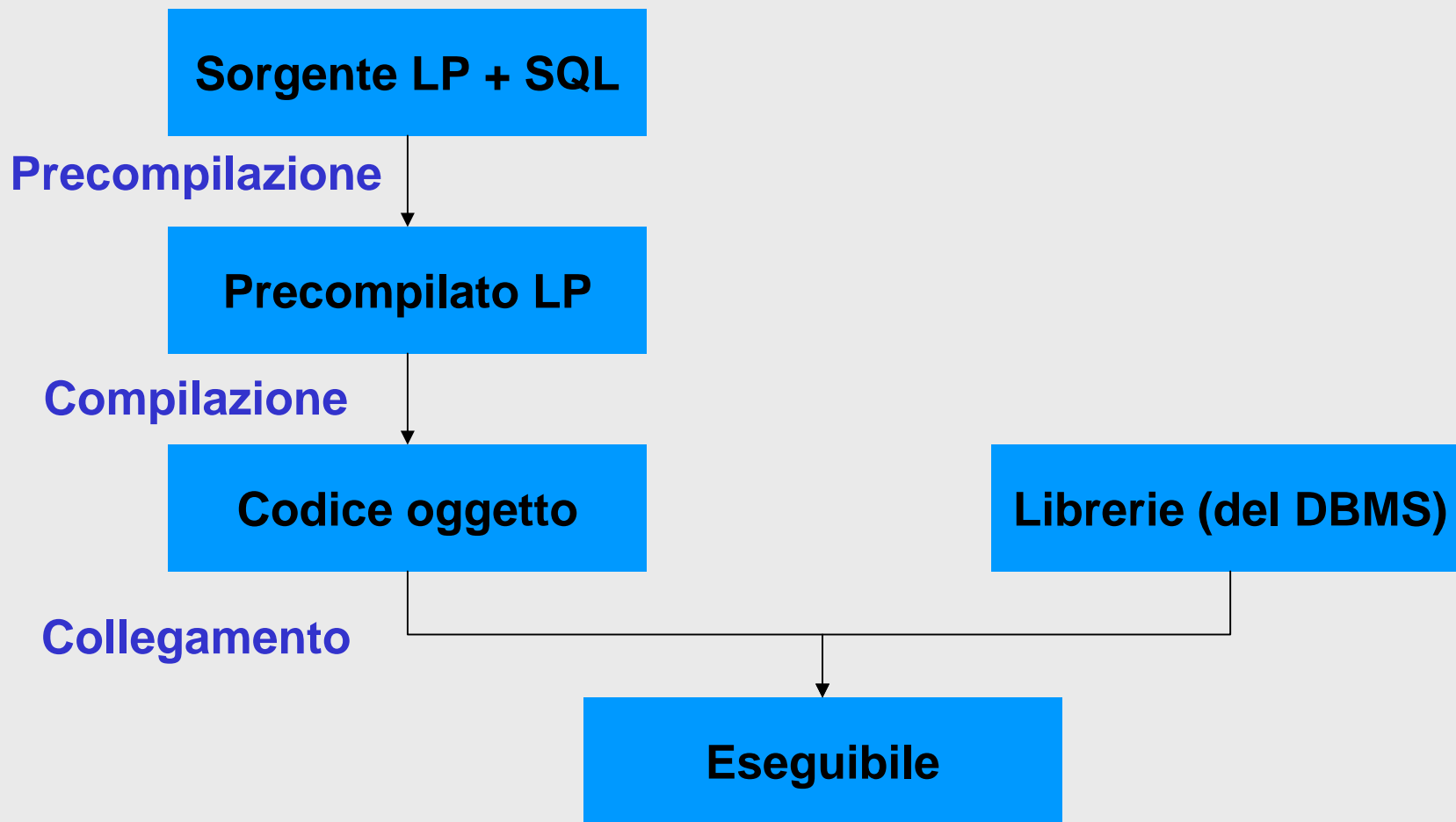
SQL immerso, commenti al codice

- EXEC SQL denota le porzioni di interesse del precompilatore:
 - definizioni dei dati
 - istruzioni SQL
- le variabili del programma possono essere usate come “parametri” nelle istruzioni SQL (precedute da “:”) dove sintatticamente sono ammesse costanti

SQL immerso, commenti al codice, 2

- `sqlca` è una struttura dati per la comunicazione fra programma e DBMS
- `sqlcode` è un campo di `sqlca` che mantiene il codice di errore dell'ultimo comando SQL eseguito:
 - zero: successo
 - altro valore: errore o anomalia

SQL immerso, fasi



Un altro esempio

```
int main() {  
    exec sql connect to universita  
        user pguser identified by pguser;  
    exec sql create table studente  
        (matricola integer primary key,  
         nome varchar(20),  
         annodicorso integer);  
    exec sql disconnect;  
    return 0;  
}
```

L'esempio "precompilato"

```
/* These include files are added by the preprocessor */
#include <ecpgtype.h>
#include <ecpglib.h>
#include <ecpgerrno.h>
#include <sqlca.h>
int main() {
    ECPGconnect(__LINE__, "universita" , "pguser" ,
        "pguser" , NULL, 0);
    ECPGdo(__LINE__, NULL, "create table studente (
matricola integer primary key , nome varchar ( 20 ) ,
annodicorso integer )", ECPGt_EOIT, ECPGt_EORT);
    ECPGdisconnect(__LINE__, "CURRENT");
    return 0;
}
```

Note

- Il precompilatore è specifico della
combinazione
linguaggio-DBMS-sistema operativo

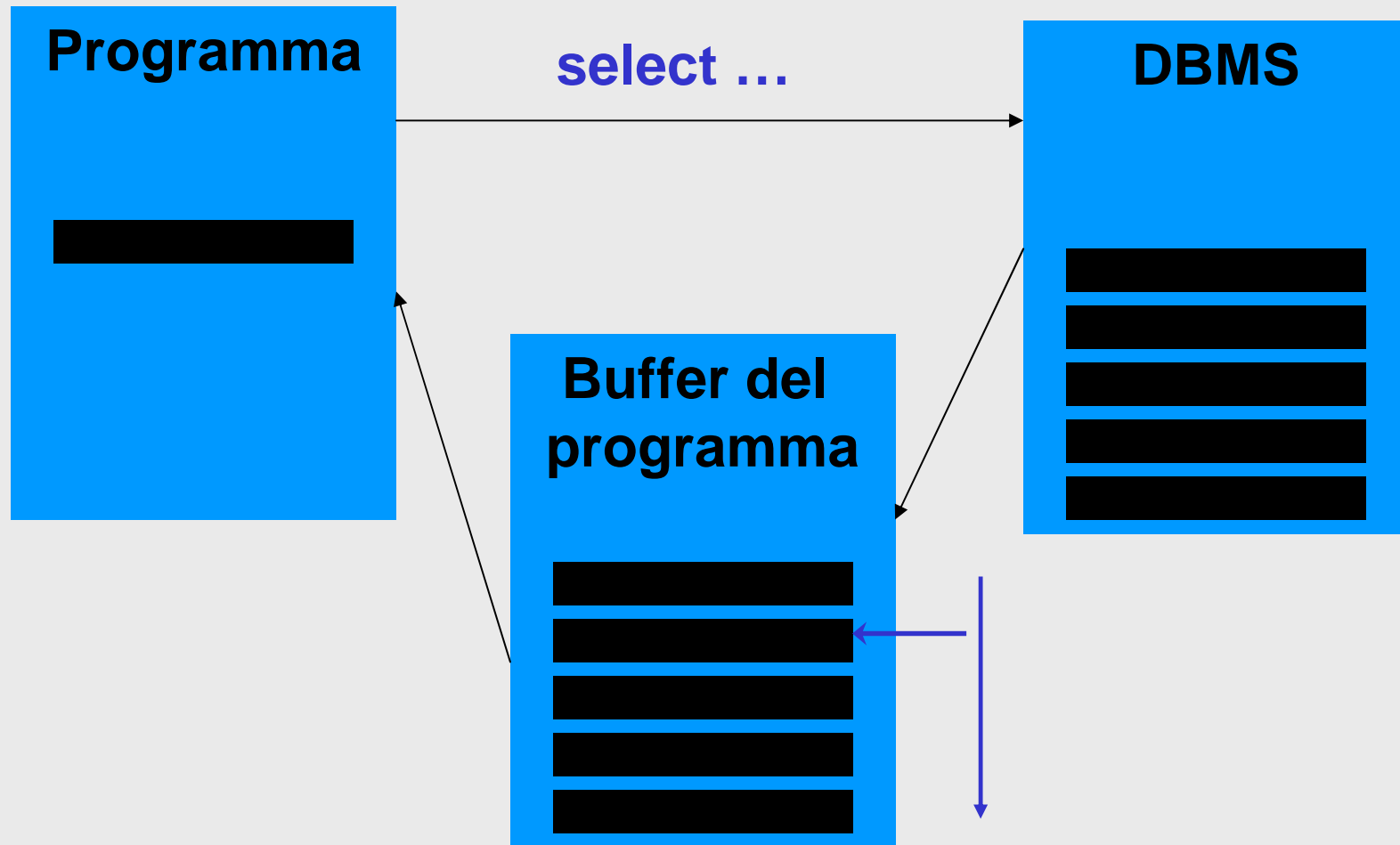
SQLJ, uno standard per SQL immerso in Java

```
import ...
#sql iterator CursoreProvaSelect(String, String);
class ProvaSelect
{
    public static void main(String argv[])
    {
        ...
        Db db = new Db(argv[0]);
        db.getDefaultContext();
        ...
        String padre = "";    String figlio = "" ; String padrePrec = "";
        CursoreProvaSelect cursore;
        #sql cursore = {SELECT Padre, Figlio FROM Paternita ORDER BY Padre};
        #sql {FETCH :cursore INTO :padre, :figlio};
        while (!cursore.endFetch()){
            if (!(padre.equals(padrePrec))) { System.out.println("Padre: " + padre + "\n Figli: " + figlio);}
            else System.out.println( "          " + figlio ) ;
            padrePrec = padre ;
            #sql {FETCH :cursore INTO :padre, :figlio};
            cursore.close();
            ...
        }
    }
}
```

Interrogazioni in SQL immerso: conflitto di impedenza

- Il risultato di una **select** è costituito da zero o piú ennuple:
 - zero o una: ok -- l'eventuale risultato puó essere gestito in un record
 - piú ennuple: come facciamo?
 - l'insieme (in effetti, la lista) non è gestibile facilmente in molti linguaggi
- **Cursore**: tecnica per trasmettere al programma una ennupla alla volta

Cursore



Nota

- Il cursore
 - accede a tutte le ennuple di una interrogazione in modo globale (tutte insieme o a blocchi – è il DBMS che sceglie la strategia efficiente)
 - trasmette le ennuple al programma una alla volta

Operazioni sui cursori

Definizione del cursore

```
declare NomeCursore [ scroll ] cursor for Select
```

...

Esecuzione dell'interrogazione

```
open NomeCursore
```

Utilizzo dei risultati (una ennupla alla volta)

```
fetch NomeCursore into ListaVariabili
```

Disabilitazione del cursore

```
close cursor NomeCursore
```

Accesso alla ennupla corrente (di un cursore su singola relazione a fini di aggiornamento)

```
current of NomeCursore
```

nella clausola **where**

```

write('nome della citta''?');
readln(citta);
EXEC SQL DECLARE P CURSOR FOR
    SELECT NOME, REDDITO
    FROM PERSONE
    WHERE CITTA = :citta ;
EXEC SQL OPEN P ;
EXEC SQL FETCH P INTO :nome, :reddito ;
while SQLCODE = 0
do begin
    write('nome della persona:', nome, 'aumento?');
    readln(aumento);
    EXEC SQL UPDATE PERSONE
        SET REDDITO = REDDITO + :aumento
        WHERE CURRENT OF P
    EXEC SQL FETCH P INTO :nome, :reddito
end;
EXEC SQL CLOSE CURSOR P

```

```

void VisualizzaStipendiDipart(char NomeDip[])
{
    char Nome[20], Cognome[20];
    long int Stipendio;
    $ declare ImpDip cursor for
        select Nome, Cognome, Stipendio
        from Impiegato
        where Dipart = :NomeDip;
    $ open ImpDip;
    $ fetch ImpDip into :Nome, :Cognome, :Stipendio;
    printf("Dipartimento %s\n",NomeDip);
    while (sqlcode == 0)
    {
        printf("Nome e cognome dell'impiegato: %s
                %s",Nome,Cognome);
        printf("Attuale stipendio: %d\n",Stipendio);
        $ fetch ImpDip into :Nome, :Cognome,
            :Stipendio;
    }
    $ close cursor ImpDip;
}

```

Cursori, commenti

- Per aggiornamenti e interrogazioni “scalari” (cioè che restituiscano una sola ennupla) il cursore non serve:

```
select Nome, Cognome  
      into :nomeDip, :cognomeDip  
from Dipendente  
where Matricola = :matrDip;
```

Cursori, commenti, 2

- I cursori possono far scendere la programmazione ad un livello troppo basso, pregiudicando la capacità dei DBMS di ottimizzare le interrogazioni:
 - se “nidifichiamo” due o più cursori, rischiamo di reimplementare il join!

Esercizio

Studenti(Matricola, Cognome, Nome)

Esami(Studente, Materia, Voto, Data)

Corsi(Codice, Titolo)

con gli ovvî vincoli di integrità referenziale

- Stampare, per ogni studente, il certificato con gli esami e il voto medio

Output

Matricola Cognome Nome
Materia Data Voto
...
Materia Data Voto
VotoMedio
Matricola Cognome Nome
Materia Data Voto
...
Materia Data Voto
VotoMedio
...

Esercizio

Studenti(Matricola, Cognome, Nome)

Esami(Studente, Materia, Voto, Data)

Corsi(Codice, Titolo)

Iscrizioni(Studente, AA, Anno, Tipo)

con gli ovvî vincoli di integrità referenziale

- Stampare, per ogni studente, il certificato con gli esami e le iscrizioni ai vari anni accademici

Output

```
Matricola Cognome Nome
  AnnoAccademico AnnoDiCorso TipolScrizione
...
  AnnoAccademico AnnoDiCorso TipolScrizione
    Materia Data Voto
...
    Materia Data Voto
Matricola Cognome Nome
  AnnoAccademico AnnoDiCorso TipolScrizione
...
  AnnoAccademico AnnoDiCorso TipolScrizione
    Materia Data Voto
...
    Materia Data Voto
```

SQL dinamico

- Non sempre le istruzioni SQL sono note quando si scrive il programma
- Allo scopo, è stata definita una tecnica completamente diversa, chiamata *Dynamic SQL* che permette di eseguire istruzioni SQL costruite dal programma (o addirittura ricevute dal programma attraverso parametri o da input)
- Non è banale gestire i parametri e la struttura dei risultati (non noti a priori)

SQL dinamico

- Le operazioni SQL possono essere:

- eseguite immediatamente

`execute immediate SQLStatement`

- prima “prepare”:

`prepare CommandName from SQLStatement`

e poi eseguite (anche più volte):

`execute CommandName [into TargetList]
[using ParameterList]`

Call Level Interface

- Indica genericamente interfacce che permettono di inviare richieste a DBMS per mezzo di parametri trasmessi a funzioni
- standard **SQL/CLI** ('95 e poi parte di SQL:1999)
- **ODBC**: implementazione proprietaria di SQL/CLI
- **JDBC**: una CLI per il mondo Java

SQL immerso vs CLI

- SQL immerso permette
 - precompilazione (e quindi efficienza)
 - uso di SQL completo
- CLI
 - indipendente dal DBMS
 - permette di accedere a più basi di dati, anche eterogenee

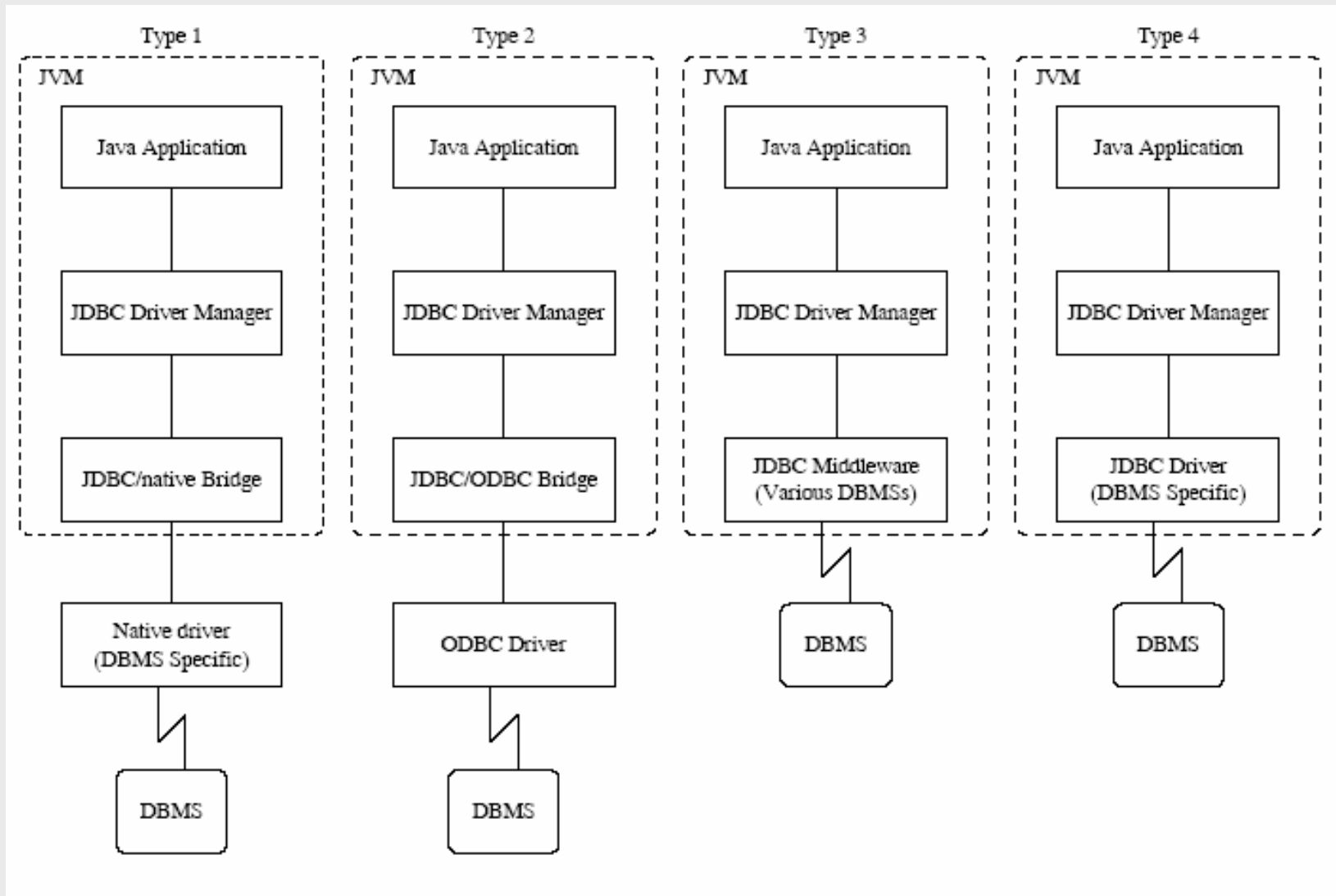
JDBC

- Una API (Application Programming Interface) di Java (intuitivamente: una libreria) per l'accesso a basi di dati, in modo indipendente dalla specifica tecnologia
- JDBC è una **interfaccia**, realizzata da classi chiamate **driver**:
 - l'interfaccia è standard, mentre i driver contengono le specificità dei singoli DBMS (o di altre fonti informative)

I driver JDBC

- (A titolo di curiosità; ne basta uno qualunque)
Esistono quattro tipi di driver (chiamati, in modo molto anonimo, tipo 1, tipo 2, tipo 3, tipo 4):
 1. Bridge JDBC-ODBC: richiama un driver ODBC, che deve essere disponibile sul client; è comodo ma potenzialmente inefficiente
 2. Driver nativo sul client: richiama un componente proprietario (non necessariamente Java) sul client
 3. Driver puro Java con server intermedio ("middleware server"): comunica via protocollo di rete con il server intermedio, che non deve risiedere sul client
 4. Driver puro Java, con connessione al DBMS: interagisce direttamente con il DBMS

I driver JDBC



Il funzionamento di JDBC, in breve

- Caricamento del driver
- Apertura della connessione alla base di dati
- Richiesta di esecuzione di istruzioni SQL
- Elaborazione dei risultati delle istruzioni SQL

Un programma con JDBC

```
import java.sql.*;
public class PrimoJDBC {
    public static void main(String[] arg){
        Connection con = null ;
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url = "jdbc:odbc:Corsi";
            con = DriverManager.getConnection(url);
        }
        catch(Exception e){
            System.out.println("Connessione fallita");
        }
        try {
            Statement query = con.createStatement();
            ResultSet result =
                query.executeQuery("select * from Corsi");
            while (result.next()){
                String nomeCorso = result.getString("NomeCorso");
                System.out.println(nomeCorso);
            }
        }
        catch (Exception e){
            System.out.println("Errore nell'interrogazione");
        }
    }
}
```

Un altro programma con JDBC, 1

```
import java.lang.*;
import java.sql.*;
class ProvaSelectJDBC
{
    public static void main(String argv[])
    {
        Connection con = null;
        try { Class.forName("com.ibm.db2.jcc.DB2Driver");
        }
        catch (ClassNotFoundException exClass) {
            System.err.println("Fallita connessione al database. Errore 1");
        }
        try {
            String url = "jdbc:db2:db04";
            con = DriverManager.getConnection(url);
        }
        catch (SQLException exSQL) {
            System.err.println("Fallita connessione al database. "+
                exSQL.getErrorCode() + " " + exSQL.getSQLState() +
                exSQL.getMessage() );
        }
    }
}
```

Un altro programma con JDBC, 2

```
try{ String padre = ""; String figlio = "" ; String padrePrec = "";
Statement query = con.createStatement();
String queryString =
    "SELECT Padre, Figlio FROM Paternita ORDER BY Padre";
ResultSet result = query.executeQuery(queryString);
while (result.next()){
    padre = result.getString("Padre");
    figlio = result.getString("Figlio");
    if (!(padre.equals(padrePrec))){
        System.out.println("Padre: " + padre +
            "\n Figli: " + figlio);}
    else System.out.println( "          " + figlio ) ;
    padrePrec = padre ;
}
}
catch (SQLException exSQL) {
System.err.println("Errore nell'interrogazione. "+
    exSQL.getErrorCode() + " " + exSQL.getMessage() );
}
}
```

Preliminari

- L'interfaccia JDBC è contenuta nel package `java.sql`

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

- Il driver deve essere caricato (trascuriamo i dettagli)

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

- Connessione: oggetto di tipo `Connection` che costituisce un collegamento attivo fra programma Java e base di dati; viene creato da

```
String url = "jdbc:odbc:Corsi";  
con = DriverManager.getConnection(url);
```

Preliminari dei preliminari: origine dati ODBC

- Per utilizzare un driver JDBC-ODBC, la base di dati (o altro) deve essere definita come "origine dati ODBC"
- In Windows (con **YYY**, avendo già definito la base di dati **xxx.yyy** da collegare):
 - Pannello di controllo
 - Strumenti di amministrazione
 - Opzione "Origini dati ODBC"
 - Bottone "Aggiungi" ("Add")
 - Nella finestra di dialogo "Crea Nuova origine dati" selezionare "**YYY Driver**" e nella successiva
 - selezionare il file **xxx.yyy**
 - attribuirgli un nome (che sarà usato da ODBC e quindi da JDBC)

Esecuzione dell'interrogazione ed elaborazione del risultato

Esecuzione dell'interrogazione

```
Statement query = con.createStatement();  
ResultSet result =  
    query.executeQuery("select * from Corsi");
```

Elaborazione del risultato

```
while (result.next()) {  
    String nomeCorso =  
        result.getString("NomeCorso");  
    System.out.println(nomeCorso);  
}
```


Statement

- Un'interfaccia i cui oggetti consentono di inviare, tramite una connessione, istruzioni SQL e di ricevere i risultati forniti
- Un oggetto di tipo `Statement` viene creato con il metodo `createStatement` di `Connection`
- I metodi dell'interfaccia `Statement`:
 - `executeUpdate` per specificare aggiornamenti o istruzioni DDL
 - `executeQuery` per specificare interrogazioni e ottenere un risultato
 - `execute` per specificare istruzioni non note a priori
 - `executeBatch` per specificare sequenze di istruzioni
- Vediamo `executeQuery`

ResultSet

- I risultati delle interrogazioni sono forniti in oggetti di tipo `ResultSet` (interfaccia definita in `java.sql`)
- In sostanza, un result set è una sequenza di ennuple su cui si può "navigare" (in avanti, indietro e anche con accesso diretto) e dalla cui ennupla "corrente" si possono estrarre i valori degli attributi
- Metodi principali:
 - `next()`
 - `getXXX(posizione)`
 - es: `getString(3); getInt(2)`
 - `getXXX(nomeAttributo)`
 - es: `getString("Cognome"); getInt("Codice")`

Specializzazioni di Statement

- **PreparedStatement** permette di utilizzare codice SQL già compilato, eventualmente parametrizzato rispetto alle costanti
 - in generale più efficiente di **Statement**
 - permette di distinguere più facilmente istruzioni e costanti (e apici nelle costanti)i metodi **setXXX(,)** permettono di definire i parametri
- **CallableStatement** permette di utilizzare "stored procedure", come quelle di Oracle PL/SQL o anche le query memorizzate (e parametriche) di Access

```

import java.sql.*;
import javax.swing.JOptionPane;
public class SecondoJDBCprep {
    public static void main(String[] arg){
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url = "jdbc:odbc:Corsi";
            Connection con = DriverManager.getConnection(url);
            PreparedStatement pquery = con.prepareStatement(
                "select * from Corsi where NomeCorso LIKE ?");
            String param = JOptionPane.showInputDialog(
                "Nome corso (anche parziale)?");
            param = "%" + param + "%";
            pquery.setString(1,param);
            ResultSet result = pquery.executeQuery();
            while (result.next()){
                String nomeCorso = result.getString("NomeCorso");
                System.out.println(nomeCorso);
            }
        } catch (Exception e){System.out.println("Errore");}
    }
}

```

```

import java.sql.*;
import javax.swing.JOptionPane;

public class TerzoJDBCcall {
    public static void main(String[] arg){
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String url = "jdbc:odbc:Corsi";
            Connection con = DriverManager.getConnection(url);
            CallableStatement pquery =
                con.prepareCall("{call queryCorso(?)}");
            String param = JOptionPane.showInputDialog(
                "Nome corso (anche parziale)?");
            param = "*" + param + "*";
            pquery.setString(1,param);
            ResultSet result = pquery.executeQuery();
            while (result.next()){
                String nomeCorso =
                    result.getString("NomeCorso");
                System.out.println(nomeCorso);
            }
        } catch (Exception e){System.out.println("Errore");}
    }
}

```

Altre funzionalità

- Molte, fra cui
 - username e password
 - aggiornamento dei ResultSet
 - richiesta di metadati
 - gestione di transazioni

Transazioni in JDBC

- Scelta della modalità delle transazioni: un metodo definito nell'interfaccia `Connection`:

`setAutoCommit(boolean autoCommit)`

- `con.setAutoCommit(true)`
 - (default) "autocommit": ogni operazione è una transazione
- `con.setAutoCommit(false)`
 - gestione delle transazioni da programma
 - `con.commit()`
 - `con.rollback()`
 - non c'è `begin transaction`