Stored Procedures & Programming Interfaces

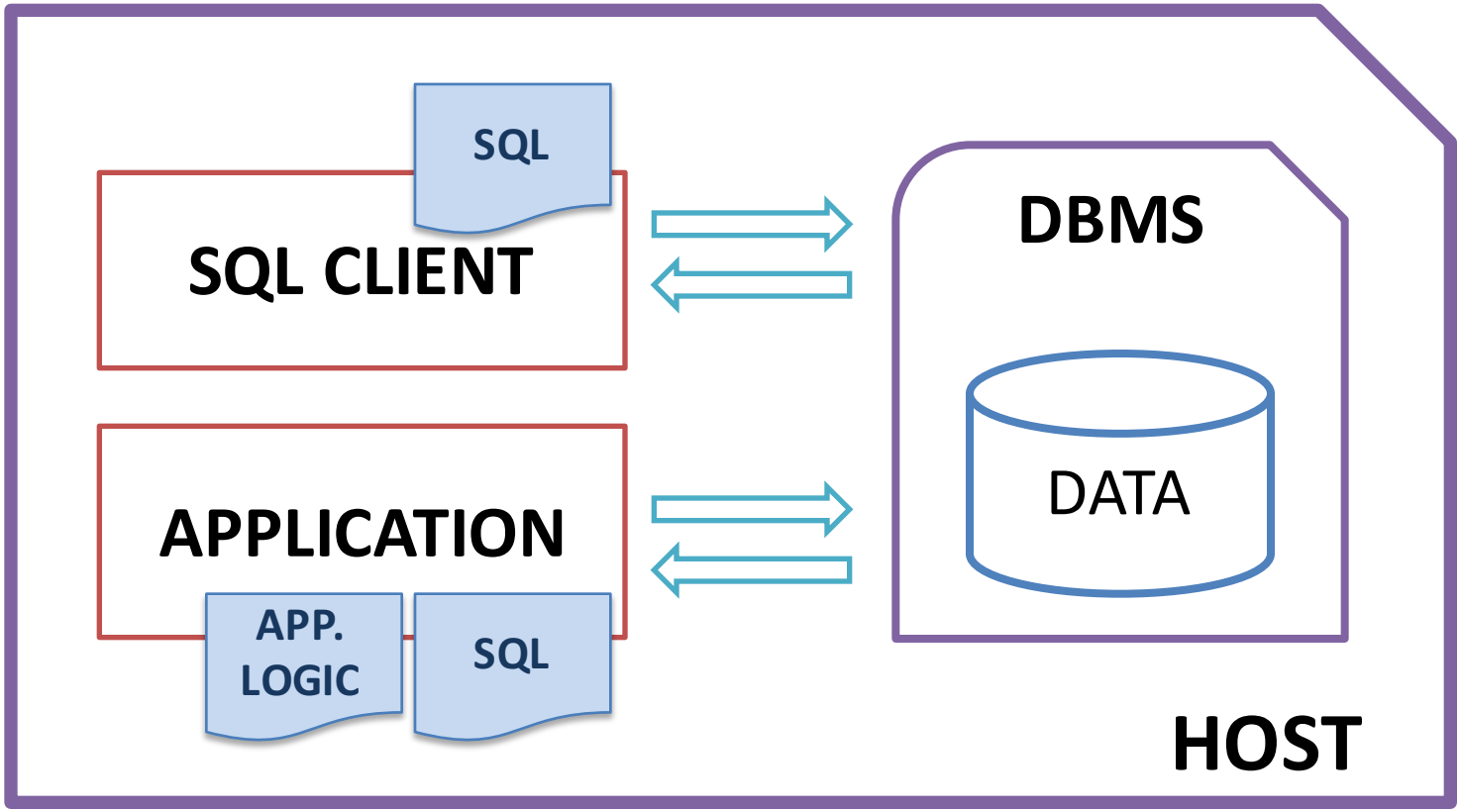
Corso di Basi di Dati

A.A. 2015/2016

Presented by
Matteo Lissandrini
ml@disi.unitn.eu

Courtesy of:
Francesco Corcoglioniti
corcoglioniti@disi.unitn.it

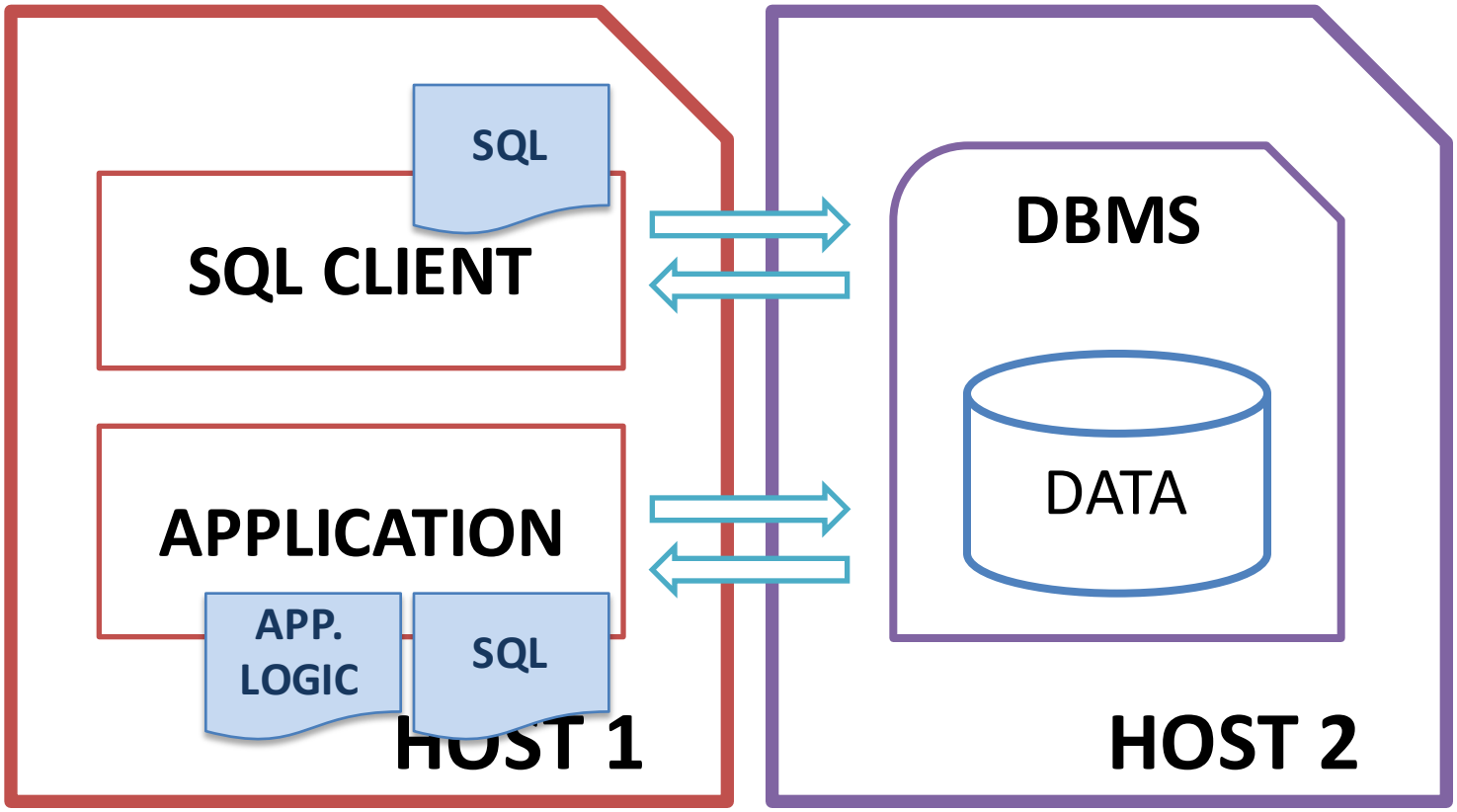
Client/Server & DBMS



Setup Locale: test, cheap hosting…

```
host$ psql -h localhost -U $USERNAME $DATABASE
```

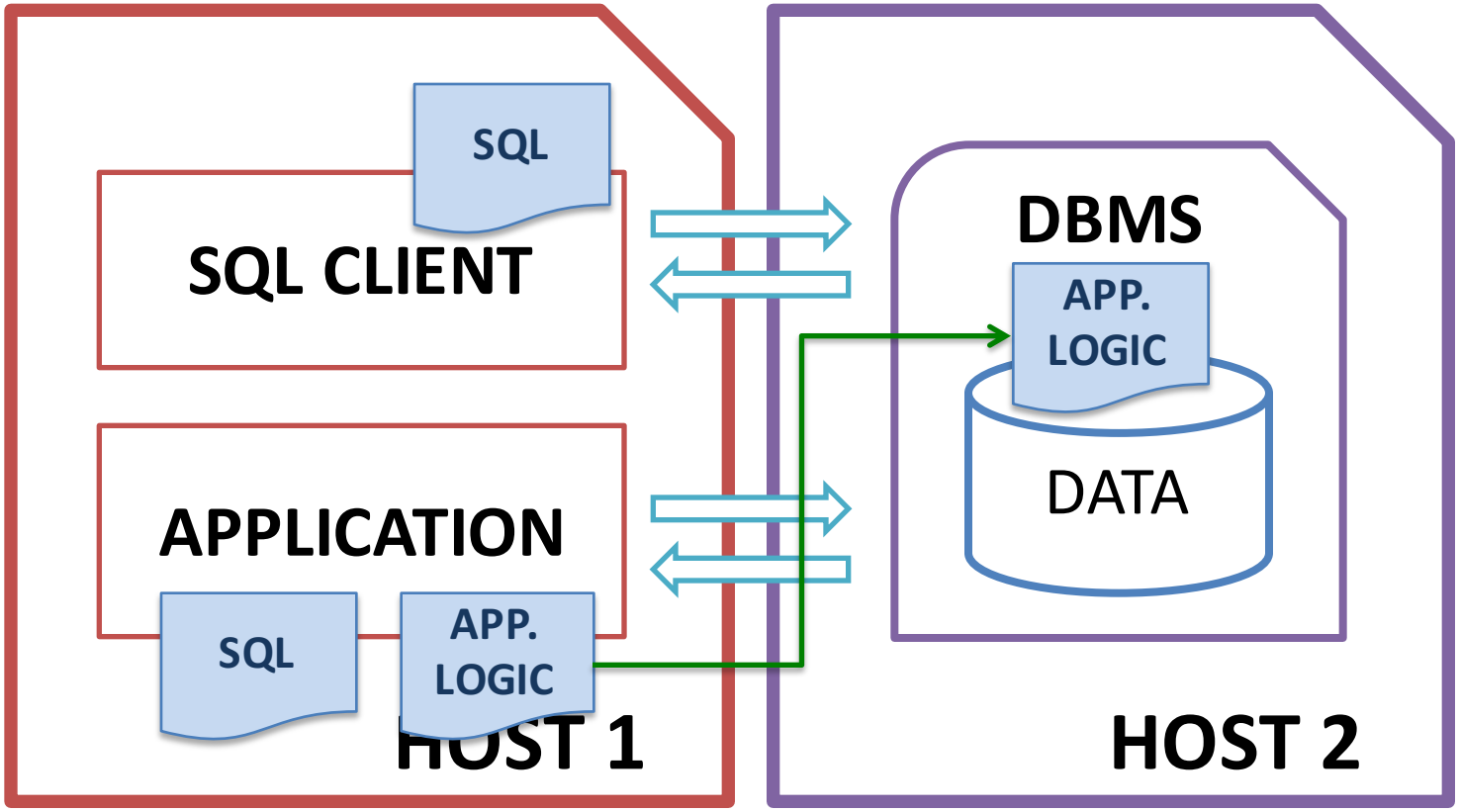
Client/Server & DBMS



Setup Enterprise: high availability, replication, restricted intranet access

```
host1$ psql -h host2 -U $USERNAME $DATABASE
```

Client/Server & DBMS + Procedural language



Parte della logica di controllo viene eseguita direttamente all'interno del DBMS

Procedural language for the PostgreSQL database system

- può essere utilizzato per creare funzioni e procedure trigger,
- aggiunge strutture di controllo per il linguaggio SQL,
- in grado di eseguire calcoli complessi,
- eredita tutti i tipi definiti dall'utente, funzioni e operatori,
- può essere definito come attendibile dal server,
- è facile da usare.

Le funzioni create con PL / pgSQL possono essere utilizzate ovunque una funzione built-in potrebbe essere utilizzata

Per “attivare” PL/pgSQL:

```
host$ createlang plpgsql -h hostname
```

```
CREATE LANGUAGE plpgsql;
```

Stored Procedures in PostgreSQL

- PostgreSQL supporta le **stored procedures** (o PSM)
  - il linguaggio è **plpgsql** la cui sintassi differisce in parte da quella del libro
  - in PostgreSQL non c'è distinzione tra funzioni e procedure: le seconde sono semplicemente funzioni che ritornano VOID

- Per definire una funzione (procedure):

```
CREATE FUNCTION name (
  parameter1 type1,
  ... )
RETURNS type AS $$
DECLARE
  variable1 type1,
  ...
BEGIN
    ... statements ...
    [RETURN expression;]
END;
$$ LANGUAGE plpgsql;
```

Dichiarazione parametri funzione/procedura

Dichiarazione tipo di ritorno, usare VOID per le procedure

Dichiarazione variabili e **cursori**. Il tipo speciale RECORD si usa per variabili che contengono una tupla.

Corpo della funzione/procedura. La return comporta l'uscita immediata dalla funzione (diversamente da libro) e può comparire più volte nelle funzioni (tipo ritornato <> VOID).

plpgsql è il linguaggio con cui è scritta la funzione corrispondente a SQL – PostgreSQL supporta l'uso di linguaggio diversi da SQL

Sintassi PLPGSQL (1)

http://www.postgresql.org/docs/9.2/static/plpgsql.html

- Blocchi di istruzioni

```
BEGIN
   ... Statements ...
END;
```

- Assegnamento variabile

```
variable := expression;
```

- Query che ritornano una sola tupla

```
SELECT ... INTO record_variable FROM ...
```

```
SELECT ... INTO variabile1, ..., variabileN FROM ...
```

- Costrutto IF/THEN/ELSIF/ELSE

```
IF condition THEN statements
[ ELSIF condition THEN statements ]…
[ ELSE statements ]
END IF;
```

Sintassi PLPGSQL (2)

- Costrutto CASE

```
CASE expression
    WHEN expression [, expression  ... ] THEN statements
    ...
  [ ELSE statements ]
END CASE;
```

- Cicli

```
LOOP ... statements ... END LOOP [label];
```

```
WHILE condition LOOP ... statements ... END LOOP [label];
```

```
FOR var IN [REVERSE] exp..exp  [BY exp] LOOP ... stmts ... END LOOP [label];
```

```
FOR var_1, ..., var_N IN query LOOP ... statements END LOOP [label];
```

```
FOR row_variable IN query LOOP ... statements END LOOP [label];
```

- EXIT e CONTINUE (usabili in ogni ciclo)

```
EXIT [label] [WHEN condition];
```

```
CONTINUE [label] [WHEN condition];
```

Sintassi PLPGSQL (3)

- Dichiarazione cursori (nella clausola DECLARE della funzione)

```
DECLARE
   ...
   cursor_var CURSOR FOR ... query ...;
```

- Apertura cursori (necessaria prima dell'uso)

```
OPEN cursor_var;
```

- Fetching tupla successiva

```
FETCH cursor_var INTO row_variable;
FETCH cursor_var INTO var1, ..., varN;
```

Subito dopo una fetch la variabile speciale boolean **FOUND** *è messa a* **TRUE** *se una tupla è stata letta, a* **FALSE** *se non c'èrano più tuple da leggere*

- Modifica tupla corrente nel database (solo con certe query)

```
UPDATE table SET ... WHERE CURRENT OF cursor_var;
DELETE FROM table WHERE CURRENT OF cursor_var;
```

- Chiusura cursori (necessaria al termine)

```
CLOSE cursor_var;
```

Sintassi PLPGSQL (4)

- Logging messaggi diagnostici

  RAISE NOTICE pattern, expression$_1$, ..., expression$_N$

  pattern è una stringa '...' che contiene il carattere % ogni qual volta si vuole inserire il valore di una espressione

- Interruzione processing sollevando eccezione

  RAISE EXCEPTION pattern, expression$_1$, ..., expression$_N$;

- Gestione eccezioni

```
BEGIN
  ... statement ...
EXCEPTION
  WHEN condition THEN ... statements ...
  WHEN condition THEN ... statements ...
  ...
END
```

la condizione può essere l'espressione **SQLSTATE 'codice'** *per gestire una particolare codice di errore di SQL (codice 02000 = non ci siano dati).*

Esempio 0

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.23;
END;
$$ LANGUAGE plpgsql;
```

Per eseguire una procedura, si può inserirla in una query, es:

```
SELECT sales_tax(100);

SELECT sales_tax(SUM(price)) FROM shop.sales;
```

Esempio 1

blocchi (scopes) & variabili

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
-- This is the outerblock
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50
    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Esempio2

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

Assumendo che esistato due tabelle: ***table1*** *e* ***table2***

La procedura ritorna un'unica stringa
Risultato della concatenazione di alcuni campi (f1, f5) di ***table1*** *e altri campi (f3, f7) di* ***table2***

Every table has an associated composite type of the same name
Thus, it does not matter in PostgreSQL whether you write %ROWTYPE or not.
But the form with %ROWTYPE is more portable.

Esercizio 1 – Stored Procedures

Sia dato lo schema:

customer (username, password, name, surname)
account (number, customer, amount)
transfer (source, target, amount, date)

Si scrivano delle stored procedures per effettuare le seguenti operazioni, usando la sintassi di PLPGSQL in PostgreSQL:

1. Inserimento cliente - dati username, password, nome e cognome, inserire un nuovo cliente con tali dati e creare un conto corrente con importo 0 per il cliente; il numero del nuovo conto deve essere scelto maggiore ai numeri di conto esistenti
2. Prelievo denaro - dato username, password, numero conto e importo richiesto, registrare il prelievo da bancomat dell'importo richiesto, modificando il conto in modo da memorizzare il nuovo importo; prima di effettuare l'operazione, si verifichino username e password e si verifichi che il conto effettivamente è associato all'utente e dispone di un importo sufficiente per l'operazione
3. Accredito interessi - dati due tassi di interesse ed una soglia, accreditare gli interessi ai conti corrente registrati nel DB secondo il seguente criterio: per la parte di importo minore o uguale alla soglia, il primo tasso di interesse è utilizzato; per la parte eccedente viene usato il secondo tasso di interesse

Esercizio 1 – Soluzioni (1/6)

1. Inserimento cliente

```
CREATE OR REPLACE FUNCTION add_customer (
  _username VARCHAR(20),
  _password VARCHAR(20),
  _name     VARCHAR(20),
  _surname  VARCHAR(20) )
RETURNS void AS $$
DECLARE
  _number INTEGER;
BEGIN
  SELECT MAX(number) + 1 INTO _number
  FROM   account;
  RAISE NOTICE 'numero nuovo conto: %', _number;

  INSERT INTO customer (username, password, name, surname)
  VALUES (_username, _password, _name, _surname);
  RAISE NOTICE 'registrato utente % - % %',
        _username, _name, _surname;
```

[CONTINUA]

Esercizio 1 – Soluzioni (2/6)

[CONTINUA]

```
  INSERT INTO account (number, customer, amount)
  VALUES (_number, _username, 0);
  RAISE NOTICE 'registrato conto % per utente %', _number, _username;
END
$$ LANGUAGE plpgsql;

-- per testare --
SELECT add_customer('marco2', 'pwd', 'marco', 'bianchi');
```

Esercizio 1 – Soluzioni (3/6)

2. Prelievo denaro

```
CREATE OR REPLACE FUNCTION withdraw (
  _username VARCHAR(20),
  _password VARCHAR(20),
  _account  INTEGER,
  _amount   INTEGER )
RETURNS VOID AS $$
DECLARE
  _max_amount INTEGER;
BEGIN
  IF ( NOT EXISTS ( SELECT *
                    FROM   customer
                    WHERE  username = _username AND
                           password = _password ) ) THEN
    RAISE EXCEPTION 'Username/password non validi';
  END IF;

  SELECT amount INTO _max_amount
  FROM   account
  WHERE  number = _account AND customer = _username;
```

[CONTINUA]

Esercizio 1 – Soluzioni (4/6)

[CONTINUA]

```
  IF (_max_amount  IS NULL) THEN
    RAISE EXCEPTION 'Account % non valido per utente %',
                    _account, _username;
  END IF;

  IF (_amount > _max_amount) THEN
    RAISE EXCEPTION 'Impossibile prelevare € %. La disponibilità sul
                    conto % è di € %', _amount, _account, _max_amount;
  END IF;

  UPDATE account
  SET    amount = amount - _amount
  WHERE  number = _account;

  RAISE NOTICE 'Prelievo di € % effettuato. La nuova disponibilità sul
        conto % è di € %', _amount, _account, _max_amount - _amount;
END
$$ LANGUAGE plpgsql;

-- per testare --
SELECT withdraw('mario', 'mario', 150014, 1000);
```

Esercizio 1 – Soluzioni (5/6)

3. Accredito interessi

```
CREATE OR REPLACE FUNCTION add_interest (
  _rate1 FLOAT,
  _rate2 FLOAT,
  _threshold INTEGER )
RETURNS VOID AS $$
DECLARE
  _row RECORD;
  _interest FLOAT;
  _cursor CURSOR FOR SELECT *
                     FROM   account
                     WHERE  amount > 0;
BEGIN
  OPEN _cursor;

  LOOP
    FETCH _cursor INTO _row;
    EXIT WHEN NOT FOUND;
```

[CONTINUA]

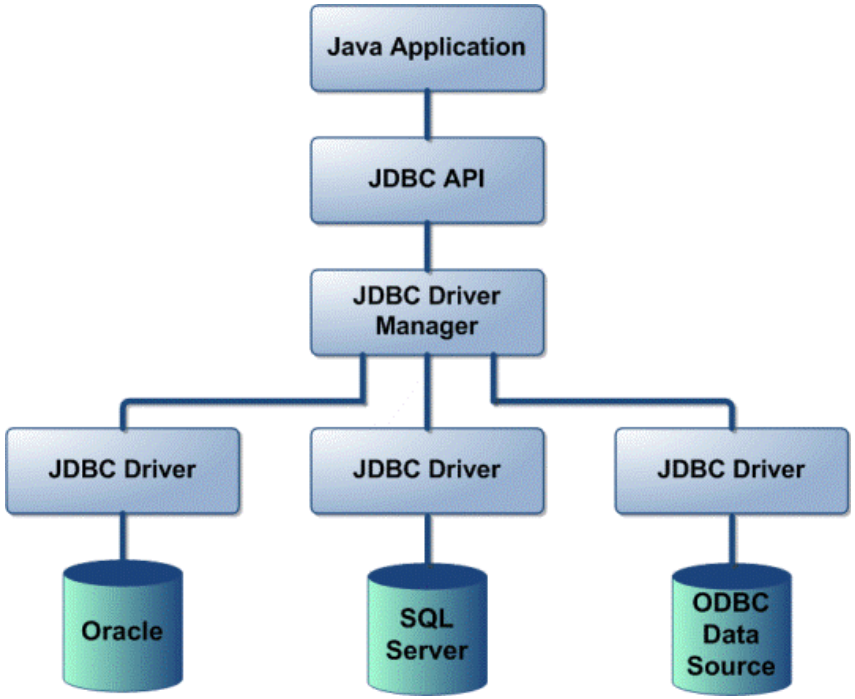
Esercizio 1 – Soluzioni (6/6)

[CONTINUA]

```
    IF (_row.amount < _threshold) THEN
      _interest = _row.amount * _rate1;
    ELSE
      _interest = _threshold * _rate1 +
                  (_row.amount - _threshold) * _rate2;
    END IF;

    UPDATE account
    SET    amount = amount + _interest
    WHERE  CURRENT OF _cursor;

    RAISE NOTICE 'aggiunti interessi € % a conto %',
                 _interest, _row.number;
  END LOOP;
END
$$ LANGUAGE plpgsql;

-- per testare --
SELECT add_interest(0.01, 0.02, 1000);
```

Java - JDBC

- JDBC (Java Data Base Connectivity) è una libreria per accedere a DB da applicazioni Java, astraendo dal particolare DBMS usato
- E' così possibile cambiare DBMS senza modificare il codice



Open Database Connectivity (ODBC) è una API standard per la connessione dal client al DBMS. Questa API è indipendente dai linguaggi di programmazione, dai sistemi di database e dal sistema operativo.

JDBC – Modell di Utilizzo

- Per usare JDBC in un'applicazione occorre:
  - procurarsi il driver per il particolare DBMS utilizzato
    - per PostgreSQL **`postgresql-X.X-XXX.jdbc4.jar`**
    - http://jdbc.postgresql.org/download.html
  - copiare il driver nella directory (o sotto-directory) dell'applicazione e assicurarsi che sia accessibile nel CLASSPATH
- Il modello di utilizzo di JDBC è il seguente
  - l'applicazione carica il driver JDBC
  - l'applicazione stabilisce una **connessione** ad un DB su un certo DBMS, fornendo le credenziali di autenticazione (username e password)

    ➔ ogni database è identificato da una URL; per PostgreSQL le URL sono del tipo **`jdbc:postgresql://host:port/database`**
  - la connessione rappresenta una sessione di lavoro al cui interno l'applicazione può mandare uno o più **statement** SQL al DBMS
  - Gli statement corrispondenti a query ritornano un **result set** che è un cursore che permette di iterare sulle tuple estratte

JDBC – Esempio di Codice

```
Class.forName("org.postgresql.Driver");
Connection conn = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/corsodb",
    NOME_UTENTE, PASSWORD);

Statement stmt = conn.createStatement();

int result = stmt2.executeUpdate(
    "INSERT_UPDATE_O_DELETE_IN_SQL");

<oppure>

ResultSet rs = stmt.executeQuery(
    "SQL_SELECT_QUERY");
while (rs.next()) {
  int attributo1 = rs.getInt(1);
  String attributo2 = rs.getString(2);
}

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

Stabilisce una connessione al DB

Crea uno statement per comando o query

Comando di modifica

Comando di query

Cicla sui risultati della query, una tupla alla volta

Ricava i campi per ciascuna tupla

Rilascia le risorse allocate

JDBC – Classi Principali

Classe	**Descrizione**	**Metodi/funzioni principali**
DriverManager	Permette di acquisire una connessione al database.	Connection conn = DriverManager.getConnection(URL_DB, NOME_UTENTE, PASSWORD)
Connection	Permette di creare statement per eseguire query o operazioni di modifica sul DB.	Statement stmt = conn.createStatement();
Statement	Permette di eseguire query o operazioni di modifica.	int numTuple = stmt.executeUpdate(SQL); ResultSet rs = stmt.executeQuery(SQL);
ResulSet	Permette di accedere ai risultati di una SELECT.	rs.next() – passa alla tupla successiva rs.getX(POS) – ritorna il campo POS di tipo X

Esercizio 2 – JDBC

Sia dato lo schema:

customer (username, password, name, surname)
account (number, customer, amount)
transfer (source, target, amount, date)

Si scrivano dei programmi Java per implementare le seguenti operazioni:

1. Stampa elenco conti corrente - connettersi al DB e stampare a video l'elenco dei conti corrente memorizzati nel sistema; per ogni conto stampare nome e cognome del proprietario, numero di conto e importo depositato.
2. Bonifico interno – connettersi al DB, quindi chiedere all'utente di
   - autenticarsi, controllando username e password immessi;
   - selezionare un conto ad egli intestato;
   - selezionare il conto verso cui effettuare il bonifico, ad esempio richiedendo nome e cognome del beneficiario e quindi chiedendo di selezionare il conto del beneficiario tra quelli ad egli intestati
   - selezionare l'importo da trasferire, verificando che sia inferiore all'importo depositato sul conto di partenza
   - effettuare il bonifico, aggiornando le disponibilità dei due conti e memorizzando l'operazione tramite aggiunta di nuova tupla a transfer

Esercizio 2 – Soluzioni (1/2)

1. Inserimento cliente

```
import java.sql.*;

public class CustomerList {

  private static final String DB_USERNAME = "studente";
  private static final String DB_PASSWORD = "studente";
  private static final String DB_DRIVER = "org.postgresql.Driver";
  private static final String DB_URL =
                          "jdbc:postgresql://localhost:5432/corsodb";

  public static void main(final String[] args) throws Throwable {

    Class.forName(DB_DRIVER);
    Connection connection =
        DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);

    try {
      Statement stmt = connection.createStatement();
      try {
        ResultSet rs = stmt.executeQuery(
          "SELECT surname, name, number, amount " +
          "FROM bank.customer c JOIN bank.account a " +
          "                     ON c.username = a.customer " +
          "ORDER BY surname, name, number");
```

[CONTINUA]

Esercizio 2 – Soluzioni (2/2)

```
        try {
          System.out.println(
          "STAMPA ELENCO CONTI CORRENTE\n\n"
          +" Cognome              | Nome                 | Conto    | Importo\n"
          +"----------------------|----------------------|----------|----------"
          );
          while (rs.next()) {
            String surname = rs.getString(1);
            String name = rs.getString(2);
            int account = rs.getInt(3);
            int amount = rs.getInt(4);
            System.out.println(String.format(" %-18s | %-18s | %8d | %8d",
                surname, name, account, amount));
          }
        } finally {
          rs.close();
        }
      } finally {
        stmt.close();
      }
    } finally {
      connection.close();
    }
  }
}
```

Python - DB-API

- Questa API è stata definita per favorire somiglianza tra i moduli Python che vengono utilizzati per accedere ai database.
  - http://www.python.org/dev/peps/pep-0249/
- Fornisce una interfaccia standard di Python ai database
- Python DB API **non** è …
  - Una singola interfaccia in cui caricare driver
  - Un pacchetto software che è possibile scaricare e installare
- L'API DB utilizza due concetti principali per l'elaborazione delle query di database:
  - Oggetti connessione
    - Connessioni Network / RPC al DB
    - Transazioni
  - Oggetti Cursore
    - Esecuzione degli Statement
    - Accesso ai risultati

Python – Esempio di Codice

Caricamento DBAPI dal pacchetto pg8000

```
from pg8000 import DBAPI
```

Connessione al Database

```
conn = DBAPI.connect(host="hostname", user="xxx", password="yyy")
```

Cursore – dall'oggetto connessione ottenuto precedentemente

```
cursor = conn.cursor()
```

Esecuzione di Query

```
cursor.execute("CREATE TEMPORARY TABLE book (id SERIAL, title
TEXT)")

cursor.execute("SELECT id, title FROM book")

cursor.execute("INSERT INTO book (title, year) VALUES (%s, %s)",
                                ("Ender's Game", 1985))
```

Possono essere eseguite query di ogni tipo: CREATE-DELETE-UPDATE-SELECT.
Le query che modificano i dati necessitano poi del commit (vedi seguente)
Con la **execute** viene eseguita un'unica query, ai segnaposti **%s** vengono sostituiti i valori nella tupla `("Ender's Game", 1985)`
Ad ogni segnaposto deve corrispondere un valore, nell'ordine in cui sono inseriti nella stringa.

Python – Esempio di Codice

Esecuzione di Query – Esecuzioni multiple

```
cursor.executemany("INSERT INTO book (title, year) VALUES (%s,
%s)",
        [
                ("Ender's Game”, 1985 ),
                ("Speaker for the Dead”, 1986)
         ])
```

Nei casi di INSERT, DELETE o UPDATE può essere necessario eseguire una query per diverse tuple.
In questo caso **executemany** esegue la query più volte per ogni tupla riportata nella lista degli argomenti.

Numero di Risultati

```
print cursor.rowcount
```

Commit

```
conn.commit()
```

Python – Esempio di Codice

Accesso ai risultati

```
cursor.execute("SELECT id, title FROM book")

rows = cursor.fetchall()

for row in rows:
     id, title = row
     print "id = %s, title = %s" % (id, title)
```

Dopo l'esecuzione della query, nell'oggetto cursor risiede il puntatore all'insieme delle tuple ritornate dalla query.

Attraverso **fetchall()** vengono ritornate tutte come una lista,

```
cursor.execute("SELECT  nome FROM users  where id = %s”, (12345))
user = cursor.fetchone()
print user[0]
```

Rilascio delle Risorse / Chiusura Connessione

```
cursor.close()
conn.close()
```

Terminare la connessione e rilasciare le risorse è molto importante!! Il numero di connessioni contemporanee viene limitato dal sistema, rilasciando le risorse quando non sono necessarie permetterà il riutilizzo più veloce delle connessioni e migliora le prestazioni.

Python

- Tutorial: http://docs.python.it/paper-a4/tut.pdf

- Programmi python possono essere scritti e salvati da un normale editor di testo con estensione **.py**
- Una volta salvati, dalla linea di comando basta digitare
  - `python nome_programma.py`
    (nome_programma.py si riferisce al nome del file che avete salvato)

- Per Windows
  - control panel > system > advanced > |Environmental Variables| > system variables -> PATH
  - In fondo al campo della variabile PATH aggiungere la direcotry di installazione di python
    - ES: C:\Python26;

Esercizio 3 – Python

Sia dato lo schema:

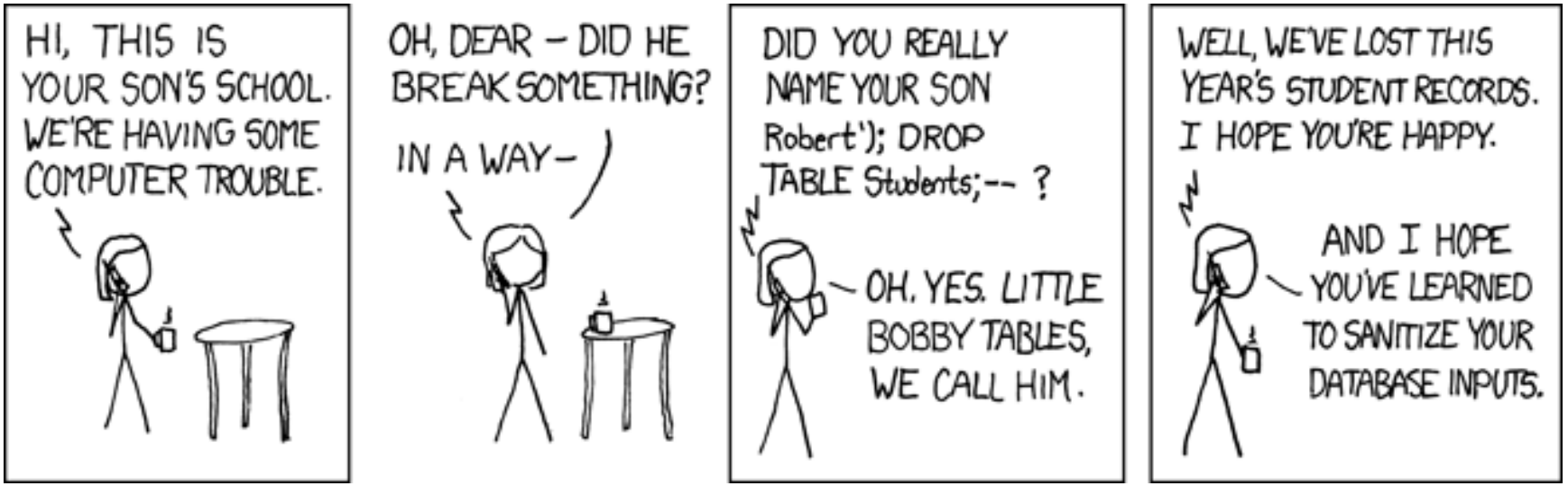
customer (username, password, salt)
book (id, title, quantity)
registry(user, book, date, returned)

Si scriva il codice Python che implementa le seguenti operazioni:

1. Stampa elenco libri disponibili - connettersi al DB e stampare a video l'elenco dei libri con quantity diversa da zero

2. Stampa elenco libri utente: dato un nome utente, stampare titolo e data di noleggio solo se non e' stato gia' restituito

3. Noleggio – connettersi al DB, quindi chiedere all'utente di
   - Autenticarsi, controllando username e password immessi;
   - Stampare elenco libri disponibili : titolo e id
   - Selezionare l'id del libro da noleggiare
   - Diminuire la quantita' se diversa da zero e aggiungere una tupla per registrare il noleggio
   - Verificare le condizioni necessarie e segnalare gli errori



https://xkcd.com/327/

Esercizio 4

TPCH – Schema

- https://bitbucket.org/tkejser/tpch-dbgen/src
- Scaricare i file:
  - postgres_dll.sql
  - postgres_ri.sql
- Eseguire i file in psql
- Scrivere il codice plpgql, oppure Java, oppure Python per popolare il database
  - Maggiori dettagli su
  - http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf