DATABASE PROGRAMMING

System architecture

Users and applications use the data which is stored in database

- Details such as SQL queries should be hidden from end users if it's possible
- Data stores should be interchangeable - replacement of one (R)DBMS with another (R)DBMS should have just minimal effects on the applications.
- (R)DBMS should be a central entity for any data access

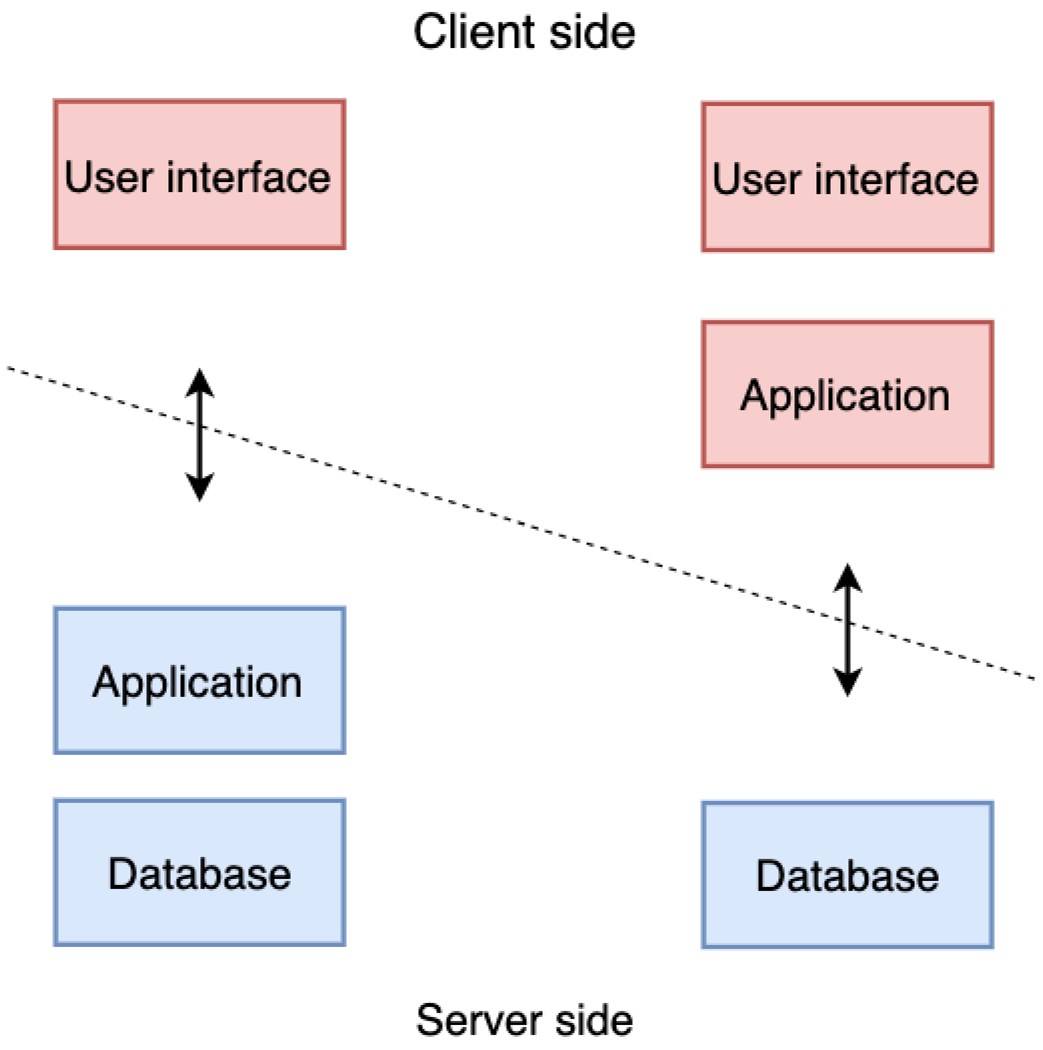
Numerous *database system architectures* are created during database management history

Basic Client/Server architectures

Client/Server architecture was developed to deal with computing environment which is connected with a network

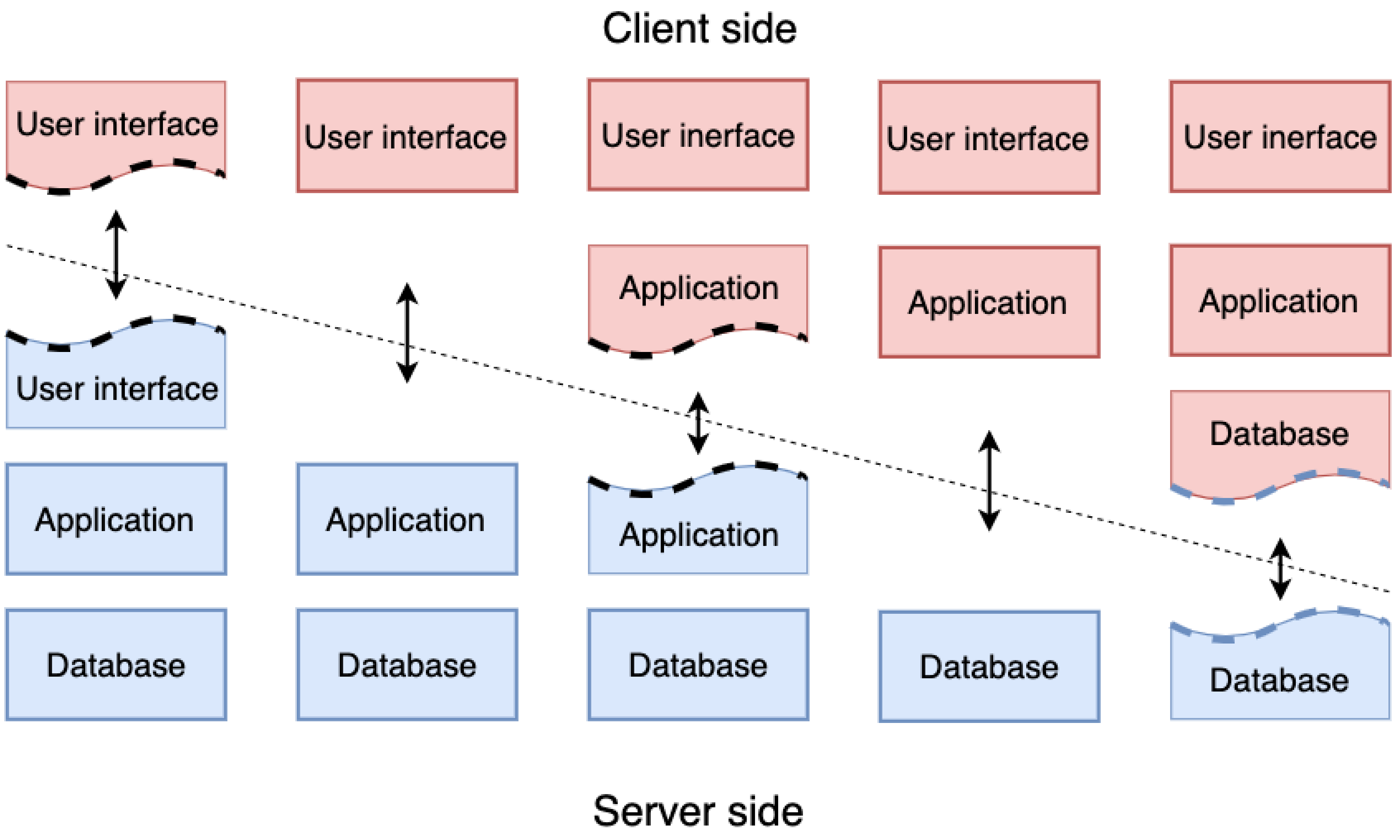
- a **client** is typically a user machine which provides user interface capabilities and to some extent local processing
- a **server** is a system which contains hardware and software that can provide services to the client machines such as file access, email services, archiving or **database access**

Simple Client/Server architecture



Two simple cases of **2-tiered architecture**

Two-tiered architecture



According to: *Distributed Systems 3rd edition (2017)* M. van Steen and A.S. Tanenbaum.

Properties of the basic 2-tiered arechitecture

Programming languages: often 4GL languages

- developed for data centered applications
- mostly automatic creation of user interfaces according to the models
- fast development for simple general cases

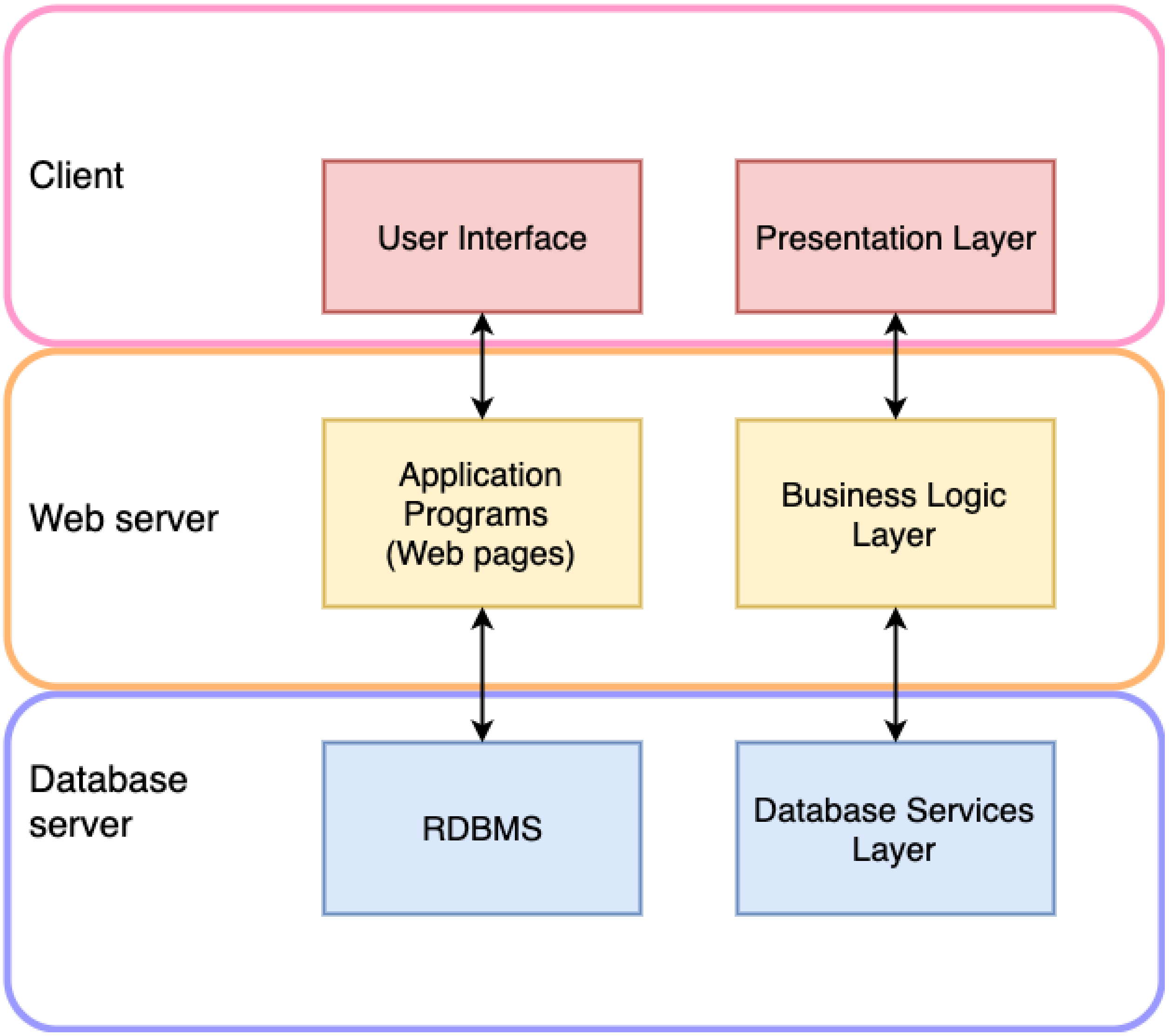
Drawbacks of this architecture

- fixed binding of two tiers
- individual components cannot be replaced without changing other components
- big systems quickly become very complex

Being very complex for maintenance and not flexible these 2-tiered architectures are nowadays rarely used

3-tiered architectures are proposed to overcome the shortcomings of 2-tier architectures

3-tiered architectures



3-tiered architectures

Many web applications use 3-tier architecture which adds a **middle tier** layer between the client and database server

- middle tier can be **application server** or the **Web server**
- intermediary role
  - running application programs
  - storing business rules (procedures and constraints)
- checking user credentials before forwarding requests to the database server
- passing partially processed data from the database server to clients

The following layers corresponding to tiers are distinguished

- **Presentation** layer - user interface
- **Business Logic** layer - application rules
- **Database Services** layer - data access

Approaches to Database Programming

Database interactions can be included in application programs in different ways:

1. **Embedding database commands** in general-purpose programming languages
   - database statements embedded into a host programming language (*EXEC SQL* statement)
2. Using **library of database functions**
   - functions to connect to a database and execute queries (queries and necessary information are included as parameters of function calls)
   - this approach provides **application programming interface (API)** for accessing a database
3. Designing **database programming language** with database model, queries and additional structures
   - additional structures include programming structures such as branching, conditions and loops (examples procedural languages such as Oracle's PL/SQL)

Impedance mismatch

Problems emerging due to the difference between database model and the programming language model

- data types of programming language can differ from the attribute data types
  - *binding* of programming language data types to compatible language types
- mapping of *query results* to *programming language structures*
  - results of queries are sets or multisets of tuples (which are sequences of attribute values)
  - **cursor** or **iterator variable** is a mechanism which loops over query results and extract values to distinct program variables
    - cursor can be seen as a pointer to a single tuple from the result (which contains more tuples)

Embedded database commands

Embedded SQL is an approach where query text is written within the program source code

- it's also called **static** database programming

Advantages of this approach

- query text is part of the program source code and can be validated over schema at compile time
- program is quite readable

Disadvantages:

- lack of flexibility to change programs at runtime
- changing of queries is going through the whole compilation process
- not really convenient for complex applications

Embedded SQL example

The example presents C program segment which reads student information from the database:

– SQL statement begins with **EXEC SQL**

```
loop = 1 ;
while (loop) {
  prompt("Enter a students ssn: ", ssn) ;
  EXEC SQL
  select first_name, last_name, address, year
  into :fname, :lname, :year, :address, :year
  from STUDENT where Ssn = :ssn ;

  if (SQLCODE == 0) printf(fname, lname, address,  year)
  else printf("SSN does not exist: ", ssn) ;
  prompt("More students (enter 1 for Yes, 0 for No): ", loop) ;
}
```

– **into** clause specifies program variables which are populated with data from the database

– program variables are prefixed with colon(:) when they are used inside of SQL statements to be distinguished from database attributes

Database programming with function calls (APIs)

This approach represents more dynamic approach for database programming

- a library of functions known as application programming interface (API) is provided to interact with a database
- more dynamic approach to database programming than embedded SQL

Advantages

- provides more flexibility because no preprocessor is needed
- function call interface makes it easier accessing multiple databases in the same application program (even from different DBMS vendors)

Disadvantages

- syntax and other checks on SQL commands are done at runtime
- requiring more complex programming to access query results

Accessing databases

Standards for connection to databases (**application programming interface (API)**)

- ODBC - integration to SQL trough a common library of functions
  - provides classical API for communication with DBMS
- OLEDB -Object Linking and Embedding successor of ODBC
- DAO - Data Access Objects
- ADO - ActiveX Data Objects - access to tabular data sources (RDBMS, CSV, etc.)
- ADO.NET - Microsoft .NET framework provided set of software components for accessing databases
- JDBC - Java Database Connectivity - similar to ODBC and developed for Java Programming language and Java virtual machine
- DB-API - SQL API for Python programming language

ODBC Open Database Connectivity

ODBC provides classical API for communication with DBMS

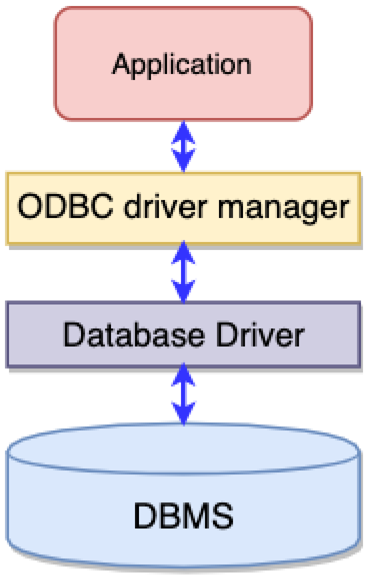
– most RDBMS vendors provide ODBC drivers for their systems

– application developers write the logic to a generic DBMS interface

– loadable **drivers** map the code to vendor-specific commands

– ODBC is based on binary libraries (usually written in C)

– nowadays, thin clients using HTML reduce the need for ODBC



JDBC - Java Database Connectivity

JDBC represents function libraries (API) for calling SQL functions using Java programming language

- Java is designed to be platform independent
  - independent of platform, vendor and DBMS
- function libraries are implemented as classes because Java is object-oriented
- API enables dynamic queries
- provides binding of data types for Java/DB impedance mismatch
  - Result set - retrieves rows and columns and some additional metadata

JDBC - Java Database Connectivity

JDBC API represents a programming interface for database connectivity

- RDBMS vendors provide **JDBC drivers** so that it is possible to access their systems via Java programs

JDBC driver is an implementation of the JDBC interface responsible for the communication with a specific database

- driver is dependent on the RDBMS vendor

JDBC - example

```
import java.io.* ;
import java.sql.*
...
class getStudentInformation {
   public static void main (String args []) throws SQLException, IOException {
     ... //loading DriverManager
     String user, year, passwrd, ssn, fname, lname ;
     passwrd = readentry("Enter password:") ;

     Connection conn = DriverManager.getConnection
       ("jdbc:postgresql//localhost/university?user=" + user + "&password" + passwr
     String stmt = "select first_name, last_name, year " +
                    "from STUDENT where Ssn = ?" ;
     PreparedStatement p = conn.prepareStatement(stmt) ;
     ssn = readentry("Enter a Social Security Number: ") ;
     p.clearParameters() ;
     p.setString(1, ssn) ;

     ResultSet r = p.executeQuery() ;
     while (r.next()) {
        fname = r.getString(1) ;
        lname = r.getString(2) ;
        year  = r.getInterger(3) ;
        System.out.println("Student " + fname +" " year) ;
      } }
}
```

DB-API (DBAPI-2.0) - SQL API for Python

DBAPI is a specification of libraries for calling SQL functions using Python

- PEP-0249 - Python Database API
- definition of APIs to encourage similarity between Python modules
- each RDBMS has to provide implementations of specific function calls (similar to JDBC)

RDBMS	**Implementation**
PosgreSQL	psycopg, pyPgSQL
Oracle	dc_oracle2k, cx_oracle
DB2	Pydb2
MySQL	MySQLdb
SQLite	sqlite3

- this is *only a specification* that should improve code portability across different DBMSs
- DB-API 2.0 doesn’t have a code on it’s own and used implementation of SQLite3 RDBMS as a reference

DBAPI - SQL API for Python

DBAPI-compliant modules (implementations of the specification) have the following elements

- imported module calls **connect()** function with parameters representing the connection string
  - connect() function should be a constructor returning **connection object**

Connection objects represents a channel for the communication with a database

- DBAPI assumes that the transaction is always in progress
  - there is no explicit begin() method for a transaction
- connection should implement methods
  - commit() - commits any pending transaction
  - rollback() - rolls back the state to the beginning of any transaction
  - close() - after this command, connection cannot be used and rollback is performed on changes which were not committed
  - cursor() - returns a cursor object if it’s implemented in the database

DBAPI - SQL API for Python

Cursor is a pointer to the memory where the data fetched from the database tables are kept once the query is executed.

- cursor can be thought of as a *pointer* to a single tuple from a *result set* which contains many tuples
- manages the context of the *fetch* operation
- cursors created by one single connection are not isolated
  - cursors can see the changes made from other cursors created by the same connection

Important cursor methods

- execute() - prepares and executes a database operation

```
cursor.execute(“select * from Student”)
```

  - cursor brings the query result from the database and sets the pointer at the beginning
  - cursor executes both DDL and DQL statements

DBAPI - fetching rows form the result set

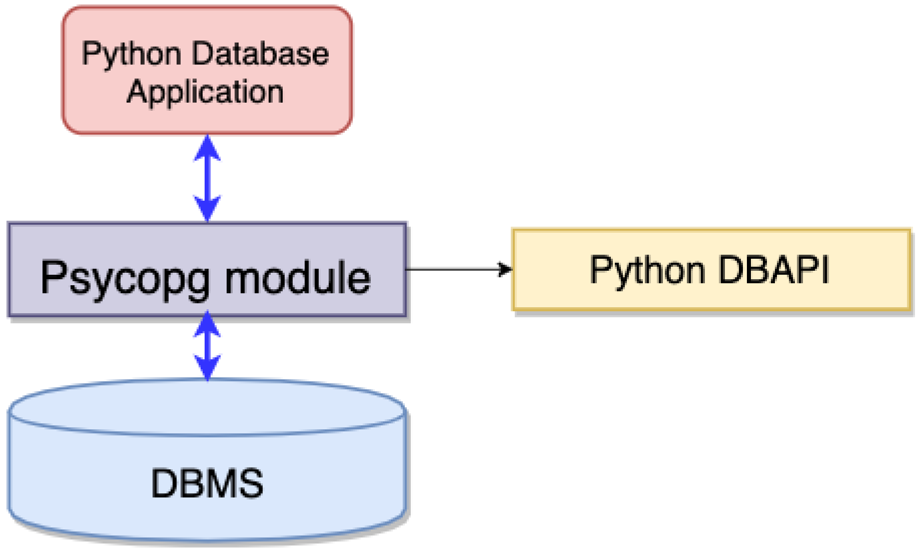
Important methods used by cursor to extract rows from the result set:

- *fetchone()* - fetches the next raw of the query result
  - corresponds to the fetch command in other languages
- *fetchmany([size])* - returns specified number of tuples from the result set
  - parameter *size* represents that number of rows that are fetched
- *fetchall()* - retrieves a sequence of all tuples rows in the result set

Psycopg2 - implementation of DB-API

Psycopg2 is a popular PostgreSQL adapter(driver) for Python programming language

- contains complete implementation of the Python DB API 2.0 specification
- mostly implemented in C and it’s very efficient and secure



Psycopg2

Psycopg is designed to be used by multi-threaded applications

- provides a system to adapt Python objects to the SQL syntax
- provides *type casting* functions to convert a PostgreSQL type to a Python object
- implements exceptions specified in DB-API such as subclasses of *DatabaseError*
    - *ProgrammingError* - table not found or already exists, syntax errors in SQL statements
    - *OperationalError* - raised for unexpected disconnect, transaction could not be processed

Psycopg2 has additionally implementation of other capabilities which are characteristic for new versions of PostgreSQL

- implements some additional objects and extends standard set of functionalities of DBAPI

Psycopg2 - executing DDL statement

Example with the execution of a create table command

```
imoport psycopg2
from psycopg2 import Error

conn_string = "host="+PGHOST+" port="+"5432"+" dbname="+PGDATABASE+ \
            " user="+PGUSER + \
            " password="+PGPASSWORD
conn=psycopg2.connect(conn_string)

try:
    create_table_query = "create table Student
          (id int primary key ,
           fname varchar(30)  not null,
           year int); "

    cursor.execute(create_table_query)
    connection.commit
    cursor.close()
except (Exception, psycopg2.DatabaseError) as error:
    print(error)
finally:
    if conn is not None:
        conn.close()
```

Psycopg2 - executing DML statement

```
conn_string = "host="+PGHOST+" port="+"5432"+" dbname="+PGDATABASE+ \
              " user="+PGUSER + \
              " password="+PGPASSWORD
conn=psycopg2.connect(conn_string)
try:
    #fetchall(example)
    cur = conn.cursor()
    query1 = ("SELECT e.ssn, e.fname, e.lname, d.dname " \
              "FROM employee e join department d " \
              "    on e.dno = d.dnumber;")

    cur.execute(query1);
    employees = cur.fetchall()
    for e in employees:
        print(f"Employee: {e[0]} {e[1]} {e[2]} {e[3]}")
    cur.close()
except (Exception, psycopg2.DatabaseError) as error:
    print(error)
finally:
    if conn is not None:
        conn.close()
```

Object relational mapping

Object-oriented systems dominantly achieve persistence using relational databases

- differences between object-oriented model and relational model make challenge for programmers

Object Relational Mapping is an approach for mapping objects to incompatible database elements

- abstraction layer creates an effect of a *virtual object database*

Characteristics

- enables writing of SQL code by making use of features of an object-oriented language
- programmer should still think in terms of SQL but can write the object oriented code
- theoretically should enable easy switching between different RDBMS
    - it should be possible to use SQLite for local development and PostgreSQL in production

Object relational mapping

Provides more object-centric perspective opposed to schema centric perspective

The following example shows how to write a simple select query in Python using ORM

```
   students = session.query(Student).all()
```

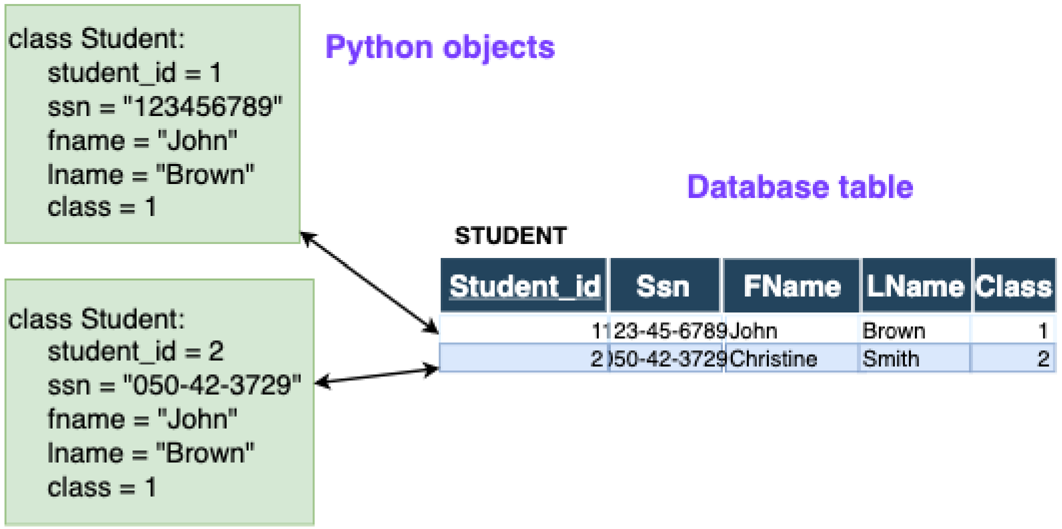
– is translated to the following query

```
SELECT * FROM Student
```

– SQL code is extracted from the object-oriented code and executed over the database

– result of the SQL code execution is wrapped up in objects and returned to the program

Object relational mapping



```
class Student:
    student_id = 1
    ssn = "123456789"
    fname = "John"
    lname = "Brown"
    class = 1
```

```
class Student:
    student_id = 2
    ssn = "050-42-3729"
    fname = "John"
    lname = "Brown"
    class = 1
```

STUDENT

Student_id	Ssn	FName	LName	Class
1	123-45-6789	John	Brown	1
2	50-42-3729	Christine	Smith	2

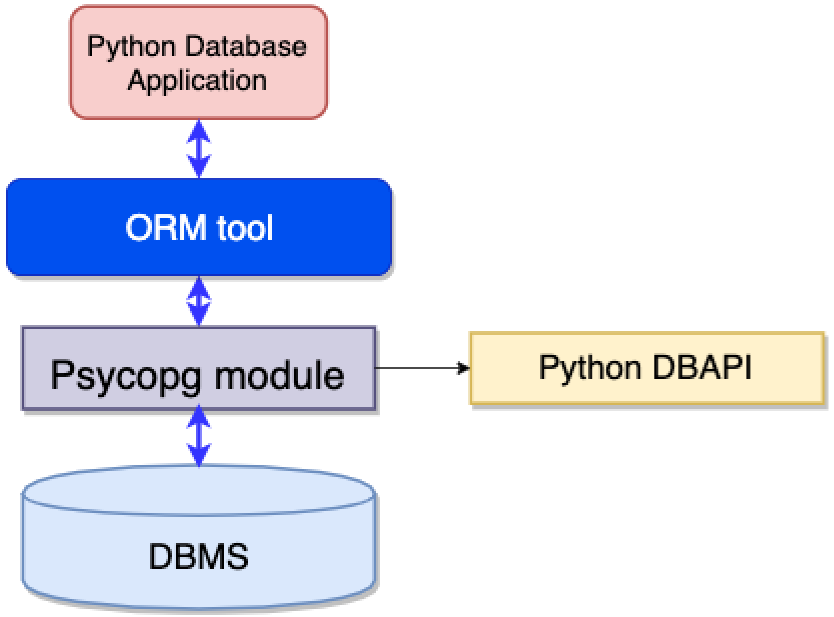
Advantages:

- reduces the amount of code that has to be written
- increases code readability and speed up application development (specially for the definition of prototypes)

Disadvantages:

- problems caused by the high level of abstraction
- due to *impedance mismatch* definition of a suitable mapping is sometimes challenging

Object-relational mapping - challenges



ORM tools can deal with the majority of modeling demands but not with all of them

- ORM modeling tend to be very complex for big projects
- improper modeling can lead to reduced performance
- good modeling with ORM requires deep understanding of ORM tools and SQL

Object-relational mapping tools

ORM approach is very popular and the majority of languages has ORM tools

- Java defines **JPA** (Java Persistence API) that provides guidelines for implementing ORM tools
    - **Hibernate** is the most popular ORM tools for java (other tools Open JPA, iBATIS etc.)

ORM with Python

- popular Python ORM tools are SqlAlchemy, Peewee ORM, Django ORM, …
- SQLAlchemy represents very well developed library for ORM
    - provides database agnostic code which is used for communication with databases

SQLAlchemy

SQLAlchemy is a well-developed object-relational mapper for Python programming language

- consists of two important parts, *Core* and *ORM*
  - Core deals with common CRUD operations on DBMS
  - ORM part is built upon the Core and takes care of mappings

SQLAlchemy implements DBAPI for interactions with databases which enables executing SQL queries as any other DBAPI adapter

- entry point to DBAPI implementation is of class *Engine* which is constructed by calling create_engine() method

```
    engine = create_engine("postgresql://user:pw;host/dbname")
    employees = engine.execute("select * from employee")
```

- cursor can be defined according to DBAPI and we can apply standard fetching functions

```
   with engine.begin() as conn:
       cursor = conn.execute("select * from employee")
       emp = cursor.fetchall()
```

SQLAlchemy - mapping definition

ORM part uses Table and Column classes to define mappings with corresponding elements in a relational database schema

```
class Employee(Base):
    __tablename__ = "employee"
    id = Column('employee_id', Integer, primary_key=True)
    first_name = Column('fname', String(45))
    last_name = Column('lname', String(45))
    dept_id = Column('department_id',Integer,
                        ForeignKey('department.department_id'))
```

- mapping is represented in a declarative manner
- model of a table is a Python class with attributes which match the column types of the corresponding database table
  - **tablename** represents the name of the table in the DBMS
  - ForeignKey() object specifies that a column is a foreign key to an attribute of the other table

SQLAlchemy - mapping definition

The second very important directive for the mapping is **relationship()**

- relationship() specifies the definition of objects members which will be used
  to access the other class in a relationship
- relationship() uses foreign keys to determine how this link between tables will behave

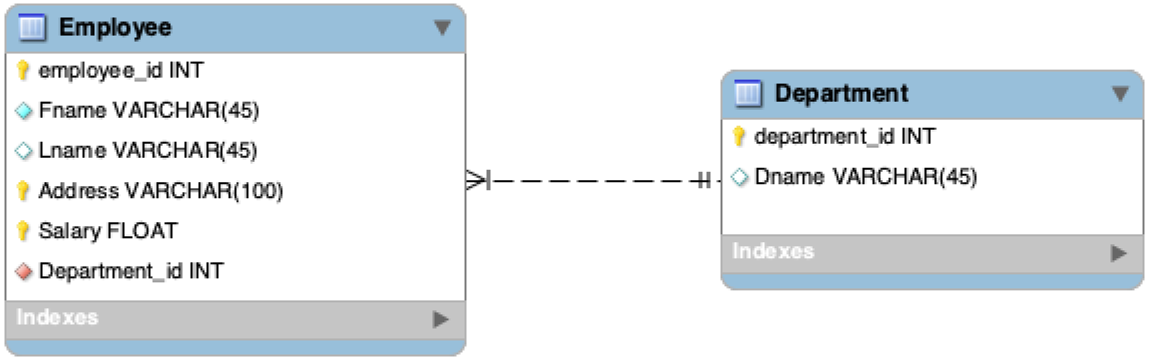
```
employees = relationship('Employee', cascade='all, delete', backref='department')
```

- backref specifies how the class member with that name will be populated in the related table

```
employees = relationship("Address",
                         order_by="desc(Address.email)",
                         primaryjoin="Address.user_id==User.id")
```

- primaryjoin can be used to specify explicitly how to generate objects

ORM - example of the one-to-many relationship



```
CREATE TABLE department (
        department_id INTEGER NOT NULL,
        "DName" VARCHAR NOT NULL,
        PRIMARY KEY (department_id)
)
CREATE TABLE employee (
        employee_id INTEGER NOT NULL,
        fname VARCHAR(45),
        lname VARCHAR(45),
        Address VARCHAR(100),
        department_id INTEGER,
        PRIMARY KEY (employee_id),
        FOREIGN KEY(department_id) REFERENCES
           department (department_id)
)
```

SQL-modeling in SQLAlchemy

Declarative mapping for the one-to-many relationship:

```
class Employee(Base):
    __tablename__ = "employee"
    id = Column('employee_id', Integer, primary_key=True)
    first_name = Column('fname', String)
    last_name = Column('lname', String)
    dept_id = Column('department_id',Integer,
        ForeignKey('department.department_id'))

class Department(Base):
    __tablename__ = "department"
    id = Column('department_id', Integer, primary_key=True)
    dname = Column('DName', String, nullable=False)
    employees = relationship('Employee', cascade='all, delete',
                          back_populates='department')
```

– relationship element specifies (with cascade** that deleting department causes that all employees working at that department are deleted

Execution of queries using ORM

Session element presents the public interface for the usage of ORM

```
session = Session(engine)
```

– session is used to execute all CRUD operations

```
session.add(employee1)
```

```
session.delete(employee1)
```

– SQL queries are defined using query() method with already defined mapping classes and filters to define restrictions

```
result = session.query(Employee).filter(first_name == 'John').all()
```

– example of the query with join statement

```
employees = session.query(Employee).\
        join(Department, Department.id == Employee.dept_id).\
        filter(Department.dname=="Accounting").all()
session.query(Department).filter(Department.dname=="Accounting").first()
```

Database Programming and Design Patterns

Model View Controller (MVC) is a design pattern which is traditionlly used for graphical user interfaces with database applications

- proved to be good for the generation of organized modular applications
- breaks an application into three modules **model**, **view** and **controller**

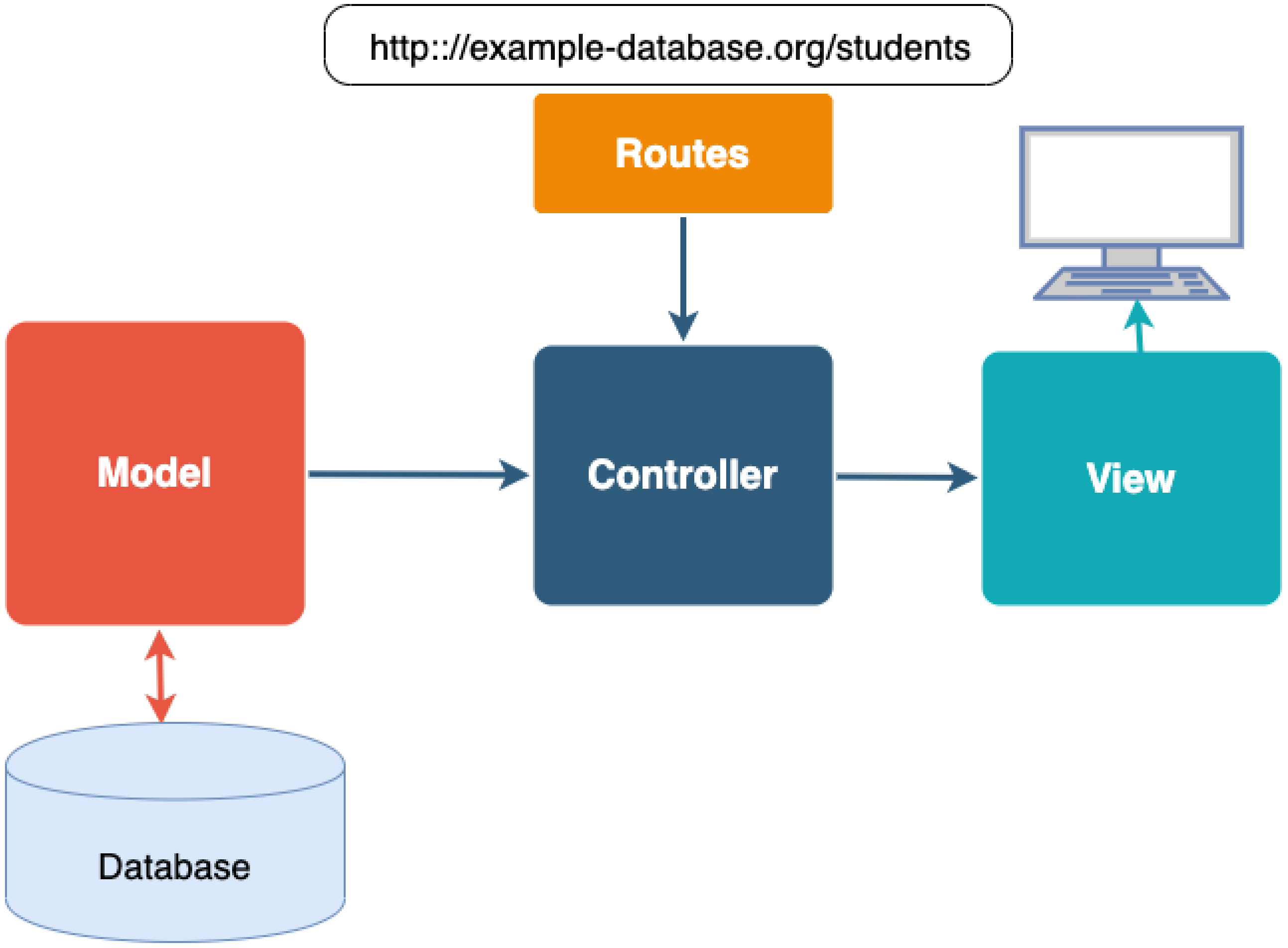
Model defines instances which are used to manipulate the database

- programmers don't have to use SQL most of the time
- this reduces syntax errors in SQL commands

Controller defines methods for handling user events

View contains methods which render appearance of data in the user interface

Model-View-Controller design pattern



Model view controller for web applications

Review questions

- What is the difference between ODBC and JDBC
- Describe 3-tier architecture.
- What is the role of cursor in database programming?
- Which cursor methods are important for fetching data according to DBAPI?
- Explain what are advantages and disadvantages of ORM.
- Write a SQLAlchemy model which specifies many-to-many relationship.
- What are main elements of a connection string?
- Explain the notion impedance mismatch

Further resources

PEP-249

Object-Relational Mapping Revisited

OrmHate - Martin Fowler