LABORATORY OF DATA SCIENCE

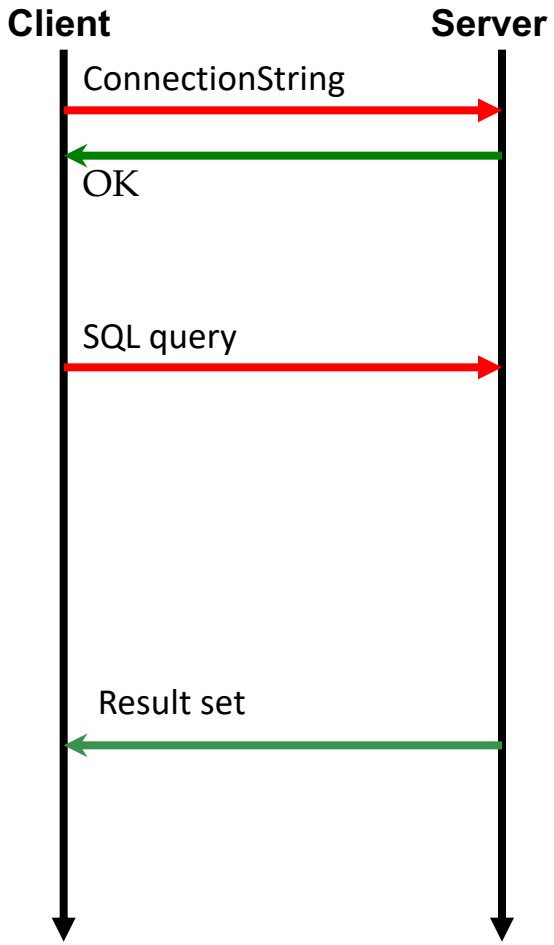
Data Access: Relational Data Bases

RDBMS data access

- Protocols and API
  - ODBC, OLE DB, ADO, ADO.NET, JDBC
- Python DBAPI with ODBC protocol

Connecting to a RDBMS

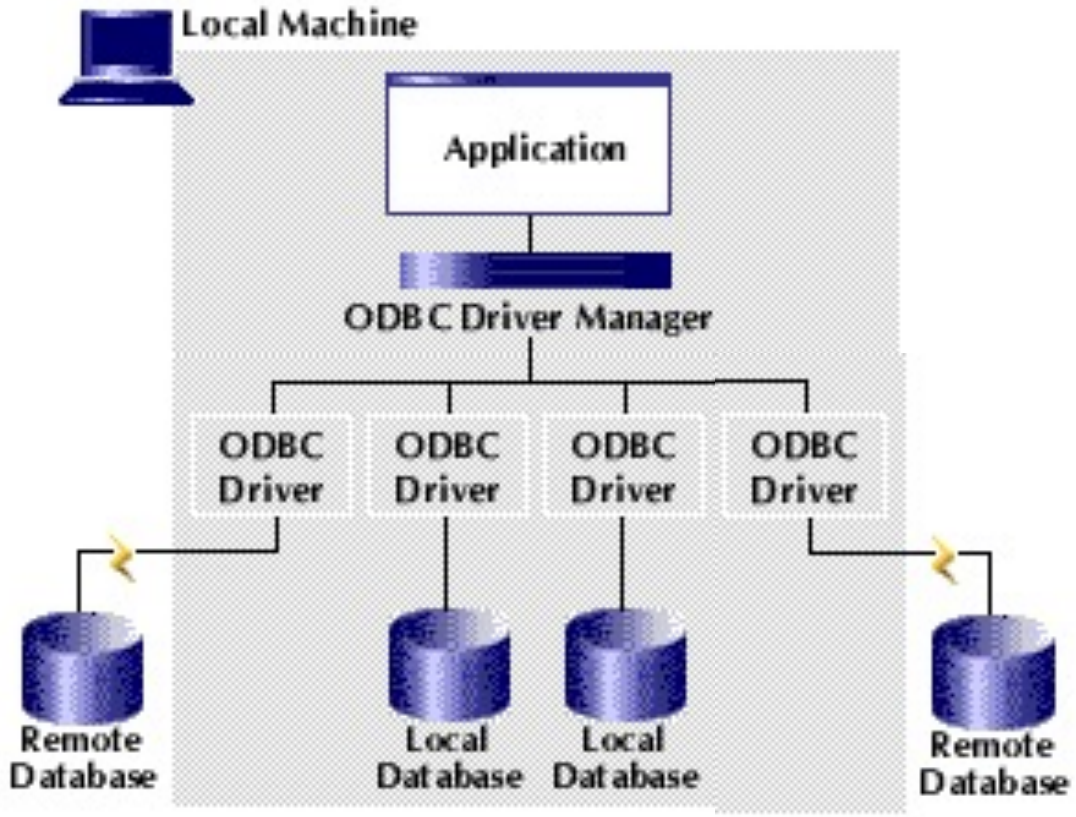
- **Connection protocol**
  - locate the RDBMS server
  - open a connection
  - user autentication
- **Querying**
  - query SQL
    - SELECT
    - UPDATE/INSERT/CREATE
  - stored procedures
  - prepared query SQL
- **Scan Result set**
  - scan row by row
  - access result meta-data



Connection Standards

- ODBC - Open DataBase Connectivity
  - Windows: odbc Linux: unixodbc, iodbc
  - Tabular Data
- JDBC – Java DataBase Connectivity
- OLE DB (Microsoft) – Object Linking and Embedding
  - Tabular data, XML, multi-dimensional data
- ADO (Microsoft) – ActiveX Data Objects
  - Object-oriented API on top of OLE DB
- ADO.NET
    - evolution of ADO in the .NET framework

ODBC Open DataBase Connectivity



ODBC Demo

- Registering an ODBC data source
  - lbi on access
  - lbi on SQL Server (driver SQL Server)
- Data access
  - copy Access table to Excel
- Linked tables
  - Linking SQL Server Table from Access

OLE DB Demo

- Creating **.udl** data links
- Data access
  - accessing Access data from Excel
- Linked tables
  - accessing Excel data from Access
- OLE DB Drivers
  - By Microsoft

RDBMS data access

- **Python DBAPI** is a standard specification for modules that interefaces with databases
- Most of Python database interfaces adhere to this standard
- **Functions:**
  - Connecting to a database
  - Submitting SQL queries
  - Scanning the results of queries
  - Accessing meta-data on tables

Support of Different RDBMS

- **Portable across several relational and non-relational databases:**
  - Microsoft SQL Server
  - Oracle
  - MySQL
  - IBM DB2
  - PostgreSQL
  - Firebird (and Interbase)
  - Cassandra
  - MongoDB
  - …..

Different Modules for a DB

Given a database we have variuos module options. For example, MySQL has the following interface modules:

- MySQL for Python (import MySQLdb)
- PyMySQL (import pymysql)
- pyODBC (import pyodbc)
- MySQL Connector/Python (import mysql.connector)
- mypysql (import mypysql)
- etc ...

DBAPI Specification

- Most of database modules conform to the specification
- no matter which kind of database and/or module you choose, the code will likely look very similar
- See details here:

  https://www.python.org/dev/peps/pep-0249/

DBAPI Specification

- Each module interface is required to have the following functions
- `connect(args):` a constructor for **Connection objects**, that makes the access available. Arguments are **database-dependent**
- `conn.close()` – close connection
- `conn.commit()` – commit pending transaction
- ….

DBAPI Specification

- `conn.cursor()` – return a Cursor object for the connection. Cursors are used fetch operations
- `c.execute(op,[params])`–prepare and execute an operation with parameters where the second argument may be a list of parameter sequences
- `c.fetch[one|many|all]([s])`– fetch next row, next s rows, or all remaining rows of result set
- `c.close()` – close cursor.
- and others.

Programming pattern

1. Import the DB module
2. Connect to the RDBMS
3. Submit a SQL query
4. Process query results
5. Close the connection

DB Module: Pyodbc

- Pyodbc is an **open source** Python module ODBC and implementing the DBAPI 2.0 specification.
- Enables an easily connection of Python applications to data sources with an ODBC driver
- Python program along with the pyodbc module will use an **ODBC driver manager** and **ODBC driver**
- The ODBC driver manager is **platform-specific**
- The ODBC driver is **database-specific**
- The ODBC driver manager and driver will connect, typically over a network, to the database server.

Connect to the RDBMS

- Access the database via the **connection object**
- Use connect constructor to create a **connection** with database

```
conn = pyodbc.connect(parameters...)
```

- Create cursor via the connection

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

- Connect function requires the “connection string”
- The connection string depends on the driver

Connection String

- The connection strings:

```
DRIVER=Driver name;      SERVER=hostname;
DATABASE=DBname;         UID=user;
PWD=password
```

- In Python:

```
conn = pyodbc.connect(
            'DRIVER={ODBC Driver 17 for SQL Server};
            SERVER=tcp:apa.di.unipi.it;
            DATABASE=Foodmart;
            UID=lbi;
            PWD=pisa')
```

ODBC DRIVER

- Microsoft have written and distributed multiple ODBC drivers for **SQL Server:**
  - {SQL Server} - released with SQL Server 2000
  - {SQL Native Client} - released with SQL Server 2005 (also known as version 9.0)
  - {SQL Server Native Client 10.0} - released with SQL Server 2008
  - …..

ODBC DRIVER

- {SQL Server Native Client 11.0} - released with SQL Server 2012
- {ODBC Driver 11 for SQL Server} - supports SQL Server 2005 through 2014
- {ODBC Driver 13 for SQL Server} - supports SQL Server 2005 through 2016
- {ODBC Driver 13.1 for SQL Server} - supports SQL Server 2008 through 2016
- {ODBC Driver 17 for SQL Server} - supports SQL Server 2008 through 2019

Submit a SQL query

Select String:

```
query = "SELECT name, age FROM students”
```

Submit the SQL query and get the result

```
cursor.execute(query)
```

UPDATE String:

```
update = “UPDATE students SET age = age + 1“;
cursor.execute(update )
Conn.commit()
```

Scan query results

FETCHALL:

```
cursor.execute("SELECT TOP 10 education, gender FROM
   customer")
rows = cursor.fetchall() // all rows in memory!!!
for row in rows:
   print (row[0], row[1])   //access by index
   print(row.gender, row.education) //access by name
```

CURSOR AS ITERATOR:

```
cursor.execute("SELECT TOP 10 education, gender FROM
   customer;"):
for row in cursor:
    print(row.gender, row.education)
```

Update and Delete

- Updating and deleting work passing the SQL to execute

```
deleted = cursor.execute("delete from products where
id <> 0001").rowcount
conn.commit()
```

`deleted` represents the number of affected rows

Close the connection


```
   …

 // close the cursor
    cursor.close();

// close connection to the database
    conn.close();
     …
```

Prepared commands with parameters

- **Problem:** read N rows from a CSV file, and insert each one into a database table
  - N SQL queries?

```
INSERT INTO names (id, name) VALUES (1, 'Luigi Rossi')
INSERT INTO names (id, name) VALUES (2, 'Mario Bianchi')
...
```

    - Inefficiency: an execution plan has to be computed for every query, yet all of them share a common structure
  - Use ? as a placeholder for parameters

```
INSERT INTO names (id, name) VALUES (?, ?)
```

Prepared commands with parameters


```
.....
conn = …… //connection
cursor = conn.cursor()
lines = fileIn.readlines()
sql ="INSERT INTO name_table(id,name)
  VALUES(?,?)“
i=0
for var in lines:
    rows = cursor.execute(sql,(i,var))
    i+=1
conn.commit()
```

Prepared commands with parameters


```
conn = …… //connection
cursor = conn.cursor()
list = ['USA', 'Canada']
query = 'SELECT education, country FROM
  customer WHERE country=?'
for el in list:
   rows =
  cursor.execute(query,el).fetchall()
   print ('Start ', el)
   for row in rows:
          print(row)
          print('\n')
```

DATA TYPE MAPPING

How Python objects passed to cursor.execute() as parameters are formatted and passed to the driver/database.

Python Datatype	Description	ODBC Datatype
None	varies	varies (1)
str	UTF-16LE (2)	SQL_VARCHAR or SQL_LONGVARCHAR (2)(3)
bytes, bytearray	binary	SQL_VARBINARY or SQL_LONGVARBINARY (3)
bool	bit	BIT
datetime.date	date	SQL_TYPE_DATE
datetime.time	time	SQL_TYPE_TIME
datetime.datetime	timestamp	SQL_TIMESTAMP
int	integer	SQL_BIGINT
float	floating point	SQL_DOUBLE
decimal	numeric	SQL_NUMERIC
UUID.uuid	UUID / GUID	SQL_GUID

DATA TYPE MAPPING

How database results are converted to Python objects

Description	ODBC Datatype	Python Datatype
NULL	any	None
1-byte text	SQL_CHAR	str via UTF-8 (1)
2-byte text	SQL_WCHAR	str via UTF-16LE (1)
UUID / GUID	SQL_GUID	str or UUID.uuid (2)
XML	SQL_XML	str via UTF-16LE (1)
binary	SQL_BINARY, SQL_VARBINARY	bytes
decimal, numeric	SQL_DECIMAL	decimal.Decimal
bit	SQL_BIT	bool
integers	SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT	int
floating point	SQL_REAL, SQL_FLOAT, SQL_DOUBLE	float
time	SQL_TYPE_TIME	datetime.time
SQL Server time	SS_TIME2	datetime.time
date	SQL_TYPE_DATE	datetime.date
timestamp	SQL_TIMESTAMP	datetime.datetime

Meta-data on ResultSet

Meta-data: column names and types of a resultset

```
for attributes in cursor.description:
    print("Name: %s, Type: %s " %
          (attributes[0], attributes[1]))
```

Meta-data on DB Tables

- `tables(table=None,catalog=None,`
  `schema=None,tableType=None)`
- Returns an iterator for generating information about the tables in the database.
- Each row has the columns:
  - Table_cat: catalog name
  - Table-schem: schema name
  - Table_name: table name
  - table_type: TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM
  - A description of the table

Meta-data on DB Tables


```
cnxn = …
cursor = cnxn.cursor()
for table in cursor.tables():
    print(table)
```

Or

```
for table in cursor.tables(table='sys%'):
    print(table)
```

Tables starting with «sys»

Columns meta-data


```
columns(table=None,catalog=None,schema=None,column=None)
```

- Creates a result set of column information on a table
- Each row has the following columns:
  - table_cat
  - table_schem
  - table_name
  - column_name
  - data_type
  - type_name
  - column_size
  - buffer_length
  - decimal_digits
  - num_prec_radix
  - nullable
  - remarks
  - column_def
  - sql_data_type
  - sql_datetime_sub
  - char_octet_length
  - ordinal_position
  - is_nullable: One of SQL_NULLABLE, SQL_NO_NULLS, SQL_NULLS_UNKNOWN.

Exercise: Stratified subsampling

- DATABASE NAME = Ibi
- Let T be a database table (e.g., census), and A a column in T (e.g., sex)
- Develop a Python program that exports on a CSV file a subset of 30% of rows of T:
  - the subset is randomly chosen;
  - but it must preserve the proportion of distinct values of column A
    - e.g., if there are 65% of male students, the subset must contain 65% of males and 35% of females.

Intuition on the solution!

Males

Nrows=100

SelRow=30

All records selected!!!

How to generate an element with probability x/y?

- Generate a number n in the range [0 …. Y]
- The element is selected if n < x the record is selected
- For random selection of a number in the above range

```
(int)(Math.random()*Y)
```