Architectures for Big Data - First assignement

Federico Cristiano Bruzzone & Andrea Longoni & Massimiliano Visconti

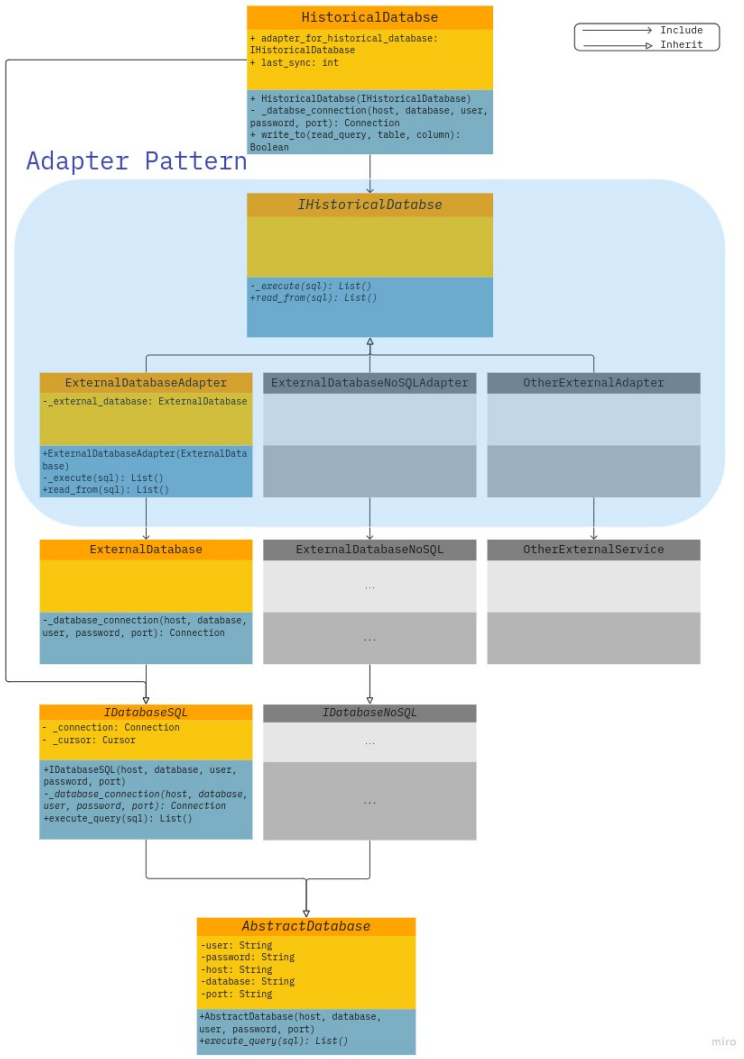
1 Architecture

1.1 Implementation

We developed the architecture later illustrated keeping in mind that we have a single immutable database on which we write (historical database) and possible multiple databases or data sources from which to read.

To interface on these databases or data sources we thought to use multiple adapters to transform the data coming in a writable format on hisotircal database.

1.2 Adapter Pattern



Please note: The gray part is not implemented. We will talk about it in the reusability chapter.

We thought to use **Adapter (Wrapper)** pattern beacuse this allow us to keep the same *historical databse* and make it to comunicate with different types of *external database*. We want to write into the *historical databse*, and we want to read from the *external database*.

This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one.

1. *AbstractDatabase* is an abstract class that contains the information to connect with any database.

   In the inherit class, you will have to implement execute_query(...) method.

2. *IDatabaseSQL* are an abstract class that define behavior of the SQL database.

   Since IDatabaseSQL inherits from AbstractDatabase it must implements execute_query(...) method.

   In the inherit class, you will have to implement _database_connection(...) method.

   Assuming that in python any database sql library implement .connector.connect(...) and .cursor() methods:

   (a) we use _connection field to store the connection to the specific database;

   (b) we use _cursor field to store the cursor to the specific database.

   Note that _cursor contanis the .execute(...) method that is used to execute the query.

3. *ExternalDatabase* is a concrete class that allow us to establish a connection.

   Since ExternalDatabase inherits from IDatabaseSQL it must implements _database_connection(...).

4. *ExternalDatabaseAdapter* is a concrete class that make the data from ExternalDatadase readable and writable for HistoricalDatabase.

   Since ExternalDatabaseAdapter inherits from IHistoricalDatabse it must implements _read_from(...) and _execute(...) methods.

   Since ExternalDatabaseAdapter has a ExternalDatabase object and it can execute query or more specifically read the data from it.

   (e.g., If the our HistoricalDatabase would like to has a list of tuple we should implement this method in the way it returns it)

5. *IHistoricalDatabse* is an interface that contains the delcarations of methods that they must be implemented by each adapters.

   We expect the read_from(...) mothod returns a fitted content for the HistoricalDatabase.

6. *HistoricalDatabse* is a concrete class that contains the methods to execute query into the historical database.

   Since HistoricalDatabse inherits from IDatabaseSQL it must implements _database_connection(...).

   Since HistoricalDatabse has a IHistoricalDatabase object it can get data from it.

   To remember the last item we read, we store the identifier of it (ordered) into a sync.json file, and when we want to execute the next query, we should read from sync.json the identifier.

   The .write_to(...) mothod use the .read_from(...) method of the adapter to get a list of tuple and then it will insert it to historical database.

   After this, it will commit the changes.

2 Software Architecture Pillars

2.1 Being the framework for satisfying requirements

Functional

Our code is able to read from the external database and write to the historical database without any problems.

Technical

We are able to read from any database and any table of them, if the adapters of the databases are setted. And we can write on the historical database if the same table exists and the right fields are setted.

Security

Our code could be vulnerable by **sql injection** if the historical and external database doesen't implement internal security feature.

For example:

- We could give the right permission to each user to avoid security issues.
- Implement the **prepare statement** database side, so having queries precomipled.

2.2 Being the technical basis for design

```
1 class Client(IDatabaseSQL):
2   clientInterface: ClientInterface = ''
3
4   [...]
```

In our code you can find an interface called *ClientInterface* and its implementation that allows modularization because the *Client* stay unchanged and you could change, add, delete and modify the *Services* as you prefer.

2.3 Being the managerial basis for cost estimation and process management

2.4 Enabling component reuse

```
class ClientInterface(ABC):
    @abstractmethod
    def _execute(self, sql): pass

    @abstractmethod
    def read_from(self, sql): pass

class ServiceAdapter(ClientInterface):
    service: Service = ''

    def __init__(self, service: Service):
        self.service = service
        print('ServiceAdapter has been created')

    def _execute(self, sql):
        query_res = self.service.execute_query(sql)
        return query_res

    def read_from(self, sql):
        query_res = self._execute(sql)
        return query_res
```

Our code is reusable beacuse if you want to change the databse where you read you should only write a new *ServiceAdapter* and *Service* to connect them to the *Client*.

So, if you to use a NoSQL type database, you will implement a new *ServiceAdapter* and *Service* that will inherit from *IDatabaseNoSQL*. Acutally, the most importat thing is that the other part of the code will not change.

2.5 Allowing a tidy scalability

```
class Client(IDatabaseSQL):

  [...]

  def write_to(self, read_query, table, column='') -> Bool:
    response = self.clientInterface.read_from(read_query)
```

```
7       new_column = f"({', '.join(column)})"
8       for i in response:
9         self.execute_query(f'INSERT INTO {table} {new_column} VALUES {(i)};')
10
11      self._connection.commit()
12      return True
```

Our code allow you to do more INSERT at a time. With an only one Query, thanks to the *ServiceAdapter*, you could read a set of tuples and write them on the historical database with a for loops.

In our code is not implemented, but a possibile solution for more scalability is to implement a multi-thread read/write structure.

2.6 Avoiding handover and people lock-in

To avoiding handover and people lock-in we could write more comments and documentation about our code.

3 Testing

We have tested the code by creating table **user1** and **user2** with *id* (Primary Key, Auto Increment), *name* and *surname* respectively and then we have tried to read from **user1** and write to **user2**.

We have used sync.json file to store the last tuple that we read from the **user1**.

Reading this file we were able to resume reading **user1** from the last writing in table **user2**, whitout having read the whole database every time.

For simplicity, we have been using the id filed of the **user1** to keep track the last tuple stored in the sync.json file, and we have read and wrote in the same database instance.

Our tests were performed succesfully.

First execution

```
1 Connection successfull to the: 127.0.0.1 test_database root welcome123
2 ExternalDatabaseAdapter has been created
3 Connection successfull to the: 127.0.0.1 test_database root welcome123
4
5 Query has been executed: SELECT name, surname FROM user where id > 0
6                          ('federico', 'bruzzone')
7                          ('andrea', 'longoni')
8                          ('massimiliano', 'visconti')
9
10 Query has been executed: INSERT INTO user2 (name, surname)
11                         VALUES ('federico', 'bruzzone');
```

```
12
13 Query has been executed: INSERT INTO user2 (name, surname)
14                         VALUES ('andrea', 'longoni');
15
16 Query has been executed: INSERT INTO user2 (name, surname)
17                         VALUES ('massimiliano', 'visconti');
```

Second execution

The second execution did not write data since in the user1 table there are only three tuples and in the our sync.json the counter is set to three after the first execution.

```
1 Connection successfull to the: 127.0.0.1 test_database root welcome123
2 ExternalDatabaseAdapter has been created
3 Connection successfull to the: 127.0.0.1 test_database root welcome123
4
5 Query has been executed: SELECT name, surname FROM user where id > 3
```