UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Sistemas de almacenamiento y procesado distribuido

Laboratorio 1

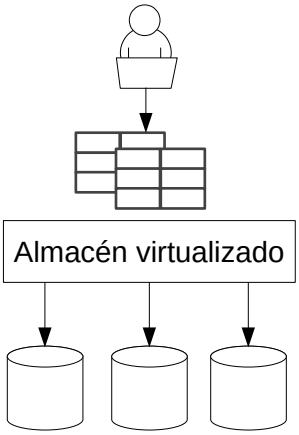
Javier Esparza Peidro – jesparza@dsic.upv.es

Contenido

1. Introducción

En este laboratorio se inicia el 1er proyecto de la asignatura. El objetivo global del proyecto consiste en diseñar un *almacén de datos virtual*, completamente funcional. El proyecto será desarrollado a lo largo de varios laboratorios.

Un **almacén de datos virtual** permite desacoplar la visión que tienen los usuarios de los datos, del soporte en el que finalmente se almacenan. Ello significa que los usuarios acceden a un esquema de datos virtual integrado, y por debajo el almacén de datos obtiene y transforma los datos procedentes de distintas fuentes. De este modo, es posible integrar bajo un esquema común diversas fuentes de datos (*persistencia políglota*), sin perder las propiedades específicas de cada una de ellas. Esta técnica también es conocida como *federación de datos.*

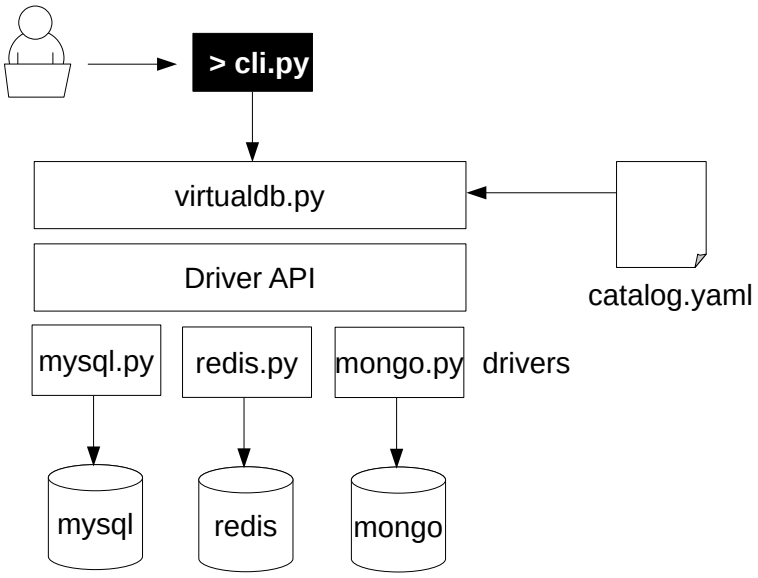


En nuestro caso, el esquema de la base de datos virtual será un **esquema relacional**, que contendrá tablas, filas y columnas. Por debajo será necesario efectuar las transformaciones necesarias desde/hacia las distintas fuentes de datos.

En este primer laboratorio nos centraremos en permitir que el almacén virtual permita efectuar consultas simples sobre el esquema virtual. En sucesivos laboratorios ampliaremos las capacidades del motor.

2. Arquitectura del sistema

Para implementar el almacén virtual, se propone la arquitectura mostrada en la siguiente figura.



Los distintos componentes de la arquitectura se describen a continuación:

- *cli.py*: este módulo implementa un cliente en línea de commandos (Command-Line-Interface), que permitirá la interacción con el almacén virtual. El usuario efectuará consultas SQL estándar y obtendrá los resultados, como si se tratara de una base de datos relacional centralizada. Para ello, utiliza las primitivas exportadas por *virtualdb.py*.
- *virtualdb.py*: módulo principal y corazón del sistema. Implementa una API sencilla que consume *cli.py*. Permite interactuar con la base de datos virtual definida en el catálogo. Para comunicarse con las distintas fuentes de datos utiliza una API homogénea, denominada Driver API.
- *catalog.yaml*: contiene la definición del catálogo del almacén virtual. Esto incluye el esquema de la base de datos relacional, y cómo se mapea a las distintas fuentes de datos.
- *Driver API*: es una API que implementan todas las fuentes de datos. Se trata de una API orientada a agregados. Esta API es implementada por distintos drivers, que permiten acceder a distintas fuentes de datos.
- *mysql.py*, *redis.py*, *mongo.py*, …: implementaciones específicas de distintos drivers, que permiten acceder a distintas fuentes de datos. Todos los drivers implementan la *Driver API,* ofreciendo una visión orientada a agregados de la fuente de datos.

En las siguientes secciones se irán diseccionando la mayoría de estos componentes.

3. Driver API

La Driver API permite al almacén virtual abstraerse de las fuentes de datos que se integran en el catálogo. Esta API ofrece una visión orientada a agregados de las fuentes de datos. De este modo, resulta posible efectuar búsquedas sobre agregados sencillos, cuyos atributos pertenecen a los tipos de datos básicos ‘str’, ‘int’ y ‘float’. Para poder acceder a una fuente de datos, es necesario que proporcione un driver que implemente esta API.

A continuación se describe la API que todo driver debe proporcionar:

`driver.connect(cfg:dict={}): Driver`

Crea una conexión a una base de datos almacenada en cierta fuente de datos.

El argumento *cfg* contiene toda la información necesaria para efectuar la conexión. Generalmente contendrá al menos las entradas “host”, “port” y “database”. Adicionalmente, podría contener otros atributos como “user” y “password”.

Si la conexión se efectúa con éxito, se devuelve un objeto de la clase *driver.Connection*

`clase driver.Connection`

Representa una conexión abierta a una base de datos en una fuente de datos.

`driver.Connection.search(collection:str, fields:list=None, query:dict=None, sort:str=None, first:int=0, count:int=None)`

Efectúa una consulta sobre la colección de agregados especificada en el argumento *collection*.

El argumento *fields* determina los campos a recuperar. Si se omite se recuperan todos.
El argumento *query* determina el filtrado a aplicar. Se utiliza la sintaxis utilizada por Mongo, y soporta al menos los operadores de consulta $eq, $neq, $gt, $gte, $lt, $lte. Si se omite entonces no se aplica filtrado.
El argumento *sort* determina el orden de los resultados. La sintaxis es “+field” o “-field” para determinar ordenado por el campo “field” ascendente o descendente respectivamente.
El argumento *first* determina la posición del primer resultado a devolver, comenzando por 0.
El argumento *count* determina el número máximo de resultados a devolver.

Si la consulta tiene éxito, se devuelve una lista con todos los agregados que la verifican. Cada agregado es un diccionario que contiene los campos del agregado.

`driver.Connection.count(collection:str)`
Devuelve el número de agregados que contiene la colección especificada en el argumento *collection*.

`driver.Connection.close()`
Cierra la conexión actual.

En las siguientes subsecciones se dan pistas acerca de cómo se podría implementar este driver para fuentes de datos de tipo Redis, MySQL y Mongo.

3.1 Redis

Redis es un almacén clave-valor muy rápido, que almacena toda la información en memoria. Primero lo arrancaremos:

```
> docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redis-stack:latest
```

Para acceder con Python primero hay que instalar el driver adecuado:

```
> python -m venv venv
> source venv/bin/activate
(venv)> pip install redis
```

La API es muy sencilla, por ejemplo:

```
import redis
con = redis.Redis(host='localhost', port=6379, db=0)
con.set('test', 'Hello world!')
con.get('test')
keys = con.keys('*')
con.delete('a')
con.close()
```

El driver de Redis debe de obtener los agregados a partir de claves. Para ello, asumiremos que cada agregado se almacena en una clave que tiene la siguiente estructura:

```
“/<database>/<collection>/<id>”
```

Donde *database* representa la base de datos, *collection* es la colección en la que se almacena el agregado e *id* representa el valor de su clave primaria.

El valor asociado a cada clave será el agregado formateado en JSON. Por ejemplo, basándonos en una hipotética colección *users*, almacenada en la base de datos *test*, el almacén Redis podría contener las siguientes entradas:

```
/test/users/1 -> ‘{“id”:1, “name”:”a”, “email”:”a”}’
/test/users/2 -> ‘{“id”:2, “name”:”b”, “email”:”b”}’
```

3.2 MySQL

MySQL es un almacén relacional clásico. Primero lo arrancaremos:

```
> docker run --name mysql -e MYSQL_ROOT_PASSWORD=root -d -p 3306:3306 mysql
```

A continuación instalaremos el driver Python:

```
(venv)> pip install mysql-connector-python
```

Ya conocemos su API:

```
import mysql.connector
con = mysql.connector.connect(host="localhost", user="root", password="root",
database="test", port= 3306)
cur = con.cursor()
cur.execute("select * from users")
res = cur.fetchall()
for x in res: print(x)
con.close();
```

En este caso, cada colección de agregados podría mapearse a una tabla, y cada agregado a una fila de dicha tabla. Por ejemplo, una colección *users* podría almacenarse en una tabla con la siguiente estructura:

id	**name**	**email**
1	‘a’	‘a’
2	‘b’	‘b’

3.3 MongoDB

MongoDB es un almacén de documentos. Primero lo arrancamos:

```
> docker run --name mongodb -d -p 27017:27017 mongodb/mongodb-community-server
```

Instalamos el driver:

```
(venv)> pip install pymongo
```

A continuación se muestra un fragmento de su API:

```
import pymongo
client = pymongo.MongoClient('localhost', 27017)
db = client.mydb # db = client['mydb']
col = db.mycollection # col = db['mycollection']
db.usuarios.insert_one({'nombre': 'Pepe', 'edad': 50});
cur = db.usuarios.find(filter={'edad': {'$gt': 50}}, projection={'nombre':1},
'skip'=1, 'limit'=1)
for doc in cur: print(doc.nombre)
db.usuarios.update_many({'nombre': 'Pepe'},{'$set':{'edad':50})
db.usuarios.delete_many({'nombre': 'Pepe'})
client.close()
```

Como se puede observar, la Driver API encaja muy bien con la API de Mongo, de manera que habrá que hacer poco trabajo de mapeo en este driver.

3.4 Carga de drivers

Al arrancar nuestro almacén debería de cargar todos los drivers disponibles en el sistema. Para ello, se recomienda implementar la función *loadDrivers()*, que deje los drivers cargados en un diccionario *drivers*:

`virtualdb.loadDrivers(path:str="./drivers"):`

Carga todos los drivers disponibles a partir del path especificado. Por defecto, se accede al path "./drivers".

Al finalizar su ejecución, los drivers estarán almacenados en un diccionario global *drivers*.

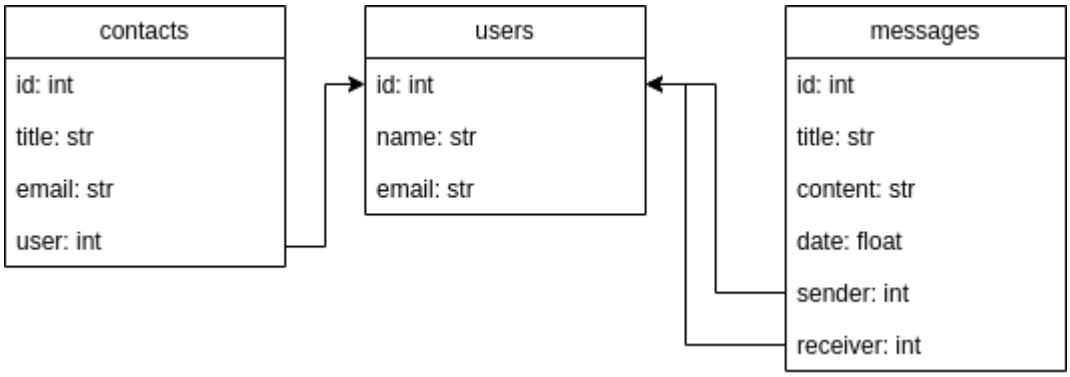
4. El catálogo

El catálogo tiene dos funciones básicas:

1. Determinar el esquema de la base de datos virtual
2. Determinar cómo se mapea a las distintas fuentes de datos.

El catálogo se describe en un fichero con formato .yaml, es una extensión de JSON que resulta más legible para humanos.

Para simplificar, trabajaremos con un ejemplo de esquema de base de datos relacional sencillo, que posee tres tablas relacionadas entre sí.



En este ejemplo, asumiremos que la tabla *users* se almacenará en Redis, la tabla *contacts* se almacenará en una base de datos en *MySQL* y la tabla *messages* se almacenará en MongoDB.

A continuación se presenta un ejemplo de la estructura básica que podría tener un fichero *catalog.yaml* que contiene el catálogo de esta base de datos virtual:

```
users:
  fields:
  - name: id
    type: int
    primary: true
  - name: name
    type: str
  - name: email
    type: str
    unique: true
  mapping:
    driver: redis
    host: localhost
    port: 6379
    database: test
    collection: users
contacts:
  fields:
  - name: id
    type: int
    primary: true
```

```
  - name: title
    type: str
    email: str
  - name: user
    type: int
  mapping:
    driver: mysql
    host: localhost
    port: 3306
    user: root
    password: root
    database: test
    collection: contacts
messages:
  fields:
  - name: id
    type: int
    primary: true
  - name: title
    type: str
  - name: content
    type: str
  - name: date
    type: float
  - name: sender
    type: int
  - name: receiver
    type: int
  mapping:
    driver: mongo
    host: localhost
    port: 27017
    database: test
    collection: messages
```

Como se puede observar, el catálogo es un diccionario que posee varias entradas. Cada entrada representa la información sobre una tabla del esquema relacional virtual. La clave es el nombre de la tabla y el valor es un diccionario que posee toda la información sobre sus campos (*fields*), y sobre como se mapean a una fuente de datos (*mapping*).

En la lista de campos (*fields*), cada campo se representa por un diccionario donde se describe al menos su nombre (*name*) y su tipo (*type*); además es posible definir otras cuestiones, como restricciones de integridad (*primary*, *unique*, etc.). Por ejemplo, el campo *email* de la tabla *users* es de tipo *str* y además posee una restricción de unicidad *unique*. Los tipos de datos que reconoceremos inicialmente son *str*, *int* y *float*. Todas las tablas deberían de poseer un campo primario (*primary*), que determina la clave primaria de la tabla.

La entrada “mapping” determina los detalles acerca de cómo se mapea una tabla a una fuente de datos. Por ejemplo, la tabla *users* se mapea a la colección *users* en la base de datos *test* en una fuente de datos Redis.

Para cargar el catálogo, se recomienda implementar la función *loadCatalog()* en el módulo *virtualdb.py*. Esta función podría encargarse de validar el fichero *catalog.yaml* y dejar en un diccionario *catalog* toda su información.

`virtualdb.loadCatalog(path:str):`

> Valida y carga el catálogo de la base de datos virtual a partir del fichero especificado. Si encuentra algún error lanza una excepción.
>
> Al finalizar su ejecución, la variable global *catalog* contiene un diccionario con los detalles acerca del catálogo.

Para cargar el fichero .yaml se recomienda utilizar la librería PyYAML:

https://pyyaml.org/

(venv) > pip install pyyaml

Una vez instalada resulta muy sencillo cargar un fichero YAML con un código similar a éste:

```
import yaml

f = open(path, "rt")
text = f.read()
f.close()
spec = yaml.load(text, yaml.Loader)
```

El catálogo podrá consultarse con la operación *describe().*

`virtualdb.describe():`
Devuelve una descripción del catálogo cargado.

5. Consultas

El almacén de datos virtual debe publicar la operación *query()* para efectuar consultas sobre la base de datos virtual.

`virtualdb.query(sql:str):`
Efectúa una query SQL sobre la base de datos virtual.

Devuelve la lista de resultados.

El lenguaje de consulta será SQL, y debería reconocer al menos la siguiente sintaxis:

```
<QUERY> ::= SELECT <FIELDS> FROM <TABLE> [<WHERE>] [<ORDER>]
            [<LIMIT> [<OFFSET>]]
<FIELDS> ::= * | <ID> {,<ID>}
<TABLE> ::= <ID>
<ID> ::= ([a-z] | [A-Z]) {[0-9] | [a-z] | [A-Z]}
<WHERE> ::= WHERE <COND> {AND <COND>}
<COND> ::= <ID> <OP> <CONST>
<OP> ::= <> | = | <= | < | >= | >
<CONST> ::= <STR> | <FLOAT> | <INT>
<STR> ::= ‘{[0-1] | [a-z] | [A-Z]}’
<FLOAT> ::= <INT> [.{[0-9]}]
<INT> ::= 0 | [1-9] {[0-9]}
<ORDER> ::= ORDER BY <ID> [ASC|DESC]
<LIMIT> ::= LIMIT <INT>
<OFFSET> ::= OFFSET <INT>
```

Algunos ejemplos serían:
SELECT * FROM users
SELECT id,name FROM users WHERE name=’Pepe’
SELECT * FROM users WHERE name=’Pepe’ AND id=1 ORDER BY name LIMIT 10 OFFSET 2

6. CLI

Para acceder al almacén de datos virtual se proporciona una herramienta en línea de comandos. A continuación se muestra un posible caso de uso de dicha herramienta:

```
$ python3 cli.py
Usage: python3 cli.py <schema.yaml>

$ python3 cli.py catalog.yaml
virtual> .help
Available commands within the prompt
    <sql>: execute SQL query or statement
    .help: print help
    .describe: print virtual schema
    .exit: close the current connection
virtual> .describe
table users:
  mapped to: redis:test/users
  id: int, primary
  name: str
  email: str, unique
...
virtual> select * from users
1, ‘a’, ‘a’
2, ‘b’, ‘b’
virtual> .exit
Bye!
```