Bases de Datos II

Francisco Javier Mercader Martínez

Índice

Tema 1: Recuperación de datos y formatos de serialización

1.1) Necesidad de formatos de serialización

- Los formatos de **serialización** son vitales para el intercambio de datos
- Más en el ámbito *Big Data* (atacan a la "V" de la *variabilidad*)
- A lo largo de los años se han diseñado formatos de serialización
- Algunos son **más eficientes** que otros.
- Algunos se han **estandarizado**.
- En cualquier caso, son fundamentales para transmitir información, ya sea a través´s de **ficheros de disco** o bien para **comunicación por red**.
- La mayoría de los formatos son **de propósito general**, por lo que una misma información se pude codificar **de varias formas** (por ejemplo, como `INSERT` en SQL, ficheros CSV, etc.)
- Nuestro objetivo es conocer los formatos más usados para elegir el correcto en cada ocasión.

1.2) Características a analizar

- **¿Es un protocolo estándar?-** Con ello nos referimos a si está avalado por cuerpo de estándares o su especificación
- **¿Permite la codificación binaria?-** A la hora de transmitir grandes cantidades de información (ya sea en forma almacenada o bien a través de la red) es fundamental un protocolo binario para ahorrar espacio/tiempo.
- **¿Es legible por los humanos?-** A veces es interesante poder depurar un protocolo por parte de un humano. Esta idea comenzó con protocolos como XML, pero se ha ido abandonando porque no resulta muy factible salvo en ocasiones muy específicas.
- **¿Soporta referencias?-** A veces tenemos que relacionar partes de un conjunto de datos. Es interesante que los mecanismos de serialización permitan referenciar otras partes de un documento o de una comunicación.
- **¿Su estructura está definida por un Esquema o IDL?-** Al igual como sucedía con el lenguaje DDL de SQL, a veces es interesante que los datos sean conformes a algún esquema, también llamado IDL (*Interface Definition Language*).
- **¿Es extensible?-** A veces es necesario acomodar datos que no se ajustan estrictamente a un esquema, o que son directamente no-estructurados.
- **¿Poseen un API estandarizado?-** Si los formatos de serialización poseen un API estandarizado será más sencillo no sólo compartir los datos, sino también compartir el código de serialización/deserialización (también llamado **marshalling**)
- Tabla resumen

	Estándar	Bin?	Humano?	Ref?	IDL?	Ext?	API?
Apache Avro	Sí	Sí	No	N/A	Sí (acoplado)	Sí	N/A
CSV	Parcial (RFC4180)	No	Sí	No	No	Parcial	No
JSON	Sí (RFC7159)	No (BSON)	Sí	Sí (RFC6901)	Parcial	Sí	No
Thrift	No	Sí	Parcial	No	Sí (acoplado)	Sí	No
XML	Sí	Parcial	Sí	Sí	Sí	No	No

1.2.1) Pandas

- Pandas es una librería open source construida sobre `Numpy`.
- Permite una preparación, limpueza y análisis rápido de los datos.
- Una de sus principales características es la de visualización de datos.
- Puede trabajar con una gran variedad de fuentes musicales.
- Para instalar pandas es tan sencillo como ejecutar una de dichas instrucciones en el terminal

```
pip install pandas
conda install pandas
```

Dataframe

- La herramienta más conocida y usada de Pandas son los `DataFrames`.
- Permite almacenar datos tabulares en dos dimensiones similar a una hoja de cálculo o una base de datos relacional.
- Las columnas de datos

```
df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y Z'.split())
```

```
          W         X         Y         Z
A -1.040684 -1.692150  1.707399 -1.257771
B -0.403809 -1.024655  2.060558 -0.242150
C -0.856354  0.173779  1.124053 -0.434952
D  0.282316 -1.349518 -0.076797  1.077644
E -0.152517 -0.603708 -0.812906  0.807102
```

1.2.2) XML (eXtensible Markup Language)

- Meta-lenguaje de etiquetas derivado de SGML.
- Motivación:
  - Intercambio de datos en Internet
- Reúne los requisitos de un lenguaje de intercambio de información:
  - Simple: al estar basado en etiquetas y legible
  - Independiente de la plataforma: codificación UNICODE
  - Estándar y amplia difusión: W3C
  - Definición de estructuras complejas: DTD, Schemas
  - Validación y transformación: DTD, XSLT
  - Integración con otros sistemas
- Facilita procesamiento lado cliente.
- **No muy utilizado para BigData. Ha perdido tracción, es demasiado complejo finalmente y es muy poco eficiente en cual al porcentaje datos/metadatos**

Ejemplo

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<DISCO CODIGO="B000067FSG">
  <TITULO>Estrella de Mar</TITULO>
  <ARTISTA>Amaral</ARTISTA>
  <ESTILO>Pop</ESTILO>
  <REFERENCIA>
    <EDITORA>Virgin</EDITORA>
    <AÑO_EDICION>2002<AÑO_EDICION>
  </REFERENCIA>
  <MUSICOS>
    <MUSICO ROL="cantante">Amaral</MUSICO>
    <MUSICO ROL="guitarra">Juan Aguirre</MUSICO>
  </MUSICOS>
</DISCO>
```

- Instrucciones de procesamiento (línea 1)
- Raíz (línea 2)
- Etiquetas y atributos

• DTD

```
<!ELEMENT DISCO (TITULO, ARTISTA, ESTILO?, REFERENCIA, MUSICOS)>
<!ATTLIST DISCO CODIGO ID #REQUIERED>
<!ATTLIST DISCO TIPO=(CD | LP | DVD) "CD">
<!ELEMENT TITULO (#PCDATA)>
...
<!ELEMENT REFERENCIA (EDITORA, AÑO_EDICION) >
<!ELEMENT MUSICOS (MUSICO*)>
...
```

Describe los documentos XML $\Longrightarrow$ **Validación**

• DTD (ii)

Documento XML:

- **Válido:** sigue la estructura de un DTD

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

- **Bien formado:** Sigue las reglas de XML

Limitaciones:

- No es XML
- Tipado limitado de datos
- No soporta espacios de nombres

- Familias de estándares
  - Schemas
    - Mismo propósito que un DTD, pero con mayor riqueza semántica.
    - Sintaxis basada en XML.
    - Contiene tipos predefinidos
    - Espacios de nombres (*namespaces*)

```
<?xml version="1.0">
<xsd:schema xmlns:xsd="http://www.w3c.org/2000/08/XMLSchema">
<xsd:element name="Disco" type="DiscoTipo"/>
<xsd:complexType name="DiscoTipo">
  <xsd:attribute name="codigo" type="String"/>
  <xsd:sequence>
  <xsd:attribute name="Titulo" type="String"/>
  <xsd:attribute name="Artista" type="String"/>
  <xsd:attribute name="Referencia" type="ReferenciaTipo"/>
  ...
```

  - NameSpaces:
    - Espacios de nombres para cualificar elementos y atributos evitando la colisión de nombres.
    - `xmlns:xsd="http://www.w3c.org/2000/08/XMLSchema"`
  - XSLT:
    - Definición de reglas de transformación de documentos
  - XSL:
    - Definición de hojas de estilos.
  - XPath:
    - Para hacer referencia a partes de un documento
    - `/DISCO[Titulo="Estrella de Mar"], /DISCO//MUSICOS[1]`
  - XLink:
    - Enlace documentos entre sí.
  - XPointer:
    - Enlace de secciones dentro de un documento.
  - XQuery:
    - Consultas XML.
- Parsers
  - API SAX:
    - Acceso secuencial al documento

  - Modelo de programación basado en eventos (*callbacks*)
  - Simple y rápido: consume pocos recursos.
  - Sólo consulta.
- API DOM:
  - Construye una estructura arbórea a partir del documento
  - Potente, pero más costoso.
  - Permite actualizaciones.
  - Ideal para estructuras complejas.
- En Python, existen numerosas formas de poder leer documentos XML.

- API SAX

Librería **xml.sax**

```
import xml.sax

class XMLHandler(xml.sax.ContentHandler):
    def __init__(self):
        # Inicializamos variables de interés

    # Se llama cuando comienza un nuevo elemento
    def StartElement(self, tag, attributes):
        pass
    # Se llama cuando un elemento acaba
    def endElement(self, tag):
        pass

parser = xml.sax.make_parser()
parser.setFeature(xml.sax.handler.feature_namespaces, 0)
Handler = XMLHandler()
parser.setContentHandler(Handler)
parser.parse('models.xml') # nombre del documento a analizar
```

Cómo podríamos procesar con **xml.sax** este documento

```
<collection shelf="New Arrivals">
    <model number="ST001">
        <price>35000</price>
        <qty>12</qty>
        <company>Samsung</company>
    </model>
    <model number="RW345">
        <price>46500</price>
        <qty>14</qty>
        <company>Onida</company>
    </model>
```

```
</collection>
```

- Parser DOM

Librería **xml.dom**

```
# Procesar un determinado fichero
file = minidom.parse('model.xml')

# Obtener los elementos con un determinado tag
modelos = fil.getElementosByTagName('modelo')

# Obtener el atributo 'nombre' del segundo
    modelo
print('modelo #2 atributos:')
print(modelos[1].attributes['nombre'].value)

# El datos de un item específico
print('\nmodelo #2 datos:')
print(modelos[1].firstChild.data)
```

```
<data>
    <modelos>
        <modelo name='modelo1'>
            modelo1abc
        </modelo>
        <modelo name="modelo2">
            modelo2abc
        </modelo>
    </modelos>
</data>
```

- Librería BeatifulSoup

```
from bs4 import BeatifulSoup

# Leemos el fichero
with open('models.xml', 'r') as f:
    data = f.read()

# Pasamos los datos al parse
bs_data = BeatifulSoup(data, 'xml')

# Buscamos todas las instancias 'unique'
b_unique = bs_data.find_all('unique')
print(b_unique)

# Usamos .find búsquedas más concretas
b_name = bs_data.find('child', {'name':'Acer'})
print(b_name)
```

```
<modelo>
    <child name="Acer" qty="12">
        Portátil Acer
    </child>
    <unique>
        Número de modelo
    </unique>
    <child name="Acer" qty="7">
        Exclusive
    </child>
    <unique>
        1.200€
    </unique>
</modelo>
```

1.2.3) CSV (Comma-Separated Values)

- Formato basado en columnas normalmente separado por coma
- Sin embargo, el formato admite variaciones:
  - Con o sin cabecera con los nombres de las columnas
  - Separador de columans (comas, tabuladores, etc.)
  - Codificación de caracteres (UTF-8, latin-1, etc.)

- Escapado de caracteres (por ejemplo, una comilla doble como dos comillas dobles(" ") ó como \")
- Comillas opcionales (sólo si hacen falta) o en todos los campos siempre

- Carga en SQL

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFLINE 'file_name'
    [RELACE | IGNORE]
    INTO TABLE tbl_name
    [PARCTITION (parctition_name, ...)]
    [CHARACTER SET charset_name]
    [{FIELDS | COLUMS}
        [TERMINATE BY 'string']
        [[OPTIONALL] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATE BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [(col_name_or_user_var, ...)]
    [SET col_name = expr, ...]
```

```
LOAD DATA LOCAL INFILE "/tmp/Posts.csv"
    INTO TABLE Posts
    COLUMNAS TERMINATED BY ',' ENCLOSED BY '"' ESCAPED BY '"'
    LINES TERMINATED BY '\r\n'
    IGNORE 1 LINES;
```

- Programación: Lectura con `Pandas DataFrame`

  - Lectura básica de un fichero CSV

```
import pandas as pd
# Read the CSV file
airbnb_data = pd.read_csv("airbnb.csv")
```

  - Si queremos establecer la columna id como índice

```
airbnb_data = pd.read_csv("airbnb.csv", index_col="id")
```

  - Si queremos leer solo un conjunto de columnas

```
usecols = ["id", "nombre", "barrio", "precio", "noches_minimias"]
airbnb_data = pd.read_csv("airbnb.csv", index_col="id", usecols=usecols)
```

  - Si queremos indicar un separador de columnas determinado

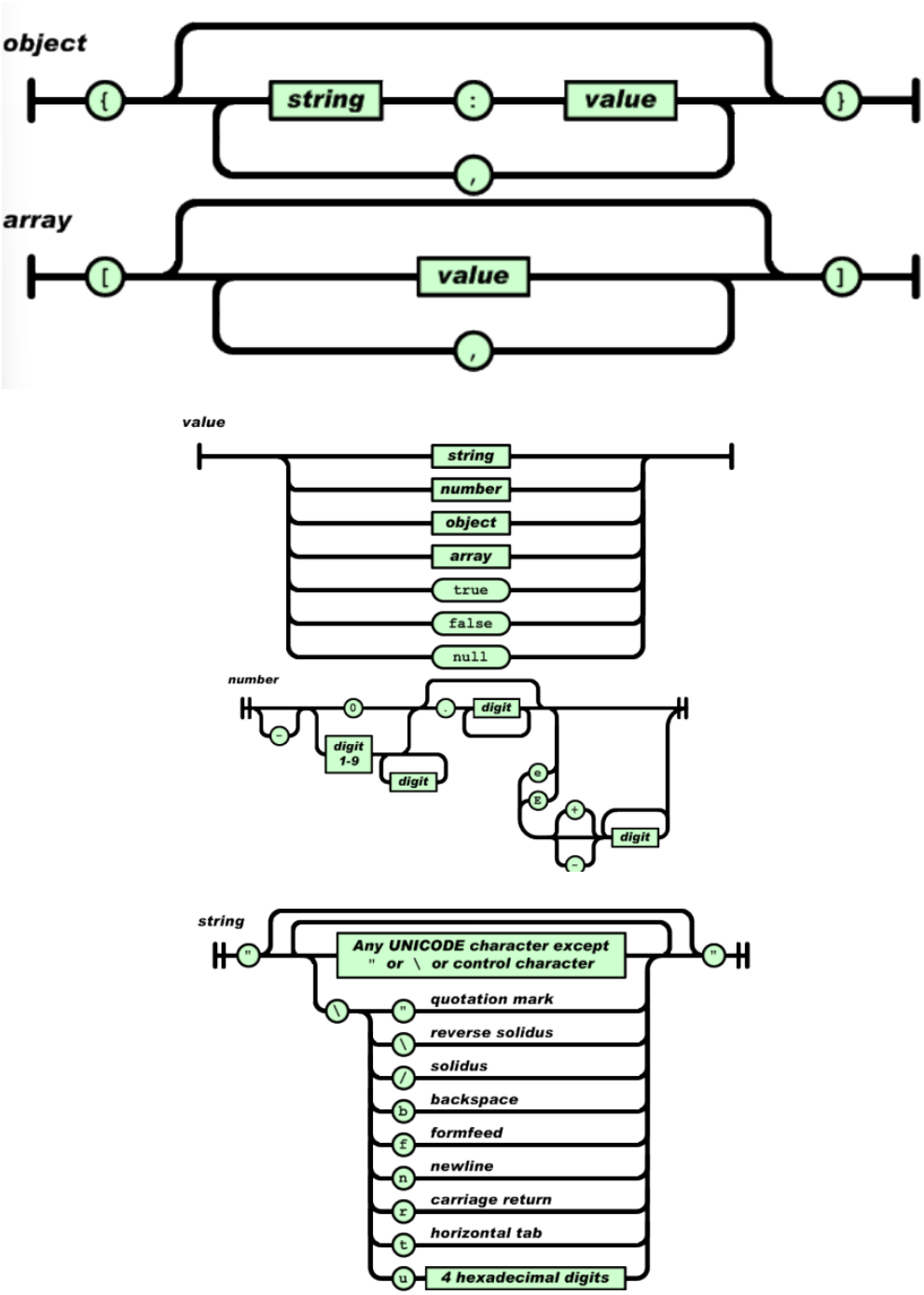
```
usecols = ["id", "nombre", "barrio", "precio", "noches_minimias"]
airbnb_data = pd.read_csv("airbnb.csv", index_col="id", usecols=usecols, sep="|")
```

- Si queremos definir el tipo de datos de determinadas columnas

```
usecols = ["id", "nombre", "barrio", "precio", "noches_minimias"]
airbnb_data = pd.read_csv("airbnb.csv", index_col="id", usecols=usecols, sep="|",
    dtype:{'precio': float, 'barrio':str, 'noches_minimas':int}, decimal=',')
```

1.2.4) JSON (JavaScript Object Notation)

Es, al mismo tiempo, un formato de archivo estándar abierto y un formato de intercambio de datos.
JSON se utiliza a menudo cuando los datos se envían desde un servidor a una página web.
En muchas ocasiones se define a JSON como *autodescriptivo* y fácil de entender.



- JSON Lines (JSONL)

JSON Lines es un formato práctico para almacenar datos estructurados que pueden procesarse de uno en uno.
Ficheros con extensión `.jsonl`

Es un gran formato para archivos de registros.

Sigue las mismas convenciones que JSON salvo que el carácter `\n` se usa como delimitador de líneas.

Cada línea de un fichero `.jsonl` es un JSON válido.

```
{"nombre": "Gilbert", "victorias": [["escalera"], ["pareja"]]}
{"nombre": "Alexa", "victorias": [["dobles parejas"], ["dobles parejas"]]}
{"nombre": "Maya", "victorias": []}
{"nombre": "Marisa", "victorias": [["trio"]]}
```

- Lectura con **`Pandas DataFrame`**
  - Lectura básica de un fichero JSON

```
import pandas as pd
df = pd.read_json('data.json')
```

  - Por defecto, Pandas sigue una orientación basada en columnas en la lectura de los ficheros {`columna -> {indice -> valor`}}

```
json_str = '{"Cursos":{"r1":"Spark"},"Tasas":{"r1":"25000"},"Duracion":{"r1":"50
    días"}}'
df = pd.read_json(json_str)
print(df)
```

```
   Cursos  Tasas Duracion
r1  Spark  25000  50 días
```

  - Otras opciones son `index`, `records`, `split` y `values`.
  - Si usaramos la orientación records ...

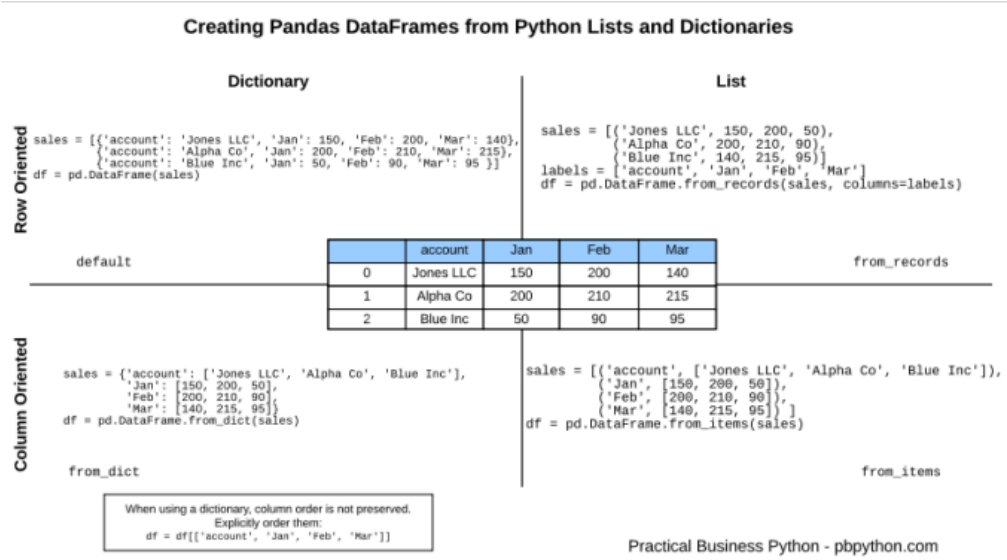
```
json_str = '[{"Cursos":"Spark","Tasas":"25000","Duracion":"50
  días","Descuento":"2000"}]'
df = pd.read_json(json_str, orient='records')
print(df)
```

```
   Cursos  Tasas Duracion Descuento
0   Spark  25000  50 días   2000
```

  - Estableciend el parámetro `lines` a `True`

```
df = pd.read_json('cursos.json', orient='records', nrows=2, lines=True)
```

- Resumen



1.2.5) Apache Avro

Avro ofrece una serie de características:

- Estructuras de datos ricas
- Un formato de datos compacto, rápido y binario
- Un lenguaje de especificación de esquema (Proto + IDL)
- Un formato de fichero contenedor para almacenar datos
- Un esquema de llamada a procedimiento remoto (RPC)
- Integración con lenguajes dinámicos, donde no hace falta recompilar el IDL (opcionalmente se puede hacer para lenguajes estáticos)

1) Schema

- Una declaración de esquema se escribe en JSON. Bien un nombre de un objeto predefinido, o bien un objeto JSON de la forma:

```
{"type": "nombreTipo", ... atributos ... }
```

- Ofrece tipos primitivos y complejos. Primitivos: `null, boolean, int, long, float, bytes, string, . . .`
- `""type": "string""` es equivalente a "`string`".
- Tipos complejos permitidos: registros (records), enums, mapas, arrays, uniones.
- Tipos complejos: Récords (registros)
  - El campo `type` se establece a `"record"`.
  - El campo `name` establece su nombre.
  - El campo `fields` establece sus campos. Cada campo: