

Bases de Datos II

Francisco Javier Mercader Martínez

Índice

1	Recuperación de datos y formatos de serialización	1
1.1	Necesidad de formatos de serialización	1
1.2	Características a analizar	1
1.2.1	Pandas	2
1.2.2	XML (eXtensible Markup Language)	2
1.2.3	CSV (Comma-Separated Values)	6
1.2.4	JSON (JavaScript Object Notation)	8
1.2.5	Apache Avro	10
1.2.6	Thrift	13
1.2.7	Pre-procesamiento de datos con Pandas	15
1.3	Conclusiones	16

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 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 puede 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, limpieza 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

```
1 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](#)

```

1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <DISCO CODIGO="B000067FSG">
3   <TITULO>Estrella de Mar</TITULO>
4   <ARTISTA>Amaral</ARTISTA>
5   <ESTILO>Pop</ESTILO>
6   <REFERENCIA>
7     <EDITORIA>Virgin</EDITORIA>
8     <AÑO_EDICION>2002<AÑO_EDICION>
9   </REFERENCIA>
10  <MUSICOS>
11    <MUSICO ROL="cantante">Amaral</MUSICO>
12    <MUSICO ROL="guitarra">Juan Aguirre</MUSICO>
13  </MUSICOS>
14 </DISCO>

```

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

• DTD

```

1 <!ELEMENT DISCO (TITULO, ARTISTA, ESTILO?, REFERENCIA, MUSICOS)>
2 <!ATTLIST DISCO CODIGO ID #REQUIRED>
3 <!ATTLIST DISCO TIPO=(CD | LP | DVD) "CD">
4 <!ELEMENT TITULO (#PCDATA)>
5 ...
6 <!ELEMENT REFERENCIA (EDITORIA, AÑO_EDICION) >
7 <!ELEMENT MUSICOS (MUSICO*)>
8 ...

```

Describe los documentos XML \Rightarrow **Validación**

• DTD (ii)

Documento XML:

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

```

1 <!DOCTYPE web-app
2   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
3   "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*)

```
1 <?xml version="1.0">
2 <xsd:schema xmlns:xsd="http://www.w3c.org/2000/08/XMLSchema">
3 <xsd:element name="Disco" type="DiscoTipo"/>
4 <xsd:complexType name="DiscoTipo">
5   <xsd:attribute name="codigo" type="String"/>
6   <xsd:sequence>
7     <xsd:attribute name="Titulo" type="String"/>
8     <xsd:attribute name="Artista" type="String"/>
9     <xsd:attribute name="Referencia" type="ReferenciaTipo"/>
10    ...
```

- 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**

```

1 import xml.sax
2
3 class XMLHandler(xml.sax.ContentHandler):
4     def __init__(self):
5         # Inicializamos variables de interés
6
7         # Se llama cuando comienza un nuevo elemento
8         def StartElement(self, tag, attributes):
9             pass
10        # Se llama cuando un elemento acaba
11        def endElement(self, tag):
12            pass
13
14 parser = xml.sax.make_parser()
15 parser.setFeature(xml.sax.handler.feature_namespaces, 0)
16 Handler = XMLHandler()
17 parser.setContentHandler(Handler)
18 parser.parse('models.xml') # nombre del documento a analizar

```

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

```

1 <collection shelf="New Arrivals">
2     <model number="ST001">
3         <price>35000</price>
4         <qty>12</qty>
5         <company>Samsung</company>
6     </model>
7     <model number="RW345">
8         <price>46500</price>
9         <qty>14</qty>
10        <company>Onida</company>
11    </model>

```

```
12 </collection>
```

- Parser DOM

Librería `xml.dom`

```
1 # Procesar un determinado fichero
2 file = minidom.parse('model.xml')
3
4 # Obtener los elementos con un determinado tag
5 modelos = fil.getElementsByTagName('modelo')
6
7 # Obtener el atributo 'nombre' del segundo
  modelo
8 print('modelo #2 atributos:')
9 print(modelos[1].attributes['nombre'].value)
10
11 # El datos de un item específico
12 print('\nmodelo #2 datos:')
13 print(modelos[1].firstChild.data)
```

```
1 <data>
2     <modelos>
3         <modelo name='modelo1'>
4             modelo1abc
5         </modelo>
6         <modelo name="modelo2">
7             modelo2abc
8         </modelo>
9     </modelos>
10 </data>
```

- Librería BeautifulSoup

```
1 from bs4 import BeautifulSoup
2
3 # Leemos el fichero
4 with open('models.xml', 'r') as f:
5     data = f.read()
6
7 # Pasamos los datos al parse
8 bs_data = BeautifulSoup(data, 'xml')
9
10 # Buscamos todas las instancias 'unique'
11 b_unique = bs_data.find_all('unique')
12 print(b_unique)
13
14 # Usamos .find búsquedas más concretas
15 b_name = bs_data.find('child', {'name': 'Acer'})
16 print(b_name)
```

```
1 <modelo>
2     <child name="Acer" qty="12">
3         Portátil Acer
4     </child>
5     <unique>
6         Número de modelo
7     </unique>
8     <child name="Acer" qty="7">
9         Exclusive
10    </child>
11    <unique>
12        1.200€
13    </unique>
14 </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 columnas (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

```

1 LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name'
2   [REPLACE | IGNORE]
3   INTO TABLE tbl_name
4   [PARTITION (partition_name, ...)]
5   [CHARACTER SET charset_name]
6   [{FIELDS | COLUMNS}
7     [TERMINATE BY 'string']
8     [[OPTIONALLY] ENCLOSED BY 'char']
9     [ESCAPED BY 'char']]
10  ]
11  [LINES
12    [STARTING BY 'string']
13    [TERMINATE BY 'string']]
14  ]
15  [IGNORE number {LINES | ROWS}]
16  [(col_name_or_user_var, ...)]
17  [SET col_name = expr, ...]

```

```

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

```

- Programación: Lectura con Pandas DataFrame

- Lectura básica de un fichero CSV

```

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

```

- Si queremos establecer la columna `id` como índice

```

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

```

- Si queremos leer solo un conjunto de columnas

```

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

```

- Si queremos indicar un separador de columnas determinado

```

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

```


- Si queremos definir el tipo de datos de determinadas columnas

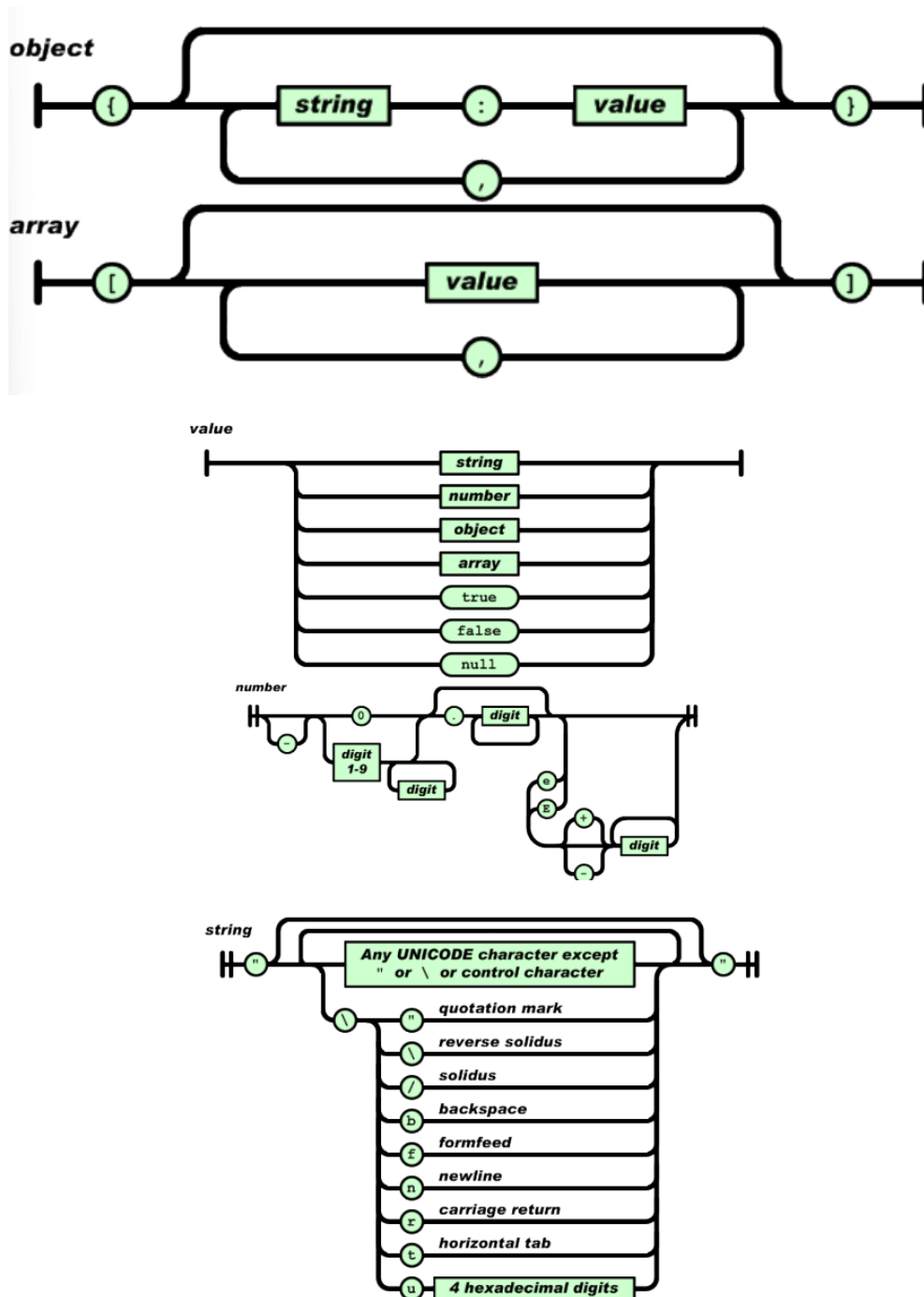
```
1 usecols = ["id", "nombre", "barrio", "precio", "noches_minimias"]
2 airbnb_data = pd.read_csv("airbnb.csv", index_col="id", usecols=usecols, sep="|",
    dtype: {'precio': float, 'barrio': str, 'noches_minimias': int}, decimal=',')
```

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.

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

• Lectura con Pandas DataFrame

- Lectura básica de un fichero JSON

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

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

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

	Cursos	Tasas	Duracion
r1	Spark	25000	50 días

- Otras opciones son `index`, `records`, `split` y `values`.

- Si usamos la orientación `records` ...

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

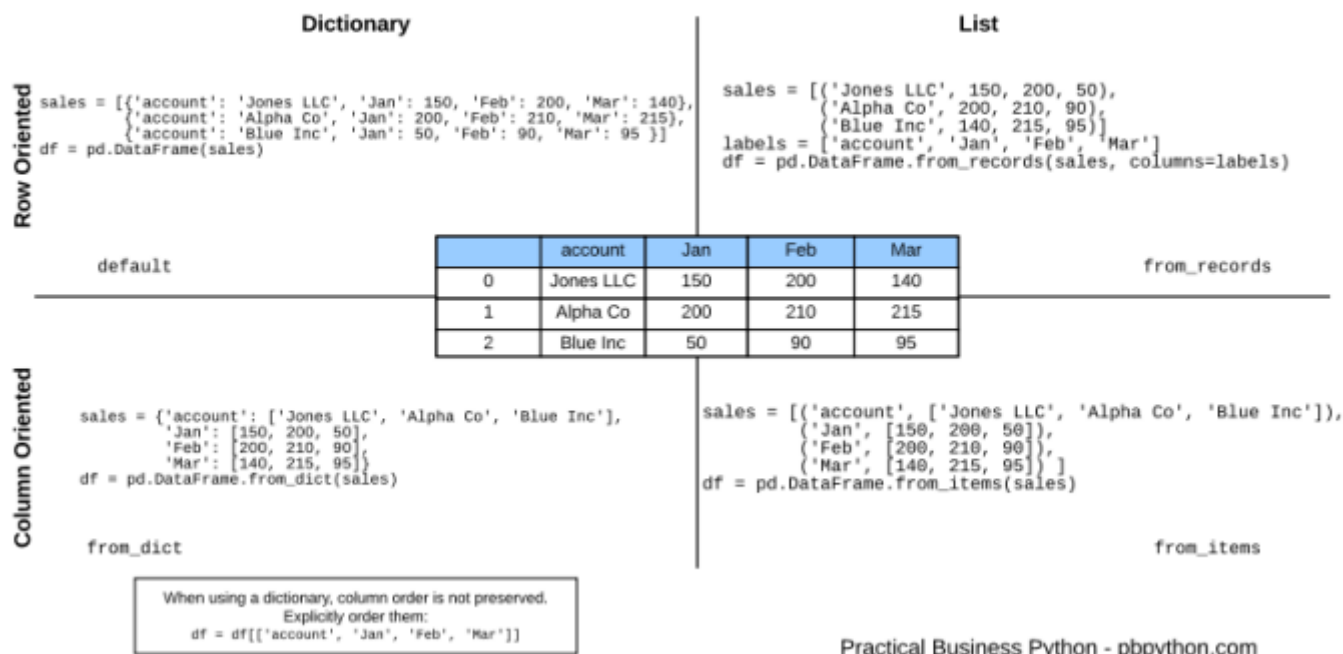
	Cursos	Tasas	Duracion	Descuento
0	Spark	25000	50 días	2000

- Estableciend el parámetro `lines` a `True`

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

• Resumen

Creating Pandas DataFrames from Python Lists and Dictionaries



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

```
1 {"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: Récor ds (registros)
 - El campo `type` se establece a `"record"`.
 - El campo `name` establece su nombre.
 - El campo `fields` establece sus campos. Cada campo:

→ Atributos name, doc, type y default

```
1 {
2   "type": "record",
3   "name": "LongList",
4   "fields" : [
5     {"name": "value", "type": "long"}, // el campo 'value' toma valores long
6     {"name": "next", "type": ["null", "LongList"]} // 'next' es opcional
7   ]
8 }
```

- Tipos complejos: Arrays

```
1 {"type": "array", "items": "string"}
```

- Tipos complejos: Mapas (los índices se suponen siempre string)

```
1 {"type": "map", "values": "long"}
```

- Los protocolos permiten especificar la parte RPC del protocolo y especificar funciones y mensajes entrantes y salientes.

- Ejemplo (mail.avpr):

```
1 {"namespace": "example.proto",
2   "protocol": "Mail",
3
4   "types": [
5     {"name": "Message", "type": "record",
6       "fields": [
7         {"name": "to", "type": "string"},
8         {"name": "from", "type": "string"},
9         {"name": "body", "type": "string"}
10      ]
11    }
12  ],
13
14  "messages": {
15    "send": {
16      "request": [{"name": "message", "type": "Message"}],
17      "response": "string"
18    }
19  }
20 }
```

- Programación

[Librería avro](#)

Instalación con ejecutando el comando

```
1 pip install avro
```

Vamos a suponer el siguiente esquema:

```
1 {"namespace": "example.avro",
2   "type": "record",
3   "name": "User",
4   "fields": [
5     {"name": "name", "type": "string"},
6     {"name": "favorite_number", "type": ["int", "null"]},
7     {"name": "favorite_color", "type": ["string", "null"]}
8   ]
9 }
```

Para serializar dos usuarios en el fichero `users.avsc`:

```
1 import avro.schema
2 from avro.datafile import DataFileReader, DataFileWriter
3 from avro.io import DatumReader, DatumWriter
4
5 schema = avro.schema.parse(open("user.avsc", "rb").read())
6
7 writer = DataFileWriter(open("users.avro", "wb"), DatumWriter(), schema)
8 writer.append({"name": "Alyssa", "favorite_number": 256})
9 writer.append({"name": "Ben", "favorite_number": 7, "favorite_color": "red"})
10 writer.close()
```

Para leer dichos usuarios de fichero

```
1 reader = DataFileReader(open("users.avro", "rb"), DatumReader())
2 for user in reader:
3     print user
4 reader.close()
```

[Librería avro con Protocols](#)

Uso del protocolo definido anteriormente.

```
1 import avro.ipc as ipc
2 import avro.protocol as protocol
3
4 PROTOCOL = protocol.Parse(open("mail.avpr").read())
5 server_addr = ('localhost', 9090)
6
7 client = ipc.HTTPTransceiver(server_addr[0], server_addr[1])
8 requestor = ipc.Requestor(PROTOCOL, client)
9
10 message = dict()
11 message['to'] = sys.argv[1]
12 message['from'] = sys.argv[2]
13 message['body'] = sys.argv[3]
14
15 params = dict()
```

```

16 params['message'] = message
17 print("Result: " + requestor.Request('send', params))
18
19 client.Close()

```

1.2.6) Thrift

Framework de serialización + RPC desarrollado por Facebook.

Requiere de compilador siempre.

Genera *stubs* en muchos lenguajes de programación, incluyendo C++, Java, Python, . . .

Incluye un lenguaje completo de especificación de interfaces (IDL) (`.thrift`), parecido a C++.

Cada elemento de estructura o parámetro de función debe ir precedido por su etiqueta de identificación numérica.

El formato de serialización puede cambiarse, y acepta formatos binarios y de texto.

Además del modelo, genera un *processor*, que hace de procesador de mensajes del servicio.

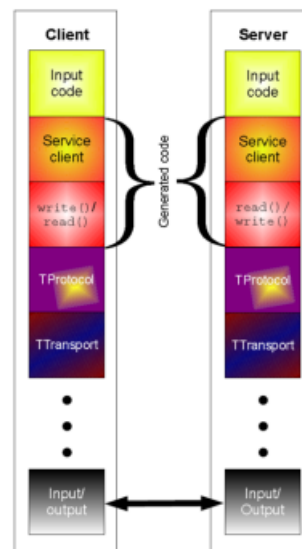
Permite el uso de varios protocolos:

- TBinaryProtocol, TDenseProtocol, TJSONProtocol, TSimpleJSONProtocol

Permite el uso de varios transportes:

- TSocket, TFileTransport (a fichero), TZlibTransport (compresión)

Processor – Permite leer y escribir mensajes del modelo de datos



- IDL

Tipos básicos: `bool`, `byte`, `i16`, `string`, etc.

Estructuras:

```

1 struct Example {
2     1:i32 number=10,
3     2:i64 bigNumber,
4     3:double decimals,
5     4:string name="thrifty"
6 }

```

Contenedores: `list`, `set`, `map` con sintaxis C++.

Servidores RPC:

```

1 service StringCache {
2     void set(1:i32 key, 2:string value),
3     string get(1:i32 key) throws (1:KeyNotFound knf),
4     void delete(1:i32 key)
5 }

```

- Código

Implementar un servicio Calculator que sume dos números

- 1) Definir el fichero .thrift con la especificación del servicio

```

1 service Calculator extends shared.SharedService {
2
3     void ping(),
4
5     i32 add (1:i32 num1, 2:i32 num2),
6 }

```

- 2) Generar el código en Python ejecutando el siguiente comando

```

1 thrift --gen py multiple.thrift

```

- 3) Definir el handler en el servidor

```

1 class CalculatorHandler:
2     def __init__(self):
3         self.log = {}
4
5     def ping(self):
6         print('ping()')
7
8     def add(self, n1, n2):
9         print('add(%d, %d)' % (n1, n2))
10        return n1+ n2

```

- 4) Arrancar el servidor

```

1 handler = CalculatorHandler()
2 processor = Calculator.Processor(handler)
3 transport = TSocket.TServerSocket(host='127.0.0.1', port=9090)
4 tfactory = TTransport.TBufferedTransportFactory()
5 pfactory = TBinaryProtocol.TBinaryProtocolFactory()
6
7 server = TServer.TSimpleServer(processor, transport, tfactory, pfactory)
8
9 print('Arrancando el servidor...', end='')
10 server.serve()
11 print('HECHO!')

```

5) Lanzar el cliente que hace uso del servicio

```
1 # Creamos un socket
2 transport = TSocket.TSocket('localhost', 9090)
3
4 # Los sockets pueden ser lentos. Necesitamos bufferizar
5 transport = TTransport.TBufferedTransport(transport)
6
7 # Definimos el protocolo Thrift a usar
8 protocol = TBinaryProtocol.TBinaryProtocol(transport)
9
10 # Creamos un cliente para usar el protocolo definido.
11 client = Calculator.Client(protocol)
12
13 #Conectamos con el servidor
14 transport.open()
15
16 client.ping()
17 print('ping()')
18
19 sum_ = client.add(1,1)
20 print('1+1=%d', % sum_)
```

1.2.7) Pre-procesamiento de datos con Pandas

• Datos faltantes

En muchos escenarios, nos encontraremos que los conjuntos de datos que con los que tenemos que trabajar se encuentran con valores faltantes.

La librería **Pandas** nos permite tratar dichos casos con algunos métodos de su herramienta **Dataframe**.

El método `.dropna` permite eliminar las filas (`axis=0`) o columnas (`axis=1`) de un dataframe

```
1 d= {'A':[1,2,np.nan], 'B':[5, np.nan, np.nan],
2     'C':[1,2,3]}
3 df= pd.DataFrame(d)
4 df.dropna() #axis=0
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

	A	B	C
0	1.0	5.0	1

El método `.dropna` permite definir cuántos valores no **NaN** deben existir en una fila o columna para no ser borrada con el parámetro `thresh`

```
1 d= {'A':[1,2,np.nan], 'B':[5, np.nan, np.nan],
2     'C':[1,2,3]}
3 df= pd.DataFrame(d)
4 df.dropna(thresh=2) #axis=0
```

Otra alternativa cuando tenemos datos faltantes puede ser sustituir dichos datos.

Esto puede realizarse con el método `.fillna` especificando un valor por defecto

```
1 d= {'A':[1,2,np.nan], 'B':[5, np.nan, np.nan],
2     'C':[1,2,3]}
3 df= pd.DataFrame(d)
4 df.fillna(value='FILL VALUE')
```

Sin embargo una estrategia más acertada podría ser usar la media de cada columna

```
1 df['A'].fillna(value=df['A'].mean())
```

La propagación de valores es también otra alternativa. Sobre todo cuando existe cierto orden pre-establecido entre filas y columnas

Con `.ffill` podemos propagar el último valor válido hacia adelante

```
1 df = pd.DataFrame([[np.nan, 2, np.nan, 0],
2                    [3, 4, np.nan, 1],
3                    [np.nan, np.nan, np.nan, np.nan],
4                    [np.nan, 3, np.nan, 4]],
5                    columns=list("ABCD"))
6 df.fffll()
```

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	NaN	1.0
2	NaN	NaN	NaN	NaN
3	NaN	3.0	NaN	4.0

	A	B	C	D
0	NaN	2.0	NaN	0.0
1	3.0	4.0	NaN	1.0
2	3.0	4.0	NaN	1.0
3	3.0	3.0	NaN	4.0

De forma similar, `.bfill` propaga hacia atrás el siguiente valor válido.

En caso de una relación temporal entre filas o columnas de un Dataframe, lo más conveniente es rellenar los valores faltantes con el método `.interpolate`.

```
1 s = pd.Series([0, 1, np.nan, 3])
2 s.interpolate()
```

0	0.0	0	0.0
1	1.0	1	1.0
2	NaN	2	2.0
3	3.0	3	3.0

Dicho método permite aplicar diferentes técnicas de regresión para imputar los valores faltantes (`quadratic`, `cubic`, `polynomial`...)

```
1 s.interpolate(method='polynomial', order=2)
```

Otro tipo de *impureza* frecuente que suele darse en los conjuntos de datos son las filas o columnas duplicadas. Podemos eliminarlas con el método `.drop_duplicates`

```
1 df.drop_duplicates()
```

También podemos forzar que determinadas columnas se conviertan al tipo de datos `Date` con `.to_datetime`

```
1 df['Date'] = pd.to_datetime(df['Date'])
```

1.3) Conclusiones

En el ecosistema digital actual, existe una gran variedad de alternativas para llevar a cabo la serialización de datos

XML, JSON y CSV son los formatos más extendidos

Otras soluciones, como Avro o Thrift, van más allá al definir un modelo de datos y un RPC para la implementación de sistemas distribuidos

No existe una *bala de plata* sino que dependiendo de la naturaleza de los datos y el contexto, deberemos optar por uno u otro enfoque

Independientemente del formato de serialización escogido es imprescindible un pre-procesado y limpieza de los datos para eliminar o sustituir valores incorrectos o imprecisos