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

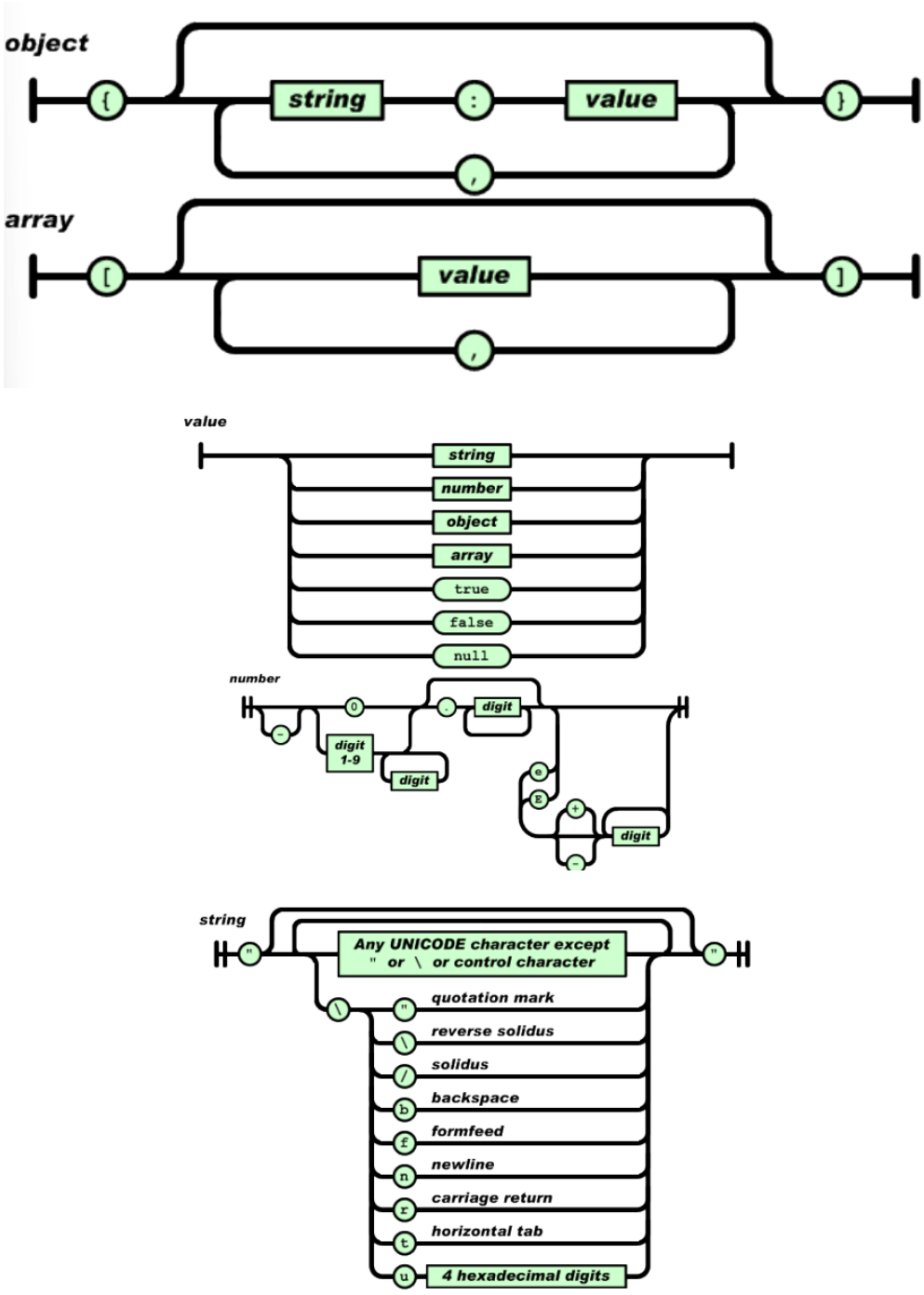
```
usecols = ["id", "nombre", "barrio", "precio", "noches_minimias"]
airbnb_data = pd.read_csv("airbnb.csv", index_col="id", usecols=usecols, sep="|")
```

- 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 ...

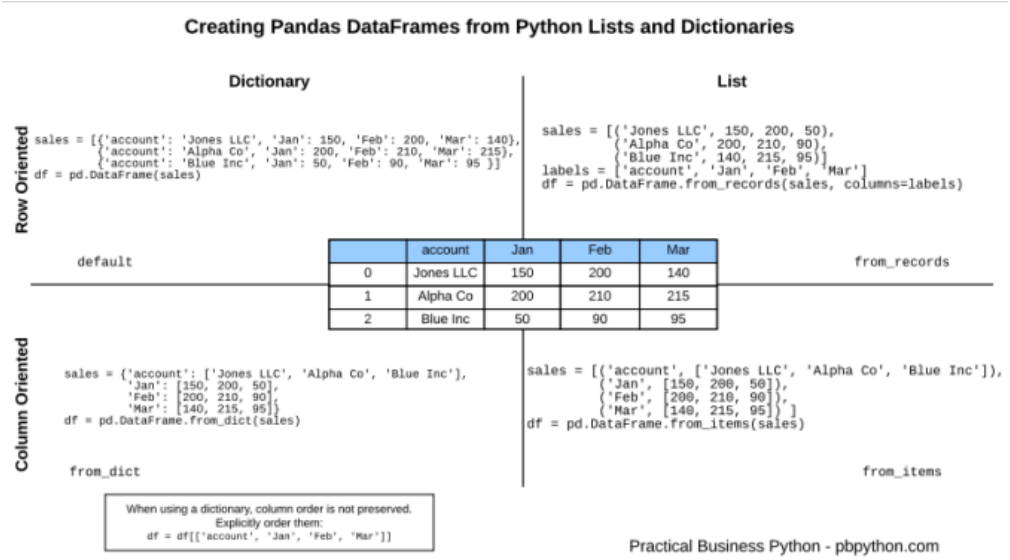
```
json_str = '[{"Cursos":"Spark","Tasas":"25000","Duracion":"50
    días","Descuento":"2000"}]'
df = pd.read_json(json_str, orient='records')
print(df)
```

```
    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)

- 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:
      → Atributos `name, doc, type` y `default`

```
{
    "type": "record",
    "name": "LongList",
    "fields" : [
        {"name": "value", "type": "long"}, // el campo 'value' toma valores long
        {"name": "next", "type": ["null", "LongList"]} // 'next' es opcional
    ]
}
```

- Tipos complejos: Arrays

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

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

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

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

- Ejemplo (`mail.avpr`):

```
{"namespace": "example.proto",
    "protocol": "Mail",

    "types": [
        {"name": "Message", "type": "record",
        "fields": [
            {"name": "to", "type": "string"},
            {"name": "from", "type": "string"},
            {"name": "body", "type": "string"}
          ]
        }
    ],

    "messages": {
        "send": {
            "request": [{"name": "message", "type": "Message"}],
            "response": "string"
        }
    }
}
```

- Programación

Librería **`avro`**

Instalación con ejecutando el comando

```
pip install avro
```

Vamos a suponer el siguiente esquema:

```
{"namespace": "example.avro",
    "type": "record",
    "name": "User",
    "fields": [
        {"name": "name", "type": "string"},
        {"name": "favorite_number", "type": ["int", "null"]},
        {"name": "favorite_color", "type": ["string", "null"]}
    ]
}
```

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

```
import avro.schema
from avro.datafile import DataFileReader, DataFileWriter
from avro.io import DatumReader, DatumWriter

schema = avro.schema.parse(open("user.avsc", "rb").read())

writer = DataFileWriter(open("users.avro", "wb"), DatumWriter(), schema)
writer.append({"name": "Alyssa", "favorite_number": 256})
writer.append({"name": "Ben", "favorite_number": 7, "favorite_color": "red"})
writer.close()
```

Para leer dichos usuarios de fichero

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

Librería **avro** con Protocols

Uso del protocolo definido anteriormente.

```
import avro.ipc as ipc
import avro.protocol as protocol

PROTOCOL = protocol.Parse(open("mail.avpr").read())
server_addr = ('localhost', 9090)

client = ipc.HTTPTransceiver(server_addr[0], server_addr[1])
requestor = ipc.Requestor(PROTOCOL, client)

message = dict()
message['to'] = sys.argv[1]
message['from'] = sys.argv[2]
message['body'] = sys.argv[3]

params = dict()
```

```
params['message'] = message
print("Result: " + requestor.Request('send', params))

client.Close()
```

1.2.6) Thrift

Framework de serialización + RPC desarrollado por Facebook.

Requiere de compilador siempre.

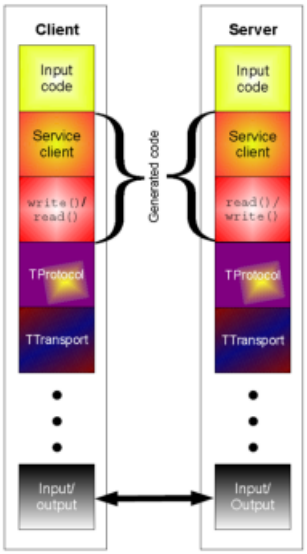
Genera *stubs* en muchos lenguajes de programación, incluyendo C++, Java, Python, . . .

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`, `TSimpleJSONProtol`

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:

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

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

Servidores RPC:

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

- Código

Implementar un servicio `Calculator` que sume dos números

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

```
service Calculator extends shared.SharedService {

    void ping(),

    i32 add (1:i32 num1, 2:i32 num2),
}
```

2) Generar el código en Python ejecutando el siguiente comando

```
thrift --gen py multiple.thrift
```

3) Definir el `handler` en el servidor

```
class CalculatorHandler:
    def __ init__(self):
        self.log = {}

    def ping(self):
        print('ping()')

    def add(self, n1, n2):
        print('add(%d, %d)') % (n1, n2)
        return n1+ n2
```

4) Arrancar el servidor

```
handler = CalculatorHandler()
processor = Calculator.Processor(handler)
transport = TSocket.TServerSocket(host='127.0.0.1', port=9090)
tfactory = TTransport.TBufferedTransportFactory()
pfactory = TBinaryProtocol.TBinaryProtocolFactory()

server = TServer.TSimpleServer(processor, transport, tfactory, pfactory)

print('Arrancando el servidor...', end='')
server.serve()
print('HECHO!')
```

5) Lanzar el cliente que hace uso del servicio

```
# Creamos un socket
transport = TSocket.TSocket('localhost', 9090)

# Los sockets pueden ser lentos. Necesitamos bufferizar
transport = TTransport.TBufferedTransport(transport)

# Definimos el protocolo Thrift a usar
protocol = TBinaryProtocol.TBinaryProtocol(transport)

# Creamos un cliente para usar el protocolo definido.
client = Calculator.Client(protocol)

#Conectamos con el servidor
transport.open()

client.ping()
print('ping()')

sum_ = client.add(1,1)
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

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

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

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

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

```
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

```
df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                   [3, 4, np.nan, 1],
                   [np.nan, np.nan, np.nan, np.nan],
                   [np.nan, 3, np.nan, 4]],
                  columnas=list("ABCD"))
df.ffill()
```

	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 res relllenar los valores faltantes con el método `.interpolate`.

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

0	0.0
1	1.0
2	NaN
3	3.0

0	0.0
1	1.0
2	2.0
3	3.0

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

```
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`

```
df.drop_duplicates()
```

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

```
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

Tema 2: Introducción a los sistemas NoSQL

2.1) Introducción a NoSQL

NoSQL ⟶ *hastag* llamativo que se eligió para una conferencia en 2009 (Johan Oskarsson de Last.fm)

Ahora se asocia a cientos de bases de datos diferentes, que se han clasificado en varios tipos (las veremos después), caracterizadas por **no usar SQL** como modelo de datos.

Más recientemente **NoSQL** ⟶ ***Not Only SQL*** (no sólo SQL) ⟶ Persistencia políglota (*polyglot persistence*)

2.1.1) ¿Por qué se plantearon?

1. **Mayor escalabilidad horizontal**
   - conjuntos de datos muy muy grandes
   - sistemas de alto volumen de escrituras (**streaming** de eventos, aplicaciones sociales)
2. **Demanda de productos de software libre** (crecimiento de las *start-ups*)
3. **Consultas especializadas** no eficientes en el modelo relacional (JOINs)
4. **Expresividad, flexibilidad, dinamismo.** Frustración con **restricciones** del modelo relacional

2.1.2) Características

No se basan en SQL

Modelos de datos más ricos

Orientadas a la **Escalabilidad**

Generalmente no obligan a definir un esquema

- **Schemaless**

Surgidos de la comunidad para solucionar problemas

- muchas **libres/*open source***

Diseño basado en **procesamiento distribuido**

Principios funcionales

- **MapReduce**

2.1.2.1) Categorías de NoSQL

- Bases de datos *key-value*
- Bases de datos documentales
- Bases de datos columnares (*wide column*)

- Bases de datos de grafos

- Bases de datos de arrays

2.1.3) Evolución desde el modelo relacional

El **modelo relacional ⇒ predominante en los últimos ˜30 años**
Tiene sus raíces en el denominado *business data processing*, procesamiento de transacciones y *batch*
Propuesto por Codd en los 70, **de alto nivel**
Actualmente los **sistemas SQL están muy optimizados:**

- el **grado de implantación es mayoritario**

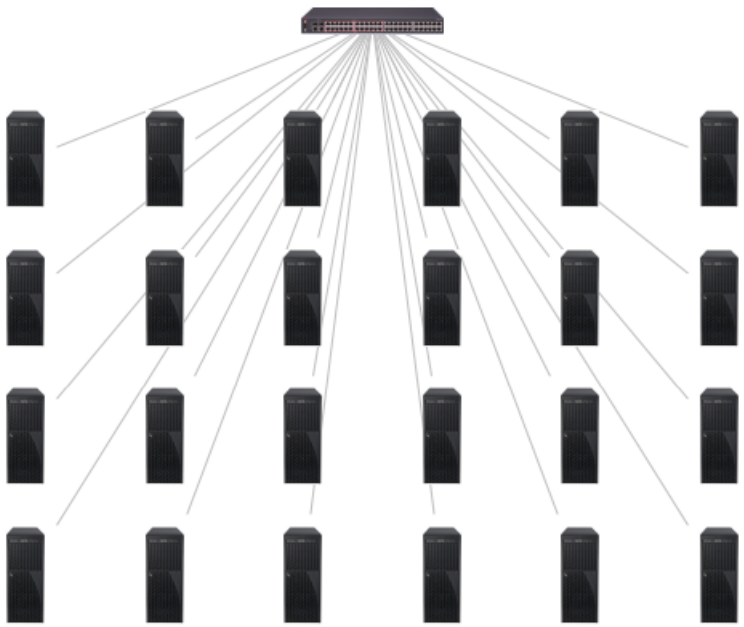
- para el 99 % de los problemas (que caben en un ordenador) es eficiente y adecuado

2.2) Adopción de NoSQL

2.2.1) Análisis

Dominan los grandes SGBDR
El *Open Source* tiene una importancia crucial (PostgreSQL, MySQL, MongoDB, etc.)
Varias bases de datos NoSQL entre las 10 primeras. Muchas en las 20 primeras
La distancia entre los grandes SGBDR y el primer NoSQL (MongoDB) es de 5×
Paradigmas más "atrevidos" como el de grafos están entre los 20 primeros (Neo4j)

2.3) Cambio de perspectiva: Red



2.3.0.1) Almacenamiento distribuido

Desde los 90's: Clústers/NOC/COW: procesamiento masivamente paralelo

Almacenamiento no distribuido

Ahora los nodos ⇒ también **almacenamiento**

Minimizar el verdadero cuello de botella: **trasiego de información por la red**.

2.3.0.2) Procesamiento distribuido

Necesidad de **paralelización máxima**
Escalabilidad
Explotar de la **localidad de los datos**:

- Datos producidos se utilizan localmente en siguientes iteraciones
- Datos recibidos directamente en los *hosts* (clientes simultáneos)

Vuelta al modelo funcional inherentemente paralelo: (e.g. **Map-Reduce**)
Almacenamiento distribuido: (e.g. **HDFS**)
Coordinación distribuida: (e.g. **Zookeeper**)

2.3.0.3) Modelo de datos

El modelo relacional limita a tablas con valores primitivos y relaciones *Primary Key/Foreign Key*
En programación se utilizan **listas, arrays, tipos de datos compuestos** (*gap semántico*)
ACID es **muy compleja y costosa** en ambientes distribuidos (quizá **no necesaria** en algunas aplicaciones).

¿Y si se pudiera ver como un **GRAN ARRAY**?

- Cada nodo almacenaría una parte del array
- Búsqueda aleatoria **muy rápida** (árboles B)
- Uso de **filtros de Bloom**
- Uso de **objetos complejos** (p. ej. **documentos JSON**), para mantener la **localidad espacial de datos relacionados**
- Transacciones limitadas al **objeto complejo**

2.4) Schemaless

Las Bases de Datos NoSQL (en general) **no quieren de un esquema**

Flexibilidad: Posibilidad de almacenar entidades con una estructura diferente

- Tratar información incompleta.
- Evolucionar la base de datos/esquema.
- Añadir nuevas características a las aplicaciones

schema-on-write ⇒	***schema-on-read***
SQL	**NoSQL**
Los datos conforman al esquema cuando se **escriben**	Los datos leídos conforman a un **esquema implícito**
Tipado estricto (estático)	***Duck-Typing*** (dinámico)
Datos homogéneos	Datos **heterogéneos**
Proceso analítico a través de **consultas**	***Use as read***

Ejemplo: Añadir el campo `first_name` a partir del campo `name`

Los nuevos objetos se crean con el nuevo formato

A la hora de leerlos, se puede hacer:

```
if (user && user.name && !user.first_name) {
    // Docs anteriores a 2013 no tienen first_name
    user.first_name = user.name.split(" ")[0];
}
```

En SQL puede ser un proceso muy costoso (procesa toda la tabla, locking, puede que haya que parar las aplicaciones):

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name =
    substring_index(name, ' ', 1);
```

2.4.1) ¿Cuándo es apropiado *schemaless*?

Objetos heterogéneos

Estructura de los datos **impuesta externamente**

Si intuimos que los datos **cambiarán en el futuro**

Sin embargo

A veces **un esquema es conveniente**

- Facilita el desarrollo y evita inconsistencias
  - `Mongoose` para MongoDB:

```
var Comment = new Schema({
    name: {type: String, default: 'Anonymous'},
    date: {type: Date, default: Date.now},
    text: Buffer
});
// a setter with on-line modification
Comment.path('name').set(function (v) {
    return capitalize(v);
});
```

2.5) Map-Reduce

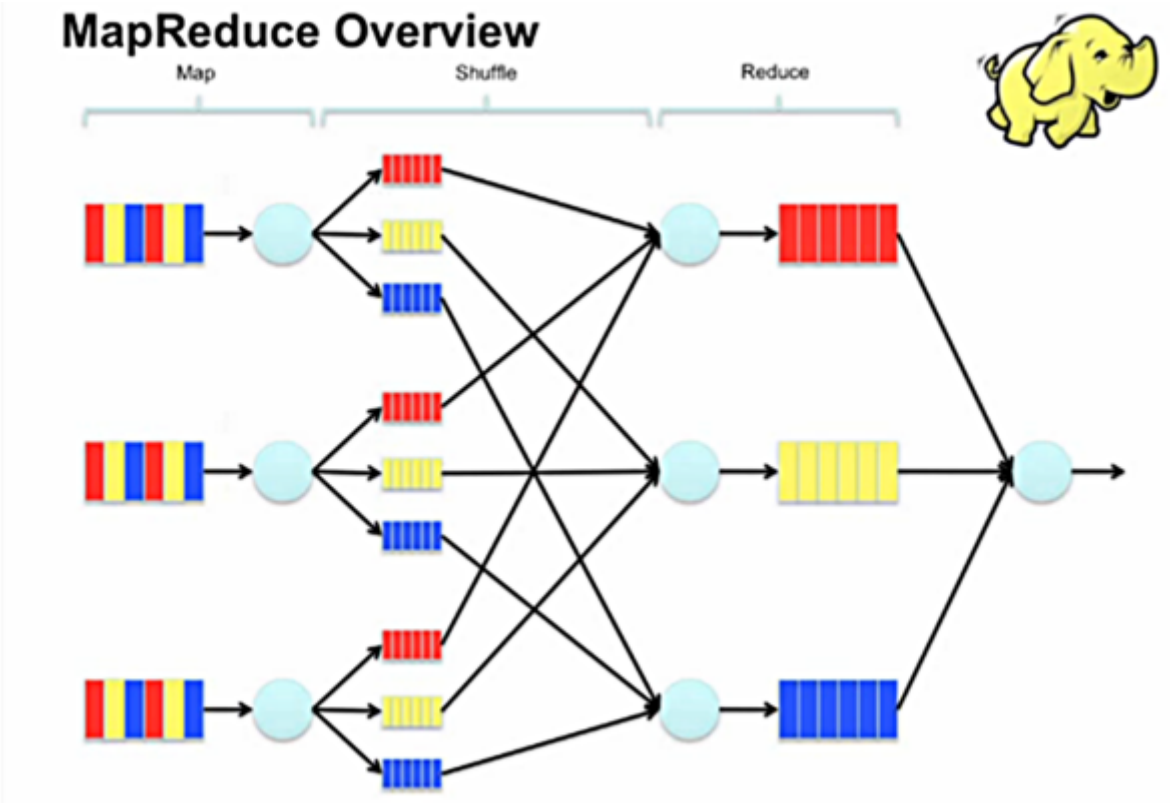
Map-Reduce: origen lenguajes funcionales:

- `map()`: Ejecuta una misma función sobre todos los elementos de un conjunto
- `reduce()`: Sumariza un conjunto de valores para producir un valor de salida

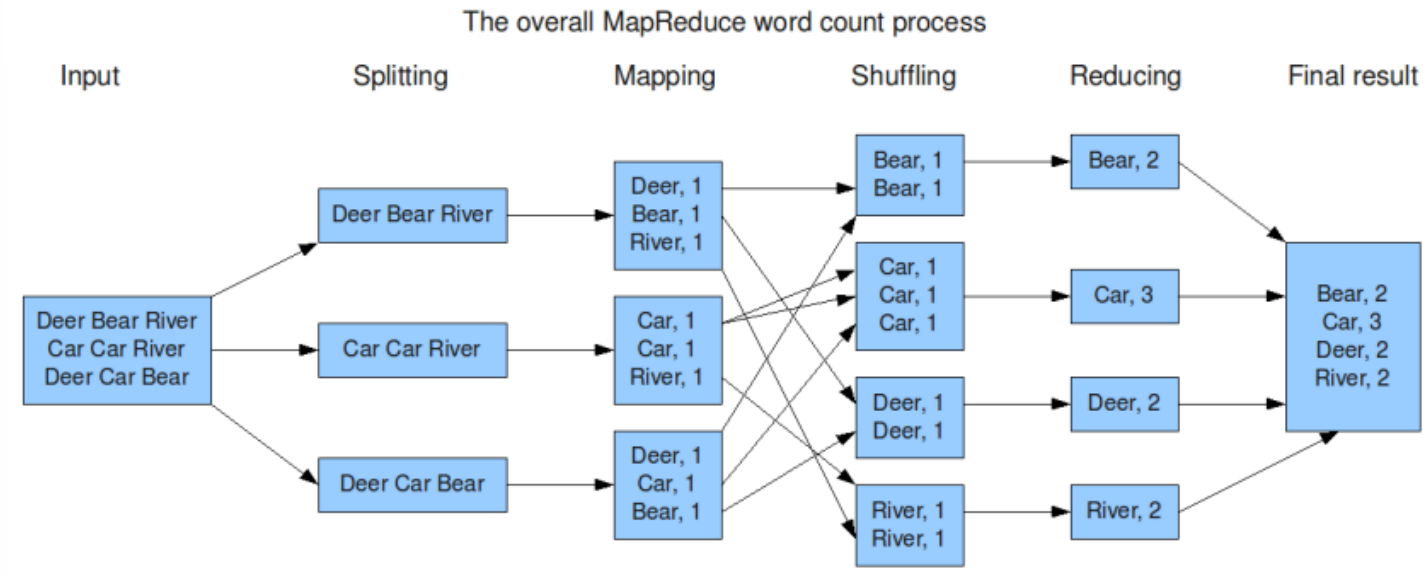
Map-Reduce combina ambas operaciones:

- Una misma operación `map()` a cada dato residente en un nodo es realizada de forma paralela en **todos** los nodos simultáneamente

- Con los resultados parciales de cada nodo, una función `reduce()` genera un resultado (o un conjunto de resultados) final
- Proceso intermedio de *shuffle* para agrupar valores **con la misma clave** antes del **reduce()**
- Resultados parciales en el mismo nodo (localidad) ⇒ procesamientos **en cadena**



- Word-Count



Map-Reduce en entornos Big-Data/NoSQL tiene una serie de particularidades:

- Se supone que los datos de entrada son siempre pares ¡ key, value ¿
- La función `map()` produce otro conjunto de valores $\{< key1, value1 >, < key2, value2 >, \dots\}$
- El **shuffle** agrupa los valores con la misma clave:

$$\{< key1, \{val1, val3, \dots\} >, < key2, \{val2, val4, \dots\} >, \dots\}$$

- `reduce()` procesa cada lista de valores con la misma clave, y produce otros elementos $< key', value' >$
- Hay procesamientos difíciles de expresar en Map-Reduce $\Rightarrow$ operaciones M/R **en cadena**

Se verá en profundidad en los siguientes temas de la asignatura

Map-Reduce puede usarse no sólo para computación distribuida, sino también como una **generalización de consultas**

Ejemplo: Imagínese un biólogo marino que hace anotaciones de cada animal que ve en el océano, y quiere saber cuántos tiburones ha visto por mes:

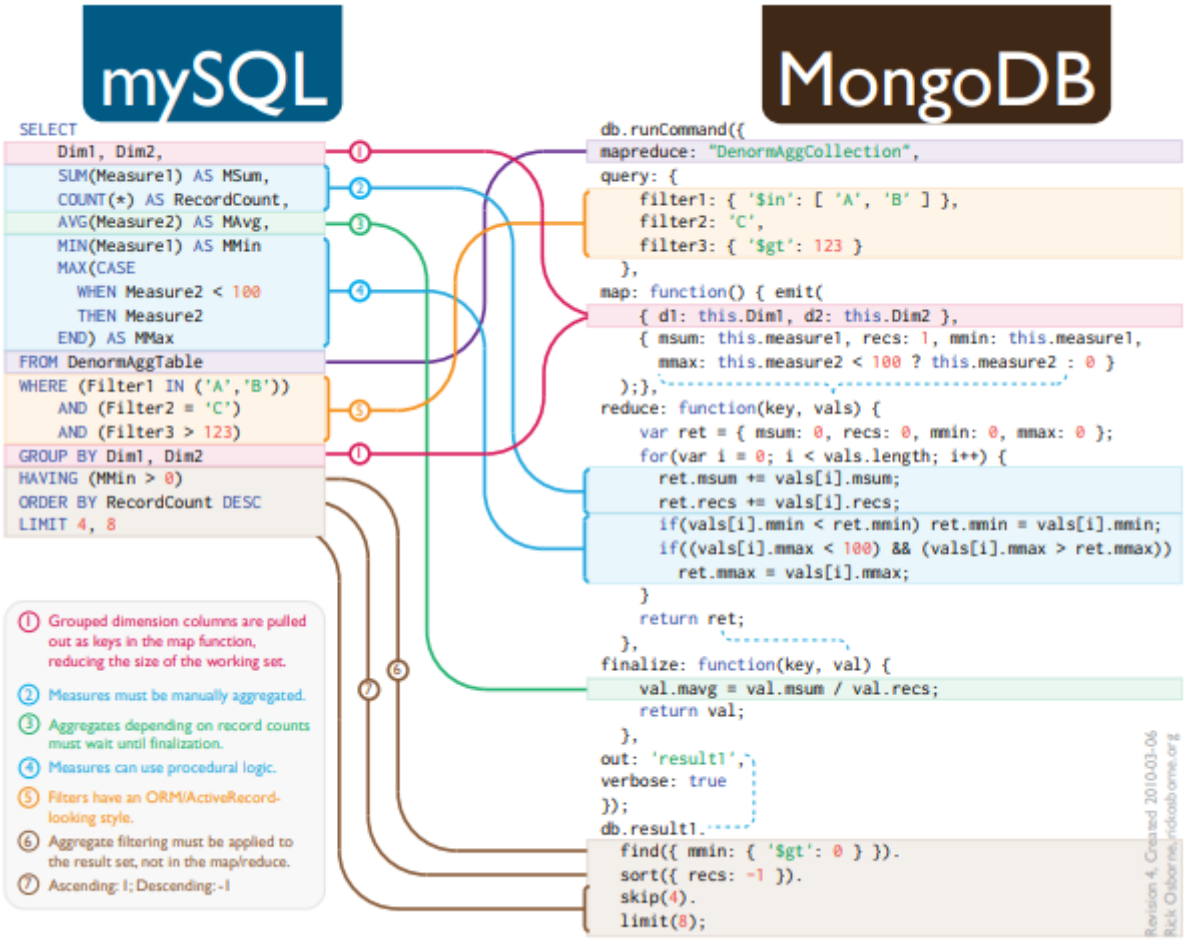
```
SELECT MONTH(observation_timestamp) AS observation_month,
       sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

MongoDB con el API de MapReduce:

```
db.observations.mapReduce(
    function map() {
        var year = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals);
    },
    function reduce(key, values) {
        return Array.sum(values);
    },
    {
        query: { family: "Sharks" },
        out: "monthlySharkReport"
    }
);
```

MongoDB ofrece además un API alternativo para funciones de agregación:

```
db.observations.aggregate([
    { $match: { family: "Sharks" } },
    { $group: {
        _id: {
            year: { $year: "$observationTimestamp" },
            month: { $month: "$observationTimestamp" }
        },
        totalAnimals: { $sum: "$sumAnimals" }
    }
    }
]);
```



2.5.1) Eficiencia *raw*

Pero los sistemas NoSQL tienen que competir también con los SQL en términos de eficiencia neta (también llamada raw).

La prueba se realizó sobre MongoDB y sobre MySQL (se ha adaptado el original, que era para PostgreSQL)

Se parte de una tabla sencilla con cuatro valores, que muestran medidas de sensores con localización, valor de la lectura y una marca de tiempo

Se realizan seis pruebas que pueden corresponder a un conjunto de consultas normales:

1) Inicialmente se insertan un millón de elementos generados al azar, con fechas que permitan la búsqueda por rango (**Fill**)

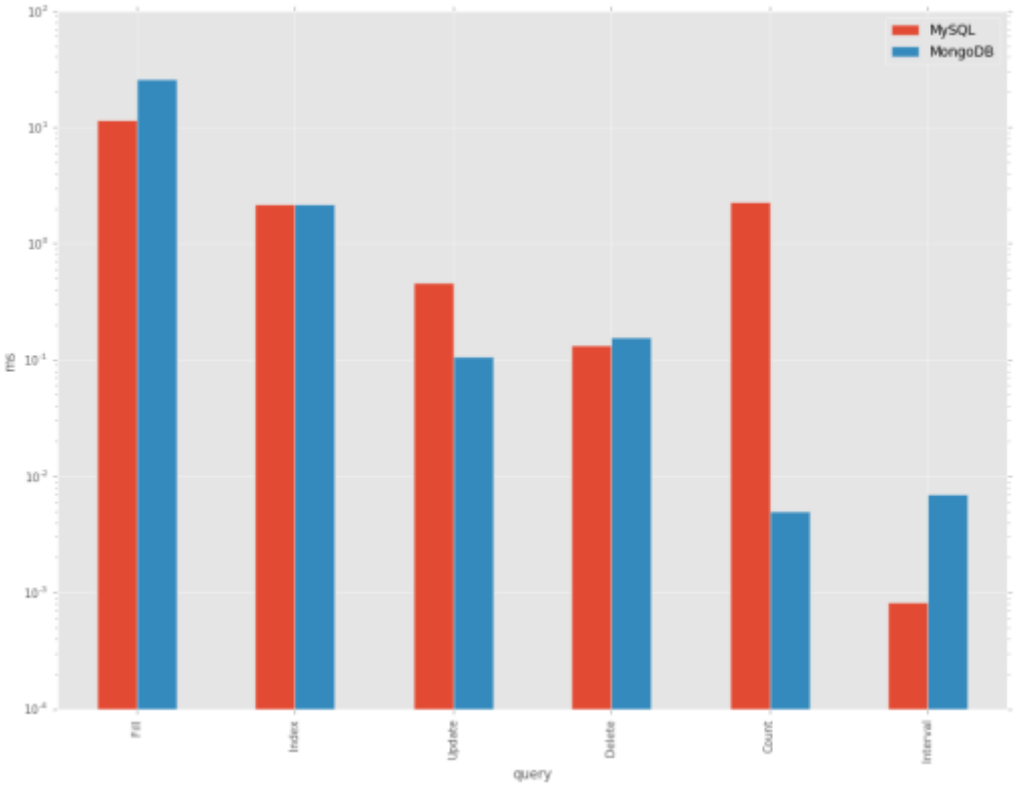
2) Se crea un índice en la tabla para la fecha de la lectura (**Index**)

3) Se actualizan los valores de un conjunto de entradas seleccionadas por rango de fechas (**Update**)

4) Se eliminan un conjunto de filas seleccionadas por rango de fechas (**Delete**)

5) Se obtiene el número de filas restantes (**Count**)

6) Se obtiene un subconjunto de filas extraído de una consulta dada por un rango de fechas **(Interval)**



El gráfico se muestra en escala logarítmica en el eje Y (las diferencias pequeñas se acentúan)

A simple vista, ambos productos están muy igualados

- SQL (MySQL) lleva *muchos años* de optimizaciones
- Mientras que productos como MongoDB tienen menos historia a sus espaldas en cuanto a optimizaciones, ...

Hay casos en los que uno es más rápido que el otro y viceversa

No se puede decir cuál es mejor

- **Depende del patrón de accesos que vaya a tener nuestra aplicación**
- (p. ej. contado en MongoDB mucho más rápido que en MySQL; actualización algo más rápida)

2.6) Tipos de sistemas NoSQL

NoSQL incluye un conjunto de tecnologías relativamente dispares

Aún así, la mayoría comparten una serie de características:

- No se basan en SQL
- Generalmente no obligan a definir un esquema
- Surgen de la comunidad para solucionar problemas, y muchas son libres/open source
- Diseño basado en procesamiento distribuido, y aplican tecnologías funcionales como MapReduce

Divididas en subcategorías:

- Bases de datos Key-Value y Documentales
- Bases de datos columnares
- Bases de datos de grafos

- Bases de datos de arrays

2.6.1) Key-Value Stores y Documentales

Cada pieza de datos tiene asignado un identificador

La diferencia entre ambas es que:

- En Key-Value, el valor es opaco, no se conoce nada de su interior (a todos los efectos es un **blob** de datos)
- En las basadas en documentos, la base de datos puede ver el contenido del agregado, y utilizar su información como parte de las búsquedas y actualizaciones

Documentos ⇒ formatos jerárquicos tipo JSON o XML

La diferencia entre ambas queda un poco difusa

- Por ejemplo, Riak es Key-Value pero permite realizar búsquedas indexadas parecidas a las de Solr/Lucene
- Redis permite que los valores de datos sean estructurados en arrays, estructuras complejas, mapas

Key-Value: Riak, Redis, Memcache, LevelDB

Documentos: Couchbase, MongoDB, OrientDB

2.6.2) Bases de Datos Columnares

Influenciadas por el Paper de Google de 2004 sobre BigTable

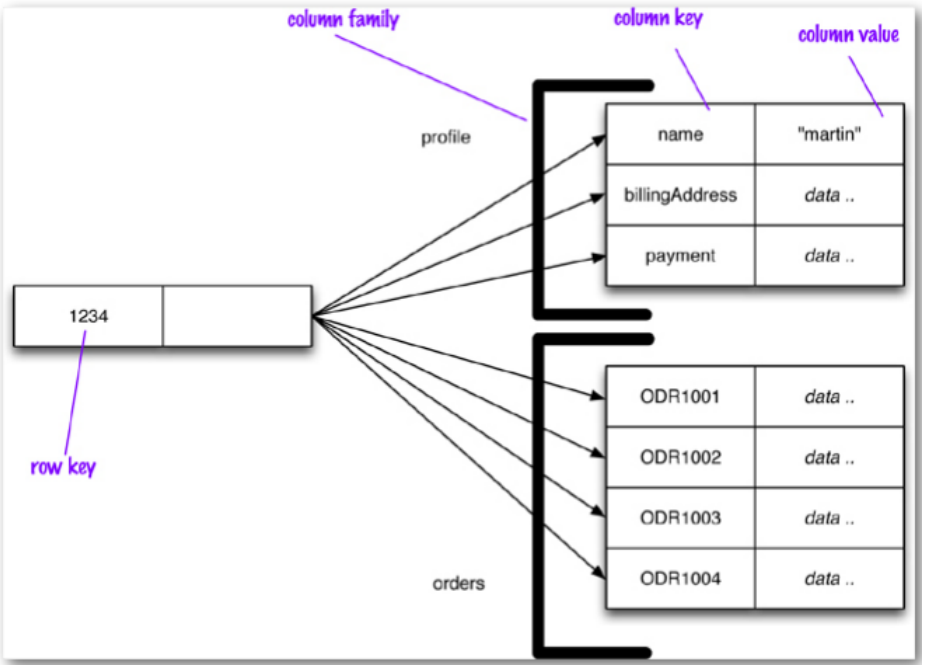
En general, parecidos a las tablas SQL, salvo que cada fila puede:

- Tener un conjunto de columnas diferente
- Almacenar *series temporales* dentro de una misma fila (varias *versiones* de un mismo conjunto de columnas)

Cada fila tiene un identificador y es un agregado de familias de columnas (*column family*)

Cambian el modo de almacenamiento para favorecer ciertas aplicaciones (almacenamiento por columnas en vez de por filas)

Bases de datos: HBase, Cassandra, Vertica, H-Store



2.6.3) Bases de Datos de Grafos

Las bases de datos de grafos llevan el mecanismo **muchos a muchos** al extremo

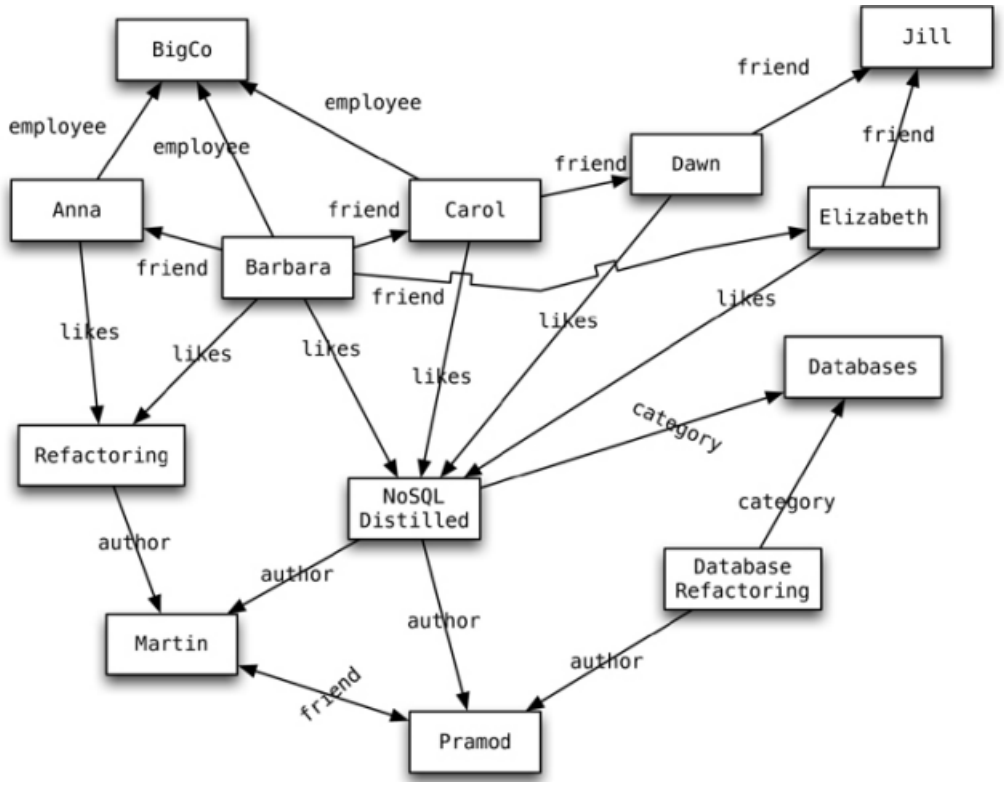
Datos en los que existen muchas relaciones entre sí y tienen un significado primordial

Las bases de datos de grafos se basan en la construcción y consulta de grafos que constan de **Vértices**, también llamados *nodos* o *entidades*, y **Aristas (*Edges*)**, también llamados *relaciones*.

Los grafos pueden capturar relaciones complejas entre entidades y ofrecen lenguajes de búsqueda, actualización y creación que permiten trabajar con subconjuntos del grafo

Origen en las bases de datos de hechos (**Datalog**)

Ejemplos: FlockDB, Neo4J, OrientDB



(Nota: Usa la sintaxis PostgreSQL para `json`)

```
CREATE TABLE vertices (
    vertex_id integer PRIMARY KEY,
    properties json
);

CREATE TABLE edges (
    edge_id integer PRIMARY KEY,
    tail_vertex integer REFERENCES vertices (vertex_id),
    head_vertex integer REFERENCES vertices (vertex_id),
    label text,
    properties json
);

CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (tail_vertex);
```

2.6.3.1) Grafo y consulta en Neo4j

```
CREATE
    (NAmerica: Location {name: 'North America', type: 'continent'}),
    (USA: Location {name: 'United States', type: 'country' }),
    (Idaho:Location {name: 'Idaho', type: 'state' }),
    (Lucy:Person {name:'Lucy' }),
    (Idaho)-[:WITHIN]->(USA)-[:WITHIN]-> (NAmerica),
    (Lucy) -[:BORN_IN]-> (Idaho)
```

Consulta:

```
MATCH
(person) -[:BORN_IN] -> () -[:WITHIN*0..] -> (us.Location {name: 'United States'})
(person) -[:LIVES_IN] -> () -[:WITHIN*0..] -> (us.Location {name: 'Europe'})
RETURN person.name
```

2.6.4) Bases de Datos basadas en Arrays

Suelen presentarse como bases de datos que soportan SQL y añaden operaciones para trabajar con conjuntos de datos especiales (arrays)

Utilizadas para tratamiento de grandes cantidades de datos de forma estadística o de modelado y OLAP

Soportan también datos geográficos, ya que pueden definir rangos numéricos de una o varias dimensiones (2D para cálculos geográficos)

Ejemplos: MonetDB, SciDB, rasdaman

Tema 3: Bases de datos basadas en documentos

3.1) Introducción a las bases de datos de documentos

3.1.1) Bases datos Documentales

En general, organizadas como un conjunto de **Colecciones** que contienen **documentos**:

- Tablas relacionales ↔ Colecciones
- Filas (*rows*) ↔ Documentos
- No restringen la estructura de cada documento y no tienen esquema

Cada colección es un *array* donde cada documento tiene asociado un identificador

La base de datos puede ver el contenido del documento, y utilizar su información como parte de las búsquedas y actualizaciones

Documentos ⇒ formatos jerárquicos tipo JSON o XML

No utilizan SQL y tienen su lenguaje de consulta propio

Utilizan normalmente **Map-Reduce** para cálculos distribuidos

Algunas implementan otros lenguajes de consulta y procesado, como N1QL (Couchbase) y Aggregation Framework (MongoDB)

Bases de datos de Documentos: Couchbase, MongoDB, OrientDB

3.2) Modelado de bases de datos de documentos

3.2.1) Modelado de datos en NoSQL

El modelado de datos debe ser:

- Realizado al mayor nivel de abstracción posible
- Independiente de la tecnología subyacente

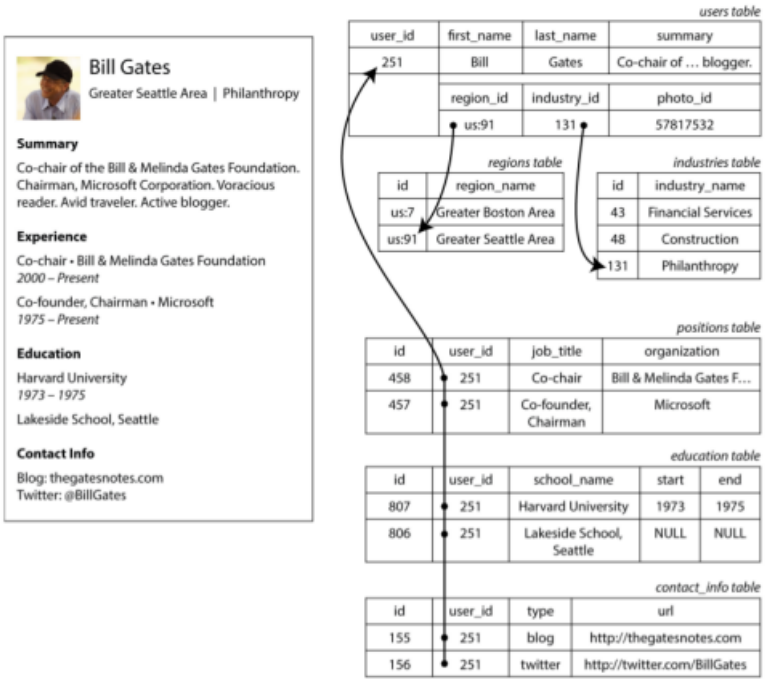
Sin embargo, en NoSQL:

- Se tiene que tener en cuenta el diseño **distribuido**
- **Optimización guiada por las consultas**

Con respecto al modelo de datos:

- Se mantienen los conceptos de entidad, relación, cardinalidades, etc.
- El modelado relacional se centra en especificar **qué datos tenemos y podemos ofrecer**
- El modelo NoSQL se centra en **optimizar qué consultas vamos a servir**
- Es "barato" **duplicar (desnormalizar)** los datos si con ello se consigue **mayor eficiencia de acceso**

3.2.2) Representación de CV como tablas



3.2.3) Representación de relaciones

3.2.3.1) Relaciones uno a muchos

Las relaciones uno a muchos (por ejemplo en el CV: `positions`) en el modelo relacional:

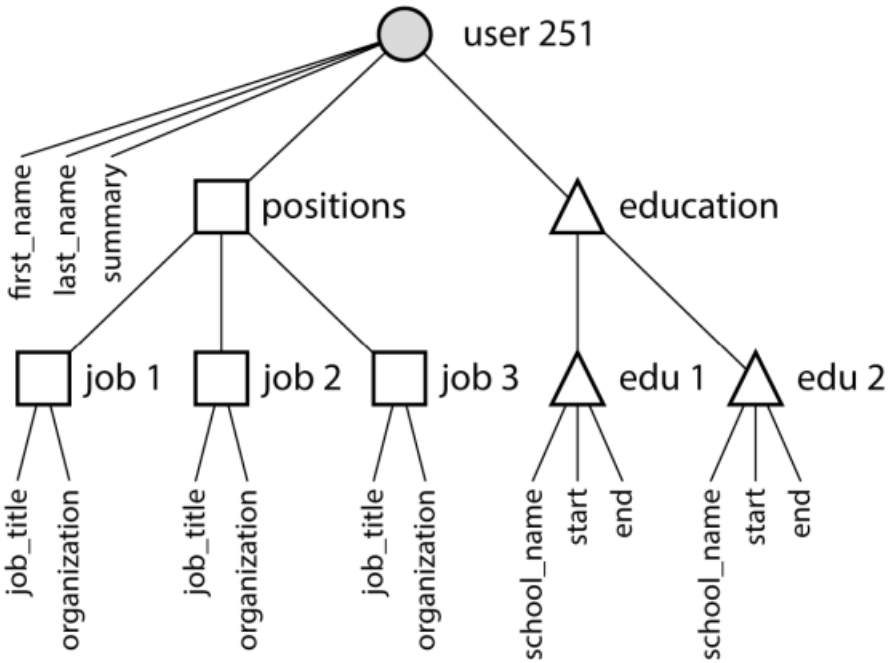
- Normalización usando varias tablas (`Positions` con `user_id`)
  - Necesidad de más de una tabla
  - Necesidad de uso de `JOIN` ⇒ ineficiencia
- Algunos SGBDR ofrecen la posibilidad de tener tipos de datos estructurados y campos XML o JSON. (P. ej. PostgreSQL)
  - Usualmente no se pueden usar para buscar dentro
  - No son estándar

3.2.4) CV como un documento

```
{
 "user_id": 251, "first_name": "Bill",
 "last_name": "Gates",
 "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
 "region_id": "us:91",
 "industry_id": 131,
 "photo_url": "p/7/000/253/05b/308dd6e.jpg",
 "positions": [
 {
  "job_title": "Co-chair",
  "organization": "Bill & Melinda Gates Foundation"
 },
 {
  "job_title": "Co-founder Chairman",
```

```
    "organization": "Microsoft"
  }],
  "education": [
  {
    "school_name": "Harvard University",
    "start": 1973,
    "end": 1975
  },
  {
    "school_name": "Lakeside School, Seattle",
    "start": null,
    "end": null
  }]
  "contact_info": {
    "blog": "http://theganotes.com"
    "twitter": "http://twitter.com/BillGates"
  }
}
```

3.2.5) CV como un árbol



3.2.6) Representación de Relaciones

3.2.6.1) Modelo de documentos

Modelo de documentos

- analogía del **array/mapa gigante**

Conjunto de documentos (objetos complejos)

- Un **identificador único**, campo *id*
- Búsqueda aleatoria eficiente por clave (**referencia**)
- Estructura jerárquica de sub-documentos contenidos ⟶ **agregación**

Más flexibilidad que el modelo relacional: (elección entre *referencia* Y *agregación*)

3.2.6.2) Uno a muchos - NoSQL

Relaciones **Uno a Muchos** (`postions`):

- **Opción 1:** Agregando la tabla `positions`
- **Opción 2:** Convertir las empresas en entidades, y utilizar una ***referencia***.

Modelado guiado por el acceso a datos:

- Si los elementos "muchos" tienen una estructura sencilla $\Longrightarrow$ **Opción 1**.
- Si los elementos "muchos" son usualmente **recuperados en una consulta** junto con el elemento "uno" $\Longrightarrow$ **Opción 1**.
- Si lo elementos "muchos" son relativamente grandes, o bien son recuperados siempre de forma separada $\Longrightarrow$ **Opción 2**.

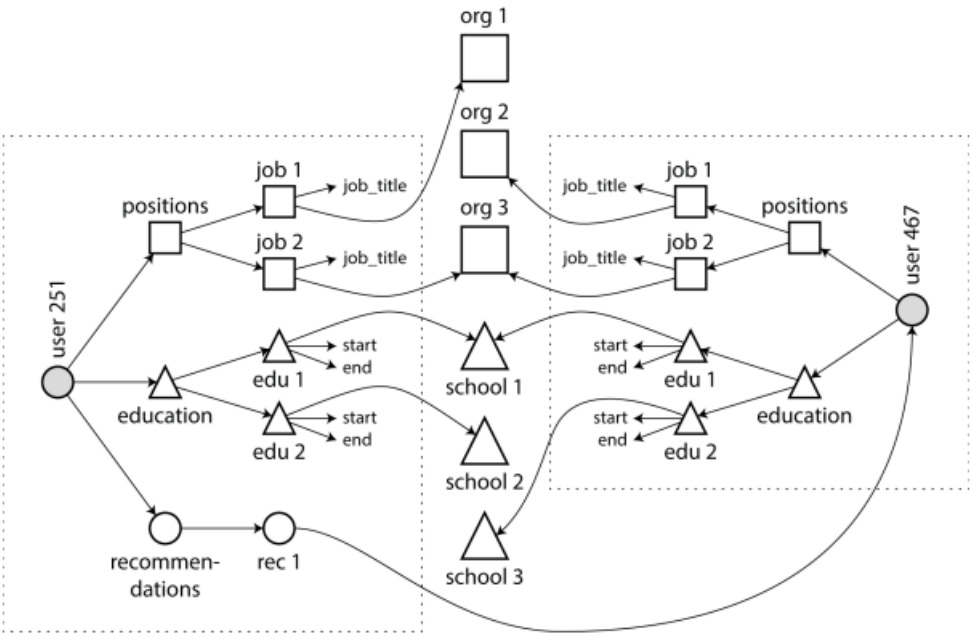
3.2.6.3) Muchos a uno y muchos a muchos

Las relaciones muchos a uno y muchos a muchos:

- Personas que viven en una región.
- Preguntas que refieren a Tags.

El modelo de documentos no aporta ventajas con respecto al modelo relacional.
Al haber muchas entidades que refieren a otra entidad, la *agregación* daría lugar a mucha **duplicación** (y a problemas de sincronización).
Referencias (sobre el ID), similar a una FK en el modelo relacional.

- **Sin embargo**, al no haber **JOINs** la aplicación tiene que hacer más de una petición a la BD.



3.3) Introducción a MongoDB

Base de datos documental

Basado en una representación binaria de JSON, llamada BSON

Cada base de datos se divide en un conjunto de *colecciones*

Una colección es un conjunto arbitrariamente grande de documentos

Cada documento contiene un campo especial `_id`, que guarda un objeto de tipo especial `ObjectID` de BSON

Permite la definición de bases de datos, colecciones (tablas *hash* de documentos) y subcolecciones

Permite realizar búsquedas secuenciales, programas MapReduce y un API propietario de consulta (llamado *aggregate*)

Ofrece un *shell* que acepta JavaScript y permite realizar operaciones directamente sobre la base de datos

También ofrece clientes en muchos lenguajes de programación

3.3.1) Uso básico de MongoDB

MongoDB ofrece un *shell*, aparte de las posibles conexiones remotas a través de *drivers* usando otros lenguajes de programación.
El shell acepta código JavaScript
Veremos inicialmente su uso
Existen una serie de objetos predefinidos

```
show databases;
use <database>;
db.<collection>;  // Permite acceder a la colección
```

Simplemente nombrando las colecciones se puede acceder a ellas para consulta y actualización.
Inserción de documentos:

```
db.colección.insert({< Objeto JSON >});
```

3.3.2) Actualización y consulta

Se puede obtener los documentos de la base de datos con:

```
var objeto = db.colección.findOne({atributos : valores})
// ó encontrar todos lo que cumplen la condición:
var objs = db.colección.find({ atributos : valores })
```

Y actualizarlos:

```
db.colección.update(
    { atributos: valores},  // búsqueda
    objeto);                                  // nuevo objeto
```

La selección del objeto se suele hacer por su `_id`, que es único.

También permite modificadores:

```
    db.analytics.update({"url" : "www.example.com"},
    {"$inc" : {"pageviews" : 1}})
```

También `$set`, `$unset`, `$push` (para *arrays*), `$pull` (para eliminar elementos de una *array* que cumplen un criterio).

3.3.3) Consultas MapReduce

Las consultas se hacen con `find`:

```
d.users.find({<criterio>},
                          {"username" : 1, "_id" : 0})
// =>
{
"username" : "joe",
}
```

Y se pueden utilizar condicionantes para la búsqueda:

```
db.user.find({"age": {"$gte": 18, "$lte": 30}});
db.user.find({"$or": [{"age": {"$gte": 18} },
                                          {"permission": true} ] });
```

Map-reduce es un paradigma de procesamiento de datos que permite condensar grandes volúmenes de datos en resultados agregados útiles.

Una operación map-reduce consta de tres fases:

1) Se aplica la función `Map (Map(k1, v1)` $\rightarrow$ `list(k2, v2)` a cada documento de entrada. La función Map se aplica en paralelo a cada documento (con clave `k1`) del conjunto de datos de entrada. Esto produce una lista de pares (codificados por `k2`) para cada llamada.

2) La función `Reduce (Reduce(k2, list (v2))` $\rightarrow$ `list((k3, v3)))` se aplica entonces en paralelo a cada grupo.

3.3.4) Validación del esquema

En sus últimas versiones MongoDB ha incorporado funcionalidades que permiten anotar y validar documentos JSON (*JSON Schema*).

A la hora de crear una colección, podemos especificar en *JSON schema* qué configuración deben seguir los documentos almacenados en ella.

Ejemplo

Vamos a crear una colección `estudiantes` y vamos a especificar su esquema asociado

```
    db.createColecction("estudiantes",{
        validator: {
            $jsonSchema: {
                bsonType: "object",
                title: "Validación del objeto Estudiante",
                required: [ "dirección", "major", "nombre", "año"],
                properties: {
                    nombre: {
                        bsonType: "string",
                        description: "'nombre' debe ser una cadena (string) y es
                           obligatorio"
                    },
                    año: {
                        bsonType: "int",
```

```
                        minimum: 2017,
                        maximum: 3017,
                        description: "'año' debe ser un entero entre [ 2017, 3017 ] y
                            es obligatorio"
                    },
                    gpa: {
                        bsonType: [ "double" ],
                        description: "'gpa' debe ser un double si el campo existe"
                    }
                }
            }
        }
    } )
```

Si ahora intentamos insertar un documento que no cumple dicho esquema

```
db.estudiantes.insertOne( {
    nombre: "Alice",
    year: Int32( 2019 ),
    major: "Historia",
    gpa: Int32(3),
    direccion: {
        city: "USA",
        street: "Fraggel Rock"
    }
} )
```

obtendremos un mensaje de error pues el campo `gpa` se ha insertado como `Int` cuando debía hacerse como `double`.

```
    MongoServerError: Document failed validation
    Additional information: {
        failingDocumentId: ObjectId("630d093a931191850b40d0a9"),
        details: {
            title: 'Validacion del objeto Estudiante',
            operatorName: '$jsonSchema',
            schemaRulesNotSatisfied: [
            {
                operatorName: 'properties',
                propertiesNotSatisfied: [
                {
                    propertyName: 'gpa',
                    description: "'gpa' debe de ser un doble si el campo existe",
                    details: [
                    {
                        operatorName: 'bsonType',
                        specifiedAs: { bsonType: [ 'double' ] },
                        reason: 'type did not match',
                        consideredValue: 3,
                        consideredType: 'int'
```

```
                    }
                    ] } ] } ]} }
```

Sin embargo, la siguiente inserción no provoca ningún fallo

```
db.students.insertOne( {
    name: "Alice",
    year: NumberInt(2019),
    major: "History",
    gpa: Double(3.0),
    address: {
        city: "NYC",
        street: "33rd Street"
    }
})
```

3.3.5) Índices y desnormalización

3.3.5.1) Índices

Sea con SQL o con datos *raw*, la búsqueda no se puede realizar secuencialmente

Los ***índices*** se utilizan para acelerar esta búsqueda

El concepto de índice está presente en casi todas las bases de datos (también es un eje principal en las NoSQL)

En el ámbito del SQL, los índices se pueden aplicar a los valores de una o más columnas

Se usan para acelerar la recuperación de información

El planificador de consultas usa los índices disponibles

Es **labor del usuario** definir los índices adecuados

Los índices aceleran las búsquedas, pero también ocupan espacio

En SQL se utiliza la construcción `CREATE INDEX, ALTER TABLE` o se incluyen en `CREATE TABLE`

`EXPLAIN` para ver si se usa:

```
mysql> explain select * from Posts where Id=5;
+----+-------------+-------+------+---------------+-----+...
| id | select_type | table | type | possible_keys | key |...
+----+-------------+-------+------+---------------+-----+...
|  1 | SIMPLE      | Posts | ref  | Id            | Id  |...
+----+-------------+-------+------+---------------+-----+...
```

Obviamente **también** se usan las base de datos NoSQL.

En general, hay tres tipos de índices:

1) Basados en árboles balanceados
2) Basado en tablas *hash*
3) Índices *full-text*

- Índices de Árboles Balanceados
  - Mantienen la clave ordenada dentro de un árbol balanceado (B-Tree, B+, ect)
  - Aceleran búsquedas con operadores =, ¡ y ¿ y rangos (`BETWEEN`)
  - También `LIKE` que no empiecen con comodín
  - Puede ayudar en `ORDER BY`
- Índices de tablas *hash*
  - Sólo para consultas de igualdad (=) y desigualdad (¡=¿)
  - Son muy rápidos
  - No se pueden usar para `ORDER BY`
  - Sólo se pueden usar con valores completos de clave (no permite prefijos, por ejemplo)
- Índices *full-text*
  - Usados en búsquedas inversas sobre campos textuales.
  - Dada una palabra o conjunto de palabras, encontrar la fila o filas en las que el campo buscado (el del índice) contiene esas palabras
  - Tiene limitaciones en `LIKE` con comodines (%).
  - Por ello los SGBD proveen operadores especiales (por ejemplo, en MySQL `"MATCH(...) AGAINST ..."`)

3.3.5.2) Desnormalización

El modelo relacional basa el modelado de datos en la **normalización**, en sus distintos niveles

Esto permite aplicar el principio de "sólo almacenar un dato en un sitio", lo que lleva a la eficiencia de datos.

Pero a veces necesitamos también **eficiencia de acceso**, y a veces está reñido con la normalización.

El principal escollo para la velocidad es la unión (`JOIN`) de datos de tablas.

¿Por qué desnormalizar?

- **Matener la historia:** Los datos de los clientes, por ejemplo, pueden cambiar. Necesitamos almacenar los valores de los mismos, por ejemplo, cuando emitimos una factura en el pasado. Para eso hay que copiar los datos de un cliente en ese momento.
- **Mejorar la velocidad de consulta:** A veces, para hacer consultas recurrentes, tenemos que usar uniones de varias tablas. Evitar la unión, por ejemplo, añadiendo claves ajenas a las tablas finales. O bien, como en el caso de `Posts.Tags`, replicando la información.
- **Precalcular valores necesitados de antemano:** Si hay datos que se necesitarán, como medias o totales, podemos calcularlos y almacenarlos de antemano.

Desventajas de desnormalizar:

- **Espacio en disco:** Al igual que ocurría con los índices, la copia y duplicación de datos ocupa más espacio
- **Anomalías de datos:** Hay que ser conscientes de que ahora un cambio en un dato puede requerir un cambio en más de una parte de la base de datos para mantener la consistencia. Los SGBDR no están preparados para mantener este tipo de consistencia. Se tienen que utilizar mecanismos de *triggers*, transacciones, etc.

- **Ralentización de las operaciones de actualización:** Dependiendo del *ratio* entre actualizaciones y consultas, este puede ser un punto importante
- **Más código, más dependencias, más propenso a errores:** Al necesitar más coordinación, la base de datos requiere más código, hay más dependencias ocultas, lo que puede llevar a más errores y a un mayor coste de mantenimiento.

3.4) Uso de MongoDB desde `pymongo`

3.4.1) Métodos de búsqueda

El método de búsqueda principal es **`find()`**, que tiene muchas opciones

En general permite especificar:

- El filtro de búsqueda
- Ordenación de resultados por algún campo
- Proyección para no obtener todos los campos del documento
- Número de resultados máximo (*limit*)
- Número de elementos iniciales a ignorar (*skip*)
- El tamaño del *batch*

Como la variabilidad es muy grande, veremos ejemplos en la hoja Jupyter Notebook de prácticas y también en la documentación

La función **`find()`** tiene un gran número de posibilidades para especificar la búsqueda. Se pueden utilizar cualificadores complejos como:

- **`$and`**
- **`$or`**
- **`$not`**

Estos calificadores unen "objetos", no valores.

Por otro lado, hay otros calificadores que se refieren a valores:

- **`$lt`** (menor)
- **`$lte`** (menor o igual)
- **`$gt`** (mayor)
- **`$gte`** (mayor o igual)
- **`$regex`** (expresión regular)

```
posts.find_one({"body": {"$regrex": "[Mm]ongo"}})
```

Condición compuesta (limitando la salida a 10 documentos):

```
posts.find({"$and": [{"PostTypeId": 2},
    {"Id" : {"$gte": 100}} ]})
        .limit(10)
```

3.4.1.1) Ejemplos introductorios

- Encontrar el primer documento en la colección `customers`

```
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = myclient["customers"]
x = mycol.find_one()
```

- Encontrar el campo de todos los documentos en la colección `customers`

```
for x in mycol.find():
    print(x)
```

También podemos indicar qué campos queremos obtener determinada búsqueda

```
for x in mycol.find({}, {"_id": 0, "name": 1, "address": 1}):
    print(x)
```

3.4.1.2) Ejemplos básicos

- Dada la siguiente colección

```
{ "_id": "manzanas", "cantidad": 5 }
{ "_id": "plátanos", "cantidad": 7 }
{ "_id": "naranjas", "cantidad": 8 }
{ "_id": "aguacate", "cantidad": 14 }
```

- Si quisieramos obtener aquellos documentos con un valor en su atributo `cantidad` superior a 4 deberíamos ejecutar el siguiente comando

```
db.collection.find( { "cantidad": { "$gt": 4} } )
```

¿Qué comando deberíamos lanzar para obtener aquellos que tengan un valor inferior o igual a 5?

```
db.collection.find( { "cantidad": { "$lte": 5} } )
```

Suponiendo una colección `bios` cuyos documentos tienen esta forma:

```
{
    "_id": <valor>,
    "nombre" : { "nombre_propio" : <string>, "primer_apellido" : <string>},
    "nacimiento" : <ISODate>,
    "muerte" : <ISODate>,
    "profesión" : <valor>,
    "trabajos" : [<string>, ...]
}
```

- Una opción interesante es el filtrado por rango de fechas en caso de que tengamos atributos de ese tipo

```
db.bios.find( { "nacimiento": { "$gt": new Date('1940-01-01'),
    "$lt": new Date('1960-01-01') } } )
```

- El operador **$in** nos permite buscar documentos con atributos dentro de una lista de valores.

```
db.bios.find(
    { "profesion": { "$in": [ 'jedi', 'samurai' ] } }
)
```

- También podemos realizar búsquedas en campos anidados

```
db.bios.find( { "nombre.primer_apellido": "Skywalker" } )
db.bios.find(
    { "nombre": { "nombre_propio": "Akira", "primer_apellido": "Kurosawa" } }
)
```

- Las búsquedas pueden realizarse también sobre campos cuyos valores son arrays

```
db.bios.find({"trabajos": "star wars"})
db.bios.find({"trabajos": { "$in": ["star wars", "alien"]} })
db.bios.find({"trabajos": { "$all": [ "star wars", "the clone wars" ]}})
db.bios.find({"trabajos": { "$size": 4 }}) #El array 'trabajos' debe tener 4
    elementos
```

- El operador **$not** es útl cuando queremos buscar elementos que **no** cumplan una condición

```
# Buscar documentos cuyo precio no sea superior a 1.99€ o que no tengan el campo
    "precio"
db.inventory.find( { "precio": { "$not": { "$gt": 1.99 } } } )
```

- Por último, los operadores **$and**, **$or** y **$not** pueden combinarse para definir condiciones complejas:

```
db.inventory.find( {
    "$and": [
        {"$or": [{"cantidad"}:{"$lt":10}, {"cantidad":{"$gt": 50} }]},
        {"$or": [{"vendido": true}, {"precio": {"$lt": 5}}]}
    ]
} )
```

3.4.2) Métodos de inserción y actualización

pymongo ofrece métodos para inserción y actualización:

- **insert_one(), insert_many()** (batch)
- **update_one()**: Permite actualizar un objeto con nuevos campos. El objeto se crea si se pone el parámetro **upsert** a **True**
- **update_many()**: Permite obtener nuevos valores calculado a un conjunto de objetos.

A la hora de insertar un elemento poder obtener su **_id** asignado

```
micoleccion = db["clientes"]
midict = { "nombre": "Luke", "direccion": "Tatooine, 83" }
x = mycol.insert_one(midict)
print(x.inserted_id)
```

Para insertar un conjunto de documentos a la vez deberemos incluirlos en un array JSON

```
miLista = [
    { "nombre": "Amy", "direccion": "Apple st 652"},
    { "nombre": "Hannah", "direccion": "Mountain 21"},
    { "nombre": "Michael", "direccion": "Valley 345"}
]
x = micoleccion.insert_many(miLista)
# Imprimimos sus _id asignados
print(x.inserted_ids)
```

A la hora de actualizar un documento debemos especificar un criterio de búsqueda y los campos y valores a actualizar.

```
micoleccion = db["clientes"]
mifiltro = { "direccion": "Valley " }
nuevosvalores = { "$set": { "direccion": "Rue del percebe, 13" } }
micoleccion.update_one(mifiltro, nuevosvalores)
```

Para actualizar múltiples documentos la estructura es la misma

```
micoleccion = db["clientes"]
mifiltro = { "direccion": { "$regrex": "^S" } }
nuevosvalores = { "$set": { "nombre": "Rey" } }
x = micoleccion.update_many(mifiltro, nuevosvalores)
print(x.modified_count, "documentos actualizados.")
```

3.4.2.1) Índices, `explain()`

La llamada **`.explain()`** a cualquier búsqueda muestra el plan de ejecución.

Se puede crear un índice si la búsqueda por ese campo va a ser crítica.

Se pueden crear más índices:

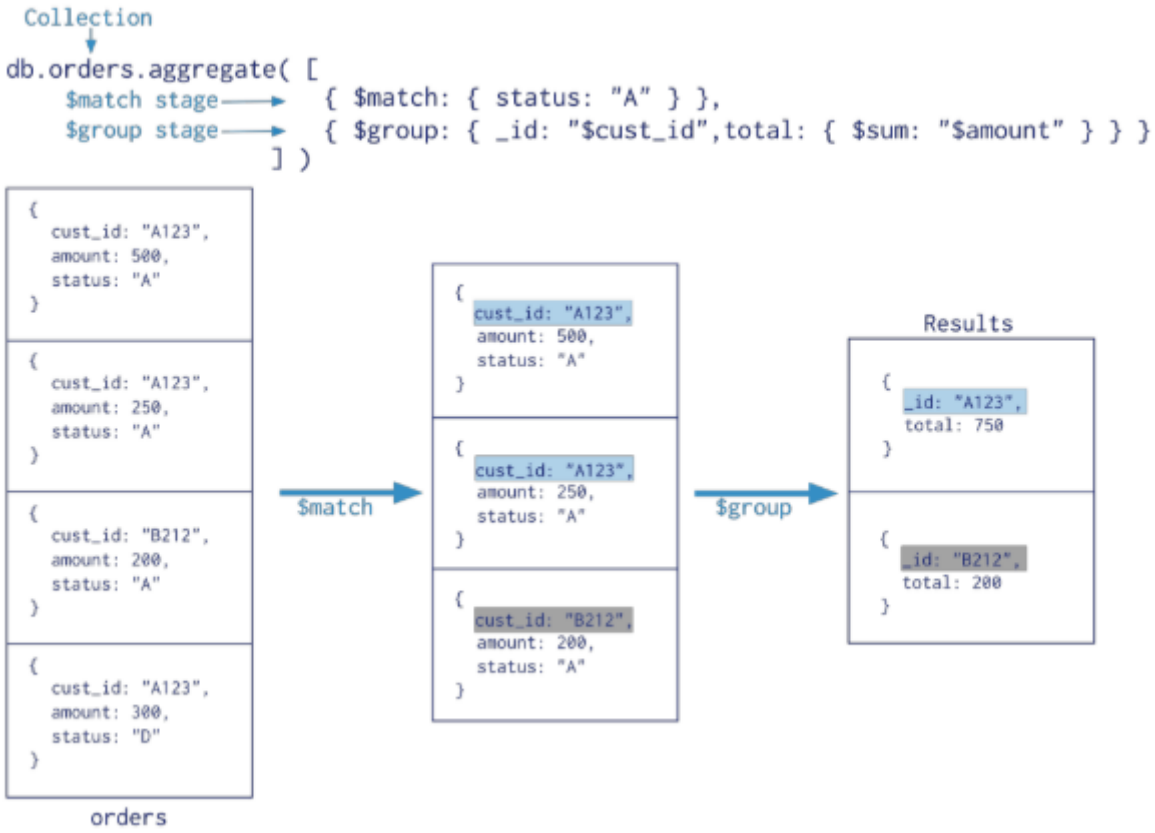
- **`ASCENDING`**
- **`DESCENDING`**
- **`HASHED`**
- Geoespaciales

3.4.3) Framework de agregación

El framework de agregación de MongoDB es una de sus características más potentes

Permite escribir expresiones, divididas en una serie de etapas, que realizan operaciones como agregaciones, transformaciones y uniones en los datos de las bases de datos MongoDB

Esto le permite realizar cálculos y análisis a través de documentos y colecciones dentro de su base de datos MongoDB



Framework de agregación: https://docs.mongodb.com/manual/reference/operator/aggregation/

```
posts.aggregate ( [
    {"$project" : { "Id" : 1 }},
    {"$limit": 20}
])
```

```
by_length = posts.aggregate( [
    {'$project': {
        'Body': {'$ifNull' : ['$Body', '']}
    }},
    {'$project': {
        'id' : {'$strLenBytes': '$Body'},
        'value' : {'$literal' : 1}
    }},
    {'$group': {
        '_id' : '$id',
        'count' : {'$sum' : '$value'}
    }},
    {'$sort' : { '_id' : 1 }}
])
```

Se pueden añadir etapas. En particular, por ejemplo, se puede **filtrar** inicialmente añadiendo al principio:

```
{'$match':  { 'Body' : {'$regex': 'HBase'}}}
```

La construcción `$lookup` permite el acceso a otra (o a la misma) colección. Es equivalente a un **JOIN**.

Por ejemplo, listar los posts con Score mayor o igual a 40 junto con el usuario que lo ha hecho:

```
posts.aggregate( [
    {'$match': { 'Score' : {'$gte': 40}}},
    {'$lookup': {
        'from': "users",
        'localField': "OwnerUserId",
        'foreingField': "Id",
        'as': "owner"}
    }
])
```

Vamos a suponer que contamos con una colección de películas (`movies`) que contienen documentos que tienen esta estructura:

```
{
    '_id': ObjectId('573a1392f29313caabcdb497'),
    'awards': {'nominations': 7,
               'text': 'Won 1 Oscar. Another 2 wins & 7 nominations.',
               'wins': 3},
    'cast': ['Janet Gaynor', 'Fredric March', 'Adolphe Menjou', 'May Robson'],
    'countries': ['USA'],
    'directors': ['Willianm A. Wellman', 'Jack Conway'],
    'fullplot': 'Esther Blodgett is just...',
    'genres': ['Drama'],
    'imdb': {'id': 29606, 'rating': 7.7, 'votes': 5005},
    'languages': ['English'],
    'lastupdated': '2015-09-01 00:55:54.333000000',
    'plot': 'A young woman ...',
    'poster': 'https://m.media-amazon.com/images/M/MV5....jpg',
    'rated': 'NOT RATED',
    'released': datetime.datetime(1937, 4, 27, 0, 0),
    'runtime': 111,
    'title': 'A Star Is Born',
    'tomatoes': {'critic': {'meter': 100, 'numReviews': 11, 'rating': 7.4},
                 'dvd': datetime.datetime(2004, 11, 16, 0, 0),
                 'fresh': 11,
                 'lastUpdated': datetime.datetime(2015, 8, 26, 18, 58, 34),
                 'production': 'Image Entertainment Inc. ',
                 'rotten': 0,
                 'viewer': {'meter': 79, 'numReviews': 2526, 'rating': 3.6},
                 'website': 'http://www.vcientertainment.com/Film-Categories?
                    product_id=73'},
    'type': 'movie',
    'writers': ['Dorothy Parker (screen play)',
                'Alan Campbell (screen play)',
                'Robert Carson (screen play)',
                'William A. Wellman (from a story by)',
                'Robert Carson (from a story by)'],
    'year': 1937}
```

Ejemplo

En primer lugar vamos a realizar una primera búsqueda sobre dicha colección filtrando por el nombre de la película (`title`) y ordenando por su año (`year`):

```
movie_colection = db["movies"]
pipeline = [
    {
        "$match": {
            "title": "A Star Is Born"
        }
    },
    {
        "$sort": {
            "year": pymongo.ASCENDING
        }
    },
]
results = movie_colecction.aggregate(pipeline)
```

Si ahora quisieramos limitar la salida a un único documento...

```
movie_colection = db["movies"]
pipeline = [
    {
        "$match": {
            "title": "A Star Is Born"
        }
    },
    {
        "$sort": {
            "year": pymongo.ASCENDING
        }
    },
    {"$limit": 1}
]
results = movie_colecction.aggregate(pipeline)
```

Ejemplo

Vamos a suponer ahora que existe otra colección llamada `comments` cuyos documentos tienen este formato

```
{
    '_id': ObjectId('5a9427648b0beebeb69579d3'),
    'movie_id': ObjectId('573a1390f2931caabcd4217'),
    'date': datetime.datetime(1983, 4, 27, 20, 39, 15),
    'email': 'cameron_duran@fakegmail.com',
    'name': 'Cameron Duran',
    'text': '...'}
```

Esto permitiría añadir un campo **`related_comments`** a cada película con sus críticas asociadas

```
stage_lookup_comments = {
    "$lookup": {
        "from": "comments",
        "localField": "_id",
        "foreignField": "movie_id",
        "as": "related_comments",
    }
}
# Limitamos a los primeros 5 documentos
stage_limit_5 = {"$limit": 5}
pipeline = [stage_lookup_comments, stage_limit_5]
results = movie_collection.aggregate(pipeline)
```

Ejemplo

Una funcionalidad muy interesante es la agrupación de documentos con el comando **$group**

Vamos a usar dicho comando para contar el número de películas por año, ordenadas de forma ascendente

```
stage_group_year = {
    "$group": {
        "_id": "$year",
        # contamos el número de películas en el grupo
        'movie_count': { "$sum": 1 },
    }
}
stage_match_years = {
    "$match": {
        "year": {
            "$type": "number",
        }
    }
}
stage_sort_year_ascending = {
    "$sort": {"_id": pymongo.ASCENDING}
}
pipeline = [stage_match_years, stage_group_year, stage_sort_year_ascending]
results = movie_colecction.aggregate(pipeline)
```

3.4.4) Índices

3.4.4.1) Creación de índices

En MongoDB, la creación de índices se realiza mediante el método **create_index**

```
db.collection.create_index([(<key and index type specification>)], <options> )
```

En el siguiente ejemplo se crea un índice con una sola clave (**nombre**) en orden descendente

```
collection.create_index([("nombre", pymongo.DESCENDING)])
```

Podemos listar los índices que tenemos creados en una colección con

```
collection.index_information()
```

En el caso de las colecciones con alta proporción de escritura por lectura, los índices son caros porque cada inserción y actualización también debe actualizar cualquier índices.

3.4.4.2) Nombrado y borrado de índices

Al crear un índice, puede darle un nombre personalizado

Ayuda a distinguir los distintos índices de la colección.

Para especificar el nombre del índice, incluya la opción de nombre al crear el índice:

```
db.collection.create_index(
    { "<campo>": "<valor>" },
    { "name": "<NombreDelIndice>" }
)
```

Para borrar un índice podemos hacerlo con el método **.drop_index**

```
colecction.drop_index("<NombreDelIndice>")
```

3.4.4.3) Tipos de índices en MongoDB

- Índices compuestos (*compound indexes*)

Los índices compuestos recopilan y ordenan datos de dos o más campos de cada documento de una colección.

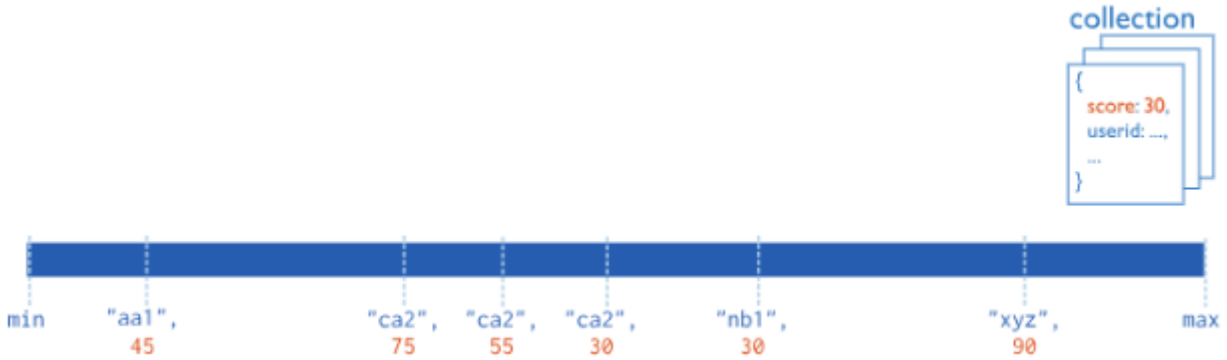
Los datos se agrupan por el primer campo del índice y, a continuación, por cada campo subsiguiente.

Para su creación se sigue un formato parecido al de los **single field index**

```
db.<collection>.create_index( {
    <campo1>: <tipoDeOrdenacion>,
    <campo2>: <tipoDeOrdenacion>,
    ...
    <campoN>: <tipoDeOrdenacion>
} )
```

Ejemplo: si quisieramos crear un índice sobre los campos `userid` (de forma ascendente) y `score` (de forma descendente) de los documentos de una colección `usuarios` deberíamos ejecutar

```
    db.usuarios.create_index({"userid": 1, "score": -1})
```

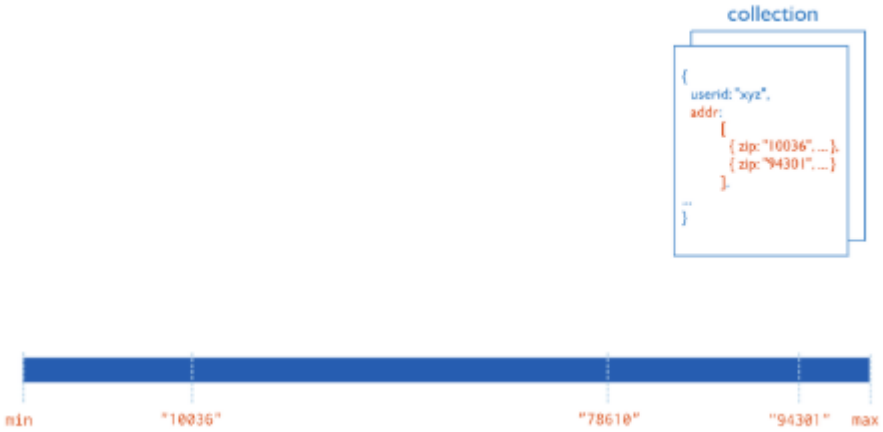


- Índices multiclave (*multikey indexes*)

Los índices multiclave recopilan y ordenan datos de campos que contienen un array como valor asociado.

Su creación es igual a los índices anteriores pero MongoDB establece automáticamnete que ese índice sea multivariante.

```
db.empleados.create_index( { "addr.zip": 1 } )
```



- Índices de texto (*text indexes*)

Los índices de texto permiten realizar búsquedas sobre campos de tipo `string`.

Los índices de texto mejoran el rendimiento cuando se buscan palabras o frases específicas dentro del contenido de una cadena.

Una colección sólo puede tener un índice de texto, pero ese índice puede abarcar varios campos.

```
db.<coleccion>.create_index( {
    <campo1>: "text",
    <campo2>: "text",
    ...
    <campoN>: "text"
} )
```

Ejemplo

Vamos a suponer que creamos una colección `blog` con los siguientes documentos

```
db.blog.insert_many( [
    {
        _id: 1,
        "contenido": "Esta mañana me he tomando una taza de café",
        "sobre": "bebida",
        "palabras_clave": ["café"]
    },
    {
        _id: 2,
        "contenido": "A quién le apetece un helado de chocolate de postre?",
        about: "comida",
        "palabras_clave": ["encuesta"]
    },
```

```
    {
        _id: 3,
        "contenido": "Mis sabores favoritos son la fresa y el chocolate",
        "sobre": "helado",
        "palabras_clave": [ "comida", "postre" ]
    }
] )
```

Puesto que vamos a querer realizar muchas búsquedas sobre el campo `contenido`, vamos a definir un índice sobre dicho campo.

```
db.blog.create_index( { "content": "text" } )
```

Esto permite realizar, por ejemplo, una búsqueda para recuperar los documentos que contengan la palabra *café* en el campo `contenido`.

```
db.blog.find(
    {
        "$text": { "$search": "café" }
    }
)
```

- Índices hash (*hash indexes*)

Recogen y almacenan los *hash* de los valores del campo indexado.

El campo que usemos para crear una clave hash de fragmento debe tener una cardinalidad alta, es decir, un grán número de valores diferentes.

Permiten la fragmentación en diferentes clusters (*sharding*) mediante claves hash

```
db.<coleccion>.create_index(
    {
        "<campo>": "hashed"
    }
)
```

3.4.4.4) La importancia de los índices en la eficiencia

Los índices son cruciales para la eficiencia

Y hay que conocer cúales usar y si realmente una búsqueda los usará

Recordemos la agregación sencilla de antes:

```
posts.aggregate( [
    {'$match': { 'Score': {'$gte': 40}}},
    {'$lookup': {
        'from': "users",
        'localField': "OwnerUserId",
        'foreignField': "Id",
        'as': "owner"}
    }
```

```
])
```

Nótese cómo se busca al usuario por el campo "Id" (ojo, no "_id")

Tiempo de ejecución sin índice: **24 segundos**

Defino un índice para el campo "Id" (valdría también *hash* porque se hace una búsqueda exacta):

```
db.users.create_index({"Id": 1})
> 'Id_1'
```

Ejecuto de nuevo: **67.2 milisegundos**

En `pymongo` hay que hacerlo de forma especial para saber que se ha usado el índice:

```
db.command('aggregate', 'posts', pipeline=pipeline, explain=True)
...
'winningPlan': {'queryPlan' : {'stage': 'EQ_LOOKUP',
    'foreignCollection' : 'stackoverflow.users',
    'localField': 'OwnerUserId',
    'foreignField': 'owner',
    'strategy': 'IndexedLoopJoin',
    'indexName': 'Id_1',
    'indexKeyPattern': {'Id': 1},
    'inputStage': {'stage': 'COLLSCAN',
    'planNodeId': 1,
    'filter': {'Score' : {'$gte': 40}},
    'direction': 'forward'}}},
...
```

3.4.5) Transacciones

En MongoDB, una operación en un solo documento es atómica.

El uso de documentos incrustados permite capturar relaciones entre datos en una sola estructura de documento.

- Elimina la necesidad de transacciones distribuidas para muchos casos

Para situaciones que requieren atomicidad de lecturas y escrituras en varios documentos, MongoDB soporta transacciones distribuidas.

Pueden ser utilizadas en múltiples operaciones, colecciones, bases de datos y documentos.

En MongoDB ofrece una `TransactionAPI` que a su vez hace uso de la `callbackAPI`

Esta API hace la siguiente secuencia de operaciones:

1) Comienza una transacción
2) Ejecuta las operaciones indicadas
3) Hace *commit* del resultado (o aborta en caso de error).

Ejemplo

1) Nos conectamos a la BD

```
client = MongoClient(MONGODB_URI)
```

2) Definimos el *callback* que especifica la secuencia de operaciones a realizar

```
def callback(sesion):
    coleccion_1 = session.client.db.foo
    coleccion_2 = session.client.db.bar
    # Importante: Se debe de pasar el parámetro de sesión a los operadores.
    coleccion_1.insert_one({"abc": 1}, session=sesion)
    coleccion_2.insert_one({"xyz": 999}, session=sesion)
```

3) Iniciamos la sesión de cliente y ejecutamos la transacción (el bloque `with` ejecuta el *commit*)

```
with client.start_session() as session:
    session.with_transaction(callback)
client.close()
```

Si algún paso falla, se puede llamar siempre a **`sessiom.abort_transaction()`**, por ejemplo en un `catch` de una excepción.