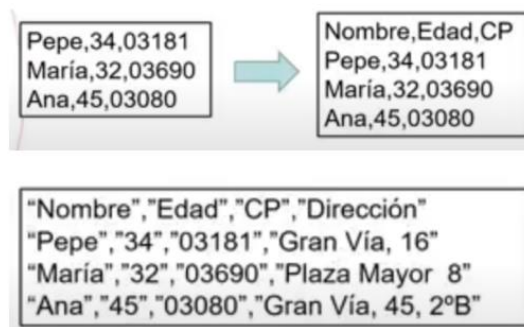


# Formatos de datos

## CSV

Comma Separated Values (valores separados por comas). Requiere que cada elemento de nuestro conjunto se presente en una línea. Dentro de esa línea, cada uno de los atributos del elemento debe estar separado por un único separador, que habitualmente es una coma, y seguir siempre el mismo orden. Además, la primera línea del fichero, a la que llamaremos *cabecera*, no contiene datos de ningún elemento, sino información de los atributos. Si el campo contiene alguna coma, utilizaremos un delimitador como por ejemplo " ".



## XML

Extensive Markup Language (lenguaje de marcas extensible). Es un lenguaje de etiquetas utilizado para almacenar datos de forma estructurada.

### Estructura en árbol

```
<libro>
  <titulo> El Quijote</titulo>
  <autor>Cervantes</autor>
</libro>
```

## JSON

JavaScript Object Notation, es un formato muy utilizado hoy en día, tiene el mismo propósito que el XML que es el intercambio de datos pero no utiliza las etiquetas abiertas y cerradas, sino que pretende que pese menos, es decir que ocupe menos espacio.

```
{
  libro:
  {
    titulo: "El Quijote",
    autor: "Cervantes"
  }
}
```

## Avro

Es un formato de almacenamiento basado en filas para Hadoop. Avro se basa en **esquemas**. Cuando los datos .avro son leídos siempre está presente el esquema con el que han sido escritos. Avro utiliza JSON para definir tipos de datos y protocolos. Es el formato utilizado para la serialización de datos ya que es más rápido y ocupa menos espacio que los JSON, la serialización de los datos la hace en un formato binario compacto.

## Parquet

Es un formato de almacenamiento basado en columnas para Hadoop. Fue creado para poder disponer de un formato de compresión y codificación eficiente. El formato Parquet está compuesto por tres piezas:

- Row group: es un conjunto de filas en formato columnar.
- Column chunk: son los datos de una columna en grupo. Se puede leer de manera independiente para mejorar las lecturas.
- Page: es donde finalmente se almacenan los datos, debe ser lo suficientemente grande para que la compresión sea eficiente.

# Formatos de datos a detalle

Conforme los datos viajan a través de los diferentes pipelines, los ingenieros de datos deben gestionar la serialización en diferentes formatos y ser capaces de convertir unos en otros.

Las propiedades que ha de tener un formato de datos son:

- independiente del lenguaje
- expresivo, con soporte para estructuras complejas y anidadas

- eficiente, rápido y reducido
- dinámico, de manera que los programas puedan procesar y definir nuevos tipos de datos.
- formato de fichero standalone y que permita dividirlo y comprimirlo.

Para que Hadoop/Spark o cualquier herramienta de analítica de datos pueda procesar documentos, es imprescindible que el formato del fichero permita su división en fragmentos (splittable in chunks).

Si los clasificamos respecto al formato de almacenamiento tenemos formatos de tipo:

- texto (más lentos, ocupan más pero son más expresivos y permiten su interoperabilidad): CSV, XML, JSON, etc...
- binario (mejor rendimiento, ocupan menos, menos expresivos): Avro, Parquet, ORC, etc...

Si comparamos los formatos más empleados a partir de las propiedades descritas tenemos:

Las ventajas de elegir el formato correcto son:

- Mayor rendimiento en la lectura y/o escritura
- Ficheros troceables (splittables)
- Soporte para esquemas que evolucionan
- Soporte para compresión de los datos (por ejemplo, mediante Snappy).

## Filas vs Columnas

Los formatos con los que estamos más familiarizados, como son CSV, XML o JSON, son formatos basados en filas, donde cada registro se almacena en una fila o documento. Estos formatos son más lentos en ciertas consultas y su almacenamiento no es óptimo.

## JSON vs JSONL

Un documento JSON contiene diferentes pares de clave-valor con los elementos que lo componen ocupando varias líneas.

Un tipo específico de JSON es JSON (JSON Lines), el cual almacena una secuencia de objetos JSON delimitando cada objeto por un saldo de línea.

### JSON

```
{“empleados”: [  
  {“nombre”: “Carlos”, “altura”: 180, “edad”: 44},  
  {“nombre”: “Juan”, “altura”: 175, “edad”: None}  
]}
```

### JSONL

```
{"nombre": "Carlos", "altura": 180, "edad": 44}  
{"nombre": "Juan", "altura": 175, "edad": None}
```

En un **formato basado en columnas**, cada columna se almacena en su conjunto de ficheros, es decir, cada registro almacena toda la información de una columna. Al basarse en columnas, ofrece mejor rendimiento para consultas de determinadas columnas y/o agregaciones, y el almacenamiento es más óptimo (como todos los datos de una columna son del mismo tipo, la compresión es mayor).

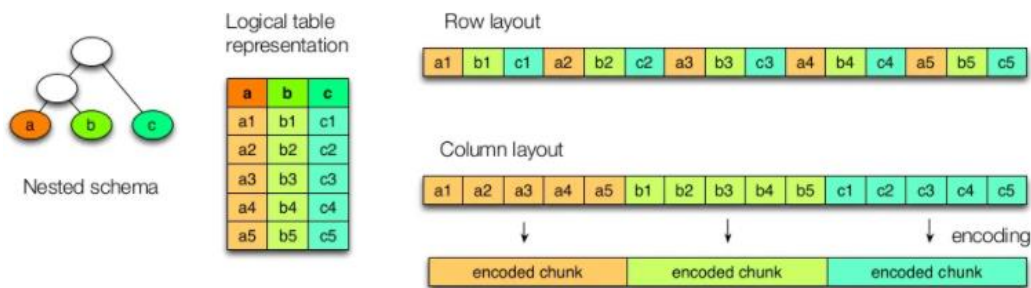
Supongamos que tenemos los siguientes datos:

user_id	country	subscription_type
1	US	Free
2	UK	Paid
3	ES	Paid

Dependiendo del almacenamiento en filas o columnas tendríamos la siguiente representación:

Row storage		Column storage	
Row 1	1	user_id	1
	US		2
	Free		3
Row 2	2	country	US
	UK		UK
	Paid		ES
Row 3	3	subscription_type	Free
	ES		Paid
	Paid		Paid

Como hemos comentado previamente, en un formato columnar los datos del mismo tipo se agrupan, lo que mejora el rendimiento de acceso y reduce el tamaño:



El almacenamiento en columnas y la compresión también presentan algunas desventajas, ya que no podemos acceder fácilmente a registros de datos individuales. Para ello, debemos reconstruir registros leyendo datos de varios archivos de columnas.

Del mismo modo, la actualización de registros supone descomprimir el archivo de columnas, modificarlo, volver a comprimirlo y escribirlo de nuevo en el almacenamiento.

Para evitar reescribir columnas completas en cada actualización, las columnas se dividen en varios archivos, normalmente utilizando estrategias de particionado y clustering que organizan los datos según los patrones de consulta y modificación de la tabla. Aun así, la sobrecarga computacional que implica actualizar una sola fila es muy alta.

Por esto, las bases de datos columnares no se adaptan bien a las cargas de trabajo transaccionales, por lo que las bases de datos transaccionales (OLTP) suelen utilizar algún tipo de almacenamiento orientado a filas o registros.

## Hablemos de tamaño

El artículo [Apache Parquet: How to be a hero with the open-source columnar data format](#) compara un formato basado en filas, como CSV, con uno basado en columnas como Parquet, en base al tiempo y el coste de su lectura en AWS (por ejemplo, AWS Athena cobra 5\$ por cada TB escaneado):

Dataset	Size on Amazon S3	Query Run time	Data Scanned	Cost
Data stored as CSV files	1 TB	236 seconds	1.15 TB	\$5.75
Data stored in Apache Parquet format*	130 GB	6.78 seconds	2.51 GB	\$0.01
Savings / Speedup	87% less with Parquet	34x faster	99% less data scanned	99.7% savings

En la tabla podemos observar como 1TB de un fichero CSV en texto plano pasa a ocupar sólo 130GB mediante *Parquet*, lo que provoca que las posteriores consultas tarden menos y, en consecuencia, cuesten menos.

En la siguiente tabla comparamos un fichero CSV compuesto de cuatro columnas almacenado en S3 mediante tres formatos cuando sólo nos interesa los datos de una única columna:

Dataset	Columns	Size on Amazon S3	Data Scanned	Cost
Data stored as CSV file	4	4TB	4TB	\$20 (4TB x \$5/TB)
Data stored as GZIP CSV file	4	1TB	1TB	\$5 (1TB x \$5/TB)
Data stored as Parquet file	4	1TB	.25TB	\$1.25 (.25TB x \$5/TB)

Comparación de costes por columnas entre CSV y Parquet

Queda claro que la elección del formato de los datos y la posibilidad de elegir el formato dependiendo de sus futuros casos de uso puede conllevar un importante ahorro en tiempo y costes.

## Avro

[Apache Avro](#) es un formato de almacenamiento basado en filas que es más rápido y ocupa menos espacio que JSON, debido a que la serialización de los datos se realiza en un formato binario compacto, con los metadatos del esquema especificados en JSON.

Tiene soporte para la compresión de bloques y es un formato que permite la división de los datos (*splittable*).

Avro es muy popular en el ecosistema de Hadoop y tiene soporte en la mayoría de herramientas cloud.

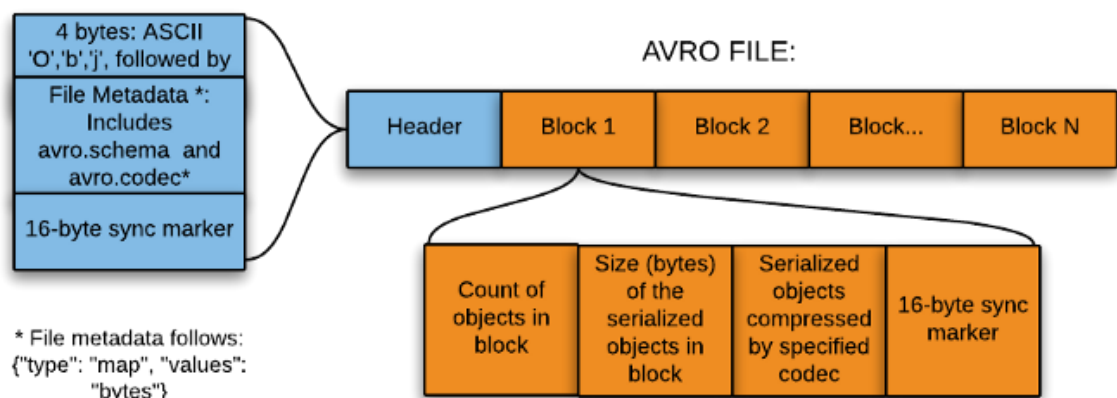
## Formato

El formato Avro se basa en el uso de esquemas, los cuales definen los tipos de datos y protocolos mediante JSON. Cuando los datos .avro son leídos siempre está presente el esquema con el que han sido escritos.

Cada fichero Avro almacena el esquema en la cabecera del fichero y luego están los datos en formato binario.

Los esquemas se componen de tipos primitivos:

(null, boolean, int, long, float, double, bytes, y string) y compuestos (record, enum, array, map, union, y fixed).



Formato de un archivo Avro

Un ejemplo de esquema podría ser:

```
empleado.avsc
1  {
2    "type" : "record",
3    "namespace" : "Severo0choa",
4    "name" : "Empleado",
5    "fields" : [
6      { "name" : "Nombre" , "type" : "string" },
7      { "name" : "Altura" , "type" : "float" },
8      { "name" : "Edad" , "type" : "int" }
9    ]
10 }
```

## Avro y Python

Para poder serializar y deserializar documentos Avro mediante *Python*, previamente debemos instalar la librería [avro](#):

```
1 pip install avro-python3
2 # o si utilizamos Anaconda
3 conda install -c conda-forge avro-python3
```

Vamos a realizar un ejemplo donde primero leemos un esquema de un archivo Avro, y con dicho esquema, escribiremos nuevos datos en un fichero. A continuación, abrimos el fichero escrito y leemos y mostramos los datos:

Código Python:

```
1 import avro
2 import copy
3 import json
4 from avro.datafile import DataFileReader, DataFileWriter
5 from avro.io import DatumReader, DatumWriter
6
7 # abrimos el fichero en modo binario y leemos el esquema
8 schema = avro.schema.parse(open("empleado.avsc", "rb").read())
9
10 # escribimos un fichero a partir del esquema leído
11 with open('empleados.avro', 'wb') as f:
12     writer = DataFileWriter(f, DatumWriter(), schema)
13     writer.append({"nombre": "Carlos", "altura": 180, "edad": 44})
14     writer.append({"nombre": "Juan", "altura": 175})
15     writer.close()
16
17 # abrimos el archivo creado, lo leemos y mostramos línea a línea
18 with open("empleados.avro", "rb") as f:
19     reader = DataFileReader(f, DatumReader())
20     # copiamos los metadatos del fichero leído
21     metadata = copy.deepcopy(reader.meta)
22     # obtenemos el schema del fichero leído
23     schemaFromFile = json.loads(metadata['avro.schema'])
24     # recuperamos los empleados
25     empleados = [empleado for empleado in reader]
26     reader.close()
27
28 print(f'Schema de empleado.avsc:\n {schema}')
29 print(f'Schema del fichero empleados.avro:\n {schemaFromFile}')
30 print(f'Empleados:\n {empleados}')
```



Resultado:

Schema de empleado.avsc:

```
{"type": "record", "name": "empleado", "namespace": "SeveroOchoa", "fields": [{"type": "string", "name": "nombre"}, {"type": "int", "name": "altura"}, {"type": ["null", "int"], "name": "edad", "default": null}]}
```

Schema del fichero empleados.avro:

```
{'type': 'record', 'name': 'empleado', 'namespace': 'SeveroOchoa', 'fields': [{'type': 'string', 'name': 'nombre'}, {'type': 'int', 'name': 'altura'}, {'type': ['null', 'int'], 'name': 'edad', 'default': None}]}
```

Empleados:

```
[{'nombre': 'Carlos', 'altura': 180, 'edad': 44}, {'nombre': 'Juan', 'altura': 175, 'edad': None}]
```

Fastavro

Para trabajar con Avro y grandes volúmenes de datos, es mejor utilizar la librería *Fastavro* (<https://github.com/fastavro/fastavro>) la cual ofrece un rendimiento mayor (en vez de estar codificada en Python puro, tiene algunos fragmentos realizados mediante Cython).

Primero, hemos de instalar la librería:

```
1 pip install fastavro
2 # o si utilizamos Anaconda
3 conda install -c conda-forge fastavro
```

Como podéis observar a continuación, hemos repetido el ejemplo y el código es muy similar:

```

1  import fastavro
2  import copy
3  import json
4  from fastavro import reader
5
6  # abrimos el fichero en modo binario y leemos el esquema
7  with open("empleado.avsc", "rb") as f:
8      schemaJSON = json.load(f)
9      schemaDict = fastavro.parse_schema(schemaJSON)
10
11  empleados = [{"nombre": "Carlos", "altura": 180, "edad": 44},
12               {"nombre": "Juan", "altura": 175}]
13
14  # escribimos un fichero a partir del esquema leído
15  with open('empleadosf.avro', 'wb') as f:
16      fastavro.writer(f, schemaDict, empleados)
17
18  # abrimos el archivo creado, lo leemos y mostramos línea a línea
19  with open("empleadosf.avro", "rb") as f:
20      reader = fastavro.reader(f)
21      # copiamos los metadatos del fichero leído
22      metadata = copy.deepcopy(reader.metadata)
23      # obtenemos el schema del fichero leído
24      schemaReader = copy.deepcopy(reader.writer_schema)
25      schemaFromFile = json.loads(metadata['avro.schema'])
26      # recuperamos los empleados
27      empleados = [empleado for empleado in reader]
28
29  print(f'Schema de empleado.avsc:\n {schemaDict}')
30  print(f'Schema del fichero empleadosf.avro:\n {schemaFromFile}')
31  print(f'Empleados:\n {empleados}')

```

Resultado:

Schema de empleado.avsc:

```
{'type': 'record', 'name': 'SeveroOchoa.empleado', 'fields': [{'name': 'nombre', 'type': 'string'}, {'name': 'altura', 'type': 'int'}, {'default': None, 'name': 'edad', 'type': ['null', 'int']}], '__fastavro_parsed': True, '__named_schemas': {'SeveroOchoa.empleado': {'type': 'record', 'name': 'SeveroOchoa.empleado', 'fields': [{'name': 'nombre', 'type': 'string'}, {'name': 'altura', 'type': 'int'}, {'default': None, 'name': 'edad', 'type': ['null', 'int']}]]}}
```

Schema del fichero empleadosf.avro:

```
{'type': 'record', 'name': 'SeveroOchoa.empleado', 'fields': [{'name': 'nombre', 'type': 'string'}, {'name': 'altura', 'type': 'int'}, {'default': None, 'name': 'edad', 'type': ['null', 'int']}]}
```

Empleados:

```
[{'nombre': 'Carlos', 'altura': 180, 'edad': 44}, {'nombre': 'Juan', 'altura': 175, 'edad': None}]
```

Fastavro y Pandas

Finalmente, vamos a realizar un último ejemplo con las dos librerías más utilizadas.

Vamos a leer un fichero CSV de [ventas](#) mediante *Pandas*, y tras limpiar los datos y quedarnos únicamente con las ventas de Alemania, almacenaremos el resultado del procesamiento en Avro.

Acceso local:

```

1 import pandas as pd
2 from fastavro import writer, parse_schema
3
4 # Leemos el csv mediante pandas
5 df = pd.read_csv('pdi_sales.csv', sep=';')
6 # Limpiamos los datos (strip a los códigos postales) y nos quedamos con Alemania
7 df['Zip'] = df['Zip'].str.strip()
8 filtro = df.Country=="Germany"
9 df = df[filtro]
10
11 # 1. Definimos el esquema
12 schema = {
13     'name': 'Sales',
14     'namespace': 'SeveroOchoa',
15     'type': 'record',
16     'fields': [
17         {'name': 'ProductID', 'type': 'int'},
18         {'name': 'Date', 'type': 'string'},
19         {'name': 'Zip', 'type': 'string'},
20         {'name': 'Units', 'type': 'int'},
21         {'name': 'Revenue', 'type': 'float'},
22         {'name': 'Country', 'type': 'string'}
23     ]
24 }
25 schemaParseado = parse_schema(schema)
26
27 # 2. Convertimos el Dataframe a una lista de diccionarios
28 records = df.to_dict('records')
29
30 # 3. Persistimos en un fichero avro
31 with open('sales.avro', 'wb') as f:
32     writer(f, schemaParseado, records)

```

## Acceso HDFS

También podemos conectarnos a HDFS para leer y persistir datos en formato Avro:

```

1  import pandas as pd
2  from fastavro import parse_schema
3  from hdfs import InsecureClient
4  from hdfs.ext.avro import AvroWriter
5  from hdfs.ext.dataframe import write_dataframe
6
7  # 1. Nos conectamos a HDFS
8  HDFS_HOSTNAME = 'iabd-virtualbox'
9  HDFSCLI_PORT = 9870
10 HDFSCLI_CONNECTION_STRING = f'http://{HDFS_HOSTNAME}:{HDFSCLI_PORT}'
11 hdfs_client = InsecureClient(HDFSCLI_CONNECTION_STRING)
12
13 # 2. Leemos el Dataframe
14 with hdfs_client.read('/user/iabd/pdi_sales.csv') as reader:
15     df = pd.read_csv(reader, sep=';')
16
17 # Limpiamos los datos (strip a los códigos postales) y nos quedamos con Alemania
18 df['Zip'] = df['Zip'].str.strip()
19 filtro = df.Country=="Germany"
20 df = df[filtro]
21
22 # 3. Definimos el esquema
23 schema = {
24     'name': 'Sales',
25     'namespace': 'Severo0choa',
26     'type': 'record',
27     'fields': [
28         {'name': 'ProductID', 'type': 'int'},
29         {'name': 'Date', 'type': 'string'},
30         {'name': 'Zip', 'type': 'string'},
31         {'name': 'Units', 'type': 'int'},
32         {'name': 'Revenue', 'type': 'float'},
33         {'name': 'Country', 'type': 'string'}
34     ]
35 }
36 schemaParseado = parse_schema(schema)
37
38 # 4a. Persistimos en un fichero avro dentro de HDFS mediante la extension AvroWriter de hdfs
39 with AvroWriter(hdfs_client, '/user/iabd/sales.avro', schemaParseado) as writer:
40     records = df.to_dict('records') # diccionario
41     for record in records:
42         writer.write(record)
43
44 # 4b. O directamente persistimos el Dataframe mediante la extension write_dataframe de hdfs
45 write_dataframe(hdfs_client, '/user/iabd/sales2.avro', df) # infiere el esquema
46 write_dataframe(hdfs_client, '/user/iabd/sales3.avro', df, schema=schemaParseado)

```

## Comprimiendo los datos

¿Y sí comprimimos los datos para ocupen menos espacio en nuestro clúster y por tanto, nos cuesten menos dinero?

La compresión de datos es compleja, pero la idea básica es fácil de entender, ya que los algoritmos de compresión buscan redundancia y repetición en los datos, recodificando los datos para reducir la redundancia.

Al comprimir los datos, además de ocupar menos espacio, la lectura de éstos será más rápida y se enviarán menos datos a través de la red, mejorando los tiempos de trabajo. Eso sí, no hemos de olvidar que comprimir y descomprimir los archivos conlleva un tiempo extra y consumo de recursos para leer o escribir los datos.

*Fastavro* soporta dos tipos de compresión: *gzip* (mediante el algoritmo *deflate*) y *snappy*. **Snappy** es una biblioteca de compresión y descompresión de datos de gran rendimiento que se utiliza con frecuencia en proyectos *Big Data*, la cual prioriza la velocidad sobre el tamaño resultante.

Antes de nada la hemos de instalar mediante:

```
1 pip install python-snappy
```

Para indicar el tipo de compresión, únicamente hemos de añadir un parámetros extra con el algoritmo de compresión en la función/constructor de persistencia:

Fastavro y gzip

```
1 writer(f, schemaParseado, records, 'deflate')
```

AvroWriter y snappy

```
1 with AvroWriter(hdfs_client, '/user/iabd/sales.avro', schemaParseado, 'snappy') as writer:
```

Write\_dataframe y snappy

```
1 write_dataframe(hdfs_client, '/user/iabd/sales3.avro', df, schema=schemaParseado, codec='snappy')
```

## Comparando algoritmos de compresión

Respecto a la compresión, sobre un fichero de 100GB, podemos considerar media si ronda los 50GB y alta si baja a los 40GB.

Algoritmo	Velocidad	Compresión
Gzip	Media	Media
Bzip2	Lenta	Alta
Snappy	Alta	Media

Más que un tema de espacio, necesitamos que los procesos sean eficientes y por eso priman los algoritmos que son más rápidos. Si te interesa el tema, es muy interesante el artículo [Data Compression in Hadoop](#).

Por ejemplo, si realizamos el ejemplo de Fast Avro y Pandas con acceso local obtenemos los siguientes tamaños:

- Sin compresión: 6,9 MiB
- Gzip: 1,9 MiB
- Snappy: 2,8 MiB

## Parquet

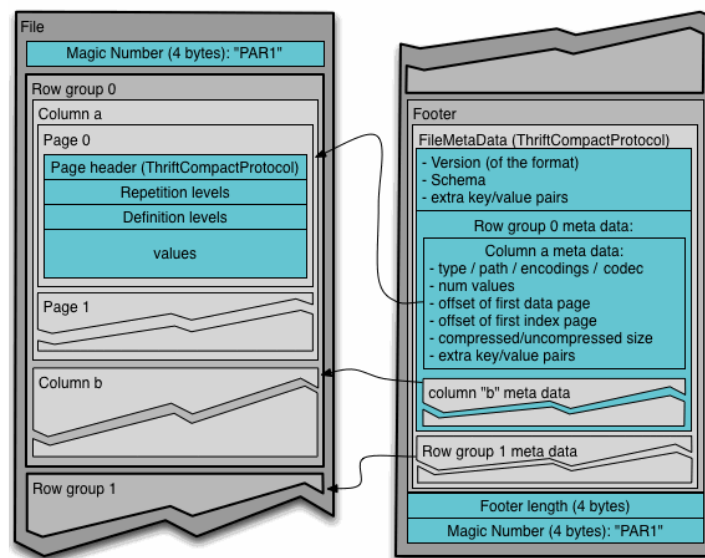
[Apache Parquet](#) es un formato de almacenamiento basado en columnas para *Hadoop*, con soporte para todos los frameworks de procesamiento de datos, así como lenguajes de programación. De la misma forma que Avro, se trata de un formato de datos auto-descriptivo, de manera que embebe el esquema o estructura de los datos con los propios datos en sí. *Parquet* es idóneo para analizar datasets que contienen muchas columnas, es decir, para lecturas de grandes cargas de trabajo.

Tiene un ratio de compresión muy alto (mediante *Snappy* ronda el 75%), y además, solo se recorren las columnas necesarias en cada lectura, lo que reduce las operaciones de entrada/salida en disco.

### Formato

Cada fichero *Parquet* almacena los datos en binario organizados en grupos de filas. Para cada grupo de filas (*row group*), los valores de los datos se organizan en columnas, lo que facilita la compresión a nivel de columna.

La columna de metadatos de un fichero *Parquet* se almacena al final del fichero, lo que permite que las escrituras sean rápidas con una única pasada. Los metadatos pueden incluir información como los tipos de datos, esquemas de codificación/compresión, estadísticas, nombre de los elementos, etc...



Formato de un archivo en Python

### Parquet y Python

Para interactuar con el formato Parquet mediante Python, la librería más utilizada es la que ofrece [Apache Arrow](#), en concreto la librería [PyArrow](#).

Así pues, la instalamos mediante pip:

```
1 pip install pyarrow
```

Apache Arrow usa un tipo de estructura denominada tabla para almacenar los datos bidimensional (sería muy similar a un *Dataframe* de *Pandas*). La documentación de PyArrow dispone de un [libro de recetas](#) con ejemplos con código para los diferentes casos de uso que se nos puedan plantear.

Vamos a simular el mismo ejemplo que hemos realizado previamente mediante Avro, y vamos a crear un fichero en formato *JSON* con empleados, y tras persistirlo en formato *Parquet*, lo vamos a recuperar:

Empleados en columnas:

```
dict-parquet.py

1 import pyarrow.parquet as pq
2 import pyarrow as pa
3
4 # 1.- Definimos el esquema
5 schema = pa.schema([ ('nombre', pa.string()),
6                      ('altura', pa.int32()),
7                      ('edad', pa.int32()) ])
8
9 # 2.- Almacenamos los empleados por columnas
10 empleados = {"nombre": ["Carlos", "Juan"],
11             "altura": [180, 44],
12             "edad": [None, 34]}
13
14 # 3.- Creamos una tabla Arrow y la persistimos mediante Parquet
15 tabla = pa.Table.from_pydict(empleados, schema)
16 pq.write_table(tabla, 'empleados.parquet')
17
18 # 4.- Leemos el fichero generado
19 table2 = pq.read_table('empleados.parquet')
20 schemaFromFile = table2.schema
21
22 print(f'Schema del fichero empleados.parquet:\n{schemaFromFile}\n')
23 print(f'Tabla de Empleados:\n{table2}')
```

Empleados en filas

Para que *pyarrow* pueda leer los empleados como documentos *JSON*, hoy en día sólo puede hacerlo leyendo documentos individuales almacenados en fichero. Por lo tanto, creamos el fichero *empleados.json* con la siguiente información:

```
1 { "nombre": "Carlos", "altura": 180, "edad": 44 }
2 { "nombre": "Juan", "altura": 175 }
```

De manera que podemos leer los datos *JSON* y persistirlos en *Parquet* del siguiente modo:



#### json-parquet.py

```
1 import pyarrow.parquet as pq
2 import pyarrow as pa
3 from pyarrow import json
4
5 # 1.- Definimos el esquema
6 schema = pa.schema([ ('nombre', pa.string()),
7                      ('altura', pa.int32()),
8                      ('edad', pa.int32()) ])
9
10 # 2.- Leemos los empleados
11 tabla = json.read_json("empleados.json")
12 # 3.- Persistimos la tabla en Parquet
13 pq.write_table(tabla, 'empleados-json.parquet')
14
15 # 4.- Leemos el fichero generado
16 table2 = pq.read_table('empleados-json.parquet')
17 schemaFromFile = table2.schema
18
19 print(f'Schema del fichero empleados-json.parquet:\n{schemaFromFile}\n')
20 print(f'Tabla de Empleados:\n{table2}')
```

En ambos casos obtendríamos algo similar a:

```
1 Schema del fichero empleados.parquet:
2 nombre: string
3 altura: int32
4 edad: int32
5
6 Tabla de Empleados:
7 pyarrow.Table
8 nombre: string
9 altura: int32
10 edad: int32
11 ----
12 nombre: [["Carlos","Juan"]]
13 altura: [[180,44]]
14 edad: [[null,34]]
```

#### Parquet y Pandas

En el caso del uso de *Pandas* el código todavía se simplifica más. Si reproducimos el mismo ejemplo que hemos realizado con *Avro* tenemos que los *Dataframes* ofrecen el método [to\\_parquet](#) para exportar a un fichero *Parquet*:

#### csv-parquet.py

```
1 import pandas as pd
2
3 df = pd.read_csv('pdi_sales.csv', sep=';')
4
5 df['Zip'] = df['Zip'].str.strip()
6 filtro = df.Country=="Germany"
7 df = df[filtro]
8
9 # A partir de un DataFrame, persistimos los datos
10 df.to_parquet('pdi_sales.parquet')
```

Para leer un archivo utilizaremos el método [read\\_parquet](#):

```
1 df_parquet = pd.read_parquet("pdi_sales.parquet")
```

## Parquet y HDFS

Si quisiéramos almacenar el archivo directamente en HDFS, necesitamos indicarle a *Pandas* la dirección del sistema de archivos que tenemos configurado en *core-site.xml*:

#### core-site.xml

```
1 <property>
2     <name>fs.defaultFS</name>
3     <value>hdfs://iabd-virtualbox:9000</value>
4 </property>
```

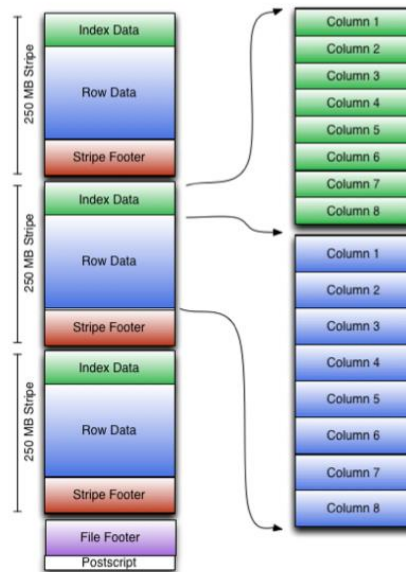
Así pues, únicamente necesitamos modificar el nombre del archivo donde serializamos los datos a *Parquet*:

```
1 df.to_parquet('hdfs://iabd-virtualbox:9000/sales.parquet')
```

## ORC

[Apache ORC](#) es un formato de datos columnar (**O**ptimized **R**ow **C**olumnar) similar a *Parquet* pero optimizado para la lectura, escritura y procesamiento de datos en *Hive*. ORC tiene una tasa de compresión alta (utiliza *zlib*), y al basarse en *Hive*, soporta sus tipos de datos simples (*datetime*, *decimal*, etc...) y los tipos complejos (como *struct*, *list*, *map* y *union*), siendo totalmente compatible con *HiveQL*.

Los fichero ORC se componen de tiras de datos (*stripes*), donde cada tira contiene un índice, los datos de la fila y un pie (con estadísticas como la cantidad, máximos y mínimos y la suma de cada columna convenientemente cacheadas)



Formato de un archivo ORC

## ORC y Python

Si queremos usar *Pandas*, para leer un fichero ORC emplearemos el método [read\\_orc](#):

```
df_orc = pd.read_orc('pdi_sales.orc')
```

Y para persistirlo, desde la versión 1.5 de *Pandas*, a partir de un *DataFrame*, podemos persistir los datos mediante el método [to\\_orc](#).

Por defecto, los archivos ORC se persisten sin comprimir. Si queremos comprimirlos, debemos pasarle los parámetros mediante el atributo `engine_kwargs` mediante un diccionario:

```
df_orc.to_orc('pdi_sales_zlib.orc', engine_kwargs={"compression": 'zlib'})
```

Además, también podemos hacerlo de nuevo desde la librería [PyArrow](#). Así pues, para la escritura de datos, por ejemplo, desde un *Dataframe*, haríamos:

```
1 import pandas as pd
2 import pyarrow as pa
3 import pyarrow.orc as orc
4
5 df = pd.read_csv('pdi_sales.csv', sep=';')
6
7 # Limpiamos los datos (strip a los códigos postales) y nos quedamos con Alemania
8 df['Zip'] = df['Zip'].str.strip()
9 filtro = df.Country=="Germany"
10 df = df[filtro]
11
12 table = pa.Table.from_pandas(df, preserve_index=False)
13 orc.write_table(table, 'pdi_sales.orc')
```

## Comparando formatos

Acabamos de ver que cada uno de los formatos tiene sus puntos fuertes.

Los formatos basados en filas ofrecen un rendimiento mayor en las escrituras que en las lecturas, ya que añadir nuevos registros es más sencillo. Si sólo vamos a hacer consultas sobre un subconjunto de las columnas, entonces un formato columnar se comportará mejor, ya que no necesita recuperar los registros enteros (cosa que sí hacen los formatos basados en filas).








Respecto a la compresión, entendiendo que ofrece una ventaja a la hora de almacenar y transmitir la información, es útil cuando trabajamos con un gran volumen de datos. Los formatos basados en columnas ofrecen un mejor rendimiento ya que todos los datos del mismo tipo se almacenan de forma contigua lo que permite una mayor eficiencia en la compresión (además, cada tipo de columna tiene su propia codificación).

Respecto a la evolución del esquema, con operaciones como añadir o eliminar columnas o cambiar su nombre, la mejor decisión es decantarse por Avro. Además, al tener el esquema en JSON facilita su gestión y permite que tengan más de un esquema.

Si nuestros documentos tienen una estructura compleja compuesta por columnas anidadas y normalmente realizamos consultas sobre un subconjunto de las subcolumnas, la elección debería ser *Parquet*, por la estructura que utiliza.

Finalmente, recordar que ORC está especialmente enfocado a su uso con *Hive*, mientras que *Spark* tiene un amplio soporte para *Parquet* y que si trabajamos con *Kafka*, *Avro* es una buena elección.

BIG DATA FORMATS COMPARISON

	Avro	Parquet	ORC
Schema Evolution Support			
Compression			
Splitability			
Most Compatible Platforms	Kafka, Druid	Impala, Arrow Drill, Spark	Hive, Presto
Row or Column	Row	Column	Column
Read or Write	Write	Read	Read

Source: Nexla analysis, April 2018

Si comparamos los tamaños de los archivos respecto al formato de datos empleado con únicamente las ventas de Alemania tendríamos:

### Actividades:

Mediante Python y utilizando Kaggle, crea un notebook a partir de los datos del dataset de [retrasos en los vuelos](#) y a partir de uno de los ficheros (el que más te guste), y teniendo en cuenta que los campos están separados por comas (,), transforma los datos y persiste los siguientes archivos:

- `air<anyo>.parquet`: el archivo CSV en formato *Parquet*.
- `air<anyo>.orc`: el archivo CSV en formato *ORC*.
- `air<anyo>_snappy.orc`: el archivo CSV comprimido en formato *Snappy* en formato *ORC*.
- `air<anyo>_small.avro`: la fecha (`FL_DATE`), el identificador de la aerolínea (`OP_CARRIER`) y el retraso de cada vuelo (`DEP_DELAY`) en formato *Avro*.

### Seleccionando columnas

Mediante Pandas, cuando tenemos un *DataFrame*, podemos seleccionar un subconjunto de las columnas de la siguiente forma:

```
1 # df es un DataFrame que contiene todas las columnas
2 df_small = df[['FL_DATE', 'OP_CARRIER', 'DEP_DELAY']]
```

- `air<anyo>_small.parquet`: con los mismos atributos pero en *Parquet*.

Adjunta una captura del cuaderno, anota los tamaños de los ficheros creados y el tiempo necesario para su creación (agrupa todos los datos en una celda a modo de tabla para poder compararlos visualmente, por ejemplo, mediante una celda de tipo *Markdown*).

Para el tamaño de los archivos, puedes comprobarlo en el panel lateral derecho de Kaggle dentro de *Output*, o utilizar el método `os.path.getsize(ruta)`:

```
1 import os
2
3 os.path.getsize("/kaggle/working/air20XX.parquet")
```

Respecto al tiempo, por ejemplo puedes obtenerlo así:

```
1  import time
2
3  inicio = time.time()
4  # operación a medir
5  fin = time.time()
6  print(fin - inicio)
```