AVRO

Agenda

- Repaso
- Avro
- Poniéndolo en práctica

Repaso

- Formatos columnares en ficheros

- Ventajas de utilizar Serializadores/Deserializadores en Hive
  - Parquet
  - Json
  - RegEx

- Repasemos las prácticas…

¿Esquemas?, ¿Por qué?

- La necesidad de los esquemas
  - ¿Qué es un esquema?
- Las complicaciones de los esquemas
  - ¿Quién gestiona/coordina/organiza los esquemas?
- Se hace necesario mitigar el impacto de los esquemas
  - ¿Cómo usaban los componentes vistos hasta ahora los esquemas?

¿Por qué alejar los datos de los esquemas que nos ayudan a interpretarlos?

¿Qué es Avro?

- Un sistema de serialización de datos
- Qué incluye:
  - Un formato compacto para la serialización
  - Un lenguaje de definición de esquemas
  - Un framework para crear una infraestructura RPC que pueda usar ese formato
  - Una colección de APIs en diversos lenguajes
- Objetivos:
  - Conseguir independencia del lenguaje
  - Soportar acceso dinámico
  - Uso de esquemas de forma sencilla pero muy potente

Especificación Avro

- La información a enviar siempre va a compañada con su esquema y se representa de una de las siguientes maneras:
  - JSON string, naming a defined type.
  - JSON object:
    - {"type": "typeName" ...attributes...}
  - JSON array

```
{
    "type": "record",
    "namespace": "com.example",
    "name": "FullName",
    "fields": [
      { "name": "first", "type": "string" },
      { "name": "last", "type": "string" }
    ]
}
```

- Tipos primitivos soportados: null, boolean, int, long, float, double, bytes, string
  - {"type": "string"}
- Tambien tipos complejos: records, enums, arrays, maps, unions, fixed

Tipos soportados en los esquemas

- **Primitive types**
  - null: no value
  - boolean: a binary value
  - int: 32-bit signed integer
  - long: 64-bit signed integer
  - float: single precision (32-bit) IEEE 754 floating-point number
  - double: double precision (64-bit) IEEE 754 floating-point number
  - bytes: sequence of 8-bit unsigned bytes
  - string: unicode character sequence
- **Complex Types**
  - records – named sets of fields
    - Fields are JSON objects with a name, type and optionally a default value
  - enums – named set of strings, instances may hold any one of the string values
  - fixed – a named fixed size set of bytes
  - arrays – a collection of a single type
  - maps – a string keyed set of key/value pairs
  - unions – an array of types, instances may assume any of the types

Más ejemplos

```
{
   "type" : "record",
   "name" : "userInfo",
   "namespace" : "my.example",
   "fields" : [{"name" : "username", "type" : "string",  "default" : "NONE"},
               {"name" : "age",  "type" : "int",  "default" : -1},
               {"name" : "phone", "type" : "string", "default" : "NONE"},
               {"name" : "housenum", "type" : "string", "default" : "NONE"},
               {"name" : "address",
                "type" : { "type" : "record",
                     "name" : "mailing_address",
                     "fields" : [ {"name" : "street",  "type" : "string",  "default" : "NONE"},
                                  {"name" : "city", "type" : "string",  "default" : "NONE"},
                                  {"name" : "state_prov", "type" : "string",  "default" : "NONE"},
                                  {"name" : "country", "type" : "string",  "default" : "NONE"},
                                  {"name" : "zip", "type" : "string", "default" : "NONE"}
                                ] },
                 "default" : {}
               }
              ]
}
```

APIs

- Python
  - Dynamic
- Java
  - Specific (generated code)
  - Generic (container-based)
  - Reflection (induces schemas from classes)
- C
- C++
- Ruby

Evolución y proyección del esquema

- Los datos de AVRO SIEMPRE viajan CON su esquema, y además permite una interpretación dinámica
- El esquema con el que se escribe no tiene que ser igual al que utilizamos en la lectura

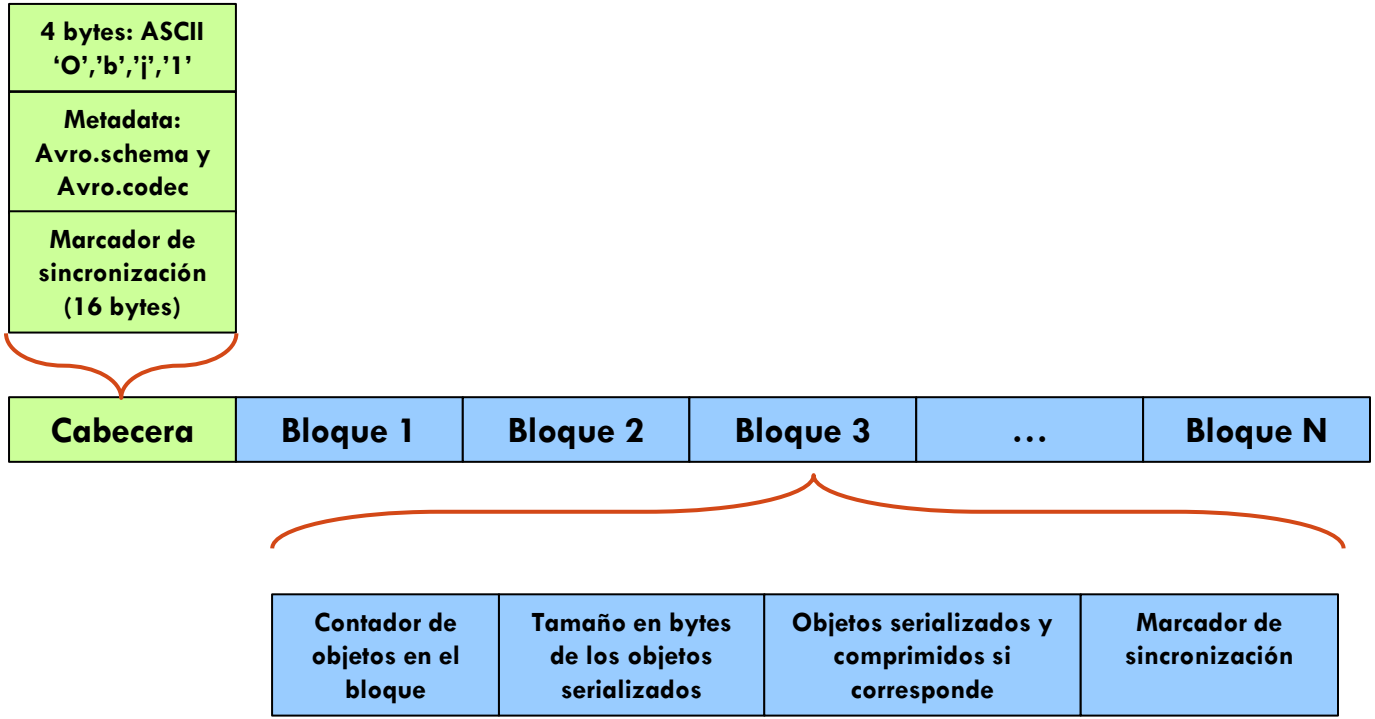
```
{ /* Writer */
"type" : "record",
"name" : "Person",
"fields" : [ {
                "name" : "first",
                "type" : "string“},
              {
                "name" : "sport",
                "type" : "string“},
               {…}
 }
```

Serialized Data:
"Alice", "Ultimate Frisbee"

```
{ /* Reader */
"type" : "record",
"name" : "Person",
"fields" : [ {
                "name" : "first",
                "type" : "string"
                }, {
                "name" : "age",
                "type" : "int",
                "default": 0},
               {…}
}
```

Data presented to application:
"Alice", 0

Contenedor de Avro



Qué es un contenedor en Avro

- Avro define un fichero como contenedor en el que alojar tanto el esquema como sus datos

- El fichero está formado entonces por:
  - Una **cabecera** que almacena su metada y un marcador de sincronización
  - Uno o más **bloques** de **datos** que contienen los datos de acuerdo al esquema definido

Cabecera

- Metadata
  - Hay dos valores definidos por defecto:
    - avro.schema – que contiene el esquema en formato JSON
    - avro.codec – que especifica el nombre del codec de compresión utilizado (si se ha utilizado uno)
      - Los codecs obligatorios son NULL y deflate, null (Sin compression)
      - Snappy es otro códec habitual
  - También se puede incluir información que el autor del fichero pueda considerarla útil
- Y una marca que se utiliza para separar e identificar bloques

Bloques de Datos (I)

- Contienen un indicador del número de objetos en el bloque, su ocupación (tamaño), los propios objetos serializados (posiblemente comprimidos) y una marca de sincronización
- Los bloques **no incluyen** ni los nombres de los datos ni información sobre el tipo de byte ni separadores de campo o registros

Bloques de Datos (II)

- El prefijo del bloque y la marca de sincronización permiten manejarlos eficientemente en las tareas de mapreduce con HDFS
  - Incluye una opción además para detectar bloques corruptos

Ordenacion

- Avro incluye un método de ordenación de los datos
  - Útil cuando tenemos que procesar ficheros de datos muy grandes
- En realidad la ordenación se realiza con los objetos en binario, no es necesario deserializarlo
- Solo se pueden comparar datos cuando tengan esquemas idénticos
- Las opciones de ordenación son:
  - Ascendente
  - Descendente
  - Ignore – Valores a ser ignorados cuando se ordenan los registros

Avro y RPC

- Avro originalmente fue pensado para ser un formato de serialización y deserialización de objetos para que los nodos de Hadoop intercambiaran los datos
- Es por ello que la especificación original además de incluir el formato de los objetos incluye (totalmente integrado) el protocolo para el intercambio de información

RPC y protocolos

- Apache Avro define cómo usar las interfaces de RPC utilizando una definición del **protocolo** a través de un **esquema** específico
- El fichero JSON define y contiene
  - Protocolo (nombre del servicio)
  - Namespace – (opcional)
  - Types – los tipos definidos dentro del protocolo
  - Mensajes – las secuencias de RPC soportadas
- Los mensajes se definen un método de RPC
  - request – lista de parámetros que se deben enviar/pasar
  - response – valor esperado a devolver
  - error – Tipo del mensaje de error a devolver (opcional)
  - one-way – mensaje opcional (boolean) solo para mensajes de request

Integración con Hadoop

- Involucración en MapReduce
  - AvroInputFormat/AvroOutputFormat (MR-815)
  - Documentado como usar AVRO en el paso de shuffle (MR-1126)
  - Recordemos que en los esquemas de AVRO se puede especificar la ordenación de los datos
- Framework
  - AVRO for Hadoop RPC (e.g., HDFS-982)

Como integrarlo con Hive

```
CREATE EXTERNAL TABLE tweets
    COMMENT "A table backed by Avro data with the Avro schema stored in HDFS"
    ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
    STORED AS
    INPUTFORMAT  'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
    OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
    LOCATION '/user/YOURUSER/examples/input/'
    TBLPROPERTIES (
        'avro.schema.url'='hdfs:///user/YOURUSER/examples/schema/twitter.avsc'
    );
```

Usando Avro

- Por defecto está disponible en java y podremos contar (entre otras cosas) con:
  - Avro-1.x.y.jar
  - Avro-tools-1.x.y.jar
- También está disponible en Python (pip install avro)
- Es importante prestar atención a las versiones que estemos utilizando para garantizar la compatibilidad. En la plataforma del Máster, si os conectáis al Docker de HIVE podréis ver, por ejemplo, distintas versiones dependiendo del componente

Practicando avro

- Descarga los ficheros zip de Moodle con los ficheros de la práctica
- Copialos en el Docker en el que quieras trabajar, por ejemplo HIVE:

  ```
  docker cp fichero_zip hive-server:/root
  ```

- Abre una sesión en el contenedor de Hive

  ```
  docker-compose exec hive bash
  ```

- El fichero zip contiene tanto las librerías de avro como los ficheros de ejemplo
- Descomprime el fichero zip que vayas a utilizar

Practicando avro (tweets)

- Utilizaremos las tools de avro para poder manejar los ficheros de ejemplo basados en tweets.

- Fíjate las opciones de las tools de avro
  - `java -jar avro-tools-1.7.7.jar`

- Los ficheros que se incluyen en el ejemplo son:
  - `.json` con ejemplo de contenido
  - `.avsc` con un posible avro schema

Extraido de: https://github.com/miguno/avro-cli-examples

Practicando avro (III)

- Como generar un fichero avro desde los ficheros originales:

```
java -jar ./avro-tools-1.7.7.jar fromjson --schema-file twitter.avsc
twitter.json > twitter.avro
```

```
java -jar ./avro-tools-1.7.7.jar fromjson --codec snappy --schema-file
twitter.avsc twitter.json > twitter.snappy_2.avro
```

- Y viceversa:

```
java -jar ./avro-tools-1.7.7.jar tojson twitter.avro > twitter.json
```

```
java -jar ./avro-tools-1.7.7.jar tojson –codec snappy twitter.snappy.avro >
twitter.json
```

- Observa que las tools de avro nos permitirían también crear la infraestructura Cliente/Servidor para invocaciones RPC

Extraido de: https://github.com/miguno/avro-cli-examples

Practicando Avro con Hive (CTAS)

- Si tienes cargadas ya tablas en hive, ¿cual sería la sentencia para crear una tabla en formato avro como CTAS?

```
CREATE TABLE ratings_avro STORED AS avro AS SELECT * FROM parquetdb.ratings;
```

- Utiliza el comando describe para ver qué características tiene

```
describe formatted avrodb.ratings_avro
```

- ¿Donde está el fichero en HDFS?

```
hdfs dfs -ls /warehouse/tablespace/managed/hive/avrodb.db/ratings_avro/delta_0000001_0000001_0000
```

Confirmando el formato avro usado por Hive

- *Tráete* el fichero de hive (sácalo de hive y llevalo al docker) y confirma que está en avro. Utiliza para ello las avro-tools

```
hdfs dfs -get
/warehouse/tablespace/managed/hive/avrodb.db/ratings_avro/delta_0000001_0000001_0000/
000000_0 ratings_avro_from_table_hive.avro
```

```
java -jar ./avro-tools-1.7.7.jar getschema ratings_avro_from_table_hive.avro
```

- Compara el schema de la tabla a partir de las avro-tools con el formato reconocido por HIVE

Mas opciones

- Se puede crear un fichero avro en cualquier plataforma y hacerlo disponible para Hive como una external table (recuerda que hicimos algo parecido con Parquet)
- El schema puede estar embebido dentro del fichero avro o puede ser especificado como una TBLPROPERTIES indicando:
  - El schema como un string completo
  - El schema indicado como un fichero recogido en hdfs
- También podemos crear una tabla vacía en formato AVRO y cargar los valores a través de Load desde un fichero en local o en hdfs