# How to Work With Avro Files

**by Dmytro Manannykov · Oct. 11, 18 · Big Data Zone · Tutorial**

## Introduction

This short article describes how to transfer data from Oracle database to S3 using Apache Sqoop utility. The data will be stored in Avro data format.

### Apache Sqoop

*Apache Sqoop* is a command-line interface application for transferring data between relational databases and Hadoop. Sqoop allows easy import and export of data from structured data stores such as relational databases, enterprise data warehouses, and NoSQL systems. Using Sqoop, you can provision the data from external system on to HDFS, and populate tables in Hive and HBase. Sqoop integrates with Oozie, allowing you to schedule and automate import and export tasks. Sqoop uses a connector-based architecture which supports plugins that provide connectivity to new external systems.

### Apache Avro

*Avro* is a row-based storage format for Hadoop which is widely used as a serialization platform. Avro stores the data definition (schema) in JSON format making it easy to read and interpret by any program. The data itself is stored in binary format making it compact and efficient. A key feature of Avro is robust support for data schemas that change over time - schema evolution. Avro handles schema changes like missing fields, added fields and changed fields; as a result, old programs can read new data and new programs can read old data. This format is the ideal candidate for storing data in a data lake landing zone, because:

1. Data from the landing zone is usually read as a whole for further processing by downstream systems (the row-based format is more efficient in this case);

2. Downstream systems can easily retrieve table schemas from files (there is no need to store the schemas separately in an external meta store);

3. Avro data format successfully handles line breaks (\n) and other non-printable characters in data (for example, a string field can contain formatted JSON or XML file);

4. Any source schema change is easily handled (schema evolution).

## Environment

The data transfer was done using the following technologies:

- Apache Sqoop 1.4.7
- Oracle 12c
- Amazon EMR 5.16.0 (Hadoop distribution 2.8.4)

## Sqoop Command to Store Data in Avro Format

Apache Sqoop 1.4.7 supports Avro data files. To store data in Avro format, the following parameters should be added to the Sqoop command:

```
--as-avrodatafile # imports data to Avro data files
--compression-codec snappy # use Hadoop codec (in this case - snappy)
```

The template of a Sqoop command is as follows:

```
sqoop import \
  --bindir ./ \
  --connect 'dbc:oracle:thin:<username>/password@<host>:<port>/<instance_name>' \
      # 'jdbc:sqlserver://<host>:<port>;databasename=<database_name>' \ # SQL Server 2008 and higher
      # 'jdbc:jtds:sqlserver://<host>:<port>/<database_name>' \ - #SQL Server 2000 \
  --username <username> \
  --driver <driver_class> # manually specify JDBC driver class to use
```

```
                          # example: --driver net.sourceforge.jtds.jdbc.Driver
  --connection-manager # Specify connection manager class to use
                       # example: --connection-manager org.apache.sqoop.manager.SQLServerManager
  --password <password> \
  --num-mappers <n> \
  --fields-terminated-by '\t' \ # sets the field separator character
  --lines-terminated-by '\n' \  # sets the end-of-line character
  --as-avrodatafile \           # imports data to Avro data files
  --compression-codec snappy \  # use Hadoop codec (in this case - snappy)
  --options-file <path_to_options_file> \
  --split-by <field_name> \ # only used if number of mappers > 1
  --target-dir s3://<path> \
      # example for HDFS: --target-dir hdfs:///<path>
  --null-string '' \
  --null-non-string ''
  --boundary-query # if used then --split-by should also be present
```

Example of Sqoop command for Oracle to dump data to S3:

```
sqoop import \
  -Dmapreduce.job.user.classpath.first=true \
  --connect "jdbc:oracle:thin:user/password@host_address.com:1521/orcl" \
  --num-mappers 1 \
  --query 'select * from employee where $CONDITIONS' \
  --target-dir s3://my-bucket/staging/employee \
  --as-avrodatafile \
  --compression-codec snappy \
  --null-string '' \
  --null-non-string ''
```

Note that when you run the command the target directory should not exist, otherwise the Sqoop command will fail.

You can use a simple AWS CLI command to delete the target directory:

```
aws s3 rm s3://my-bucket/staging/employee --recursive
```

Example of a Sqoop command for Oracle to dump data to Hadoop:

```
sqoop import \
  -Dmapreduce.job.user.classpath.first=true \
  --connect "jdbc:oracle:thin:user/password@host_address.com:1521/orcl" \
  --num-mappers 1 \
  --query 'select * from employee where $CONDITIONS' \
  --delete-target-dir
  --target-dir /user/hive/warehouse/employee \
  --as-avrodatafile \
  --compression-codec snappy \
  --null-string '' \
  --null-non-string ''
```

Note, that there is a parameter, *--delete-target-dir,* in the command that deletes the target directory and can only be used if the target directory is located in HDFS.

Sqoop can transfer data to either Hadoop (HDFS) or AWS (S3). To query transferred data you need to create tables on top of physical files. If the data was transferred to Hadoop you can create Hive tables. If the data was transferred to S3 you can create either Hive tables or Amazon Athena tables. In both cases, you will need a table schema which you can retrieve from physical files. Starting from version 1.4.7 (EMR 5.14.0, Hadoop distribution: Amazon 2.8.3) Sqoop automatically retrieves table schema and stores it in an **AutoGeneratedSchema.avsc** file in the same folder. If Sqoop version 1.4.6 (a part of EMR 5.13.0) or lower is used, then the table schema can be retrieved manually.

If the destination of your data is HDFS, you can use the below command to retrieve the table schema:

```
hadoop jar avro-tools-1.8.1.jar getschema /user/hive/warehouse/employee/part-m-00000.avro > employee.avs
```

If the destination of your data is S3, you need to copy the Avro data file to local file system and then retrieve the schema:

```
java -jar avro-tools-1.8.1.jar getschema part-m-00000.avro > employee.avsc
```

*Avro-tools-1.8.1.jar* is a part of Avro Tools that provide CLI interface to work with Avro files.

After the table schema has been retrieved, it can be used for further table creation.

## Create Avro Table in Hive

To create an Avro table in Hive (on Hadoop Cluster or on EMR) you have to provide a table schema location retrieved from the Avro data file:

```
CREATE TABLE employee
STORED AS AVRO
LOCATION '/user/hive/warehouse/employee'
TBLPROPERTIES('avro.schema.url'='hdfs:///user/hive/warehouse/avsc/employee.avsc');
```

You can also specify a table location in S3::

```
CREATE TABLE employee
STORED AS AVRO
location 's3://my-bucket/staging/employee'
```

```
TBLPROPERTIES ('avro.schema.url'='hdfs:///user/hive/warehouse/avsc/employee.avsc');
```

You can even keep a table schema in S3:

```
CREATE EXTERNAL TABLE employee
STORED AS AVRO
location 's3:/my-bucket/staging/employee'
TBLPROPERTIES ('avro.schema.url'='s3://my-bucket/staging/avsc/employee.avsc');
```

The Avro schema for the EMPLOYEE table looks like this:

```
   {
     "type" : "record",
     "name" : "AutoGeneratedSchema",
     "doc" : "Sqoop import of QueryResult",
     "fields" : [ {
       "name" : "ID",
       "type" : [ "null", "string" ],
       "default" : null,
       "columnName" : "ID",
       "sqlType" : "2"
     }, {
       "name" : "NAME",
       "type" : [ "null", "string" ],
       "default" : null,
       "columnName" : "NAME",
       "sqlType" : "12"
     }, {
       "name" : "AGE",
       "type" : [ "null", "string" ],
       "default" : null,
       "columnName" : "AGE",
       "sqlType" : "2"
     }, {
       "name" : "GEN",
       "type" : [ "null", "string" ],
       "default" : null,
       "columnName" : "GEN",
       "sqlType" : "12"
     }, {
       "name" : "CREATE_DATE",
       "type" : [ "null", "long" ],
       "default" : null,
       "columnName" : "CREATE_DATE",
       "sqlType" : "93"
     }, {
       "name" : "PROCESS_NAME",
       "type" : [ "null", "string" ],
       "default" : null,
       "columnName" : "PROCESS_NAME",
       "sqlType" : "12"
     }, {
       "name" : "UPDATE_DATE"
```

```
    "name" : "UPDATE_DATE",
    "type" : [ "null", "long" ],
    "default" : null,
    "columnName" : "UPDATE_DATE",
    "sqlType" : "93"
  } ],
  "tableName" : "QueryResult"
}
```

Note that all timestamp columns are defined as *long*.

**Important**: All tables created in Hive using *create table* statement are managed tables. It means that if a table is deleted the corresponding directory in HDFS or S3 will also be deleted. To retain the data is HDFS or S3 a table should be created as external:

```
CREATE EXTERNAL TABLE employee
```

In this case, even if the external table is deleted, the physical files in HDFS or S3 will remain untouched.

## Create an Avro Table in Amazon Athena

Amazon Athena does not support the table property *avro.schema.url* — the schema needs to be added explicitly in *avro.schema.literal*:

```
CREATE EXTERNAL TABLE employee
(
  ID string,
  NAME string,
  AGE string,
  GEN string,
  CREATE_DATE bigint,
  PROCESS_NAME string,
  UPDATE_DATE bigint
)
STORED AS AVRO
LOCATION 's3://my-bucket/staging/employees'
TBLPROPERTIES (
'avro.schema.literal'='
{
    "type" : "record",
    "name" : "AutoGeneratedSchema",
    "doc" : "Sqoop import of QueryResult",
    "fields" : [ {
      "name" : "ID",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "ID",
      "sqlType" : "2"
    }, {
      "name" : "NAME",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "NAME",
      "sqlType" : "12"
    }, {
      "name" : "AGE",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "AGE",
      "sqlType" : "2"
    }, {
      "name" : "GEN",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "GEN",
      "sqlType" : "12"
    }, {
      "name" : "CREATE_DATE",
      "type" : [ "null", "long" ],
      "default" : null,
      "columnName" : "CREATE_DATE",
      "sqlType" : "93"
    }, {
      "name" : "PROCESS_NAME",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "PROCESS_NAME",
      "sqlType" : "12"
    }, {
      "name" : "UPDATE_DATE",
      "type" : [ "null", "long" ],
      "default" : null,
      "columnName" : "UPDATE_DATE",
      "sqlType" : "93"
    } ],
    "tableName" : "QueryResult"
  }
');
```

Note that all timestamp columns in the table definition are defined as *bigint.* The explanation for this is given below.

## Working With Timestamps in Avro

When Sqoop imports data from Oracle to Avro (using --*as-avrodatafile*) it stores all "timestamp" values in Unix time format (Epoch time), i.e. *long*.

### In Hive

No changes occur when creating an Avro table in Hive:

```
CREATE TABLE employee
STORED AS AVRO
LOCATION '/user/hive/warehouse/employee'
TBLPROPERTIES ('avro.schema.url'='hdfs:///user/hive/warehouse/avsc/employee.avsc');
```

When querying the data, you just need to convert milliseconds to *string*:

```
from_unixtime(<Unix time column> div 1000)
```

The resulting dataset without using timestamp conversion looks like this:

```
hive> select id, name, age, gen, create_date, process_name, update_date
    > from employee limit 2;
OK
id  name    age  gen  create_date    process_name  update_date
--  ----    ---  ---  -----------    ------------  -----------
2   John    30   M    1538265652000  BACKFILL      1538269659000
3   Jennie  25   F    1538265652000  BACKFILL      1538269659000
```

The resulting dataset using timestamp conversion looks like this:

```
hive> select
    >     id, name, age, gen,
    >     from_unixtime(create_date div 1000) as create_date,
    >     process_name,
    >     from_unixtime(update_date div 1000) as update_date
    > from employee limit 2;
OK
id  name    age  gen  create_date          process_name  update_date
--  ----    ---  ---  -----------          ------------  -----------
2   John    30   M    2018-09-30 00:00:52  BACKFILL      2018-09-30 01:07:39
3   Jennie  25   F    2018-09-30 00:00:52  BACKFILL      2018-09-30 01:07:39
```

**Important**: In Hive, if reserved words are used as column names (like*timestamp*) you need to use backquotes to escape them:

```
select from_unixtime(`timestamp` div 1000) as time_stamp
from employee;
```

## In Amazon Athena

When creating Athena tables, all *long* fields should be created as *bigint* in a CREATE TABLE statement (not in Avro schema!):

```
    CREATE EXTERNAL TABLE employee
```

```
(
  ID string,
  NAME string,
  AGE string,
  GEN string,
  CREATE_DATE bigint,
  PROCESS_NAME string,
  UPDATE_DATE bigint
)
STORED AS AVRO
LOCATION 's3://my-bucket/staging/employee'
TBLPROPERTIES (
'avro.schema.literal'='
{
    "type" : "record",
    "name" : "AutoGeneratedSchema",
    "doc" : "Sqoop import of QueryResult",
    "fields" : [ {
      "name" : "ID",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "ID",
      "sqlType" : "2"
    }, {
      "name" : "NAME",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "NAME",
      "sqlType" : "12"
    }, {
      "name" : "AGE",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "AGE",
      "sqlType" : "2"
    }, {
      "name" : "GEN",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "GEN",
      "sqlType" : "12"
    }, {
      "name" : "CREATE_DATE",
      "type" : [ "null", "long" ],
      "default" : null,
      "columnName" : "CREATE_DATE",
      "sqlType" : "93"
    }, {
      "name" : "PROCESS_NAME",
      "type" : [ "null", "string" ],
      "default" : null,
      "columnName" : "PROCESS_NAME",
      "sqlType" : "12"
    }, {
      "name" : "UPDATE_DATE",
      "type" : [ "null", "long" ],
      "default" : null,
      "columnName" : "UPDATE_DATE",
      "sqlType" : "93"
    } ],
    "tableName" : "QueryResult"
  }
');
```

When querying the data, you just need to convert milliseconds to *string*:

```
from_unixtime(<Unix time column> / 1000)
```

The resulting dataset without using timestamp conversion looks like this:

```
select id, name, age, gen, create_date, process_name, update_date
from employee limit 2;
```

```
id  name    age  gen  create_date    process_name  update_date
--  ----    ---  ---  -----------    ------------  -----------
2   John    30 M    1538265652000  BACKFILL      1538269659000
3   Jennie  25 F    1538265652000  BACKFILL      1538269659000
```

The resulting dataset using timestamp conversion looks like this:

```
select id, name, age, gen,
  from_unixtime(create_date / 1000) as create_date,
  process_name,
  from_unixtime(update_date / 1000) as update_date
from employee limit 2;
```

```
id  name    age  gen  create_date              process_name  update_date
--  ----    ---  ---  -----------              ------------  -----------
2   John    30   M    2018-09-30 00:00:52.000  BACKFILL      2018-09-30 01:07:39.000
3   Jennie  25   F    2018-09-30 00:00:52.000  BACKFILL      2018-09-30 01:07:39.000
```

## Storing Timestamp as Text

If you do not want to convert the timestamp from Unix time every time you run a query, you can store timestamp values as text by adding the following parameter to Sqoop:

```
--map-column-java CREATE_DATE=String,UPDATE_DATE=String
```

After applying this parameter and running Sqoop the table schema will look like this:

```
{
  "type" : "record",
  "name" : "AutoGeneratedSchema",
  "doc" : "Sqoop import of QueryResult",
  "fields" : [ {
    "name" : "ID",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "ID",
    "sqlType" : "2"
  }, {
    "name" : "NAME",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "NAME",
    "sqlType" : "12"
  }, {
    "name" : "AGE",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "AGE",
    "sqlType" : "2"
  }, {
    "name" : "GEN",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "GEN",
    "sqlType" : "12"
  }, {
    "name" : "CREATE_DATE",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "CREATE_DATE",
    "sqlType" : "93"
  }, {
    "name" : "PROCESS_NAME",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "PROCESS_NAME",
    "sqlType" : "12"
  }, {
    "name" : "UPDATE_DATE",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "UPDATE_DATE",
    "sqlType" : "93"
  } ],
  "tableName" : "QueryResult"
}
```

Note that the timestamp columns in the table schema are defined as *string*.

The Sqoop command for storing timestamp fields in string format:

```
sqoop import \
  -Dmapreduce.job.user.classpath.first=true \
  --connect "jdbc:oracle:thin:user/password@host_address.com:1521/orcl" \
  --num-mappers 1 \
  --query 'select * from employee where $CONDITIONS' \
  --target-dir s3://my-bucket/staging/employee_ts_str \
  --as-avrodatafile \
  --compression-codec snappy \
  --null-string '' \
  --null-non-string '' \
  --map-column-java CREATE_DATE=String,UPDATE_DATE=String
```

For dumping data to HDFS, the Sqoop command will be the same except for the *--target-dir* parameter:

```
--target-dir hdfs:.///user/hive/warehouse/employee_ts_str
```

## In Hive

Create a new table in Hive using the new table schema:

```
CREATE TABLE employee_ts_str
STORED AS AVRO
LOCATION '/user/hive/warehouse/employee_ts_str'
TBLPROPERTIES('avro.schema.url'='hdfs:///user/hive/warehouse/avsc/employee_ts_str.avsc');
```

Select the data without using timestamp conversion:

```
hive> select id, name, age, gen, create_date, process_name, update_date
    > from employee_ts_str limit 2;
OK
id  name   age  gen  create_date          process_name  update_date
--  ----   ---  ---  -----------          ------------  -----------
2  John    30   M    2018-09-30 00:00:52  BACKFILL      2018-09-30 01:07:39
3  Jennie  25   F    2018-09-30 00:00:52  BACKFILL      2018-09-30 01:07:39
```

## In Amazon Athena

Create a new table in Amazon Athena using the new table schema:

```
    CREATE EXTERNAL TABLE employee_ts_str
    (
      ID string,
      NAME string,
      AGE string,
      AGE string,
      GEN string,
      CREATE_DATE string,
      PROCESS_NAME string,
      UPDATE_DATE string
    )
    STORED AS AVRO
    LOCATION 's3://my-bucket/staging/employee_ts_str'
    TBLPROPERTIES (
    'avro.schema.literal'='
    {
        "type" : "record",
        "name" : "AutoGeneratedSchema",
        "doc" : "Sqoop import of QueryResult",
        "fields" : [ {
          "name" : "ID",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "ID",
          "sqlType" : "2"
        }, {
          "name" : "NAME",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "NAME",
          "sqlType" : "12"
        }, {
          "name" : "AGE",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "AGE",
          "sqlType" : "2"
        }, {
          "name" : "GEN",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "GEN",
          "sqlType" : "12"
        }, {
          "name" : "CREATE_DATE",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "CREATE_DATE",
          "sqlType" : "93"
        }, {
          "name" : "PROCESS_NAME",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "PROCESS_NAME",
          "sqlType" : "12"
        }, {
          "name" : "UPDATE_DATE",
          "type" : [ "null", "string" ],
          "default" : null,
          "columnName" : "UPDATE_DATE",
          "sqlType" : "93"
        } ],
      "tableName" : "QueryResult"
    }
  ');
```

Note that the timestamp columns in the table definition are defined as *string*.

Select the data without using timestamp conversion:

```
select id, name, age, gen, create_date, process_name, update_date
from employee_ts_str limit 2;
```

```
id  name    age gen  create_date          process_name  update_date
--  ----    --- ---  -----------          ------------  -----------
2   John    30  M    2018-09-30 00:00:52  BACKFILL      2018-09-30 01:07:39
3   Jennie  25  F    2018-09-30 00:00:52  BACKFILL      2018-09-30 01:07:39
```

## Avro Files Concatenation

If there are several output files (there were more than one number of mappers) and you want to combine them into one file you can use a concatenation:

```
hadoop jar avro-tools-1.8.1.jar part-m-00000.avro part-m-00001.avro cons_file.avro
```

Files can be local or in S3:

```
hadoop jar avro-tools-1.8.1.jar concat s3://my-bucket/staging/employee/part-m-00000.avro s3://my-bucket/
```

## Summary

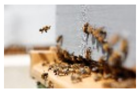
This article explained how to transfer data from a relational database (Oracle) to S3 or HDFS and store it in Avro data files using Apache Sqoop. The article also demonstrated how to work with Avro table schema and how to handle timestamp fields in Avro (to keep them in Unix time (Epoch time) or to convert to *string* data type).

Dmytro Manannykov,

Big Data Architect

Make New Friends and Escape Compliance Hell: Read tl

## Like This Article? Read More From DZone

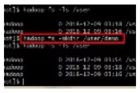


**DZone Article**

**Hive Incremental Update Using Sqoop [Snippet]**

**DZone Article**

**Converting Apache Sqoop Commands Into StreamSets Data Collector Pipelines**



**DZone Article**

**Drilling into Big Data: Data Ingestion**

**Free DZone Refcard**

**Introduction to Text Mining**

Topics: APACHE SQOOP , BIG DATA , BIG DATA SETS , DATA TRANSFER , TUTORIAL

Opinions expressed by DZone contributors are their own.