AvroSerDe

Availability

Earliest version AvroSerde is available

The AvroSerde is available in Hive 0.9.1 and greater.

Overview – Working with Avro from Hive

The AvroSerde allows users to read or write Avro data as Hive tables. The AvroSerde's bullet points:

- Infers the schema of the Hive table from the Avro schema. Starting in Hive 0.14, the Avro schema can be inferred from the Hive table schema.
- Reads all Avro files within a table against a specified schema, taking advantage of Avro's backwards compatibility abilities
- Supports arbitrarily nested schemas.
- Translates all Avro data types into equivalent Hive types. Most types map exactly, but some Avro types don't exist in Hive and are automatically converted by the AvroSerde.
- Understands compressed Avro files.
- Transparently converts the Avro idiom of handling nullable types as Union[T, null] into just T and returns null when appropriate.
- Writes any Hive table to Avro files.
- Has worked reliably against our most convoluted Avro schemas in our ETL process.
- Starting in Hive 0.14, columns can be added to an Avro backed Hive table using the Alter Table statement.

For general information about SerDes, see Hive SerDe in the Developer Guide. Also see SerDe for details about input and output processing.

Requirements

The AvroSerde has been built and tested against Hive 0.9.1 and later, and uses Avro 1.7.5 as of Hive 0.13 and 0.14.

Hive Versions	Avro Version
Hive 0.9.1	Avro 1.5.3
Hive 0.10, 0.11, and 0.12	Avro 1.7.1
Hive 0.13 and 0.14	Avro 1.7.5

Avro to Hive type conversion

While most Avro types convert directly to equivalent Hive types, there are some which do not exist in Hive and are converted to reasonable equivalents. Also, the AvroSerde special cases unions of null and another type, as described below:

Avro type	Becomes Hive type	Note
null	void	
boolean	boolean	

int	int	
long	bigint	
float	float	
double	double	
bytes	binary	Bytes are converted to Array[smallint] prior to Hive 0.12.0.
string	string	
record	struct	
map	map	
list	array	
union	union	Unions of [T, null] transparently convert to nullable T, other types translate directly to Hive's unions of those types. However, unions were introduced in Hive 7 and are not currently able to be used in where/group-by statements. They are essentially look-at-only. Because the AvroSerde transparently converts [T,null], to nullable T, this limitation only applies to unions of multiple types or unions not of a single type and null.
enum	string	Hive has no concept of enums.
fixed	binary	Fixeds are converted to Array[smallint] prior to Hive 0.12.0.

Creating Avro-backed Hive tables

Avro-backed tables can be created in Hive using AvroSerDe.

All Hive versions

To create an Avro-backed table, specify the serde as org.apache.hadoop.hive.serde2.avro.AvroSerDe, specify the inputformat as org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat, and the outputformat as org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat. Also provide a location from which the AvroSerde will pull the most current schema for the table. For example:

```
CREATE TABLE kst
  PARTITIONED BY (ds string)
  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'
  TBLPROPERTIES (
    'avro.schema.url'='http://schema_provider/kst.avsc');
```

In this example we're pulling the source-of-truth reader schema from a webserver. Other options for providing the schema are described below.

Add the Avro files to the database (or create an external table) using standard Hive operations (http://wiki.apache.org/hadoop/Hive/LanguageManual/DML).

This table might result in a description as below:

```
hive> describe kst;
OK
string1 string  from deserializer
string2 string  from deserializer
int1    int     from deserializer
boolean1        boolean from deserializer
long1   bigint  from deserializer
float1  float   from deserializer
double1 double  from deserializer
inner_record1   struct<int_in_inner_record1:int,string_in_inner_record1:string> from deserializer
enum1   string  from deserializer
array1  array<string>   from deserializer
map1    map<string,string>      from deserializer
union1  uniontype<float,boolean,string> from deserializer
fixed1  binary  from deserializer
null1   void    from deserializer
unionnullint    int     from deserializer
bytes1  binary  from deserializer
```

At this point, the Avro-backed table can be worked with in Hive like any other table.

Hive 0.14 and later versions

Starting in Hive 0.14, Avro-backed tables can simply be created by using "STORED AS AVRO" in a DDL statement. AvroSerDe takes care of creating the appropriate Avro schema from the Hive table schema, a big win in terms of Avro usability in Hive.

For example:

```
CREATE TABLE kst (
    string1 string,
    string2 string,
    int1 int,
    boolean1 boolean,
    long1 bigint,
    float1 float,
    double1 double,
    inner_record1 struct<int_in_inner_record1:int,string_in_inner_record1:string>,
    enum1 string,
    array1 array<string>,
    map1 map<string,string>,
    union1 uniontype<float,boolean,string>,
    fixed1 binary,
    null1 void,
    unionnullint int,
    bytes1 binary)
  PARTITIONED BY (ds string)
  STORED AS AVRO;
```

This table might result in a description as below:

```
hive> describe kst;
OK
string1 string  from deserializer
string2 string  from deserializer
int1    int     from deserializer
boolean1        boolean from deserializer
long1   bigint  from deserializer
float1  float   from deserializer
double1 double  from deserializer
inner_record1   struct<int_in_inner_record1:int,string_in_inner_record1:string> from deserializer
enum1   string  from deserializer
array1  array<string>   from deserializer
map1    map<string,string>      from deserializer
union1  uniontype<float,boolean,string> from deserializer
fixed1  binary  from deserializer
null1   void    from deserializer
unionnullint    int     from deserializer
bytes1  binary  from deserializer
```

Writing tables to Avro files

The AvroSerde can serialize any Hive table to Avro files. This makes it effectively an any-Hive-type to Avro converter. In order to write a table to an Avro file, you must first create an appropriate Avro schema (except in Hive 0.14.0 and later, as described below). Create as select type statements are not currently supported.

Types translate as detailed in the table above. For types that do not translate directly, there are a few items to keep in mind:

- **Types that may be null must be defined as a union of that type and Null within Avro.** A null in a field that is not so defined will result in an exception during the save. No changes need be made to the Hive schema to support this, as all fields in Hive can be null.
- Avro Bytes type should be defined in Hive as lists of tiny ints. The AvroSerde will convert these to Bytes during the saving process.
- Avro Fixed type should be defined in Hive as lists of tiny ints. The AvroSerde will convert these to Fixed during the saving process.
- Avro Enum type should be defined in Hive as strings, since Hive doesn't have a concept of enums. Ensure that only valid enum values are present in the table – trying to save a non-defined enum will result in an exception.

Hive is very forgiving about types: it will attempt to store whatever value matches the provided column in the equivalent column position in the new table. No matching is done on column names, for instance. Therefore, it is incumbent on the query writer to make sure the target column types are correct. If they are not, Avro may accept the type or it may throw an exception; this is dependent on the particular combination of types.

Example

Consider the following Hive table, which covers all types of Hive data types, making it a good example:

```
CREATE TABLE test_serializer(string1 STRING,
                             int1 INT,
                             tinyint1 TINYINT,
                             smallint1 SMALLINT,
                             bigint1 BIGINT,
                             boolean1 BOOLEAN,
                             float1 FLOAT,
                             double1 DOUBLE,
                             list1 ARRAY<STRING>,
                             map1 MAP<STRING,INT>,
                             struct1 STRUCT<sint:INT,sboolean:BOOLEAN,sstring:STRING>,
                             union1 uniontype<FLOAT, BOOLEAN, STRING>,
                             enum1 STRING,
                             nullableint INT,
                             bytes1 BINARY,
                             fixed1 BINARY)
 ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' COLLECTION ITEMS TERMINATED BY ':' MAP KEYS TERMINATED BY '#'
LINES TERMINATED BY '\n'
 STORED AS TEXTFILE;
```

If the table were backed by a csv file such as:

why hello there	42	3	100	1412341	true	42.43	85.23423424	alpha:beta:gamma	Earth#42:Control#86:Bob#31	17:true:Abe Linkedin	0:3.141459	BLUE	72	^A^B^C	^A^B^C
another record	98	4	101	9999999	false	99.89	0.00000009	beta	Earth#101	1134:false:wazzup	1:true	RED	NULL	^D^E^F^G	^D^E^F
third record	45	5	102	999999999	true	89.99	0.00000000000009	alpha:gamma	Earth#237:Bob#723	102:false:BNL	2:Time to go home	GREEN	NULL	^H	^G^H^I

then you could write it out to Avro as described below.

All Hive versions

To save this table as an Avro file, create an equivalent Avro schema (the namespace and actual name of the record are not important):

```
{
  "namespace": "com.linkedin.haivvreo",
  "name": "test_serializer",
  "type": "record",
  "fields": [
    { "name":"string1", "type":"string" },
    { "name":"int1", "type":"int" },
    { "name":"tinyint1", "type":"int" },
    { "name":"smallint1", "type":"int" },
    { "name":"bigint1", "type":"long" },
    { "name":"boolean1", "type":"boolean" },
    { "name":"float1", "type":"float" },
    { "name":"double1", "type":"double" },
    { "name":"list1", "type":{"type":"array", "items":"string"} },
    { "name":"map1", "type":{"type":"map", "values":"int"} },
    { "name":"struct1", "type":{"type":"record", "name":"struct1_name", "fields": [
          { "name":"sInt", "type":"int" }, { "name":"sBoolean", "type":"boolean" }, { "name":"sString", "type":"
string" } ] } },
    { "name":"union1", "type":["float", "boolean", "string"] },
    { "name":"enum1", "type":{"type":"enum", "name":"enum1_values", "symbols":["BLUE","RED", "GREEN"]} },
    { "name":"nullableint", "type":["int", "null"] },
    { "name":"bytes1", "type":"bytes" },
    { "name":"fixed1", "type":{"type":"fixed", "name":"threebytes", "size":3} }
  ] }
```

Then you can write it out to Avro with:

```
CREATE TABLE as_avro
  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'
  TBLPROPERTIES (
    'avro.schema.url'='file:///path/to/the/schema/test_serializer.avsc');

INSERT OVERWRITE TABLE as_avro SELECT * FROM test_serializer;
```

Hive 0.14 and later

In Hive versions 0.14 and later, you do not need to create the Avro schema manually. The procedure shown above to save a table as an Avro file reduces to just a DDL statement followed by an insert into the table.

```
CREATE TABLE as_avro(string1 STRING,
                     int1 INT,
                     tinyint1 TINYINT,
                     smallint1 SMALLINT,
                     bigint1 BIGINT,
                     boolean1 BOOLEAN,
                     float1 FLOAT,
                     double1 DOUBLE,
                     list1 ARRAY<STRING>,
                     map1 MAP<STRING,INT>,
                     struct1 STRUCT<sint:INT,sboolean:BOOLEAN,sstring:STRING>,
                     union1 uniontype<FLOAT, BOOLEAN, STRING>,
                     enum1 STRING,
                     nullableint INT,
                     bytes1 BINARY,
                     fixed1 BINARY)
STORED AS AVRO;
INSERT OVERWRITE TABLE as_avro SELECT * FROM test_serializer;
```

Avro file extension

The files that are written by the Hive job are valid Avro files, however, MapReduce doesn't add the standard .avro extension. If you copy these files out, you'll likely want to rename them with .avro.

Specifying the Avro schema for a table

There are three ways to provide the reader schema for an Avro table, all of which involve parameters to the serde. As the schema evolves, you can update these values by updating the parameters in the table.

Use avro.schema.url

Specifies a URL to access the schema from. For http schemas, this works for testing and small-scale clusters, but as the schema will be accessed at least once from each task in the job, this can quickly turn the job into a DDOS attack against the URL provider (a web server, for instance). Use caution when using this parameter for anything other than testing.

The schema can also point to a location on HDFS, for instance: hdfs://your-nn:9000/path/to/avsc/file. The AvroSerde will then read the file from HDFS, which should provide resiliency against many reads at once. Note that the serde will read this file from every mapper, so it's a good idea to turn the replication of the schema file to a high value to provide good locality for the readers. The schema file itself should be relatively small, so this does not add a significant amount of overhead to the process.

Use schema.literal and embed the schema in the create statement

You can embed the schema directly into the create statement. This works if the schema doesn't have any single quotes (or they are appropriately escaped), as Hive uses this to define the parameter value. For instance:

```
CREATE TABLE embedded
  COMMENT "just drop the schema right into the HQL"
  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'
  TBLPROPERTIES (
    'avro.schema.literal'='{
      "namespace": "com.howdy",
      "name": "some_schema",
      "type": "record",
      "fields": [ { "name":"string1","type":"string"}]
    }');
```

Note that the value is enclosed in single quotes and just pasted into the create statement.

Use avro.schema.literal and pass the schema into the script

Hive can do simple variable substitution and you can pass the schema embedded in a variable to the script. Note that to do this, the schema must be completely escaped (carriage returns converted to \n, tabs to \t, quotes escaped, etc). An example:

```
set hiveconf:schema;
DROP TABLE example;
CREATE TABLE example
  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'
  TBLPROPERTIES (
    'avro.schema.literal'='${hiveconf:schema}');
```

To execute this script file, assuming $SCHEMA has been defined to be the escaped schema value:

```
hive --hiveconf schema="${SCHEMA}" -f your_script_file.sql
```

Note that $SCHEMA is interpolated into the quotes to correctly handle spaces within the schema.

Use none to ignore either avro.schema.literal or avro.schema.url

Hive does not provide an easy way to unset or remove a property. If you wish to switch from using URL or schema to the other, set the to-be-ignored value to **none** and the AvroSerde will treat it as if it were not set.

HBase Integration

Hive 0.14.0 onward supports storing and querying Avro objects in HBase columns by making them visible as structs to Hive. This allows Hive to perform ad hoc analysis of HBase data which can be deeply structured. Prior to 0.14.0, the HBase Hive integration only supported querying primitive data types in columns. See Avro Data Stored in HBase Columns for details.

If something goes wrong

Hive tends to swallow exceptions from the AvroSerde that occur before job submission. To force Hive to be more verbose, it can be started with *__hive --hiveconf hive.root.logger=INFO,console__*, which will spit orders of magnitude more information to the console and will likely include any information the AvroSerde is trying to get you about what went wrong. If the AvroSerde encounters an error during MapReduce, the stack trace will be provided in the failed task log, which can be examined from the JobTracker's web interface. The AvroSerde only emits the AvroSerdeException; look for these. Please include these in any bug reports. The most common is expected to be exceptions while attempting to serializing an incompatible type from what Avro is expecting.

FAQ

- Why do I get **error-error-error-error-error-error-error** and a message to check avro.schema.literal and avro.schema.url when describing a table or running a query against a table?

> *The AvroSerde returns this message when it has trouble finding or parsing the schema provided by either the avro.schema.literal or avro.avro.schema.url value. It is unable to be more specific because Hive expects all calls to the serde config methods to be successful, meaning we are unable to return an actual exception. By signaling an error via this message, the table is left in a good state and the incorrect value can be corrected with a call to **alter table T set TBLPROPERTIES**.*