Working with Unstructured and Semi-Structured Data

The sections below describe two SerDes for working with unstructured and semi-structured data: JsonSerDe and RegexSerDe.

JsonSerDe

Hive's JsonSerDe is useful for semi-structured data stored in JSON (JavaScript Object Notation) format. To use the SerDe, the CREATE TABLE statement for the table should include the clause ROW FORMAT SERDE followed by the fully qualified name of the Java class that implements the JsonSerDe, enclosed in quotes.

For example, a couple of records in a JSON file might look like this:

```
{"id":393930, "name":"Aleks Norkov", "city":"Berkeley", "state":"CA"}

{"name":"Christine Goldbaum", "city":"Boulder City", "state":"NV", "id":82800}
```

The CREATE TABLE statement might look like this:

```
CREATE TABLE subscribers

    (id INT, name STRING, city STRING, state STRING)

  ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';
```

The table would include the correct data, even though the records were not organized exactly the same.

RegexSerDe

Hive's RegexSerDe is useful for semi-structured or unstructured data. It identifies the fields within each record based on a *regular expression* (a way to describe patterns in text). For example, using the RegexSerDe, Hive can directly read a log file that lacks consistent delimiters. As with the other SerDes, the CREATE TABLE statement for the table should include the clause ROW FORMAT SERDE followed by the fully qualified name of the Java class that implements the RegexSerDe, enclosed in quotes. To specify the regular expression, use WITH SERDEPROPERTIES("input.regex"="*regular expression*"). (If you're unfamiliar with regular expressions, see this Regular Expressions Tutorial.)

For example, here are two sample records from a log file. The fields are separated by a space, but there are also spaces within the quoted comment string. This makes it difficult to use the usual ROW FORMAT DELIMITED clause, because it would not be able to distinguish the spaces within the quotes from the delimiting spaces.

```
05/23/2016 19:45:19 312-555-7834 CALL_RECEIVED ""

05/23/2016 19:48:37 312-555-7834 COMPLAINT "Item damaged"
```

You could do some processing of the data to convert it, but the RegexSerDe allows you to work directly with the raw data through Hive. (This could be helpful for log files that are constantly being generated, adding to the data you want to use.) To do this, use a CREATE TABLE statement like the following.

```
CREATE TABLE calls (

    event_date STRING, event_time STRING,

    phone_num STRING, event_type STRING, details STRING)

  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
```

```
    WITH SERDEPROPERTIES ("input.regex" =

      "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) \"([^\"]*)\"");
```

Inside the regular expression, the parentheses define capture groups. The value of each field is the text matched by the pattern within each pair of parentheses. The first four fields capture any number of non-space characters, with the fields separated by one space. The last field begins after the literal quote character and captures any number of non-quote characters, then ends before the following quote character. The quote characters must be escaped with backslashes because they are themselves within a quoted string.

The five captured fields become the five columns of the table: event_date, event_time, phone_num, event_type, and details. Although this example uses the data type STRING for all five columns, the RegexSerDe does support other data types.

Note that RegexSerDe is for deserialization (reading), but it doesn't support serialization (writing). You can use LOAD DATA to load an existing file, but INSERT will not work.

Try It!

In the VM, try creating a couple of tables using these SerDes.

First, do the following to create a table using a JSON file for its data.

1. Examine the file in /user/hive/warehouse/subscribers/. This is the same example data used above.

2. Create the subscribers table. In Hive, execute the following statement:

```
  CREATE EXTERNAL TABLE default.subscribers

      (id INT, name STRING, city STRING, state STRING)
```

ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';

3. Execute a SELECT * statement to verify that the table has been created with the correct values in each column. Note that although the two rows gave the column values in different orders, each row has the correct values (the two IDs are 39390 and 82800, for example).

4. *Optional:* You can drop the table if you like. If you used EXTERNAL as noted above, dropping the table will not delete the data, so you can come back and recreate it later, if you like.

Now use the RegExSerDe. The data (see the example below) has fields that are not delimited; instead they use fixed widths.

1030929610759620160829012215Oakland CA94618

Field	Length
cust_id	7
order_id	7
order_dt	8
order_tm	6
city	20

state	2
zip	5

5. *Optional:* The sample data is in /user/hive/warehouse/fixed. (It's just the one row provided above.) You can take a look if you like.

6. Take a look at the regular expression in this CREATE TABLE statement, and see how it will split the row of data into the seven fields. In a regular expression, \d matches any digit, a dot (.) matches any character, and \w matches any word character (letters, numbers, or the underscore character). The first \ in \\d and \\w is to escape the second \ character. This lets Hive know this second \ is part of the regular expression and not the start of an escape sequence representing a special character.

```
CREATE EXTERNAL TABLE fixed
    (cust_id INT, order_id INT, order_dt STRING, order_tm STRING,
    city STRING, state STRING, zip STRING)
  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
  WITH SERDEPROPERTIES ("input.regex" =
    "(\\d{7})(\\d{7})(\\d{8})(\\d{6})(.{20})(\\w{2})(\\d{5})");
```

7. In Hive, execute the statement above.

8. Verify the results using the data source panel or a SELECT * query.

9. *Optional:* You can drop the table if you like. If you used EXTERNAL as noted above, dropping the table will not delete the table, so you can come back and recreate it later, if you like.