Hive has two built-in functions, get\_json\_object and json\_tuple, for dealing with JSON. There are also a couple of JSON SerDe's (Serializer/Deserializers) for Hive

I will document using these **three options** here.

Let's start with a simple JSON document and then move to a complex document with nested subdocuments and arrays of subdocuments.

Here's the first document:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {      "Foo": "ABC",      "Bar": "20090101100000",      "Quux": {          "QuuxId": 1234,          "QuuxName": "Sam"      }  } |

We are going to store this as a Text document, so it is best to have the whole JSON entry on a single line in the text file you point the Hive table to.

Here it is on one line for easy copy and pasting:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1 | {"Foo":"ABC","Bar":"20090101100000","Quux":{"QuuxId":1234,"QuuxName":"Sam"}} |

Let's create a Hive table to reference this. I've put the above document in a file called sample.json:

Use below command to copy Json file from windows folder to the Hadoop sandbox folder.

scp -P 2222 sample.json root@127.0.0.1:/root/shelly/myHive

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3 | CREATE TABLE json\_table ( json string );    LOAD DATA LOCAL INPATH  '/root/shelly/myHive/sample.json' INTO TABLE json\_table; |

Since there are no delimiters, we leave off the ROW FORMAT section of the table DDL

**First Option**: -**Built in function #1: get\_json\_object**

The get\_json\_object takes two arguments: tablename.fieldname and the JSON field to parse, where '$' represents the root of the document.

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1 | select get\_json\_object(json\_table.json, '$') from json\_table; |

Returns the full JSON document.

So do this to query all the fields:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | select get\_json\_object(json\_table.json, '$.Foo') as foo,         get\_json\_object(json\_table.json, '$.Bar') as bar,         get\_json\_object(json\_table.json, '$.Quux.QuuxId') as qid,         get\_json\_object(json\_table.json, '$.Quux.QuuxName') as qname  from json\_table; |

You should get the output:

foo bar qid qname

ABC 20090101100000 1234 Sam

(Note: to get the header fields, enter set hive.cli.print.header=true at the hive prompt or in your $HOME/.hiverc file.)

This works and has a nice JavaScript like "dotted" notation, but notice that you have to parse the same document once for every field you want to pull out of your JSON document, so it is rather inefficient.

The Hive wiki recommends using json\_tuple for this reason.

**Second Option**: -**Built in function #2: json\_tuple**

So let's see what json\_tuple looks like. It has the benefit of being able to pass in multiple fields, but it only works to a single level deep. You also need to use Hive's slightly odd LATERAL VIEW notation:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4 | select v.foo, v.bar, v.quux, v.qid  from json\_table jt       LATERAL VIEW json\_tuple(jt.json, 'Foo', 'Bar', 'Quux', 'Quux.QuuxId') v       as foo, bar, quux, qid; |

This returns:

foo bar quux qid

ABC 20090101100000 {"QuuxId":1234,"QuuxName":"Sam"} NULL

It doesn't know how to look inside the Quux subdocument. And this is where json\_tuple gets clunky fast - you have to create another lateral view for each subdocument you want to descend into:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | select v1.foo, v1.bar, v2.qid, v2.qname  from json\_table jt       LATERAL VIEW json\_tuple(jt.json, 'Foo', 'Bar', 'Quux') v1       as foo, bar, quux       LATERAL VIEW json\_tuple(v1.quux, 'QuuxId', 'QuuxName') v2       as qid, qname; |

This gives us the output we want:

foo bar qid qname

ABC 20090101100000 1234 Sam

With a complicated highly nested JSON doc, json\_tuple is also quite inefficient and clunky as hell. So let's turn to a custom SerDe to solve this problem.

**Third Option**: -**The best option: Hive-JSON SerDe**

A SerDe is a better choice than a json function (UDF) for at least two reasons:

1. it only has to parse each JSON record once
2. you can define the JSON schema in the Hive table schema, making it much easier to issue queries against.

Now let's define the Hive schema that this SerDe expects and load the simple.json doc:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE json\_serde (  Foo string,  Bar string,  Quux struct<QuuxId:int, QuuxName:string>  )  ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';    LOAD DATA LOCAL INPATH '/root/shelly/myHive/sample.json' INTO TABLE json\_serde; |

With the JsonSerDe, you can define subdocuments as maps or structs. I prefer structs, as it allows you to use the convenient dotted-path notation (e.g., Quux.QuuxId) and you can match the case of the fields. With maps, all the keys you pass in have to be lowercase, even if you defined them as upper or mixed case in your JSON.

The query to match the above examples is beautifully simple:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2 | SELECT Foo, Bar, Quux.QuuxId, Quux.QuuxName  FROM json\_serde; |

Result:

foo bar quuxid quuxname

ABC 20090101100000 1234 Sam

And now let's do a more complex JSON document:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | {    "DocId": "ABC",    "User": {      "Id": 1234,      "Username": "sam1234",      "Name": "Sam",      "ShippingAddress": {        "Address1": "123 Main St.",        "Address2": null,        "City": "Durham",        "State": "NC"      },      "Orders": [        {          "ItemId": 6789,          "OrderDate": "11/11/2012"        },        {          "ItemId": 4352,          "OrderDate": "12/12/2012"        }      ]    }  } |

Collapsed version:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1 | {"DocId":"ABC","User":{"Id":1234,"Username":"sam1234","Name":"Sam","ShippingAddress":{"Address1":"123 Main St.","Address2":"","City":"Durham","State":"NC"},"Orders":[{"ItemId":6789,"OrderDate":"11/11/2012"},{"ItemId":4352,"OrderDate":"12/12/2012"}]}} |

Hive Schema:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | CREATE TABLE complex\_json (    DocId string,    User struct<Id:int,                Username:string,                Name: string,                ShippingAddress:struct<Address1:string,                                       Address2:string,                                       City:string,                                       State:string>,                Orders:array<struct<ItemId:int,                                    OrderDate:string>>>  )  ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'; |

Load the data:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2 | LOAD DATA LOCAL INPATH '/tmp/complex.json'  OVERWRITE INTO TABLE complex\_json; |

First let's query something from each document section. Since we know there are two orders in the orders array we can reference them both directly:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT DocId, User.Id, User.ShippingAddress.City as city,         User.Orders[0].ItemId as order0id,         User.Orders[1].ItemId as order1id  FROM complex\_json; |

Result:

docid id city order0id order1id

ABC 1234 Durham 6789 4352

But what if we don't know how many orders there are and we want a list of all a user's order Ids? This will work:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2 | SELECT DocId, User.Id, User.Orders.ItemId  FROM complex\_json; |

Result:

docid id itemid

ABC 1234 [6789,4352]

Oooh, it returns an array of ItemIds. Pretty cool. One of Hive's nice features.

Finally, does the openx JsonSerDe require me to define the whole schema? Or what if I have two JSON docs (say version 1 and version 2) where they differ in some fields? How constraining is this Hive schema definition?

Let's add two more JSON entries to our JSON document - the first has no orders; the second has a new "PostalCode" field in Shipping Address.

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | {      "DocId": "ABC",      "User": {          "Id": 1235,          "Username": "fred1235",          "Name": "Fred",          "ShippingAddress": {              "Address1": "456 Main St.",              "Address2": "",              "City": "Durham",              "State": "NC"          }      }  }    {      "DocId": "ABC",      "User": {          "Id": 1236,          "Username": "larry1234",          "Name": "Larry",          "ShippingAddress": {              "Address1": "789 Main St.",              "Address2": "",              "City": "Durham",              "State": "NC",              "PostalCode": "27713"          },          "Orders": [              {                  "ItemId": 1111,                  "OrderDate": "11/11/2012"              },              {                  "ItemId": 2222,                  "OrderDate": "12/12/2012"              }          ]      }  } |

Collapsed version:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2 | {"DocId":"ABC","User":{"Id":1235,"Username":"fred1235","Name":"Fred","ShippingAddress":{"Address1":"456 Main St.","Address2":"","City":"Durham","State":"NC"}}}  {"DocId":"ABC","User":{"Id":1236,"Username":"larry1234","Name":"Larry","ShippingAddress":{"Address1":"789 Main St.","Address2":"","City":"Durham","State":"NC","PostalCode":"27713"},"Orders":[{"ItemId":1111,"OrderDate":"11/11/2012"},{"ItemId":2222,"OrderDate":"12/12/2012"}]}} |

Add those records to complex.json and reload the data into the complex\_json table.

Now try the query:

[?](http://thornydev.blogspot.com/2013/07/querying-json-records-via-hive.html)

|  |  |
| --- | --- |
| 1  2 | SELECT DocId, User.Id, User.Orders.ItemId  FROM complex\_json; |

It works just fine and gives the result:

docid id itemid

ABC 1234 [6789,4352]

ABC 1235 null

ABC 1236 [1111,2222]

Any field not present will just return null, as Hive normally does even for non-JSON formats.

Note that we cannot query for User.ShippingAddress.PostalCode because we haven't put it on our Hive schema. You would have to revise the schema and then reissue the query.

To store Json file in HDFS use this:

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

STORED AS TEXTFILE

LOCATION '/user/data/';

**To upload the state vector json file**

scp -P 2222 StateVector.query.json [root@127.0.0.1:/root/shelly/myHive](mailto:root@127.0.0.1:/root/shelly/myHive)

CREATE TABLE state\_vector\_serde(BEMSID struct<Query:string,Type:string,Value:string,Label:string>,

DataSecurity struct<Constraint:string,Description:string,Flag:string,Persist:string,Type:string,Mask:string,

Value:struct<DataBlockFilename:struct<Description:string,Constraint:string,Mask:string,Value:string,Flag:string,Persist:string,Type:string>>>)

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

LOAD DATA LOCAL INPATH '/root/shelly/myHive/StateVector.query.json' INTO TABLE state\_vector\_serde;

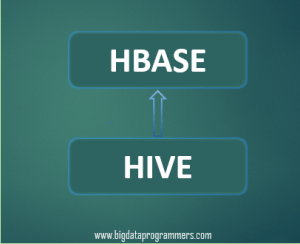
select BEMSID.Query as Query,BEMSID.Type as Type,BEMSID.Value as Value,BEMSID.Label as Label,DataSecurity.Constraint cons,DataSecurity.Description desc,DataSecurity.Flag f,DataSecurity.Persist per,DataSecurity.Type Type,DataSecurity.Mask Mask,DataSecurity.value.DataBlockFilename.Description Description,DataSecurity.value.DataBlockFilename.Value Value from state\_vector\_serde;



JSON file contents below:

{"DataSecurity": {"Description": null, "Constraint": null, "Mask": "6666", "Value": {"DataBlockFilename": {"Description": "File containing data block definition", "Constraint": null, "Mask": "4444", "Value": "DataSecurity.in", "Flag": "D", "Persist": "Plan", "Type": "string"}}, "Flag": "O", "Persist": "archive", "Type": "DataBlock"},"BEMSID": {"Query": true, "Type": "int", "Value": 10310, "Label": "BEMSID"}}

**Now create a Hive table to map a HBase table:** the HBase table will be created in HBase with name state\_vector



CREATE TABLE hbase\_hive\_state\_vector(Query STRING,Type STRING,Value STRING,Label STRING,Constraint STRING,

ds\_Description string,ds\_Flag string,ds\_Persist string,ds\_Type string,ds\_Mask string,dbf\_Description String,dbf\_Constraint string,dbf\_Mask string,dbf\_Value string,dbf\_Flag string,dbf\_Persist string,dbf\_Type string)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,BEMSID:Type,BEMSID:Value,BEMSID:Label,DataSecurity:Constraint,DataSecurity:Description,DataSecurity:Flag,DataSecurity:Persist,DataSecurity:Type,DataSecurity:Mask,DataBlockFilename:Description,DataBlockFilename:Constraint,DataBlockFilename:Mask,DataBlockFilename:Value,DataBlockFilename:Flag,DataBlockFilename:Persist,DataBlockFilename:Type")

TBLPROPERTIES ("hbase.table.name" = "state\_vector");

**Now fill the new Hive table with the data from the already created Hive table:** this will populate the HBAse table too

INSERT INTO TABLE hbase\_hive\_state\_vector select BEMSID.Query,BEMSID.Type,BEMSID.Value,BEMSID.Label,DataSecurity.Constraint,DataSecurity.Description,DataSecurity.Flag ,DataSecurity.Persist,DataSecurity.Type Type,DataSecurity.Mask,DataSecurity.value.DataBlockFilename.Description,DataSecurity.value.DataBlockFilename.Constraint,DataSecurity.value.DataBlockFilename.Mask,DataSecurity.value.DataBlockFilename.Value,DataSecurity.value.DataBlockFilename.Flag,DataSecurity.value.DataBlockFilename.Persist,DataSecurity.value.DataBlockFilename.Type from state\_vector\_serde;

