1. Create the Document “json.txt”

{"Foo": "ABC", "Bar": "20","Quux":{ "QuuxId": 1234,"QuuxName": "Sam"} }

1. Creating the Hive table and accessing the data

hive> create table json1(line string);  
hive> load data local inpath 'json.txt' into table json1;  
hive> select \* from json1;  
 {"Foo": "ABC", "Bar": "20","Quux":{ "QuuxId": 1234,"QuuxName": "Sam"} }

hive> select get\_json\_object(line,'$.Foo') j from json1;  
ABC  
  
hive> select get\_json\_object(line,'$.Quux') from json1;  
{"QuuxId":1234,"QuuxName":"Sam"}  
  
hive> select get\_json\_object(line,'$.Quux.QuuxId') j from json1;  
1234  
  
hive> select get\_json\_object(line,'$.Quux.QuuxName') j from json1;  
Sam

1. Create “simple.json” with below content

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

1. Create the Hive and load the above file.
2. CREATE TABLE json\_table ( json string );
3. LOAD DATA LOCAL INPATH  '/tmp/simple.json' INTO TABLE json\_table;

Note: Since there are no delimiters, Leave the ROW FORMAT section of the table DDL

1. **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.

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

Result : Returns the full JSON document



|  |
| --- |
| 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; |

Result :

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.

1. **Built in function #2: json\_tuple**

json\_tuple  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.

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;

Result:

foo bar quux qid

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

Note: 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:

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;

Result :

foo bar qid qname

ABC 20090101100000 1234 Sam

Note: With a complicated highly nested JSON doc, json\_tuple is also quite inefficient

1. **The best option: rcongiu's 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. The JSON schema can be defined in the Hive table schema, making it much easier to issue queries against.

I reviewed a couple of SerDe's and by far the best one I've found is

[rcongiu's Hive-JSON SerDe](https://github.com/rcongiu/Hive-JSON-Serde).  
To get that SerDe, clone the project from GitHub and run mvn package. It creates a json-serde-1.1.6.jar in the target directory. If you have a place you like to put your jars for runtime referencing move it there.   
Then tell Hive about it with:

ADD JAR /path/to/json-serde-1.1.6.jar;

You can do this either at the hive prompt or put it in your $HOME/.hiverc file.

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

1.

CREATE TABLE json\_serde (

   Foo string,

   Bar string,

   Quux struct<QuuxId:int, QuuxName:string>

)

ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe';

2.

LOAD DATA LOCAL INPATH '/tmp/simple.json' INTO TABLE json\_serde;

With the openx 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:

SELECT Foo, Bar, Quux.QuuxId, Quux.QuuxName

FROM json\_serde;

Result:

foo bar quuxid quuxname

ABC 20090101100000 1234 Sam

1. Complex JSON

{

  "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:

{"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"}]}}

1. Hive Schema

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';

1. Load the data

LOAD DATA LOCAL INPATH '/tmp/complex.json'

OVERWRITE INTO TABLE complex\_json;

1. Querying the JSON

c.1

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

C.2

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:

SELECT DocId, User.Id, User.Orders.ItemId

FROM complex\_json;

Result:

docid id itemid

ABC 1234 [6789,4352]



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.

{

    "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:

{"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"}]}}

d.1 Add those records to complex.json and reload the data into the complex\_json table.

d.2

SELECT DocId, User.Id, User.Orders.ItemId

FROM complex\_json;

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.