11 Exploring Semi-Structured JSON Data

11.1 Lab Introduction

11.1.1 Purpose

As you know, Snowflake is designed to natively store semi-structured data in various formats. The purpose of this lab is for you to learn how to analyze JSON formatted semi-structured data stored in Snowflake.

If you're a data analyst, you'll learn critical skills for analyzing semi-structured data. If you perform a role that ensures data is available for data analysts and end users, you'll gain insight into their specific needs with regards to semi-structured data.

11.1.2 What you'll learn

- How to query a column containing semi-structured data without using the FLATTEN function.
- How to write a query that uses the FLATTEN table function to query semi-structured data.
- How to query a column containing semi-structured data that is flattened by using the FLATTEN function.
- How to leverage the PATH and KEY output values of the FLATTEN function to get the data you want.
- How to leverage DESCRIBE RESULT to determine the data types of a query's columns.

11.1.3 How to complete this lab

To complete this lab, you can key the SQL commands below directly into a worksheet. It is not recommended that you cut and paste from the workbook pdf as that sometimes results in errors.

You can also use the code file for this lab that was provided at the start of the class. You would simply need to open it in TextEdit (Mac) or Notepad (Windows), and then copy and paste the code directly into a worksheet.

11.2 Working with JSON data

In the first half of this lab there are two exercises to introduce you to querying JSON data. In this first exercise you will query the JSON data accessing the top-level key-value pairs.

11.2.1 Scenario

You've been given a table with JSON data that contains a customer ID and the customer's first and last names. You've been asked to produce a data set that includes a row for each ID. The row must display the customer's ID, first, and last names. Your task is to write a query that produces the requested data.

Let's get started!

11.2.2 Create a new folder and call it Semi-Structured Data.

11.2.3 Create a new worksheet inside the folder and call it Working with Semi-Structured Contact Data.

11.2.4 Create the data.

Let's create the data we'll need for this first scenario. Run the SQL statements below:

```
USE ROLE TRAINING_ROLE;

CREATE DATABASE IF NOT EXISTS LEARNER_DB;
CREATE OR REPLACE SCHEMA LEARNER_DB.LEARNER_SCHEMA;

USE WAREHOUSE LEARNER_WH;
USE DATABASE LEARNER_DB;
USE SCHEMA LEARNER_SCHEMA;

CREATE OR REPLACE TABLE persons AS
SELECT
    column1 AS id
  , parse_json(column2) AS c
FROM
  VALUES
   (12712555,
   '{ "name":  { "first": "John", "last": "Smith"}}'),
   (98127771,
   '{ "name":  { "first": "Jane", "last": "Doe"}}') v;
```

Let's look at the data we're loading. The first item is a scalar value, which is the ID. Next, inside the curly braces, we have the JSON key–value pairs. Using the first data value as an example - name is a key and the values are the first + John and last + Smith. We also have nested key–value pairs. First and last are the keys. The customer's first and last names, respectively, are the values.

Now let's begin writing our queries. Let's start with the following SQL statement (you don't need to run it):

```
 SELECT
         *
 FROM
         persons p;
```

In this statement we have the numeric ID in the first column titled "ID". In the column titled "C" we have the key-value pairs for name, both first and last. The data is still in JSON format with quotation marks and curly braces.

Let's parse the JSON name data values and cast them as varchar so that they're easy to read.

```
SELECT
    p.ID
   ,p.c:name.first::varchar AS first_name
   ,p.c:name.last::varchar AS last_name
 FROM
    persons p;
```


To access the data from the Persons table we specify its access path and the format for the result. For the ID – no formatting is required. For the first and last names we specify the path to this data embedded in a JSON structure and the data type. The syntax is:

- p - the table alias
- c - the column alias for the VARIANT column
- :name.first - the first JSON key-value pair for which we need the value
- :name.last - the second JSON key-value pair for which we need the value
- ::varchar - casts the result as varchar

NOTE: if we do not CAST the result from a VARIANT column the returned value will be of data type variant.

We made no change to the result format of the ID. We did change the result format of the name data. In addition to being tidy and easy to read, with our result data in tabular format we can now use it like any other structured data. For example, it could be inserted into a table, or joined with other structured data.

11.3 Using FLATTEN

In the second exercise you will query JSON data stored in an array. We're going to expand our JSON data set to introduce you to the FLATTEN function. After that we offer an exercise for you to try that involves a Snowbear Air-based scenario using more complex data.

11.3.1 Scenario

The JSON data has been expanded to include contact details in addition to the customer ID, first and last names. You've now been asked to produce a data set that includes a single row for each contact. The row must display the ID, first and last names, phone number, and email address. Your task is to explode the data and write a query that produces the requested result data.

11.3.2 Create the data

Let's create the data we'll need for this second exercise. Run the SQL statement below:

```
CREATE OR REPLACE TABLE persons AS
SELECT
    column1 AS id
  , parse_json(column2) AS c
FROM
  VALUES
   (12712555,
   '{ "name":  { "first": "John", "last": "Smith"},
     "contact": [
     { "business":[
       { "type": "phone", "content":"555-1234" },
       { "type": "email", "content":"j.smith@company.com" } ] } ] }'),
   (98127771,
```



```
  '{ "name":  { "first": "Jane", "last": "Doe"},
    "contact": [
    { "business":[
      { "type": "phone", "content":"555-1236" },
      { "type": "email", "content":"j.doe@company.com" } ] } ] }') v;
```

Let's look at the JSON data using the following SQL statement:

```
SELECT
       *
FROM
       persons p;
```

Just as with our initial exercise, the first column value is the ID. But now we have two paths to follow in this JSON data column "C": name and contact. We know that name is a JSON key-value pair with name as the key. Contact is a key, and an array named business is nested within contact. We know this because of the square brackets used to indicate arrays. Business contains a set of key-value pairs.

To get the data we need from the arrays, we're going to use the FLATTEN table function. Take a moment to review the definition below.

FLATTEN
FLATTEN is a table function that explodes the contents of a variant column (for purposes of this lab, a column with semi-structured data) and produces a relational representation that correlates to the table from which it was derived, and which precedes it in the FROM clause.
ARGUMENTS:

1. INPUT: A reference to a column containing semi-structured data (required).
2. PATH: A path defined by a key from one of the top-level key-value pairs in the semi-structured data (optional).

OUTPUT

1. EXPLODED DATA: Semi structured data presented in a tabular structure and in variant format.
2. PATH: A path to the element within the semi-structured data that needs to be flattened.
3. KEY: The key of a key-value pair for an exploded value.

Now that we have a high-level understanding of how the FLATTEN function works, let's use it in our query. We'll input the column with the JSON and indicate the path is contact. We'll provide column p.c for the required argument INPUT as well as the value 'contact' for the optional argument PATH:

```
SELECT
     *
FROM
     persons p
   , LATERAL FLATTEN(INPUT => p.c, PATH =>'contact') f;
```

There are numerous fields that are part of the flatten output by default: SEQ, KEY, PATH, INDEX, VALUE and THIS. For this lab we will concern ourselves only with PATH and KEY, which will be discussed in greater detail below.

Now let's fetch the contact's name from the JSON:

```
SELECT
     id AS "ID"
   , n.value AS name_value

FROM
     persons p
   , LATERAL FLATTEN (INPUT => p.c, PATH => 'name') n;
```

Here we provided two parameters - the column with the JSON and the path that we wanted to follow, which is the name path. Note that there are two key-value pairs (first and last). Flattening fetches both values from the two pairs and presents them as two rows. We have another way to fetch the first and last name so they're on the first row. We'll do that shortly.

Note the quotes around the first and last names. Run the SQL statement below to find out why.

```
DESCRIBE RESULT LAST_QUERY_ID();
```

Note that the NAME_VALUE column is a VARIANT. It would be better to have it in VARCHAR format so we can filter and sort on that column and get predictable results. Let's fix that now:

```
SELECT
     id AS "ID"
   , n.value::varchar AS name_value

FROM
     persons p
   , LATERAL FLATTEN (INPUT => p.c, PATH => 'name') n;
```

Now we've cast the value as varchar and it looks correct. Run the DESCRIBE RESULT statement again to check the data type.

```
DESCRIBE RESULT LAST_QUERY_ID();
```

You can see the column name_value is in varchar format. Now let's add the contact details into our query:

```
SELECT
     id AS "ID"
   , n.value::varchar AS name_value
   , f.value AS contact_value
FROM
     persons p
   , LATERAL FLATTEN (INPUT => p.c, PATH => 'name') n
   , LATERAL FLATTEN (INPUT => p.c, PATH => 'contact') f;
```

The contact column values are in JSON format. We executed a new lateral flatten and provided it two values: the column with the JSON we wanted to flatten, and the path within the JSON we wanted to drill down into (contact). This is because we want the CONTENT field which has the data we need.

Now run the following to fetch the CONTENT column:

```
SELECT
     id AS "ID"
```



```
    , n.value::varchar AS name_value
    , f1.value:content::varchar AS "Content"
    , f.value AS contact_value

FROM
      persons p
    , LATERAL FLATTEN (INPUT => p.c, PATH => 'name') n
    , LATERAL FLATTEN (INPUT => p.c, PATH =>'contact') f
    , LATERAL FLATTEN (INPUT => f.value:business) f1;
```

Here we accessed the content column and cast it as varchar. We accomplished this by adding another flatten clause that accesses the nested array business within contact. We can see that we have pretty much extracted the content we want. The problem is that the values are repeating based on how the JSON was structured. Let's fix this problem. Run the SQL statement below:

```
SELECT
      p.ID
    , p.c:name.first::varchar AS first_name
    , p.c:name.last::varchar AS last_name
    , f.value AS contact_value

FROM
      persons p
    , LATERAL FLATTEN (INPUT => p.c, PATH =>'contact') n
    , LATERAL FLATTEN (INPUT => n.value:business) f;
```

First we removed the last flatten for now so we can focus on the name. Next, we extracted the first and last name. We started by referencing the column, p.c. Then, we put a colon, which indicates we're going to drill down into the key-value pair. Then for the first and last names, we referenced the path and key, name.first and name.last respectively. Finally, we cast the value to varchar. This put the first and last names together on a single line. The syntax can be something like this:

table.column:key1.key2.keyN

Now we'll extract the phone number and email from the JSON:

```
SELECT
      p.ID
    , p.c:name.first::varchar AS first_name
    , p.c:name.last::varchar AS last_name
    , f.value:content::varchar AS content
    , f.path
FROM
      persons p
    , LATERAL FLATTEN (INPUT => p.c, PATH =>'contact') n
    , LATERAL FLATTEN (INPUT => n.value:business) f;
```

Notice that we still have two rows for John Smith and two rows for Jane Doe. However we've managed to extract the phone number and email address from the JSON. Focusing on the first row of each person, we see the content column contains the phone number while the second row for each person contains the email address. So now the content is in relational format. The problem is that we have repeating rows, which is not what we wanted. But from the path column we added, we can see that the paths are from a zero-based array. The path with the

phone number is [0] and the path with the email address is [1]. Let's flatten n.value:business twice and apply a path to each. Run the following statement:

```
SELECT
      p.ID
    , p.c:name.first::varchar AS first_name
    , p.c:name.last::varchar AS last_name
    , f.value::varchar as phone_number
    , f1.value::varchar as email_address
    , f.key AS PATH_0_KEY
    , f1.key AS PATH_1_KEY

FROM
      persons p
    , LATERAL FLATTEN (INPUT => p.c, path =>'contact') n
    , LATERAL FLATTEN (INPUT => n.value:business, PATH => '[0]') f
    , LATERAL FLATTEN (INPUT => n.value:business, PATH => '[1]') f1;
```

So now we've put the phone number and the email address on the same lines but there are repeating rows that we need to deal with. We've added in the KEY output from the flatten clauses for paths [0] and [1]. When we look at the rows we can see that we need only the rows where both keys state 'content'. Let's apply that filter now:

```
SELECT
      p.ID
    , p.c:name.first::varchar AS first_name
    , p.c:name.last::varchar AS last_name
    , f.value::varchar AS phone_number
    , f1.value::varchar AS email_address
FROM
      persons p
    , LATERAL FLATTEN (INPUT => p.c, PATH =>'contact') n
    , LATERAL FLATTEN (INPUT => n.value:business, PATH => '[0]') f
    , LATERAL FLATTEN (INPUT => n.value:business, PATH => '[1]') f1
WHERE
      f.key = 'content'
      AND
      f1.key = 'content';
```

Now we have exactly the content we wanted. Run the statement below to see how you can get the same result but with a join:

```
SELECT
      p.ID
    , p.c:name.first::varchar AS first_name
    , p.c:name.last::varchar AS last_name
    , f.value::varchar as phone_number
    , f1.value::varchar as email_address

FROM
      persons p INNER JOIN persons p1 ON p.ID = p1.ID
    , LATERAL FLATTEN (INPUT => p.c, PATH =>'contact') n
    , LATERAL FLATTEN (INPUT => n.value:business, PATH => '[0]') f
    , LATERAL FLATTEN (INPUT => p1.c, PATH =>'contact') n1
    , LATERAL FLATTEN (INPUT => n1.value:business, PATH => '[1]') f1
```



```
WHERE
      f.key = 'content'
      AND
      f1.key = 'content';
```

11.3.3 Run the statements below to drop your database and schema:

```
DROP DATABASE LEARNER_DB;
```

11.4 Working with Weather Data

Ok, now that we understand how to use the FLATTEN function, let's apply what we've learned to some real weather data.

11.4.1 Scenario

As you know, Snowbear Air does more than just sell cool apparel branded with its logo. It actually flies customers to fun destinations all over the world! Recently Snowbear Air has decided to analyze weather conditions at airports around the world so it can better understand those trends and how they may impact flight arrival times.

You've been provided with a set of weather quality control check data in JSON format. Your task is to calculate the average, max and min air temperature in Celsius and Fahrenheit for each weather station. You've also been asked to limit the results to records for an 8-day period starting on 2019-08-14 and ending on 2019-08-21, and to air temperature values that have passed all quality control checks.

Let's get started!

11.4.2 Review the Weather Data

Much of this lab will involve extracting and examining the weather data. Since it is in semi-structured format, it will be important to understand how it is structured, how it is stored, and the specific syntax you will need to execute SELECT statements against it.

11.4.3 Create a new worksheet inside folder LAB 04 - Semi-Structured Weather Data and call it Working with Semi-Structured Weather Data.

11.4.4 Setting the context

For this portion of the lab you're going to use the database TRAINING_DB and the schema WEATHER.

Set your context using the code shown below:


```
USE ROLE TRAINING_ROLE;
USE WAREHOUSE LEARNER_WH;
USE DATABASE TRAINING_DB;
USE SCHEMA TRAINING_DB.WEATHER;
```

11.4.5 Use the DESCRIBE TABLE command to describe the ISD_2019_DAILY table:

The table that contains the data we're interested in is called isd_2019_daily table. Let's examine its structure now. Run the following query to see what fields it contains:

```
DESCRIBE TABLE isd_2019_daily;
```

Notice that this table contains just two columns: V (variant) and T (date).

11.4.6 Select all columns from the ISD_2019_DAILY table, limit the results to 20 rows, and explore the data and its content:

```
SELECT
      *
FROM
      weather.isd_2019_daily w

LIMIT 20;
```

11.4.7 Click on the first cell of JSON data in column V.

Once you click on the first cell of JSON data in column V, to the right of the results pane you should see the JSON data displayed in a small side pane. Click "TEXT" to view it in a formatted way. What you see should then look like this stub of JSON data:

```
{
  "data": {
    "observations": [
      {
        "air": {
          "dew-point": 26.8,
          "dew-point-quality-code": "1",
          "temp": 29,
          "temp-quality-code": "1"
        },
        "atmospheric": {
          "pressure": 10086,
          "pressure-quality-code": "1"
        },
        "dt": "2019-08-01T00:00:00",
        "sky": {
          "ceiling": 22000,
          "ceiling-quality-code": "1"
```



```
        },
        "visibility": {
          "distance": 999999,
          "distance-quality-code": "9"
        },
        "wind": {
          "direction-angle": 200,
          "direction-quality-code": "1",
          "speed-quality-code": "1",
          "speed-rate": 10
        }
      },
      {
        "air": {
          "dew-point": 25.1,
          "dew-point-quality-code": "1",
          "temp": 32.3,
          "temp-quality-code": "1"
        },
        "atmospheric": {
          "pressure": 10083,
          "pressure-quality-code": "1"
        },
        "dt": "2019-08-01T03:00:00",
        "sky": {
          "ceiling": 99999,
          "ceiling-quality-code": "9"
        },
        "visibility": {
          "distance": 7000,
          "distance-quality-code": "1"
        },
        "wind": {
          "direction-angle": 160,
          "direction-quality-code": "1",
          "speed-quality-code": "1",
          "speed-rate": 30
        }
      },...SOME DATA REMOVED FOR BREVITY

      }
    ]
  },
  "station": {
    "USAF": "547250",
    "WBAN": 99999,
    "coord": {
      "lat": 37.5,
      "lon": 117.533
    },
    "country": "CH",
    "elev": 12,
    "id": "54725099999",
    "name": "HUIMIN"
  }
}
```


11.4.8 Figure out your strategy

Note that the top-level paths are data and station. At the next level down but on the same level are the key-value pairs country, elev, id and name.

The array called observations contains repeating sets of key-value pairs (air, atmospheric, dt, visibility, wind, etc.) each of which is a key for a value that consists of a nested set of key-value pairs (except for dt, which is a key for a simple key-value pair).

Note that temp and temp-quality-code (the field that indicates if the air temperature values for a specific check passed all quality control checks) are located under data, observations, air. This means that we will have to flatten observations to get to that data.

The other pieces of data we need are station, country, date. We can get date straight from the structured column T. We should then be able to use our normal semi-structured querying syntax to get to country and station. We can then calculate the Fahrenheit measurements from the Celsius measurements.

Let's get our data!

11.4.9 Fetch country, date, station and filter for rows between '2019-08-14' AND '2019-08-21'

Run the following SQL command:

```
 SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , observations.value
FROM
          weather.isd_2019_daily weather
        , LATERAL FLATTEN(input => v:data.observations) observations
WHERE
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'

ORDER BY
        DATE;
```

Note that we flattened the observations array in order to get to its nested data. If you click on the value field of the first row in the results pane, in the side pane to the right you can see the air measurements for that row's country-date-station combination.

Now let's fetch the air temperature:

```
SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , observations.value:air.temp
        , observations.value
FROM
          weather.isd_2019_daily weather
        , LATERAL FLATTEN(input => v:data.observations) observations
```



```
WHERE
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'

ORDER BY
        DATE;
```

Note that we now have the air temperature in our query.

11.4.10 Check the air temp column's data type:

Run the following statement to check the data type:

```
 DESCRIBE RESULT LAST_QUERY_ID();
```

As we can see it's a variant. Let's cast it to a number:

11.4.11 Cast air.temp to NUMBER(38,1)

Run the following SQL commands and verify that air.temp is now cast as a number.

```
 SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , observations.value:air.temp::NUMBER(38,1) AS temp_c
FROM
          weather.isd_2019_daily weather
        , LATERAL FLATTEN(input => v:data.observations) observations
WHERE
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'

ORDER BY
        DATE;

 DESCRIBE RESULT LAST_QUERY_ID();
```

You should see that air.temp is now a number and has been aliased as temp_c.

11.4.12 Fetch the average temperature in Celsius

Now let's apply the AVG function to our temperature field and add a GROUP BY clause:

```
SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , AVG(observations.value:air.temp)::NUMBER(38,1) as avg_temp_c
FROM
          weather.isd_2019_daily weather
        , LATERAL FLATTEN(input => v:data.observations) observations
WHERE
```



```
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'

GROUP BY

        country, date, station

ORDER BY
        DATE;
```

11.4.13 Add columns for the MIN and MAX temperatures in Celsius

Note below that we only had to add the same field two more times but apply either a MIN or MAX to each:

```
SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , AVG(observations.value:air.temp)::NUMBER(38,1) as avg_temp_c
        , MIN(observations.value:air.temp) as min_temp_c
        , MAX(observations.value:air.temp) as max_temp_c
FROM
          weather.isd_2019_daily weather
        , LATERAL FLATTEN(input => v:data.observations) observations
WHERE
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'

GROUP BY

        country, date, station

ORDER BY
        DATE;
```

11.4.14 Check the MIN and MAX temp field for unusually large or high numbers

Check the MIN and MAX temp field for unusually large or high numbers and note that there is occasionally a 999.9 value in these columns. This is because in some instances the check wasn't performed, or something went wrong during the check so the value wasn't recorded.

Let's add a filter in our WHERE clause that filters only for rows where all temperature quality checks were successfully completed and recorded. Run the code below to add the filter.

```
SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , AVG(observations.value:air.temp)::NUMBER(38,1) as avg_temp_c
        , MIN(observations.value:air.temp) as min_temp_c
        , MAX(observations.value:air.temp) as max_temp_c
FROM
          weather.isd_2019_daily weather
```



```
        , LATERAL FLATTEN(input => v:data.observations) observations
WHERE
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'
        AND
        observations.value:air."temp-quality-code" = '1'

GROUP BY

        country, date, station

ORDER BY
        DATE;
```

11.4.15 Add the Fahrenheit values

We basically have the query written. Now all we have to do is convert Celsius to Fahrenheit and apply the AVG, MIN and MAX functions to the results of the conversion:

```
SELECT
          weather.v:station.country::VARCHAR AS country
        , weather.t as date
        , weather.v:station.name::VARCHAR AS station
        , AVG(observations.value:air.temp)::NUMBER(38,1) as avg_temp_c
        , MIN(observations.value:air.temp) as min_temp_c
        , MAX(observations.value:air.temp) as max_temp_c
        , (AVG(observations.value:air.temp) * 9/5 + 32)::NUMBER(38,1) as avg_temp_f
        , (MIN(observations.value:air.temp) * 9/5 + 32) as min_temp_f
        , (MAX(observations.value:air.temp) * 9/5 + 32) as max_temp_f
FROM
          weather.isd_2019_daily weather
        , LATERAL FLATTEN(input => v:data.observations) observations
WHERE
        weather.t BETWEEN '2019-08-14' AND '2019-08-21'
        AND
        observations.value:air."temp-quality-code" = '1'

GROUP BY

        country, date, station

ORDER BY
        DATE;
```

You have now completed the query. Great job!

11.5 Key takeaways

- You can use use the table function FLATTEN to access data within arrays in semi-structured data.
- When you're not working with an array, you can use this syntax to fetch the value of a key-value pair without having to flatten any data: table.column:key1.key2. . . . keyN.

- When you have JSON data that you need to flatten and there is more than one top-level path, you can use the PATH argument of the FLATTEN command to determine which path to navigate.
- You can use DESCRIBE RESULT to determine the data types of an executed query's output columns.
- You can leverage the PATH and KEY output values of the FLATTEN function to get the data you want.

