**Working with Semi-structured server data**

### **What you will learn**

* Upload sample JSON data from a public S3 bucket into a column of the variant type in a Snowflake table.
* Test simple queries for JSON data in the table.
* Explore the FLATTEN function to flatten JSON data into a relational representation and save it in another table.
* Explore ways to ensure uniqueness as you insert rows in the flattened version of the data.

### **About the sample data file**

For this tutorial, you use the following sample application events JSON data provided in a public S3 bucket.

**{**

"device\_type"**:** "server"**,**

"events"**:** **[**

**{**

"f": 83**,**

"rv"**:** "15219.64,783.63,48674.48,84679.52,27499.78,2178.83,0.42,74900.19"**,**

"t": 1437560931139**,**

"v"**:** **{**

"ACHZ": 42869**,**

"ACV": 709489**,**

"DCA": 232**,**

"DCV": 62287**,**

"ENJR": 2599**,**

"ERRS": 205**,**

"MXEC": 487**,**

"TMPI": 9

**},**

"vd": 54**,**

"z": 1437644222811

**},**

**{**

"f": 1000083**,**

"rv"**:** "8070.52,54470.71,85331.27,9.10,70825.85,65191.82,46564.53,29422.22"**,**

"t": 1437036965027**,**

"v"**:** **{**

"ACHZ": 6953**,**

"ACV": 346795**,**

"DCA": 250**,**

"DCV": 46066**,**

"ENJR": 9033**,**

"ERRS": 615**,**

"MXEC": 0**,**

"TMPI": 112

**},**

"vd": 626**,**

"z": 1437660796958

**}**

**],**

"version": 2**.**6

**}**

The data represents sample events that applications upload to S3. A variety of devices and applications, such as servers, cell phones, and browsers publish events. In a common data collection scenario, a scalable web endpoint collects POSTed data from different sources and writes them to a queuing system. An ingest service/utility then writes the data to a S3 bucket, from which you can load the data into Snowflake.

The sample data illustrates the following concepts :

* Applications can choose to group events in batches. A batch is a container that holds header information common to all of the events in the batch. For example, the preceding JSON is a batch of two events with common header information: device\_type and version that generated these events.
* Amazon S3 supports using folders concept to organize a bucket. Applications can leverage this feature to partition event data. Partitioning schemes typically identify details, such as application or location that generated the event, along with an event date when it was written to S3. Such a partitioning scheme enables you to copy any fraction of the partitioned data to Snowflake with a single COPY command. For example, you can copy event data by the hour, data, month, or year when you initially populate tables.  
  For example:  
  s3://*bucket\_name*/*application\_a*/2016/07/01/11/  
  s3://*bucket\_name*/*application\_b*/*location\_c*/2016/07/01/14/  
  Note the application\_a, application\_b, location\_c, etc. identify details for the source of all data in the path. The data can be organized by the date when it was written. An optional 24-hour directory reduces the amount of data in each directory.  
  **Note**  
  S3 transmits a directory list with each COPY statement used by Snowflake, so reducing the number of files in each directory improves the performance of your COPY statements. You may even consider creating 10-15 minute increment folders in each hour.  
  The sample data provided in the S3 bucket uses a similar partitioning scheme. In a COPY command you will specify a specific folder path to copy events data.

### **Creating the database, table, warehouse, and external stage**

Execute the following statements to create a database, a table, a virtual warehouse, and an external stage needed for this tutorial. After you complete the tutorial, you can drop these objects.

**CREATE** **OR** **REPLACE** **DATABASE** mydatabase**;**

**USE** **SCHEMA** mydatabase**.public;**

**CREATE** **OR** **REPLACE** **TABLE** raw\_source **(**

SRC **VARIANT);**

**CREATE** **OR** **REPLACE** **WAREHOUSE** mywarehouse **WITH**

**WAREHOUSE\_SIZE=**'X-SMALL'

**AUTO\_SUSPEND** **=** 120

**AUTO\_RESUME** **=** **TRUE**

**INITIALLY\_SUSPENDED=TRUE;**

**USE** **WAREHOUSE** mywarehouse**;**

**CREATE** **OR** **REPLACE** **STAGE** my\_stage

**URL** **=** 's3://snowflake-docs/tutorials/json'**;**

Note the following:

* The CREATE DATABASE statement creates a database. The database automatically includes a schema named ‘public’.
* The USE SCHEMA statement specifies an active database and schema for the current user session. Specifying a database now enables you to perform your work in this database without having to provide the name each time it is requested.
* The CREATE TABLE statement creates a target table for JSON data.
* The CREATE WAREHOUSE statement creates an initially suspended warehouse. The statement also sets AUTO\_RESUME = true, which starts the warehouse automatically when you execute SQL statements that require compute resources. The USE WAREHOUSE statement specifies the warehouse you created as the active warehouse for the current user session.
* The CREATE STAGE statement creates an external stage that points to the S3 bucket containing the sample file for this tutorial.

## **Copy data into the target table**

Execute [COPY INTO <table>](https://docs.snowflake.com/en/sql-reference/sql/copy-into-table) to load your staged data into the target RAW\_SOURCE table.

**COPY** **INTO** raw\_source

**FROM** **@**my\_stage**/server/**2**.**6**/**2016**/**07**/**15**/**15

**FILE\_FORMAT** **=** **(TYPE** **=** **JSON);**

The command copies all new data from the specified path on the external stage to the target RAW\_SOURCE table. In this example, the specified path targets data written on the 15th hour (3 PM) of July 15th, 2016. Note that Snowflake checks each file’s S3 ETag value to ensure it is copied only once.

Execute a SELECT query to verify the data is copied successfully.

**SELECT** **\*** **FROM** raw\_source**;**

In this sample JSON data, there are two events. The device\_type, and version key values identify a data source and version for events from a specific device.

## **Query data**

In this section, you explore SELECT statements to query the JSON data.

Retrieve device\_type.

**SELECT** src:device\_type

**FROM** raw\_source**;**

1. The query uses the src:device\_type notation to specify the column name and the JSON element name to retrieve. This notation is similar to the familiar SQL table.column notation. Snowflake allows you to specify a sub-column within a parent column, which Snowflake dynamically derives from the schema definition embedded in the JSON data. For more information, refer to [Querying Semi-structured Data](https://docs.snowflake.com/en/user-guide/querying-semistructured).  
   **Note**  
   The column name is case-insensitive, however JSON element names are case-sensitive.

Retrieve the device\_type value without the quotes.  
The preceding query returns the JSON data value in quote. You can remove the quotes by casting the data to a specific data type, in this example a string.  
This query also optionally assigns a name to the column using an alias.  
**SELECT** src:device\_type**::string** **AS** device\_type

**FROM** raw\_source**;**

The query retuns the following result:  
**+**-------------+

| DEVICE\_TYPE |

|-------------|

| server |

**+**-------------+

1. Retrieve repeating f keys nested within the array event objects.  
   The sample JSON data includes events array. Each event object in the array has the f field as shown.

**{**

"device\_type"**:** "server"**,**

"events"**:** **[**

**{**

"f": 83**,**

**..**

**}**

**{**

"f": 1000083**,**

**..**

**}**

**]}**

To retrieve these nested keys, you can use the [FLATTEN](https://docs.snowflake.com/en/sql-reference/functions/flatten) function. The function flattens the events into separate rows.  
**SELECT**

**value**:f**::number**

**FROM**

raw\_source

**,** **LATERAL** FLATTEN**(** **INPUT** **=>** SRC:events **);**

The query returns the following result:  
**+**-----------------+

| VALUE:F::NUMBER |

|-----------------|

| 83 |

| 1000083 |

**+**-----------------+

1. Note the value is one of the columns that FLATTEN function returns. The next step provides more details about using the FLATTEN function.

## **Flatten data**

[FLATTEN](https://docs.snowflake.com/en/sql-reference/functions/flatten) is a table function that produces a lateral view of a VARIANT, OBJECT, or ARRAY column. In this step, you use this function to explore different levels of flattening.

### **Flatten array objects in a variant column**

You can flatten the event objects in the events array into separate rows using the FLATTEN function. The function output includes a VALUE column that stores these individual events.

You can then use the LATERAL modifier to join the FLATTEN function output with any information outside of the object — in this example, the device\_type and version.

1. Query the data for each event:

**SELECT** src:device\_type**::string,**

src:version**::String,**

**VALUE**

**FROM**

raw\_source**,**

**LATERAL** FLATTEN**(** **INPUT** **=>** SRC:events **);**

1. Use a CREATE TABLE AS SELECT statement to store the preceding query result in a table:

**CREATE** **OR** **REPLACE** **TABLE** flattened\_source **AS**

**SELECT**

src:device\_type**::string** **AS** device\_type**,**

src:version**::string** **AS** **version,**

**VALUE** **AS** src

**FROM**

raw\_source**,**

**LATERAL** FLATTEN**(** **INPUT** **=>** SRC:events **);**

1. Query the resulting table.

**SELECT** **\*** **FROM** flattened\_source**;**

### 

### 

### **Flatten object keys in separate columns**

In the preceding example, you flattened the event objects in the events array into separate rows. The resulting flattened\_source table retained the event structure in the src column of the VARIANT type.

One benefit of retaining the event objects in the src column of the VARIANT type is that when event format changes, you don’t have to recreate and repopulate such tables. But you also have the option to copy individual keys in the event object into separate typed columns as shown in the following query.

The following CREATE TABLE AS SELECT statement creates a new table named events with the event object keys stored in separate columns. Each value is cast to a data type that is appropriate for the value, using a double-colon (::) followed by the type. If you omit the casting, the column assumes the VARIANT data type, which can hold any value:

**create** **or** **replace** **table** events **as**

**select**

src:device\_type**::string** **as** device\_type

**,** src:version**::string** **as** **version**

**,** **value**:f**::number** **as** f

**,** **value**:rv**::variant** **as** rv

**,** **value**:t**::number** **as** **t**

**,** **value**:v**.**ACHZ**::number** **as** achz

**,** **value**:v**.**ACV**::number** **as** acv

**,** **value**:v**.**DCA**::number** **as** dca

**,** **value**:v**.**DCV**::number** **as** dcv

**,** **value**:v**.**ENJR**::number** **as** enjr

**,** **value**:v**.**ERRS**::number** **as** errs

**,** **value**:v**.**MXEC**::number** **as** mxec

**,** **value**:v**.**TMPI**::number** **as** tmpi

**,** **value**:vd**::number** **as** vd

**,** **value**:z**::number** **as** z

**from**

raw\_source

**,** **lateral** flatten **(** **input** **=>** SRC:events **);**

The statement flattens the nested data in the EVENTS.SRC:V key, adding a separate column for each value. The statement outputs a row for each key/value pair.

## **Update data**

So far in this tutorial, you did the following:

* Copied sample JSON event data from an S3 bucket into the RAW\_SOURCE table and explored simple queries.
* You also explored the FLATTEN function to flatten the JSON data and obtain a relational representation of the data. For example, you extracted event keys and stored the keys in separate columns in another EVENTS table.

At the beginning, the tutorial explains the application scenario where multiple sources generate events and a web endpoint saves it to your S3 bucket. As new events are added to the S3 bucket, you might use a script to continuously copy new data into the RAW\_SOURCE table. But how do insert only new event data into the EVENTS table.

There are numerous ways to maintain data consistency. This section explains two options.

### **Use primary key columns for comparison**

In this section you add a primary key to the EVENTS table. The primary key then guarantees uniqueness.

1. Examine your JSON data for any values that are naturally unique and would be good candidates for a primary key. For example, assume that the combination of src:device\_type and value:rv can be a primary key. These two JSON keys correspond to the DEVICE\_TYPE and RV columns in the EVENTS table.  
   **Note**  
   Snowflake does not enforce the primary key constraint. Rather, the constraint serves as metadata that identifies the natural key in the Information Schema.
2. Add the primary key constraint to the EVENTS table:  
   **ALTER** **TABLE** events **ADD** **CONSTRAINT** pk\_DeviceType **PRIMARY** **KEY** **(**device\_type**,** rv**);**
3. Insert a new JSON event record into the RAW\_SOURCE table:

insert into raw\_source

select

PARSE\_JSON ('{

"device\_type": "cell\_phone",

"events": [

{

"f": 79,

"rv": "786954.67,492.68,3577.48,40.11,343.00,345.8,0.22,8765.22",

"t": 5769784730576,

"v": {

"ACHZ": 75846,

"ACV": 098355,

"DCA": 789,

"DCV": 62287,

"ENJR": 2234,

"ERRS": 578,

"MXEC": 999,

"TMPI": 9

},

"vd": 54,

"z": 1437644222811

}

],

"version": 3.2

}');

1. Insert the new record that you added to the RAW\_SOURCE table into the EVENTS table based on a comparison of the primary key values:

1. Querying the EVENTS table shows the added row:  
   **select** **\*** **from** EVENTS**;**

The query returns the following result:  
**+**-------------+---------+---------+----------------------------------------------------------------------+---------------+-------+--------+-----+-------+------+------+------+------+-----+---------------+

| DEVICE\_TYPE | VERSION | F | RV | T | ACHZ | ACV | DCA | DCV | ENJR | ERRS | MXEC | TMPI | VD | Z |

|-------------+---------+---------+----------------------------------------------------------------------+---------------+-------+--------+-----+-------+------+------+------+------+-----+---------------|

| server | 2.6 | 83 | "15219.64,783.63,48674.48,84679.52,27499.78,2178.83,0.42,74900.19" | 1437560931139 | 42869 | 709489 | 232 | 62287 | 2599 | 205 | 487 | 9 | 54 | 1437644222811 |

| server | 2.6 | 1000083 | "8070.52,54470.71,85331.27,9.10,70825.85,65191.82,46564.53,29422.22" | 1437036965027 | 6953 | 346795 | 250 | 46066 | 9033 | 615 | 0 | 112 | 626 | 1437660796958 |

| cell\_phone | 3.2 | 79 | "786954.67,492.68,3577.48,40.11,343.00,345.8,0.22,8765.22" | 5769784730576 | 75846 | 98355 | 789 | 62287 | 2234 | 578 | 999 | 9 | 54 | 1437644222811 |

**+**-------------+---------+---------+----------------------------------------------------------------------+---------------+-------+--------+-----+-------+------+------+------+------+-----+---------------+

### **Use all columns for comparison**

If the JSON data does not have fields that can be primary key candidates, you could compare all repeating JSON keys in the RAW\_SOURCE table with the corresponding column values in the EVENTS table.

No changes to your existing EVENTS table are required.

1. Insert a new JSON event record into the RAW\_SOURCE table:

insert into raw\_source

select

parse\_json ('{

"device\_type": "web\_browser",

"events": [

{

"f": 79,

"rv": "122375.99,744.89,386.99,12.45,78.08,43.7,9.22,8765.43",

"t": 5769784730576,

"v": {

"ACHZ": 768436,

"ACV": 9475,

"DCA": 94835,

"DCV": 88845,

"ENJR": 8754,

"ERRS": 567,

"MXEC": 823,

"TMPI": 0

},

"vd": 55,

"z": 8745598047355

}

],

"version": 8.7

}');

1. Insert the new record in the RAW\_SOURCE table into the EVENTS table based on a comparison of all repeating key values:

insert into events

select

src:device\_type::string

, src:version::string

, value:f::number

, value:rv::variant

, value:t::number

, value:v.ACHZ::number

, value:v.ACV::number

, value:v.DCA::number

, value:v.DCV::number

, value:v.ENJR::number

, value:v.ERRS::number

, value:v.MXEC::number

, value:v.TMPI::number

, value:vd::number

, value:z::number

from

raw\_source

, lateral flatten( input => src:events )

where not exists

(select 'x'

from events

where events.device\_type = src:device\_type

and events.version = src:version

and events.f = value:f

and events.rv = value:rv

and events.t = value:t

and events.achz = value:v.ACHZ

and events.acv = value:v.ACV

and events.dca = value:v.DCA

and events.dcv = value:v.DCV

and events.enjr = value:v.ENJR

and events.errs = value:v.ERRS

and events.mxec = value:v.MXEC

and events.tmpi = value:v.TMPI

and events.vd = value:vd

and events.z = value:z);

1. Querying the EVENTS table shows the added row:  
   **select** **\*** **from** EVENTS**;**

## **Congratulations**

Congratulations, you have successfully completed the tutorial.

### **Tutorial key points**

* Partitioning the event data in your S3 bucket using logical, granular paths allows you to copy a subset of the partitioned data into Snowflake with a single command.
* Snowflake’s column:key notation, similar to the familiar SQL table.column notation, allows you to effectively query a column within the column (i.e., a sub-column), which is dynamically derived based on the schema definition embedded in the JSON data.
* The [FLATTEN](https://docs.snowflake.com/en/sql-reference/functions/flatten) function allows you to parse JSON data into separate columns.
* Several options are available to update table data based on comparisons with staged data files.

### **Tutorial clean up (optional)**

Execute the following [DROP <object>](https://docs.snowflake.com/en/sql-reference/sql/drop) commands to return your system to its state before you began the tutorial:

**DROP** **DATABASE** **IF** **EXISTS** mydatabase**;**

**DROP** **WAREHOUSE** **IF** **EXISTS** mywarehouse**;**

Dropping the database automatically removes all child database objects such as tables.

Happy Cloud computing

Regards

Saransh Jain