snowflake®

HOW TO ANALYZE JSON WITH SQL

Schema-on-Read made easy

Author: Kent Graziano

CHAMPION GUIDES

TABLE OF CONTENTS

SEMI-STRUCTURED BRINGS NEW INSIGHTS TO BUSINESS

If you're an experienced data architect, data engineer, or data analyst, you've probably been exposed to semi-structured data such as JSON. IoT devices, social media sites, and mobile devices all generate endless streams of JSON log files. Handling JSON data is unavoidable, but it can't be managed the same way as the more familiar structured data. Yet, to thrive in today's world of data, knowing how to manage and derive value from this form of semi-structured data is crucial to delivering valuable insights to your organization. One of the key differentiators in Snowflake Cloud Data Platform is the ability to natively ingest semi-structured data such as JSON, store it efficiently, and then access it quickly using simple extensions to standard SQL. This ebook will give you a modern approach to produce analytics easily and affordably from JSON data using SQL.

SCHEMA? NO NEED!

Load your semi-structured data directly into a relational table

Over the last several years, we have all heard the phrase "Schema-on-Read" to explain the benefit of loading semi-structured data, such as JSON, into a NoSQL platform such as Hadoop. The idea here: Data modeling and schema design could be delayed until long after you loaded the data. Delaying these tasks avoids slowing down getting the data into a repository because you had to wait for a data modeler to first design the tables.

Schema-on-Read implies there is a knowable schema. So, even though organizations can quickly load semi-structured data into Hadoop or a NoSQL platform, there is still more work required to actually parse the data into an understandable schema before it can be analyzed with a standard SQL-based tool. Experienced data professionals often have the burden of determining the schema and writing code to extract the data. Unlike structured data in a relational database, this requirement impedes an organization's ability to access and utilize semi-structured data in a timely manner.

INSTANTLY QUERY SEMI-STRUCTURED DATA WITH SNOWFLAKE

With Snowflake, you can load your semi-structured data directly into a relational table. Then, you can query that data with a SQL statement and join it to other structured data, while not fretting about future changes to the "schema" of that data. Snowflake keeps track of the self-describing schema so you don't have to; no ETL or fancy parsing algorithms are required.

The built-in support to load and query semi-structured data—including JSON, XML, and AVRO—is one of the remarkable benefits of Snowflake. With most of today's big data environments and traditional, on-premises and cloud-washed data warehouses, you have to first load this type of data to a Hadoop or NoSQL platform. Then you need to parse it, for example, with MapReduce, in order to load it into columns in a relational database. Then, and only then, can you visualize or transform that data using a BI/analytics tool or a Notebook/data science tool. All of this means more time, money, and headache for you to allow business users to see that data.

The idea here:

Data modeling and schema design could be delayed until long after you loaded the data. Delaying these tasks avoids slowing down getting the data into a repository because you had to wait for a data modeler to first design the tables.

HOW SNOWFLAKE SOLVED THIS PROBLEM

It's simple. Snowflake has a new data type called VARIANT that allows semi-structured data to be loaded, as is, into a column in a relational table.

When Snowflake loads semi-structured data, it optimizes how it stores that data internally by automatically discovering the attributes and structure that exist in the data, and using that knowledge to optimize how the data is stored. Snowflake also looks for repeated attributes across records, organizing and storing those repeated attributes separately. This enables better compression and faster access, similar to the way that a columnar database optimizes storage of columns of data.

The upshot: No Hadoop or NoSQL is needed in your enterprise data landscape for the sole purpose of holding semi-structured data. The result is a modern data platform that uses SQL, which you and your staff already know how to write. And as the data source evolves and changes over time with new attributes, nesting, or arrays, there's no need to redo ETL or ELT code. The VARIANT data type does not care if the schema varies.

DATA IN, INSIGHT OUT

But that's only half the equation. Once the data is in, how do you get the insight out? Snowflake has extensions to SQL to reference the internal schema of the data. Because the data is self-describing, you can query the components and join the data to columns in other tables, as if you already parsed the data into a standard relational format, except no coding, ETL, or other parsing is required to prep the data.

In addition, statistics about the subcolumns are also collected, calculated, and stored in Snowflake's metadata repository. This gives Snowflake's advanced query optimizer metadata about the semi-structured data, to optimize access to it. The collected statistics allow the optimizer to use pruning to minimize the amount of data needed for access, thus speeding the return of data.

DATA WAREHOUSING AND ANALYTICS, REIMAGINED FOR THE CLOUD

No other on-premises or cloud-washed solution offers Snowflake's optimized level of support for processing semi-structured data. Even though some traditional vendors have added features to store and access JSON and XML, those are add-ons to legacy code, using existing data types such as character large objects (CLOBs), and they are not natively optimized.

With these solutions, getting any kind of performance optimization requires additional performance tuning by DBAs. For example, in its documentation, one of the newer, cloud-washed data warehouse providers states that customers should not try to use their JSON feature at scale. This is yet another example of how cloud-washed legacy code can't magically solve data problems.

ENOUGH THEORY.
LET'S GET STARTED.

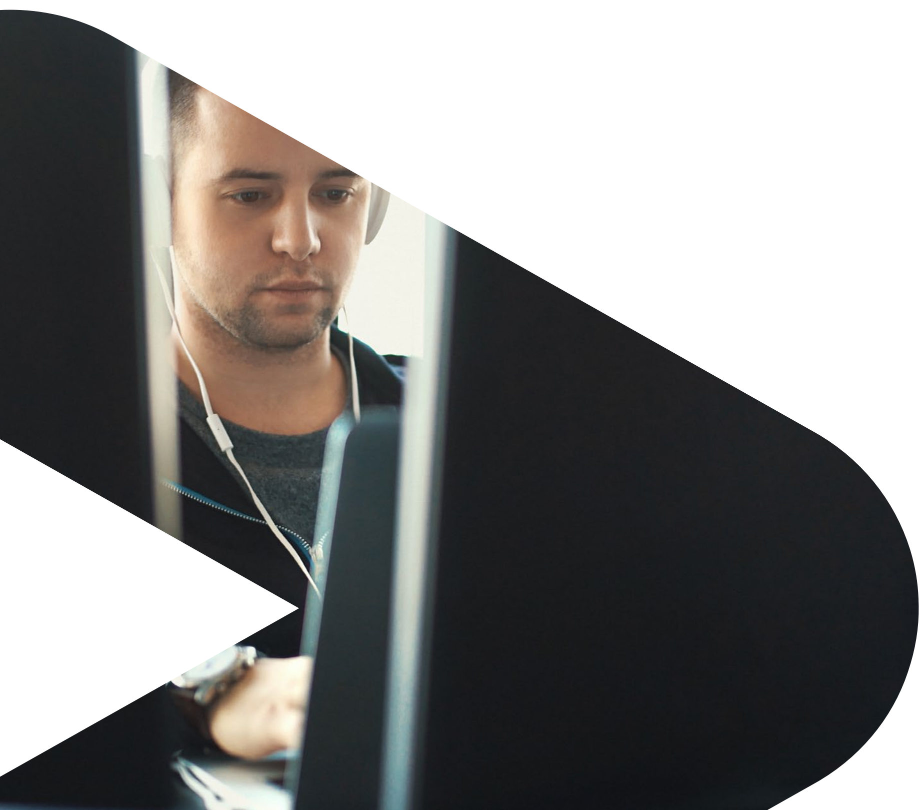
How you can load semi-structured data directly into Snowflake

1. CREATE A TABLE

I already have a Snowflake account, a database, and a multi-cluster warehouse set up, so just like I would in any other database, I simply issue a DDL statement to create a table:

```
create or replace table json_demo (v variant);
```

Now I have a table with one column ("v") with a declared data type of **VARIANT**.



2. LOAD SOME DATA

Now I load a sample JSON document using an **INSERT** and Snowflake's **PARSE_JSON** function. We're not simply loading the document as text but rather storing it as an object in the **VARIANT** data type, while at the same time converting it to an **optimized columnar** format (to query later):

```
insert into json_demo
 select
 parse_json(
 '{
    "fullName": "Johnny Appleseed",
    "age": 42,
    "gender": "Male",
    "phoneNumber": {
                "areaCode": "415",
                "subscriberNumber": "5551234"
               },
    "children": [
               { "name": "Jayden", "gender": "Male", "age": "10" },
               { "name": "Emma", "gender": "Female", "age": "8" },
               { "name": "Madelyn", "gender": "Female", "age": "6" }
              ],
    "citiesLived": [
                { "cityName": "London",
                  "yearsLived": [ "1989", "1993", "1998", "2002" ]
                },
                { "cityName": "San Francisco",
                  "yearsLived": [ "1990", "1993", "1998", "2008" ]
                },
                { "cityName": "Portland",
                  "yearsLived": [ "1993", "1998", "2003", "2005" ]
                },
                { "cityName": "Austin",
                  "yearsLived": [ "1973", "1998", "2001", "2005" ]
                }
               ]
  }');
```

While this approach is useful for testing, normally JSON would be loaded into a Snowflake table from your Snowflake staging area using a simple **COPY** command.

```
copy into myjsontable
  from @my_json_stage/tutorials/dataloading/contacts.json
  on_error = 'skip_file';
```

For more details on the many options and features of the COPY command, see Snowflake's data loading tutorial.

3. START PULLING DATA OUT

Now that we've loaded an example, let's work through how we access it. We'll start with just getting the data from the NAME subcolumn:

```
select v:fullName from json_demo;
```

1 row produced	
row#	V:FULLNAME
1	"Johnny Appleseed"

Where:

v = the column name in the json_demo table (from our create table command)

fullName = attribute in the JSON schema

v:fullName = notation to indicate which attribute in column "v" we want to select

Similar to the table.column notation all SQL people are familiar with, Snowflake has the ability to effectively specify a column within the column––a subcolumn. However, we cannot use the dot notation for our separator, because SQL syntax has already claimed that. So, the Snowflake team chose the next obvious thing: a colon to reference the JSON subcolumns and navigate that hierarchy. This structural information is dynamically derived based on the schema definition embedded in the JSON string. Snowflake's advanced metadata engine records this information at the time it loads the JSON document into the table.

4. CASTE THE DATA

Usually we don't want to see the double quotes around the data in the report output unless we're going to create an extract file. Instead, we can format it as a string and give it a nicer column alias, similar to what we would do with a normal column:

```
select v:fullName::string as full_name
from json_demo;
```

1 row produced	
row#	FULL_NAME
1	Johnny Appleseed

Next, let's look at a bit more of the data using the same syntax from above:

```
select
   v:fullName::string as full_name,
   v:age::int as age,
   v:gender::string as gender
from json_demo;
```

1 row produced			
row#	FULL_NAME	AGE	GENDER
1	Johnny Appleseed	42	Male

Again, we use simple SQL and the output is similar to the results from any table you might have built in a traditional data warehouse.

At this point, you could look at a table in Snowflake with a VARIANT column and quickly start "shredding" the JSON with SQL. You can query semi-structured data without learning a new programming language or using a framework required with Hadoop or NoSQL. Instead, you have a much lower learning curve to get the same result.

A MORE COMPLEX DATA LOAD

Nested data and adding new attributes

Yes, those examples are very simple. So let's look at something a bit more complex. Notice that the original sample document contains some nesting of the data:

```
{
   "fullName": "Johnny Appleseed",
   "age": 42,
   "gender": "Male",
   "phoneNumber": {
                "areaCode": "415",
                "subscriberNumber": "5551234"
                    },
...
```

How do we pull that apart? We use a very familiar table.column dot notation:

```
select
  v:phoneNumber.areaCode::string as area_code,
  v:phoneNumber.subscriberNumber::string as subscriber_number
from json_demo;
```

Just as fullName, age and gender are subcolumns, so too is phoneNumber. And subsequently, areaCode and subscriberNumber are subcolumns of the subcolumn. We can pull apart nested objects like this and easily adapt if the schema changes and we add another subcolumn.

WHAT HAPPENS IF THE STRUCTURE CHANGES?

One of the benefits of storing data in JSON is that the schema can easily change. But imagine if, in a subsequent load, the data provider changed the specification to this:

```
{
   "fullName": "Johnny Appleseed",
   "age": 42,
   "gender": "Male",
   "phoneNumber": {
                "areaCode": "415",
                "subscriberNumber": "5551234",
                "extensionNumber": "24"
                  },
...
```

A new attribute, **extensionNumber**, was added to phoneNumber! What happens to the load? Nothing. It keeps working because we ingest the string into the VARIANT column in the table.

You may ask, "What about the ETL/ELT code?" What code? There is no code, so there's nothing to break. And what about existing reports? They keep working, too. The previous query will work just fine. If you want to see the new column, the SQL needs to be refactored to account for the change:

```
select
  v:phoneNumber.areaCode::string as area_code,
  v:phoneNumber.subscriberNumber::string as subscriber_number,
  v:phoneNumber.extensionNumber::string as extension_number
from json_demo;
```

In addition, if the reverse happens and an attribute is dropped, the query will not fail. Instead, it simply returns a NULL value. In this way, Snowflake insulates all the code you write from these types of dynamic changes.

HOW TO HANDLE ARRAYS OF DATA

One of JSON's many cool features is the ability to specify and embed an array of data within the document. In this example, one such array is children:

```
"children": [
    { "name": "Jayden", "gender": "Male", "age": "10" },
    { "name": "Emma", "gender": "Female", "age": "8" },
    { "name": "Madelyn", "gender": "Female", "age": "6" }
 ]
```

You will notice there are three rows in the array and each row has three subcolumns: name, gender, and age. Each of those rows constitutes the value of that array entry, which includes all the subcolumn labels and data. (Remember this for later.) So how do you know how many rows there are if you don't have access to the raw data? Like this:

```
select array_size(v:children) from json_demo;
```

The function **ARRAY_SIZE** determines it for us. To pull the data for each row in the array, we can use the previous dot notation, but with the added specification for the row number of the array located inside the brackets:

```
select v:children[0].name from json_demo
union all
select v:children[1].name from json_demo
union all
select v:children[2].name from json_demo;
```

3 rows produced	
row#	V:CHILDREN[0].NAME
1	"Jayden"
2	"Emma"
3	"Madelyn"

If another element is added to the array, such as a fourth child, we will not have to change the SQL. FLATTEN allows us to determine the structure and content of the array on the fly. This makes the SQL resilient to changes in the JSON document.

You can now get all the array sub-columns and format them just like a relational table:

```
select
  f.value:name::string as child_name,
  f.value:gender::string as child_gender,
  f.value:age::string as child_age
from json_demo, table(flatten(v:children)) f;
```

3 rows produced			
row#	CHILD_NAME	CHILD_GENDER	CHILD_AGE
1	Jayden	Male	10
2	Emma	Female	8
3	Madelyn	Female	6

Putting all this together, you can write a query to get the parent's name and the children's names:

```
select
  v:fullName::string as parent_name,
  f.value:name::string as child_name,
  f.value:gender::string  as child_gender,
  f.value:age::string as child_age
from json_demo, table(flatten(v:children)) f;
```

3 rows produced				
row#	PARENT_NAME	CHILD_NAME	CHILD_GENDER	CHILD_AGE
1	Johnny Appleseed	Jayden	Male	10
2	Johnny Appleseed	Emma	Female	8
3	Johnny Appleseed	Madelyn	Female	6

If you just want a quick count of children by parent, you do not need to use FLATTEN but instead you refer back to ARRAY_SIZE:

```
select
  v:fullName::string as Parent_Name,
  array_size(v:children) as Number_of_Children
from json_demo;
```

1 row produced		
row#	PARENT_NAME	NUMBER_OF_CHILDREN
1	Johnny Appleseed	3

Notice no GROUP BY clause is needed because the nested structure of the JSON has naturally grouped the data for us.

HOW TO HANDLE MULTIPLE ARRAYS

Simplifying an array with an array

You may recall there are multiple arrays in the sample JSON string. You can pull from several arrays at once with no problem:

```
select
   v:fullName::string as Parent_Name,
   array_size(v:citiesLived) as Cities_lived_in,
   array_size(v:children) as Number_of_Children
from json_demo;
```

1 row produced			
row#	PARENT_NAME	CITIES_LIVED_IN	NUMBER_OF_CHILDREN
1	Johnny Appleseed	4	3

What about an array within an array? Snowflake can handle that, too. From the sample data, you can see **yearsLived** is an array nested inside the array described by **citiesLived**:

```
"citiesLived": [
 { "cityName": "London",
   "yearsLived": [ "1989", "1993", "1998", "2002" ]
 },
 { "cityName": "San Francisco",
   "yearsLived": [ "1990", "1993", "1998", "2008" ]
 },
 { "cityName": "Portland",
   "yearsLived": [ "1993", "1998", "2003", "2005" ]
 },
 { "cityName": "Austin",
   "yearsLived": [ "1973", "1998", "2001", "2005" ]
 }
 ]
```

To pull that data out, we add a **second** FLATTEN clause that transforms the **yearsLived** array within the FLATTENed **citiesLived** array.

```
select
  cl.value:cityName::string as city_name,
  yl.value::string as year_lived
from json_demo,
     table(flatten(v:citiesLived)) cl,
     table(flatten(cl.value:yearsLived)) yl;
```

In this case the second FLATTEN (alias "**yl**") transforms, or pivots, the **yearsLived** array for each **value** returned from the first FLATTEN of the **citiesLived** array ("cl").

The resulting output shows the year lived by city name:

16 rows produced		
row#	CITY_NAME	Year_Lived
1	London	1989
2	London	1993
3	London	1998
4	London	2002
5	San Francisco	1990
6	San Francisco	1993
7	San Francisco	1998
8	San Francisco	2008
9	Portland	1993
10	Portland	1998
11	Portland	2003
12	Portland	2005

Similar to the previous example, you can augment this result by adding the parent's name to show who lived where:

```
select
  v:fullName::string as parent_name,
  cl.value:cityName::string as city_name,
  yl.value::string as year_lived
from json_demo,
     table(flatten(v:citiesLived)) cl,
     table(flatten(tf.value:yearLived)) yl;
```

16 rows produced			
row#	PARENT_NAME	CITY_NAME	Year_Lived
1	Johnny Appleseed	London	1989
2	Johnny Appleseed	London	1993
3	Johnny Appleseed	London	1998
4	Johnny Appleseed	London	2002
5	Johnny Appleseed	San Francisco	1990
6	Johnny Appleseed	San Francisco	1993
7	Johnny Appleseed	San Francisco	1998
8	Johnny Appleseed	San Francisco	2008
9	Johnny Appleseed	Portland	1993
10	Johnny Appleseed	Portland	1998
11	Johnny Appleseed	Portland	2003
12	Johnny Appleseed	Portland	2005

AGGREGATIONS

How to execute standard SQL aggregations on semi-structured data

To answer the question you're probably thinking: Yes! You can even execute standard SQL aggregations on the semi-structured data. So, just as with ANSI SQL, you can do a **COUNT(*)** and a **GROUP BY**:

```
select
  cl.value:cityName::string as city_name,
  count(*) as year_lived
from json_demo,
    table(flatten(v:citiesLived)) cl,
    table(flatten(tf.value:yearLived)) yl
group by 1;
```

4 rows produced		
row#	CITY_NAME	Year_Lived
1	London	4
2	San Francisco	4
3	Portland	4
4	Austin	4

You can also create much more complex analyses using the library of standard SQL aggregation and windowing functions including LEAD, LAG, RANK, and STDDEV.

FILTERING YOUR DATA

How to focus your data analytics to only the data you need

What if you don't want to return every row in an array? Similar to standard SQL, you add a **WHERE** clause:

```
select
  cl.value:cityName::string as city_name,
  count(*) as years_lived
from json_demo,
     table(flatten(v:citiesLived)) cl,
     table(flatten(tf.value:yearsLived)) yl
where city_name = 'Portland'
group by 1;
```

4 rows produced		
row#	CITY_NAME	Year_Lived
1	Portland	4

To make it easier to read the SQL, notice you can even reference the sub-column alias **city_name** in the predicate. You can also use the full, subcolumn specification **cl.value:cityName**.

SCHEMA-ON-READ IS A REALITY

Get access to all your data with the ease of SQL

The examples we've walked through show how very easy it is to load and analyze semi-structured data with SQL, using Snowflake as both your big data and data warehouse solution. Snowflake invented a new, optimized data type, **VARIANT**, which lives in a relational table structure in a relational database. VARIANT offers native support for querying JSON without the need to analyze the structure ahead of time or design appropriate database tables and columns, subsequently parsing the data string into that predefined schema.

VARIANT provides the same performance as all the standard relational data types. In the examples, you saw easy-to-learn extensions to ANSI-standard SQL for accessing that data in a very flexible, resilient manner. With Snowflake, you get the bonus of on-demand resource scalability that no traditional or cloud-washed data warehouse solution delivers.

With these features, Snowflake gives you a fast path to the enterprise endgame: the true ability to quickly and easily load semi-structured data into a modern cloud data platform and make it available for immediate analysis

ABOUT SNOWFLAKE

Snowflake delivers the Data Cloud—a global network where thousands of organizations mobilize data with near-unlimited scale, concurrency, and performance. Inside the Data Cloud, organizations unite their siloed data, easily discover and securely share governed data, and execute diverse analytic workloads. Wherever data or users live, Snowflake delivers a single and seamless experience across multiple public clouds. Snowflake's platform is the engine that powers and provides access to the Data Cloud, creating a solution for data warehousing, data lakes, data engineering, data science, data application development, and data sharing. Join Snowflake customers, partners, and data providers already taking their businesses to new frontiers in the Data Cloud. **snowflake.com**.

About the author

Kent Graziano is a recognized industry expert, keynote speaker, and published author in the areas of data modeling, data warehousing, and agile data. He has over 30 years of experience in information technology, including data modeling, data analysis, and relational database design, as well as large scale data warehouse architecture, design, and implementation.

snowflake.com #MobilizeYourData