Semi-structured data types

The following Snowflake data types can contain other data types:

- VARIANT (can contain a value of any other data type).
- OBJECT (can directly contain a VARIANT value, and thus indirectly contain a value of any other data type, including itself).
- ARRAY (can directly contain a VARIANT value, and thus indirectly contain a value of any other data type, including itself).

We often refer to these data types as *semi-structured* data types. Strictly speaking, OBJECT is the only one of these data types that, by itself, has all of the characteristics of a true semi-structured data type. However, combining these data types allows you to explicitly represent arbitrary hierarchical data structures, which can be used to load and operate on data in semi-structured formats (such as JSON, Avro, ORC, Parquet, or XML).

Note
For information about *structured data types* (for example, ARRAY(INTEGER), OBJECT(city VARCHAR), or MAP(VARCHAR, VARCHAR), see Structured data types.

VARIANT

Characteristics of a VARIANT value

A VARIANT value can have a maximum size of up to 16 MB of uncompressed data. However, in practice, the maximum size is usually smaller due to internal overhead. The maximum size is also dependent on the object being stored.

Note

If the 2025_03 behavior change bundle is enabled, the maximum size for a VARIANT value is 128 MB. For more information, see Size limits for database objects.

Inserting VARIANT data

To insert VARIANT data directly, use `INSERT INTO ... SELECT`. The following example shows how to insert JSON-formatted data into a VARIANT value:

```
CREATE OR REPLACE TABLE variant_insert (v VARIANT);
INSERT INTO variant_insert (v)
  SELECT PARSE_JSON('{"key3": "value3", "key4": "value4"}');
SELECT * FROM variant_insert;
```

```
+---------------------+
| V                   |
|---------------------|
| {                   |
| }                    |
+----------------------+
```

Using VARIANT values

To convert a value to or from the VARIANT data type, you can explicitly cast using the CAST function, the TO_VARIANT function, or the `::` operator (for example, `<expression>::VARIANT`).

In some situations, a value can be implicitly cast to a VARIANT value. For details, see Data type conversion.

The following example shows how to use a VARIANT value, including how to convert from a VARIANT value and to a VARIANT value.

Create a table and insert a value:

```
CREATE OR REPLACE TABLE varia (float1 FLOAT, v VARIANT, float2 FLOAT);
INSERT INTO varia (float1, v, float2) VALUES (1.23, NULL, NULL);
```

The first UPDATE converts a FLOAT value to a VARIANT value. The second UPDATE converts a VARIANT value to a FLOAT value.

```
UPDATE varia SET v = TO_VARIANT(float1);  -- converts from a FLOAT value to a VARIANT value.
UPDATE varia SET float2 = v::FLOAT;       -- converts from a VARIANT value to a FLOAT value.
```

```
SELECT * FROM varia;
```

```
+--------+-----------------------+--------+
| FLOAT1 | V                     | FLOAT2 |
|--------+-----------------------+--------|
|   1.23 | 1.230000000000000e+00 |   1.23 |
+--------+-----------------------+--------+
```

As shown in the previous example, to convert a value from the VARIANT data type, cast the VARIANT value to the target data type. For example, the following statement uses the `::` operator to convert the VARIANT to a FLOAT:

```
SELECT my_variant_column::FLOAT * 3.14 FROM ...;
```

VARIANT data stores both the value and the data type of the value. Therefore, you can use VARIANT values in expressions where the value's data type is valid without first casting the VARIANT. For example, if VARIANT column `my_variant_column` contains a numeric value, then you can directly multiply `my_variant_column` by another numeric value:

```
SELECT my_variant_column * 3.14 FROM ...;
```

You can retrieve the value's native data type by using the TYPEOF function.

```
SELECT 'Sample', 'Sample'::VARIANT, 'Sample'::VARIANT::VARCHAR;
```

```
+----------+------------------+---------------------------+
| 'SAMPLE' | 'SAMPLE'::VARIANT | 'SAMPLE'::VARIANT::VARCHAR |
|----------+------------------+---------------------------|
| Sample   | "Sample"         | Sample                    |
+----------+------------------+---------------------------+
```

A VARIANT value can be missing (contain SQL NULL), which is different from a VARIANT **null** value, which is a real value used to represent a null value in semi-structured data. VARIANT **null** is a true value that compares as equal to itself. For more information, see NULL values.

If data was loaded from JSON format and stored in a VARIANT column, then the following considerations apply:

- For data that is mostly regular and uses only native JSON types (such as strings and numbers), the performance is very similar for storage and query operations on relational data and data in a VARIANT column.
- For non-native data (such as dates and timestamps), the values are stored as strings when loaded into a VARIANT column. Therefore, operations on these values might be slower and also consume more space than when stored in a relational column with the corresponding data type.

For more information about using the VARIANT data type, see Considerations for semi-structured data stored in VARIANT.

For more information about querying semi-structured data stored in a VARIANT column, see Querying Semi-structured Data.

Common uses for VARIANT data

VARIANT data is typically used when:

- You want to create hierarchical data by explicitly defining a hierarchy that contains two or more ARRAY values or OBJECT values.
- You want to load JSON, Avro, ORC, or Parquet data directly, without explicitly describing the hierarchical structure of the data.

  Snowflake can convert data from JSON, Avro, ORC, or Parquet format to an internal hierarchy of ARRAY, OBJECT, and VARIANT data and store that hierarchical data directly in a VARIANT value. Although you can manually construct the data hierarchy yourself, it is usually easier to let Snowflake do it for you.

  For more information about loading and converting semi-structured data, see Loading Semi-structured Data.

OBJECT

A Snowflake OBJECT value is analogous to a JSON "object". In other programming languages, the corresponding data type is often called a "dictionary," "hash," or "map."

An OBJECT value contains key-value pairs.

Characteristics of an OBJECT value

In Snowflake semi-structured OBJECT data, each key is a VARCHAR value, and each value is a VARIANT value.

and a person's age as an INTEGER value. In the following example, both the name and the age are cast to VARIANT values.

```
SELECT OBJECT_CONSTRUCT(
  'name', 'Jones'::VARIANT,
  'age',  42::VARIANT);
```

The following considerations apply to OBJECT data:

- Currently, Snowflake doesn't support explicitly-typed objects.
- In a key-value pair, the key shouldn't be an empty string, and neither the key nor the value should be NULL.
- The maximum length of an OBJECT value is 16 MB.

  If the 2025_03 behavior change bundle is enabled, the maximum size for an OBJECT value is 128 MB. For more information, see Size limits for database objects.
- An OBJECT value can contain semi-structured data.
- An OBJECT value can be used to create hierarchical data structures.

Note

Snowflake also supports the structured OBJECT data type, which allows for values other than VARIANT values. A structured OBJECT type also defines the keys that must be present in an OBJECT value of that type. For more information, see Structured data types.

The following example uses the OBJECT_CONSTRUCT function to construct the OBJECT value that it inserts.

```
CREATE OR REPLACE TABLE object_example (object_column OBJECT);
INSERT INTO object_example (object_column)
  SELECT OBJECT_CONSTRUCT('thirteen', 13::VARIANT, 'zero', 0::VARIANT);
SELECT * FROM object_example;
```

```
+-------------------+
| OBJECT_COLUMN     |
|-------------------|
| {                 |
|   "thirteen": 13, |
|   "zero": 0       |
| }                 |
+-------------------+
```

In each key-value pair, the value was explicitly cast to VARIANT. Explicit casting wasn't required in these cases. Snowflake can implicitly cast to VARIANT. For information about implicit casting, see Data type conversion.

You can also use an OBJECT constant to specify the OBJECT value to insert. For more information, see OBJECT constants.

OBJECT constants

A *constant* (also known as a *literal*) refers to a fixed data value. Snowflake supports using constants to specify OBJECT

OBJECT constants have the following syntax:

```
{ [<key>: <value> [, <key>: <value> , ...]] }
```

Where:

`<key>`

The key in a key-value pair. The `<key>` must be a string literal.

`<value>`

The value that is associated with the key. The `<value>` can be a literal or an expression. The `<value>` can be any data type.

The following are examples that specify OBJECT constants:

- `{}` is an empty OBJECT value.
- `{ 'key1': 'value1' , 'key2': 'value2' }` contains the specified key-value pairs for the OBJECT value using literals for the values.
- `{ 'key1': c1+1 , 'key2': c1+2 }` contains the specified key-value pairs for the OBJECT value using expressions for the values.
- `{*}` is a wildcard that constructs the OBJECT value from the specified data using the attribute names as keys and the associated values as values.

```
SELECT {*} FROM my_table;

SELECT {my_table1.*}
  FROM my_table1 INNER JOIN my_table2
    ON my_table2.col1 = my_table1.col1;
```

You can use the ILIKE and EXCLUDE keywords in an object constant. To select specific columns, use the ILIKE keyword. For example, the following query selects columns that match the pattern `col1%` in the table `my_table`:

```
SELECT {* ILIKE 'col1%'} FROM my_table;
```

To exclude specific columns, use the EXCLUDE keyword. For example, the following query excludes `col1` in the table `my_table`:

```
SELECT {* EXCLUDE col1} FROM my_table;
```

The following query excludes `col1` and `col2` in the table `my_table`:

```
SELECT {* EXCLUDE (col1, col2)} FROM my_table;
```

Wildcards can’t be mixed with key-value pairs. For example, the following wildcard specification isn’t allowed:

```
SELECT {*, 'k': 'v'} FROM my_table;
SELECT {t1.*, t2.*} FROM t1, t2;
```

The following statements use an OBJECT constant and the OBJECT_CONSTRUCT function to perform an insert of OBJECT data into a table. The OBJECT values contain the names and capital cities of two Canadian provinces.

```
CREATE OR REPLACE TABLE my_object_table (my_object OBJECT);

INSERT INTO my_object_table (my_object)
  SELECT { 'PROVINCE': 'Alberta'::VARIANT , 'CAPITAL': 'Edmonton'::VARIANT };

INSERT INTO my_object_table (my_object)
  SELECT OBJECT_CONSTRUCT('PROVINCE', 'Manitoba'::VARIANT , 'CAPITAL', 'Winnipeg'::VARIANT );

SELECT * FROM my_object_table;
```

```
+--------------------------+
| MY_OBJECT                |
|--------------------------|
| {                        |
|   "CAPITAL": "Edmonton", |
|   "PROVINCE": "Alberta"  |
| }                        |
| {                        |
|   "CAPITAL": "Winnipeg", |
|   "PROVINCE": "Manitoba" |
| }                        |
```

The following example uses a wildcard (`{*}`) to insert OBJECT data by getting the attribute names and values from the FROM clause. First, create a table named `demo_ca_provinces` with VARCHAR values that contain the province and capital names:

```
CREATE OR REPLACE TABLE demo_ca_provinces (province VARCHAR, capital VARCHAR);
INSERT INTO demo_ca_provinces (province, capital) VALUES
  ('Ontario', 'Toronto'),
  ('British Columbia', 'Victoria');

SELECT province, capital
  FROM demo_ca_provinces
  ORDER BY province;
```

```
+------------------+----------+
| PROVINCE         | CAPITAL  |
|------------------+----------|
| British Columbia | Victoria |
| Ontario          | Toronto  |
+------------------+----------+
```

Insert object data into the `my_object_table` using the data in the `demo_ca_provinces` table:

```
INSERT INTO my_object_table (my_object)
  SELECT {*} FROM demo_ca_provinces;

SELECT * FROM my_object_table;
```

```
+-----------------------------------+
| MY_OBJECT                         |
|-----------------------------------|
| {                                 |
|   "CAPITAL": "Edmonton",          |
|   "PROVINCE": "Alberta"           |
| }                                 |
| {                                 |
|   "CAPITAL": "Winnipeg",          |
|   "PROVINCE": "Manitoba"          |
| }                                 |
| {                                 |
|   "CAPITAL": "Toronto",           |
|   "PROVINCE": "Ontario"           |
| }                                 |
| {                                 |
|   "CAPITAL": "Victoria",          |
|   "PROVINCE": "British Columbia"  |
| }                                 |
+-----------------------------------+
```

The following example uses expressions for the values in an OBJECT constant:

```
SET my_variable = 10;
SELECT {'key1': $my_variable+1, 'key2': $my_variable+2};
```

```
+---------------------------------------------------+
|   "key2": 12                                      |
| }                                                 |
+---------------------------------------------------+
```

SQL statements specify string literals inside an OBJECT value with single quotes (as elsewhere in Snowflake SQL), but string literals inside an OBJECT value are displayed with double quotes:

```
SELECT { 'Manitoba': 'Winnipeg' } AS province_capital;
```

```
+--------------------------+
| PROVINCE_CAPITAL         |
|--------------------------|
| {                        |
|   "Manitoba": "Winnipeg" |
| }                        |
+--------------------------+
```

Accessing elements of an OBJECT value by key

To retrieve the value in an OBJECT value, specify the key in square brackets, as shown below:

```
SELECT object_column['thirteen'] FROM object_example;
SELECT object_column['thirteen'],
       object_column:thirteen
  FROM object_example;
```

```
+---------------------------+-------------------------+
| OBJECT_COLUMN['THIRTEEN'] | OBJECT_COLUMN:THIRTEEN |
|---------------------------+------------------------|
| 13                        | 13                     |
+---------------------------+-------------------------+
```

For more information about the colon operator, see Dot Notation, which describes the use of the `:` and `.` operators to access nested data.

Common uses for OBJECT data

OBJECT data is typically used when one or more of the following are true:

- You have multiple pieces of data that are identified by strings. For example, if you want to look up information by province name, you might want to use an OBJECT value.
- You want to store information about the data with the data. The names (keys) aren't merely distinct identifiers, but are meaningful.
- The information has no natural order, or the order can be inferred solely from the keys.
- The structure of the data varies, or the data can be incomplete. For example, if you want to create a catalog of books

ARRAY

A Snowflake array is similar to an array in many other programming languages. An array contains 0 or more pieces of data. Each element is accessed by specifying its position in the array.

Characteristics of an array

Each value in a semi-structured array is of type VARIANT. A VARIANT value can contain a value of any other data type.

Values of other data types can be cast to VARIANT values and then stored in an array. Some functions for arrays, including ARRAY_CONSTRUCT, can implicitly cast values to VARIANT values.

Because arrays store VARIANT values, and because VARIANT values can store other data types within them, the underlying data types of the values in an array can be different. However, in most cases, the data elements are of the same or compatible types, so they can all be processed the same way.

The following considerations apply to arrays:

- Snowflake doesn't support arrays of elements of a specific non-VARIANT type.
- A Snowflake array is declared without specifying the number of elements. An array can grow dynamically based on operations such as ARRAY_APPEND. Snowflake doesn't currently support fixed-size arrays.
- An array can contain both SQL NULL values and JSON null values. For more information, see NULL values.
- The theoretical maximum combined size of all values in an array is 16 MB. However, arrays have internal overhead. The practical maximum data size is usually smaller, depending upon the number and values of the elements.

Note

Snowflake also supports structured arrays, which allow for elements of types other than VARIANT. For more information, see Structured data types.

Inserting ARRAY data

To insert ARRAY data directly, use `INSERT INTO ... SELECT`.

The following code uses the ARRAY_CONSTRUCT function to construct the array that it inserts.

```
CREATE OR REPLACE TABLE array_example (array_column ARRAY);
INSERT INTO array_example (array_column)
  SELECT ARRAY_CONSTRUCT(12, 'twelve', NULL);
```

You can also use an ARRAY constant to specify the array to insert. For more information, see ARRAY constants.

ARRAY constants

A *constant* (also known as a *literal*) refers to a fixed data value. Snowflake supports using constants to specify ARRAY values. ARRAY constants are delimited with square brackets (`[` and `]`).

ARRAY constants have the following syntax:

```
[<value> [, <value> , ...]]
```

Where:

`<value>`

The value that is associated with an array element. The `<value>` can be a literal or an expression. The `<value>` can be any data type.

The following are examples that specify ARRAY constants:

- `[]` is an empty ARRAY value.
- `[ 1 , 'value1' ]` contains the specified values for the ARRAY constant using literals for the values.
- `[ c1+1 , c1+2 ]` contains the specified values for the ARRAY constant using expressions for the values.

The following example uses an ARRAY constant to specify the array to insert.

```
INSERT INTO array_example (array_column)
  SELECT [ 12, 'twelve', NULL ];
```

The following statements use an ARRAY constant and the ARRAY_CONSTRUCT function to perform the same task:

```
UPDATE my_table SET my_array = [ 1, 2 ];
```

The following example uses expressions for the values in an ARRAY constant:

```
SET my_variable = 10;
SELECT [$my_variable+1, $my_variable+2];
```

```
+----------------------------------+
| [$MY_VARIABLE+1, $MY_VARIABLE+2] |
|----------------------------------|
| [                                |
|   11,                            |
|   12                             |
| ]                                |
+----------------------------------+
```

SQL statements specify string literals inside an array with single quotes (as elsewhere in Snowflake SQL), but string literals inside an array are displayed with double quotes:

```
SELECT [ 'Alberta', 'Manitoba' ] AS province;
```

```
+--------------+
| PROVINCE     |
|--------------|
| [            |
|   "Alberta", |
```

Accessing elements of an array by index or by slice

Array indexes are 0-based, so the first element in an array is element 0.

Values in an array are accessed by specifying an array element's index number in square brackets. For example, the following query reads the value at index position `2` in the array stored in `my_array_column`.

```
SELECT my_array_column[2] FROM my_table;
```

Arrays can be nested. The following query reads the zeroth element of the zeroth element of a nested array:

```
SELECT my_array_column[0][0] FROM my_table;
```

Attempting to access an element beyond the end of an array returns NULL.

A *slice* of an array is a sequence of adjacent elements (that is, a contiguous subset of the array).

You can access a slice of an array by calling the ARRAY_SLICE function. For example:

```
SELECT ARRAY_SLICE(my_array_column, 5, 10) FROM my_table;
```

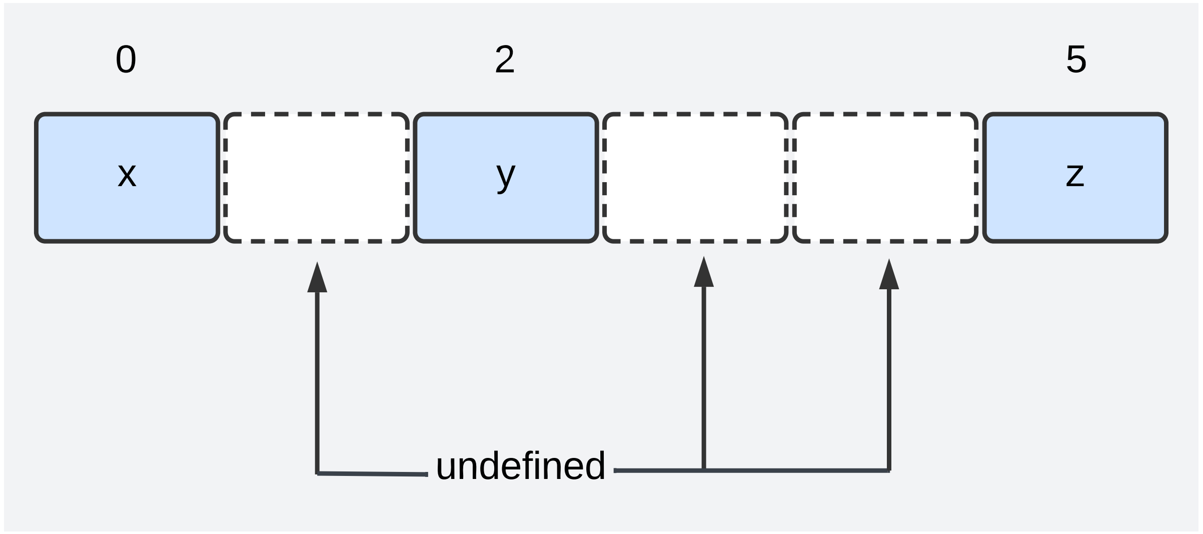
The ARRAY_SLICE function returns elements from the specified starting element (5 in the example above) up to ***but not including*** the specified ending element (10 in the example above).

Dense and sparse arrays

An array can be *dense* or *sparse*.

In a dense array, the index values of the elements start at zero and are sequential (0, 1, 2, and so on). However, in a sparse array, the index values can be non-sequential (for example, 0, 2, 5). The values don't need to start at 0.

If an index has no corresponding element, then the value corresponding to that index is said to be *undefined*. For example, if a sparse array has three elements, and those elements are at indexes 0, 2, and 5, then the elements at indexes 1, 3, and 4 are `undefined`.



An undefined element is treated as an element. For example, consider the earlier example of a sparse array that contains elements at indexes 0, 2, and 5 (and doesn't have any elements after index 5). If you read the slice containing elements at indexes 3 and 4, then the output is similar to the following:

```
[ undefined, undefined ]
```

Attempting to access a slice beyond the end of an array results in an empty array, not an array of `undefined` values. The following SELECT statement attempts to read beyond the last element in the sample sparse array:

```
SELECT ARRAY_SLICE(array_column, 6, 8) FROM table_1;
```

The output is an empty array:

```
+----------------------------------+
| array_slice(array_column, 6, 8) |
+----------------------------------+
| [ ]                              |
+----------------------------------+
```

Note that `undefined` is different from NULL. A NULL value in an array is a defined element.

In a dense array, each element consumes storage space, even if the value of the element is NULL. In a sparse array, `undefined` elements don't directly consume storage space.

In a dense array, the theoretical range of index values is from 0 to 16777215. (The maximum theoretical number of elements is 16777216 because the upper limit on size is 16 MB, or 16777216 bytes, and the smallest possible value is one byte.)

In a sparse array, the theoretical range of index values is from 0 to 2^{31} - 1. However, due to the 16 MB limitation, a sparse array can't hold 2^{31} values. The maximum theoretical number of values is still limited to 16777216.

Note

If the 2025_03 behavior change bundle is enabled, the maximum size for an array is 128 MB. For more information, see Size limits for database objects.

You can create a sparse array by using the ARRAY_INSERT function to insert values at specific index points in an array (leaving other array elements `undefined`). Because ARRAY_INSERT pushes elements to the right, which changes the index values required to access them, it is normally best to fill a sparse array from left to right (that is, from 0 up, increasing the index value for each new value inserted).

Common uses for ARRAY data

ARRAY data is typically used when one or more of the following are true:

- There is a collection of data, and each piece in the collection is structured the same or similarly.
- Each piece of data is processed similarly. For example, you might loop through the data, processing each piece the same way.
- The data has a natural order, for example, chronological.

Examples

The following example shows the output of a DESC TABLE command on a table with VARIANT, ARRAY, and OBJECT data.

```
  arr ARRAY,
  obj OBJECT);

DESC TABLE test_semi_structured;
```

```
+------+---------+--------+-------+---------+-------------+------------+-------+------------+---------
| name | type    | kind   | null? | default | primary key | unique key | check | expression | comment
|------+---------+--------+-------+---------+-------------+------------+-------+------------+---------
| VAR  | VARIANT | COLUMN | Y     | NULL    | N           | N          | NULL  | NULL       | NULL
| ARR  | ARRAY   | COLUMN | Y     | NULL    | N           | N          | NULL  | NULL       | NULL
| OBJ  | OBJECT  | COLUMN | Y     | NULL    | N           | N          | NULL  | NULL       | NULL
+------+---------+--------+-------+---------+-------------+------------+-------+------------+---------
```

This example shows how to load simple values into a table, and what those values look like when you query the table.

Create a table and load the data:

```
CREATE OR REPLACE TABLE demonstration1 (
  ID INTEGER,
  array1 ARRAY,
  variant1 VARIANT,
  object1 OBJECT);

INSERT INTO demonstration1 (id, array1, variant1, object1)
  SELECT
    1,
    ARRAY_CONSTRUCT(1, 2, 3),
    PARSE_JSON(' { "key1": "value1", "key2": "value2" } '),
INSERT INTO demonstration1 (id, array1, variant1, object1)
  SELECT
    2,
    ARRAY_CONSTRUCT(1, 2, 3, NULL),
    PARSE_JSON(' { "key1": "value1", "key2": NULL } '),
    PARSE_JSON(' { "outer_key1": { "inner_key1A": "1a", "inner_key1B": NULL }, '
              ||
               '   "outer_key2": { "inner_key2": 2 } '
              ||
              ' } ');
```

Now show the data in the table.

```
SELECT *
  FROM demonstration1
  ORDER BY id;
```

```
+----+-------------+---------------------+-------------------------+
| ID | ARRAY1      | VARIANT1            | OBJECT1                 |
|----+-------------+---------------------+-------------------------|
|  1 | [           | {                   | {                       |
|    |   1,        |   "key1": "value1", |   "outer_key1": {       |
|    |   2,        |   "key2": "value2"  |     "inner_key1A": "1a", |
|    |   3         | }                   |     "inner_key1B": "1b"  |
|    | ]           |                     |   },                    |
|    |             |                     |   "outer_key2": {       |
|    |             |                     |     "inner_key2": 2     |
|    |             |                     |   }                     |
|    |   3,        | }                    |     "inner_key1B": null  |
|    |   undefined |                      |   },                     |
|    | ]           |                      |   "outer_key2": {        |
|    |             |                      |     "inner_key2": 2      |
|    |             |                      |   }                      |
|    |             |                      | }                        |
+----+-------------+----------------------+--------------------------+
```

For additional examples, see Querying Semi-structured Data.