Structured data types

FEATURE — GENERALLY AVAILABLE

Structured types are generally available for Apache Iceberg™ tables.

PREVIEW FEATURE — OPEN

Enabled for all accounts.

Structured types are in preview for standard Snowflake tables (non-Iceberg), views, and materialized views. Structured types aren't supported for dynamic, hybrid, or external tables.

The Snowflake structured types are ARRAY, OBJECT, and MAP. Structured types contain elements or key-value pairs with specific Snowflake data types. The following are examples of structured types:

- An ARRAY of INTEGER elements.
- An OBJECT with VARCHAR and NUMBER key-value pairs.
- A MAP that associates a VARCHAR key with a DOUBLE value.

You can use structured types in the following ways:

In an Apache Iceberg™ table, the Apache Iceberg™ data types `list`, `struct`, and `map` correspond to the structured ARRAY, structured OBJECT, and MAP types in Snowflake.

- You use structured types when accessing data from a structured type column in a table.
- You can cast a semi-structured ARRAY, OBJECT, or VARIANT value to a corresponding structured type (for example, an ARRAY value to an ARRAY value of INTEGER elements). You can also cast a structured type of a semi-structured type.

This topic explains how to use structured types in Snowflake.

Specifying a structured type

When defining a structured type column or casting a value to a structured type, use the syntax described in the following sections:

- Specifying a structured ARRAY type
- Specifying a structured OBJECT type
- Specifying a MAP type

Specifying a structured ARRAY type

To specify a structured ARRAY type, use the following syntax:

Where:

- `<element_type>` is the Snowflake data type of the elements in this ARRAY.

  You can also specify a structured ARRAY, a structured OBJECT, or a MAP as the type of the element.

  Note
  In the definition of a standard Snowflake table (non-Iceberg) column, you can't specify GEOGRAPHY as the type of the ARRAY element.

  In the definition of an Iceberg table column, you can't specify VARIANT, semi-structured ARRAY, or semi-structured OBJECT as the type of the ARRAY element.

- NOT NULL specifies that the ARRAY can't contain any elements that are NULL.

For example, compare the types returned by the SYSTEM$TYPEOF function in the following statement:

- The first column expression casts a semi-structured ARRAY value to a structured ARRAY value (an ARRAY of NUMBER elements).
- The second column expression specifies a semi-structured ARRAY value.

```
SELECT
  SYSTEM$TYPEOF(
    [1, 2, 3]::ARRAY(NUMBER)
  ) AS structured_array,
  SYSTEM$TYPEOF(
    [1, 2, 3]
```

```
+--------------------------------+------------------------+
| STRUCTURED_ARRAY               | SEMI_STRUCTURED_ARRAY  |
|--------------------------------+------------------------|
| ARRAY(NUMBER(38,0))[LOB]       | ARRAY[LOB]             |
+--------------------------------+------------------------+
```

Specifying a structured OBJECT type

To specify a structured OBJECT type, use the following syntax:

```
OBJECT(
  [
    <key> <value_type> [ NOT NULL ]
    [ , <key> <value_type> [ NOT NULL ] ]
    [ , ... ]
  ]
)
```

Where:

- `<key>` specifies a key for the OBJECT type.
  - Each `<key>` in an OBJECT definition must be unique.
  - The order of the keys is part of the OBJECT definition. Comparing two OBJECT values that have the same keys in a different order isn't allowed. (A compile-time error occurs.)

OBJECT.

- `<value_type>` is the Snowflake data type of the value corresponding to the key.

  You can also specify a structured ARRAY, a structured OBJECT, or a MAP as the type of the value.

  Note
  In the definition of a standard Snowflake table (non-Iceberg) column, you can't specify GEOGRAPHY as the type of the value corresponding to the OBJECT key.

  In the definition of an Iceberg table column, you can't specify VARIANT, semi-structured ARRAY, or semi-structured OBJECT as the type of the value corresponding to the OBJECT key.

- NOT NULL specifies that the value corresponding to the key can't be NULL.

For example, compare the types returned by the SYSTEM$TYPEOF function in the following statement:

- The first column expression casts a semi-structured OBJECT value to a structured OBJECT value that contains the following keys and values:
  - A key named `str` with a VARCHAR value that is not NULL.
  - A key named `num` with a NUMBER value.
- The second column expression specifies a semi-structured OBJECT value.

```
SELECT
  SYSTEM$TYPEOF(
    {
```

```
      num NUMBER
    )
  ) AS structured_object,
  SYSTEM$TYPEOF(
    {
      'str': 'test',
      'num': 1
    }
  ) AS semi_structured_object;
```

```
+-------------------------------------------------------------------+-------------------------+
| STRUCTURED_OBJECT                                                 | SEMI_STRUCTURED_OBJECT  |
|-------------------------------------------------------------------+-------------------------|
| OBJECT(str VARCHAR(16777216) NOT NULL, num NUMBER(38,0))[LOB] | OBJECT[LOB]             |
+-------------------------------------------------------------------+-------------------------+
```

Specifying a MAP type

To specify a MAP type, use the following syntax:

```
MAP( <key_type> , <value_type> [ NOT NULL ] )
```

Where:

- *<key_type>* is the Snowflake data type of the key for the map. You must use one of the following types for keys:

You can't use a floating point data type as the type for the key.

Map keys can't be NULL.

- `<value_type>` is the Snowflake data type of the values in the map.

  You can also specify a structured ARRAY, a structured OBJECT, or a MAP as the type of the values.

  Note

  In the definition of a standard Snowflake table (non-Iceberg) column, you can't specify GEOGRAPHY as the type of the value in the MAP.

  In the definition of an Iceberg table column, you can't specify VARIANT, semi-structured ARRAY, or semi-structured OBJECT as the type of the value in the MAP.

- NOT NULL specifies that the value corresponding to the key can't be NULL.

The following example casts a semi-structured OBJECT value to a MAP value and uses the SYSTEM$TYPEOF function to print the resulting type of the value. The MAP associates VARCHAR keys with VARCHAR values.

```
SELECT
  SYSTEM$TYPEOF(
    {
      'a_key': 'a_val',
      'b_key': 'b_val'
    }::MAP(VARCHAR, VARCHAR)
  ) AS map_example;
```

```
|-------------------------------------------------|
| MAP(VARCHAR(16777216), VARCHAR(16777216))[LOB] |
+-------------------------------------------------+
```

Creating a table with a structured type column

When you use the CREATE TABLE command to create a table, you can use the syntax described in Specifying a structured type to define a column that contains a structured type.

The following examples demonstrate how to specify a structured type column:

- Example of creating a table with a structured ARRAY column
- Example of creating a table with a structured OBJECT column
- Example of creating a table with a MAP column

Example of creating a table with a structured ARRAY column

The following statement creates a table with a column for a structured ARRAY:

```
CREATE TABLE my_table_with_structured_array_column (
  numeric_array ARRAY(NUMBER)
);
```

```
INSERT INTO my_table_with_structured_array_column SELECT
  [10, 20, 30]::ARRAY(NUMBER);
```

Note the following:

- Because the example uses an ARRAY constant for the value to insert, the example uses a query (SELECT) rather than the VALUES clause.

  The VALUES clause does not support OBJECT constants, ARRAY constants, and some functions like OBJECT_CONSTRUCT and ARRAY_CONSTRUCT.
- Because an ARRAY constant specifies a semi-structured ARRAY (not a structured ARRAY), you must cast the resulting semi-structured ARRAY to a structured ARRAY.

Example of creating a table with a structured OBJECT column

The following statement creates a table with a column for a structured OBJECT:

```
CREATE TABLE customer (
  c_id VARCHAR,
  c_name VARCHAR,
  c_address OBJECT(
    state VARCHAR,
    city VARCHAR,
    street VARCHAR,
    zip_code NUMBER
```

The following statement inserts a row into the table:

```
INSERT INTO customer SELECT
  '1',
  'customer_name',
  {
    'state': 'CA',
    'city': 'San Mateo',
    'street': '450 Concar Drive',
    'zip_code': 94402
  }::OBJECT(
    state VARCHAR,
    city VARCHAR,
    street VARCHAR,
    zip_code NUMBER
  );
```

Note the following:

- Because the example uses an OBJECT constant for the value to insert, the example uses a query (SELECT) rather than the VALUES clause.

  The VALUES clause does not support OBJECT constants, ARRAY constants, and some functions like OBJECT_CONSTRUCT and ARRAY_CONSTRUCT.
- Because an OBJECT constant specifies a semi-structured OBJECT (not a structured OBJECT), you must cast the resulting semi-structured OBJECT to a structured OBJECT.

```
CREATE OR REPLACE TABLE my_table_with_map_column(my_map MAP(VARCHAR, VARCHAR));
```

The following statement inserts a row into the table:

```
INSERT INTO my_table_with_map_column SELECT
  {'key123': 'value123'}::MAP(VARCHAR, VARCHAR);
```

Note the following:

- Because the example uses an OBJECT constant for the value to insert, the example uses a query (SELECT) rather than the VALUES clause.

  The VALUES clause does not support OBJECT constants, ARRAY constants, and some functions like OBJECT_CONSTRUCT and ARRAY_CONSTRUCT.
- Because an OBJECT constant specifies a semi-structured OBJECT (not a MAP), you must cast the resulting semi-structured OBJECT to a MAP.

Adding a structured type column

To add a column containing a structured type, use ALTER TABLE … ADD COLUMN with the syntax described in Specifying a structured type. For example:

Dropping and renaming structured type columns

To drop or rename a structured type column, you can use ALTER TABLE ... DROP COLUMN and ALTER TABLE ... RENAME COLUMN (as you would with a column with a semi-structured object).

Using structured types in semi-structured types

You can't use a MAP, structured OBJECT, or structured ARRAY value in a VARIANT, semi-structured OBJECT, or semi-structured ARRAY value. An error occurs in the following situations:

- You use a MAP, structured OBJECT, or structured ARRAY value in an OBJECT constant or ARRAY constant.
- You pass a MAP, structured OBJECT, or structured ARRAY value to an OBJECT or ARRAY constructor function.

Converting structured and semi-structured types

The following table summarizes rules for converting structured OBJECT, structured ARRAY, and MAP values to semi-structured OBJECT, ARRAY, and VARIANT values (and vice versa).

Source data type	Target data type	Castable	Coercible
Semi-structured ARRAY	Structured ARRAY	✓	✗
Semi-structured OBJECT	• Structured OBJECT	✓	✗

Source data type	Target data type	Castable	Coercible
Semi-structured VARIANT	• Structured ARRAY • Structured OBJECT • MAP	✓	✗
Structured ARRAY	Semi-structured ARRAY	✓	✗
• Structured OBJECT • MAP	Semi-structured OBJECT	✓	✗
• Structured ARRAY • Structured OBJECT • MAP	Semi-structured VARIANT	✓	✗

The following sections explain these rules in more detail.

- Explicitly casting a semi-structured type to a structured type
- Explicitly casting a structured type to a semi-structured type
- Implicit casting a value (coercion)
- Casting from one structured type to another

Explicitly casting a semi-structured type to a structured type

To explicitly cast a value of a semi-structured type to a value of a structured type, you can call the CAST function or use the :: operator.

> **Note**
> TRY_CAST isn't supported for structured types.

You can only cast values of the following semi-structured types to values of the corresponding structured type; otherwise, a runtime error occurs.

Semi-structured type	Structured type that you can cast to
ARRAY	Structured ARRAY
OBJECT	MAP or structured OBJECT
VARIANT	MAP or structured ARRAY or OBJECT

The next sections provide more detail about how the types are cast:

- Casting semi-structured ARRAY and VARIANT values to structured ARRAY values
- Casting semi-structured OBJECT and VARIANT values to structured OBJECT values
- Casting semi-structured OBJECT and VARIANT values to MAP values

The following steps demonstrate how to cast a semi-structured ARRAY or VARIANT value to an ARRAY value of NUMBER elements:

```
SELECT
  SYSTEM$TYPEOF(
    CAST ([1,2,3] AS ARRAY(NUMBER))
  ) AS array_cast_type,
  SYSTEM$TYPEOF(
    CAST ([1,2,3]::VARIANT AS ARRAY(NUMBER))
  ) AS variant_cast_type;
```

Or:

```
SELECT
  SYSTEM$TYPEOF(
    [1,2,3]::ARRAY(NUMBER)
  ) AS array_cast_type,
  SYSTEM$TYPEOF(
    [1,2,3]::VARIANT::ARRAY(NUMBER)
  ) AS variant_cast_type;
```

```
+--------------------------+--------------------------+
| ARRAY_CAST_TYPE          | VARIANT_CAST_TYPE        |
|--------------------------+--------------------------|
| ARRAY(NUMBER(38,0))[LOB] | ARRAY(NUMBER(38,0))[LOB] |
+--------------------------+--------------------------+
```

- Each element of the ARRAY value is cast to the specified type of the ARRAY.

  Casting the ARRAY column to ARRAY(VARCHAR) converts each value to a VARCHAR value:

```
SELECT
  CAST ([1,2,3] AS ARRAY(VARCHAR)) AS varchar_array,
  SYSTEM$TYPEOF(varchar_array) AS array_cast_type;
```

```
+---------------+--------------------------------+
| VARCHAR_ARRAY | ARRAY_CAST_TYPE                |
|---------------+--------------------------------|
| [             | ARRAY(VARCHAR(16777216))[LOB] |
|   "1",        |                                |
|   "2",        |                                |
|   "3"         |                                |
| ]             |                                |
+---------------+--------------------------------+
```

- If the element can't be cast to the specified type (for example, casting `['a', 'b', 'c']` to ARRAY(NUMBER)), the cast fails.
- If the ARRAY value contains NULL elements and the ARRAY type specifies NOT NULL (for example, casting `[1, NULL, 3]` to ARRAY(NUMBER NOT NULL), the cast fails.
- Elements that are JSON null values are converted to NULL, if the target element type doesn't support JSON nulls (that is, the target type isn't a semi-structured ARRAY, OBJECT, or VARIANT).

  For example, if you are casting to ARRAY(NUMBER), JSON null values are converted to NULL because NUMBER doesn't support JSON nulls.

Casting semi-structured OBJECT and VARIANT values to structured OBJECT values

The following steps demonstrate how to cast a semi-structured OBJECT or VARIANT value to a structured OBJECT value containing the `city` and `state` key-value pairs (which are VARCHAR values):

```
SELECT
  SYSTEM$TYPEOF(
    CAST ({'city':'San Mateo','state':'CA'} AS OBJECT(city VARCHAR, state VARCHAR))
  ) AS object_cast_type,
  SYSTEM$TYPEOF(
    CAST ({'city':'San Mateo','state':'CA'}::VARIANT AS OBJECT(city VARCHAR, state VARCHAR))
  ) AS variant_cast_type;
```

Or:

```
SELECT
  SYSTEM$TYPEOF(
     {'city':'San Mateo','state':'CA'}::OBJECT(city VARCHAR, state VARCHAR)
  ) AS object_cast_type,
  SYSTEM$TYPEOF(
     {'city':'San Mateo','state':'CA'}::VARIANT::OBJECT(city VARCHAR, state VARCHAR)
  ) AS variant_cast_type;
```

```
+-------------------------------------------------------------+-------------------------------------------
| OBJECT_CAST_TYPE                                            | VARIANT_CAST_TYPE
|-------------------------------------------------------------+-------------------------------------------
```

When you cast a semi-structured OBJECT or VARIANT value to a structured OBJECT value, note the following:

- The OBJECT value can't contain any additional keys that aren't specified in the OBJECT type.

  If there are additional keys, the cast fails.
- If the OBJECT value is missing a key that is specified in the OBJECT type, the cast fails.
- The value of each key in the OBJECT value is converted to the specified type for that key.

  If a value can't be cast to the specified type, the cast fails.
- If the value for a key is a JSON null value, the value is converted to NULL when the target value type doesn't support JSON nulls (that is, the target type is not a semi-structured ARRAY, OBJECT, or VARIANT).

  For example, if you are casting to OBJECT(city VARCHAR), JSON null values are converted to NULL because VARCHAR doesn't support JSON nulls.

  On the other hand, if you are casting to OBJECT(city VARIANT), JSON null values aren't converted to NULL because VARIANT supports JSON nulls.

Casting semi-structured OBJECT and VARIANT values to MAP values

The following statements demonstrate how to cast a semi-structured OBJECT or VARIANT value to a MAP value that associates a VARCHAR key with a VARCHAR value:

```
SELECT
  SYSTEM$TYPEOF(
    CAST ({'my_key':'my_value'} AS MAP(VARCHAR, VARCHAR))
  ) AS map_cast_type,
  SYSTEM$TYPEOF(
```

Or:

```
SELECT
  SYSTEM$TYPEOF(
    {'my_key':'my_value'}::MAP(VARCHAR, VARCHAR)
  ) AS map_cast_type,
  SYSTEM$TYPEOF(
    {'my_key':'my_value'}::VARIANT::MAP(VARCHAR, VARCHAR)
  ) AS variant_cast_type;
```

```
+-------------------------------------------------+-------------------------------------------------+
| MAP_CAST_TYPE                                   | VARIANT_CAST_TYPE                               |
|-------------------------------------------------+-------------------------------------------------|
| MAP(VARCHAR(16777216), VARCHAR(16777216))[LOB] | MAP(VARCHAR(16777216), VARCHAR(16777216))[LOB] |
+-------------------------------------------------+-------------------------------------------------+
```

When you cast a semi-structured OBJECT or VARIANT value to a MAP value, note the following:

- If the keys and values do not match the specified types, the keys and values are converted to the specified types.
- If the keys and values can't be cast to the specified types, the cast fails.
- If the value for a key is a JSON null value, the value is converted to NULL when the target value type doesn't support JSON nulls (that is, the target type is not a semi-structured ARRAY, OBJECT, or VARIANT).

  For example, if you are casting to MAP(VARCHAR, VARCHAR), JSON null values are converted to NULL because VARCHAR doesn't support JSON nulls.

  On the other hand, if you are casting to MAP(VARCHAR, VARIANT), JSON null values aren't converted to NULL

Explicitly casting a structured type to a semi-structured type

To explicitly cast a value of a structured type to a value of a semi-structured type, you can call the CAST function, use the :: operator, or call one of the conversion functions (for example, TO_ARRAY, TO_OBJECT, or TO_VARIANT).

Note
TRY_CAST isn't supported with structured types.

Structured type	Semi-structured type that you can cast to
Structured ARRAY	ARRAY
MAP or structured OBJECT	OBJECT
MAP, structured ARRAY, or structured OBJECT	VARIANT

For example:

- If `col_structured_array` is ARRAY(VARCHAR) type:
  - CAST(col_structured_array AS ARRAY) returns a semi-structured ARRAY value.
  - CAST(col_structured_array AS VARIANT) returns a VARIANT value that holds a semi-structured ARRAY value.
- If `col_structured_object` is OBJECT(name VARCHAR, state VARCHAR) type:
  - CAST(col_structured_object AS OBJECT) returns a semi-structured OBJECT value.
  - CAST(col_structured_object AS VARIANT) returns a VARIANT value that holds a semi-structured OBJECT value.

- CAST(col_map AS VARIANT) returns a VARIANT value that holds a semi-structured OBJECT value.

Note the following:

- When you are casting to a semi-structured OBJECT value, the order of keys in the structured OBJECT value isn't preserved.
- When you are casting a structured OBJECT or MAP value to a semi-structured OBJECT or VARIANT value, any NULL values are converted to JSON null values.

  If you are casting a structured ARRAY value to a VARIANT value, NULL values are preserved as is.

```
SELECT [1,2,NULL,3]::ARRAY(INTEGER)::VARIANT;
```

```
+---------------------------------------+
| [1,2,NULL,3]::ARRAY(INTEGER)::VARIANT |
|---------------------------------------|
| [                                     |
|   1,                                  |
|   2,                                  |
|   undefined,                          |
|   3                                   |
| ]                                     |
+---------------------------------------+
```

- If you are casting a MAP value that uses a NUMBER type for keys, the MAP keys are converted to strings in the returned OBJECT value.

The following rules apply to implicitly casting (coercion) from a value of one structured type to a value of another structured type:

- A structured type value can be coerced to another structured type value if the two basic types are the same:
  - An ARRAY value of one type can be coerced to an ARRAY value of another type, provided that the first element type is coercible to the second element type.

    An element type can be coerced to another element type in either of the following cases:

    - Both types are numeric. The following cases are supported:
      - Both use the same numeric type but possibly differ in precision and/or scale.
      - Coercing NUMBER to FLOAT (and vice versa).
    - Both types are timestamps. The following cases are supported:
      - Both use the same type but possibly differ in precision.
      - Coercing TIMESTAMP_LTZ to TIMESTAMP_TZ (and vice versa).

    For example:

    - An ARRAY(NUMBER) value can be coerced to an ARRAY(DOUBLE) value.
    - An ARRAY(DATE) value can't be coerced to an ARRAY(NUMBER) value.
  - An OBJECT value with one type definition can be coerced to an OBJECT value of with another type definition only if all of the following are true:
    - Both OBJECT types have the same number of keys.
    - Both OBJECT types use the same names for keys.
    - The keys in both OBJECT types are in the same order.
    - The type of each value in one OBJECT type can be coerced to the type of the corresponding value in the

- Both types are numeric. The following cases are supported:
  - Both use the same numeric type but possibly differ in precision and/or scale.
  - Coercing NUMBER to FLOAT (and vice versa).
- Both types are timestamps. The following cases are supported:
  - Both use the same type but possibly differ in precision.
  - Coercing TIMESTAMP_LTZ to TIMESTAMP_TZ (and vice versa).

For example:

- An OBJECT(city VARCHAR, zipcode NUMBER) value can be coerced to an OBJECT(city VARCHAR, zipcode DOUBLE) value.
- An OBJECT(city VARCHAR, zipcode NUMBER) value can't be coerced to an OBJECT(city VARCHAR, zipcode DATE) value.

- A MAP value with one value type can be coerced to a MAP value with a different value type if:
  - Both value types are numeric. The following cases are supported:
    - Both use the same numeric type but possibly differ in precision and/or scale.
    - Coercing NUMBER to FLOAT (and vice versa).
  - Both value types are timestamps. The following cases are supported:
    - Both use the same type but possibly differ in precision.
    - Coercing TIMESTAMP_LTZ to TIMESTAMP_TZ (and vice versa).

  For example, a MAP(VARCHAR, NUMBER) value can be coerced to a MAP(VARCHAR, DOUBLE) value.
- A MAP value with one key type can be coerced to a MAP value with a different key type if both key types use the same integer NUMERIC type that differ only in precision.

  For example, a MAP(VARCHAR, NUMBER) value can't be coerced to a MAP(NUMBER, NUMBER) value.

Casting from one structured type to another

You can call the CAST function or use the :: operator to cast from a value of one structured type to a value of another structured type. You can cast values from and to the following structured types:

- For structured ARRAYs:

  You can cast an ARRAY value of one type to an ARRAY value of another type.
- For structured OBJECTs:
  - You can use a cast to change the order of key-value pairs in an OBJECT value.
  - You can use a cast to change the names of the keys in an OBJECT value.
  - You can use a cast to add keys to an OBJECT value.
  - You can cast a structured OBJECT value to a MAP value.
- For MAP values:
  - You can cast a MAP value with keys and values of a specific type to a MAP value with keys and values of a different type.
  - You can cast a MAP value to a structured OBJECT value.

Note
TRY_CAST isn't supported with structured types.

If it isn't possible to cast the values from one type to the other, the cast fails. For example, attempting to cast an ARRAY(BOOLEAN) value to an ARRAY(DATE) value fails.

The following example casts an ARRAY(NUMBER) value to an ARRAY(VARCHAR) value:

```
SELECT CAST(
  CAST([1,2,3] AS ARRAY(NUMBER))
  AS ARRAY(VARCHAR)) AS cast_array;
```

```
+------------+
| CAST_ARRAY |
|------------|
| [          |
|   "1",     |
|   "2",     |
|   "3"      |
| ]          |
+------------+
```

Example: Changing the order of key-value pairs in an OBJECT value

The following example changes the order of key-value pairs in a structured OBJECT value:

```
SELECT CAST(
  {'city': 'San Mateo','state': 'CA'}::OBJECT(city VARCHAR, state VARCHAR)
  AS OBJECT(state VARCHAR, city VARCHAR)) AS object_value_order;
```

```
+-----------------------+
| OBJECT_VALUE_ORDER    |
|-----------------------|
| {                     |
|   "state": "CA",      |
|   "city": "San Mateo" |
| }                     |
+-----------------------+
```

Example: Changing the key names in an OBJECT value

To change the key names in a structured OBJECT value, specify the RENAME FIELDS keywords at the end of CAST. For example:

```
SELECT CAST({'city':'San Mateo','state': 'CA'}::OBJECT(city VARCHAR, state VARCHAR)
  AS OBJECT(city_name VARCHAR, state_name VARCHAR) RENAME FIELDS) AS object_value_key_names;
```

```
+-----------------------------+
| OBJECT_VALUE_KEY_NAMES      |
|-----------------------------|
| {                           |
|   "city_name": "San Mateo", |
|   "state_name": "CA"        |
| }                           |
```

Example: Adding keys to an OBJECT value

If the type that you are casting to has additional key-value pairs that aren't present in the original structured OBJECT value, specify the ADD FIELDS keywords at the end of CAST. For example:

```
SELECT CAST({'city':'San Mateo','state': 'CA'}::OBJECT(city VARCHAR, state VARCHAR)
  AS OBJECT(city VARCHAR, state VARCHAR, zipcode NUMBER) ADD FIELDS) AS add_fields;
```

```
+-------------------------+
| ADD_FIELDS              |
|-------------------------|
| {                       |
|   "city": "San Mateo",  |
|   "state": "CA",        |
|   "zipcode": null       |
| }                       |
+-------------------------+
```

The values for the newly added keys are set to NULL. If you want to assign a value to these keys, call the OBJECT_INSERT function instead.

Constructing structured ARRAY, structured OBJECT, and MAP values

- Using ARRAY and OBJECT constants to construct structured ARRAY and OBJECT values
- Constructing a MAP value

Using SQL functions to construct structured ARRAY and OBJECT values

The following functions construct semi-structured ARRAY values:

- ARRAY_CONSTRUCT
- ARRAY_CONSTRUCT_COMPACT
- ARRAY_AGG
- TO_ARRAY

The following functions construct semi-structured OBJECT values:

- OBJECT_CONSTRUCT
- OBJECT_CONSTRUCT_KEEP_NULL
- OBJECT_AGG
- TO_OBJECT

To construct a structured ARRAY or OBJECT value, use these functions and explicitly cast the return value of the function. For example:

```
SELECT OBJECT_CONSTRUCT(
  'oname', 'abc',
  'created_date', '2020-01-18'::DATE
)::OBJECT(
  oname VARCHAR,
  created_date DATE
);
```

For details, refer to Explicitly casting a semi-structured type to a structured type.

Note

You can't pass structured ARRAY, structured OBJECT, or MAP values to these functions. Doing so would result in a structured type being implicitly cast to a semi-structured type, which isn't allowed, as noted in Implicit casting a value (coercion).

Using ARRAY and OBJECT constants to construct structured ARRAY and OBJECT values

When you specify an ARRAY constant or an OBJECT constant, you are specifying a semi-structured ARRAY or OBJECT value.

To construct a structured ARRAY or OBJECT value, you must explicitly cast the expression. For example:

```
```

```
SELECT {
  'oname': 'abc',
  'created_date': '2020-01-18'::DATE
}::OBJECT(
  oname VARCHAR,
  created_date DATE
);
```

For details, refer to Explicitly casting a semi-structured type to a structured type.

Constructing a MAP value

To construct a MAP value, construct a semi-structured OBJECT value, and cast the OBJECT value to a MAP value.

For example, the following statements both produce the MAP value `{'city'->'San Mateo','state'->'CA'}`:

```
SELECT OBJECT_CONSTRUCT(
  'city', 'San Mateo',
  'state', 'CA'
)::MAP(
  VARCHAR,
  VARCHAR
);
```

```
}::MAP(
  VARCHAR,
  VARCHAR
);
```

The following statement produces the MAP value `{-10->'CA',-20->'OR'}`:

```
SELECT {
  '-10': 'CA',
  '-20': 'OR'
}::MAP(
  NUMBER,
  VARCHAR
);
```

For details, refer to Casting semi-structured OBJECT and VARIANT values to MAP values.

Working with keys, values, and elements in values of structured types

The following sections explain how to use keys, values, and elements in values of structured types.

- Getting the list of keys from a structured OBJECT value
- Getting the list of keys from a MAP value
- Accessing values and elements from values of structured types

- Looking up elements in a structured ARRAY value
- Determining if a MAP value contains a key

Getting the list of keys from a structured OBJECT value

To get the list of keys in a structured OBJECT value, call the OBJECT_KEYS function:

```
SELECT OBJECT_KEYS({'city':'San Mateo','state':'CA'}::OBJECT(city VARCHAR, state VARCHAR));
```

If the input is a structured OBJECT value, the function returns an ARRAY(VARCHAR) value containing the keys. If the input is a semi-structured OBJECT value, the function returns an ARRAY value.

Getting the list of keys from a MAP value

To get the list of keys in a MAP value, call the MAP_KEYS function:

```
SELECT MAP_KEYS({'my_key':'my_value'}::MAP(VARCHAR,VARCHAR));
```

Accessing values and elements from values of structured types

- The GET function
- The GET_IGNORE_CASE function
- The GET_PATH function
- Dot Notation
- Bracket Notation

The returned values and elements have the type specified for the structured value, rather than VARIANT.

The following example passes the first element of a semi-structured ARRAY value and an ARRAY(VARCHAR) value to the SYSTEM$TYPEOF function to return the data type of that element:

```
SELECT
  SYSTEM$TYPEOF(
    ARRAY_CONSTRUCT('San Mateo')[0]
  ) AS semi_structured_array_element,
  SYSTEM$TYPEOF(
    CAST(
      ARRAY_CONSTRUCT('San Mateo') AS ARRAY(VARCHAR)
    )[0]
  ) AS structured_array_element;
```

```
+--------------------------------+-------------------------------+
| SEMI_STRUCTURED_ARRAY_ELEMENT | STRUCTURED_ARRAY_ELEMENT     |
|--------------------------------+-------------------------------|
| VARIANT[LOB]                   | VARCHAR(16777216)[LOB]       |
```

- When you pass a structured OBJECT value to the GET or GET_IGNORE_CASE function, you must specify a constant for the key.

  You don't need to specify a constant if you are passing a MAP or structured ARRAY value to the GET function.

  You also don't need to specify a constant if you are passing a MAP value to the GET_IGNORE_CASE function.
- When you pass a structured OBJECT, structured ARRAY, or MAP value to the GET_PATH function, you must specify a constant for the path name.
- For a structured OBJECT value, if you use an OBJECT key or a path that doesn't exist, a compile-time error occurs.

  In contrast, when you use an index, key, or path that doesn't exist with a semi-structured OBJECT value, the function returns NULL.

Determining the size of a structured ARRAY value

To determine the size of a structured ARRAY value, pass the ARRAY value to the ARRAY_SIZE function:

```
SELECT ARRAY_SIZE([1,2,3]::ARRAY(NUMBER));
```

Determining the size of a MAP value

To determine the size of a MAP value, pass the MAP value to MAP_SIZE function:

Looking up elements in a structured ARRAY value

To determine if an element is present in a structured ARRAY value, call the ARRAY_CONTAINS function. For example:

```
SELECT ARRAY_CONTAINS(10, [1, 10, 100]::ARRAY(NUMBER));
```

To determine the position of an element in a structured ARRAY value, call the ARRAY_POSITION function. For example:

```
SELECT ARRAY_POSITION(10, [1, 10, 100]::ARRAY(NUMBER));
```

Note

For both functions, use an element of a type that is comparable to the type of the ARRAY value.

Don’t cast the expression for the element to a VARIANT value.

Determining if a MAP value contains a key

To determine if a MAP value contains a key, call the MAP_CONTAINS_KEY function:

For example:

```
SELECT MAP_CONTAINS_KEY(10, my_map);
```

Comparing values

The following sections explain how to compare values:

- Comparing structured values with semi-structured values
- Comparing structured values with other structured values
- Determining if two ARRAY values overlap

Comparing structured values with semi-structured values

You can't compare a structured ARRAY, structured OBJECT, or MAP value with a semi-structured ARRAY, OBJECT, or VARIANT value.

Comparing structured values with other structured values

You can compare two values of the same type (for example, two structured ARRAY values, two structured OBJECT values, or two MAP values).

- `!=`
- `<`
- `<=`
- `>=`
- `>`

When you compare two structured values for equality, note the following:

- If one type can't be coerced to the other type, the comparison fails.
- When you compare MAP values that have numeric keys, the keys are compared as numbers (not as VARCHAR values).

When you compare two structured values using `<`, `<=`, `>=`, or `>`, the structured value fields are compared in alphabetical order. For example, the following value:

```
{'a':2,'b':1}::OBJECT(b INTEGER,a INTEGER)
```

is greater than:

```
{'a':1,'b':2}::OBJECT(b INTEGER,a INTEGER)
```

```
SELECT ARRAYS_OVERLAP(numeric_array, other_numeric_array);
```

The ARRAY values must be of comparable types.

You can't pass a semi-structured ARRAY value and a structured ARRAY value to this function. Both ARRAY values must either be structured or semi-structured.

Transforming values of structured types

The following sections explain how to transform structured ARRAY, structured OBJECT, and MAP values:

- Transforming structured ARRAY values
- Transforming structured OBJECT values
- Transforming MAP values

Transforming structured ARRAY values

When you pass a structured ARRAY value to these functions, the functions return a structured ARRAY value of the same type:

- ARRAY_APPEND
- ARRAY_CAT

- ARRAY_INSERT
- ARRAY_INTERSECTION
- ARRAY_PREPEND
- ARRAY_SLICE
- ARRAY_UNION_AGG

The next sections explain how these functions work with structured ARRAY values.

- Functions that add elements to ARRAY values
- Functions that accept multiple ARRAY values as input

Functions that add elements to ARRAY values

The following functions add elements to an ARRAY values:

- ARRAY_APPEND
- ARRAY_INSERT
- ARRAY_PREPEND

For these functions, the type of the element must be coercible to the type of the ARRAY value.

For example, the following call succeeds because a NUMBER value can be coerced to a DOUBLE value (the type of the ARRAY value):

```
SELECT ARRAY_APPEND( [1,2]::ARRAY(DOUBLE), 3::NUMBER );
```

The following call succeeds because VARCHAR values can be coerced to DOUBLE values:

```
SELECT ARRAY_APPEND( [1,2]::ARRAY(DOUBLE), '3' );
```

The following call fails because DATE values can't be coerced to NUMBER values:

```
SELECT ARRAY_APPEND( [1,2]::ARRAY(NUMBER), '2022-02-02'::DATE );
```

Functions that accept multiple ARRAY values as input

The following functions accept multiple ARRAY values as input arguments:

- ARRAY_CAT
- ARRAY_EXCEPT
- ARRAY_INTERSECTION

When you call these functions, both arguments must either be structured ARRAY values or semi-structured ARRAY values. For example, the following calls fail because one argument is a structured ARRAY value and the other argument is a semi-

```
SELECT ARRAY_CAT( [1,2]::ARRAY(NUMBER), ['3','4'] );
```

```
SELECT ARRAY_CAT( [1,2], ['3','4']::ARRAY(VARCHAR) );
```

The ARRAY_EXCEPT function returns an ARRAY value of the same type as the ARRAY value in the first argument.

The ARRAY_CAT and ARRAY_INTERSECTION functions return an ARRAY value of a type that can accommodate the types of both input values.

For example, the following call to ARRAY_CAT passes in two structured ARRAY values:

- The first structured ARRAY value doesn't allow NULLs and contains NUMBER values with the scale of 0 (NUMBER(38, 0)).
- The second structured ARRAY value contains a NULL and a NUMBER value that has the scale of 1.

The ARRAY value returned by ARRAY_CAT allows NULLs and contains NUMBER values with the scale of 1.

```
SELECT
  ARRAY_CAT(
    [1, 2, 3]::ARRAY(NUMBER NOT NULL),
    [5.5, NULL]::ARRAY(NUMBER(2, 1))
  ) AS concatenated_array,
  SYSTEM$TYPEOF(concatenated_array);
```

```
+--------------------+-------------------------------------+
| CONCATENATED_ARRAY | SYSTEM$TYPEOF(CONCATENATED_ARRAY)   |
|--------------------+-------------------------------------|
| [                  | ARRAY(NUMBER(38,1))[LOB]            |
|   1,               |                                     |
|   2,               |                                     |
|   3,               |                                     |
|   5.5,             |                                     |
|   undefined        |                                     |
| ]                  |                                     |
+--------------------+-------------------------------------+
```

For the ARRAY_CAT function, the ARRAY value in the second argument must be coercible to the type in the first argument.

For the ARRAY_EXCEPT and ARRAY_INTERSECTION functions, the ARRAY value in the second argument must be comparable to the ARRAY value in the first argument.

For example, the following call succeeds because an ARRAY(NUMBER) value is comparable to an ARRAY(DOUBLE) value:

```
SELECT ARRAY_EXCEPT( [1,2]::ARRAY(NUMBER), [2,3]::ARRAY(DOUBLE) );
```

The following call fails because an ARRAY(NUMBER) value isn't comparable to an ARRAY(VARCHAR) value:

```
SELECT ARRAY_EXCEPT( [1,2]::ARRAY(NUMBER), ['2','3']::ARRAY(VARCHAR) );
```

Transforming structured OBJECT values

The following sections explain how to return a structured OBJECT value that has been transformed from another OBJECT value:

- Removing key-value pairs
- Inserting key-value pairs and updating values
- Selecting key-value pairs from an existing OBJECT

To change the order of key-value pairs, rename keys, or add keys without specifying values, use the CAST function or :: operator. For details, see Casting from one structured type to another.

Removing key-value pairs

To return a new OBJECT value that contains the key-value pairs from an existing OBJECT value with specific key-value pairs removed, call the OBJECT_DELETE function.

When calling this function, note the following:

- For the arguments that are keys, you must specify constants.
- If the specified key isn't part of the OBJECT type definition, the call fails. For example, the following call fails because the OBJECT value doesn't contain the specified key `zip_code`:

```
SELECT OBJECT_DELETE( {'city':'San Mateo','state':'CA'}::OBJECT(city VARCHAR,state VARCHAR), 'zip_
```

```
093201 (23001): Function OBJECT_DELETE: expected structured object to contain field zip_code but i
```

- The function returns a structured OBJECT value. The type of the OBJECT value excludes the deleted key. For example, suppose that you remove the `city` key:

```
SELECT
  OBJECT_DELETE(
    {'city':'San Mateo','state':'CA'}::OBJECT(city VARCHAR,state VARCHAR),
    'city'
  ) AS new_object,
  SYSTEM$TYPEOF(new_object);
```

  The function returns an OBJECT value of the type `OBJECT(state VARCHAR)`, which doesn't include the `city` key.

```
+-----------------+----------------------------------------+
| NEW_OBJECT      | SYSTEM$TYPEOF(NEW_OBJECT)              |
|-----------------+----------------------------------------|
| {               | OBJECT(state VARCHAR(16777216))[LOB] |
|   "state": "CA" |                                        |
| }               |                                        |
+-----------------+----------------------------------------+
```

- If the function removes all keys from the OBJECT value, the function returns an empty structured OBJECT value of the type `OBJECT()`.

```
SELECT
```

```
  ) AS new_object,
  SYSTEM$TYPEOF(new_object);
```

```
+------------+---------------------------+
| NEW_OBJECT | SYSTEM$TYPEOF(NEW_OBJECT) |
|------------+---------------------------|
| {}         | OBJECT()[LOB]             |
+------------+---------------------------+
```

When the type of a structured OBJECT value includes key-value pairs, the names and types of those pairs are included in parentheses in the type (for example, OBJECT(city VARCHAR)). Because an empty structured OBJECT value contains no key-value pairs, the parentheses are empty.

Inserting key-value pairs and updating values

To return a new OBJECT value that contains the key-value pairs from an existing OBJECT value with additional key-value pairs or new values for keys, call the OBJECT_INSERT function.

When calling this function, note the following:

- For the arguments that are keys, you must specify constants.
- When the `<updateFlag>` argument is FALSE (when you are inserting a new key-value pair):
  - If you specify a key that already exists in the OBJECT value, an error occurs.

```
  false
);
```

```
093202 (23001): Function OBJECT_INSERT:
  expected structured object to not contain field city but it did.
```

- The function returns a structured OBJECT value. The type of the OBJECT value includes the newly inserted key. For example, suppose that you add the `zipcode` key with the FLOAT value `94402`:

```
SELECT
  OBJECT_INSERT(
    {'city':'San Mateo','state':'CA'}::OBJECT(city VARCHAR,state VARCHAR),
    'zip_code',
    94402::FLOAT,
    false
  ) AS new_object,
  SYSTEM$TYPEOF(new_object) AS type;
```

```
+-------------------------------------+-------------------------------------------------------
| NEW_OBJECT                          | TYPE
|-------------------------------------+-------------------------------------------------------
| {                                   | OBJECT(city VARCHAR(16777216), state VARCHAR(16777216)
|   "city": "San Mateo",              |
|   "state": "CA",                    |
|   "zip_code": 9.440200000000000e+04 |
| }                                   |
+-------------------------------------+-------------------------------------------------------
```

The type of the inserted value determines the type added to the OBJECT type definition. In this case, the value for `zipcode` is a value cast to a FLOAT, so the type of `zipcode` is FLOAT.

- When the *`<updateFlag>`* argument is TRUE (when you are replacing an existing key-value pair):
  - If you specify a key that doesn't exist in the OBJECT value, an error occurs.
  - The function returns a structured OBJECT value of the same type.
  - The type of the inserted value is coerced to the type of the existing key.

Selecting key-value pairs from an existing OBJECT

To return a new OBJECT value that contains selected key-value pairs from an existing OBJECT value, call the OBJECT_PICK function.

When calling this function, note the following:

- For the arguments that are keys, you must specify constants.
- You can't pass in an ARRAY of keys as the second argument. You must specify each key as a separate argument.
- The function returns a structured OBJECT value. The type of the OBJECT value includes the keys in the order in which they are specified.

  For example, suppose that you select the `state` and `city` keys in that order:

```
SELECT
  OBJECT_PICK(
    {'city':'San Mateo','state':'CA','zip_code':94402}::OBJECT(city VARCHAR,state VARCHAR,zip_code
    'state',
```

The function returns an OBJECT value of the type `OBJECT(state VARCHAR, city VARCHAR)`.

```
+-----------------------+-----------------------------------------------------------------+
| NEW_OBJECT            | SYSTEM$TYPEOF(NEW_OBJECT)                                       |
|-----------------------+-----------------------------------------------------------------|
| {                     | OBJECT(state VARCHAR(16777216), city VARCHAR(16777216))[LOB] |
|   "state": "CA",      |                                                                 |
|   "city": "San Mateo" |                                                                 |
| }                     |                                                                 |
+-----------------------+-----------------------------------------------------------------+
```

Transforming MAP values

To transform MAP values, use the following functions:

- MAP_CAT
- MAP_DELETE
- MAP_INSERT
- MAP_PICK

Working with structured types

The following sections explain how to use different SQL functions and set operators with values of structured types:

- Using structured types with set operators and CASE expressions
- Working with other semi-structured functions

Using the FLATTEN function with values of structured types

You can pass structured ARRAY, structured OBJECT, and MAP values to the FLATTEN function. As is the case with semi-structured data types, you can use the PATH argument to specify the value being flattened.

- If the value being flattened is a structured ARRAY value and the RECURSIVE argument is FALSE, the `value` column contains a value of the same type as the ARRAY value.

  For example:

```
SELECT value, SYSTEM$TYPEOF(value)
  FROM TABLE(FLATTEN(INPUT => [1.08, 2.13, 3.14]::ARRAY(DOUBLE)));
```

```
+-------+----------------------+
| VALUE | SYSTEM$TYPEOF(VALUE) |
|-------+----------------------|
|  1.08 | FLOAT[DOUBLE]        |
|  2.13 | FLOAT[DOUBLE]        |
|  3.14 | FLOAT[DOUBLE]        |
+-------+----------------------+
```

- If the value being flattened is a MAP value and the RECURSIVE argument is FALSE, the `key` column contains a key of

```
SELECT key, SYSTEM$TYPEOF(key), value, SYSTEM$TYPEOF(value)
  FROM TABLE(FLATTEN(INPUT => {'my_key': 'my_value'}::MAP(VARCHAR, VARCHAR)));
```

```
+--------+-----------------------+----------+-----------------------+
| KEY    | SYSTEM$TYPEOF(KEY)    | VALUE    | SYSTEM$TYPEOF(VALUE)  |
|--------+-----------------------+----------+-----------------------|
| my_key | VARCHAR(16777216)[LOB] | my_value | VARCHAR(16777216)[LOB] |
+--------+-----------------------+----------+-----------------------+
```

- Otherwise, the `key` and `value` columns have the type VARIANT.

For MAP values, the order of keys and values returned is indeterminate.

Using the PARSE_JSON function

The PARSE_JSON function doesn't return structured types.

Using structured types with set operators and CASE expressions

You can use structured ARRAY, structured OBJECT, and MAP values in:

- Query expressions combined by set operators (e.g. UNION ALL).

For set operators, if different types are used in the different expressions (for example, if one type is ARRAY(NUMBER) and the other is ARRAY(DOUBLE)), one type is coerced to the other.

Working with other semi-structured functions

The following functions don't accept a structured ARRAY, structured OBJECT, or MAP values as an input argument:

- AS_ARRAY
- AS_OBJECT
- IS_ARRAY
- IS_OBJECT
- TYPEOF

Passing a structured type value as input results in an error.

Accessing structured types in applications using drivers

In applications that use drivers (for example, the ODBC or JDBC driver, the Snowflake Connector for Python, etc.), structured type values are returned as semi-structured type values. For example:

- The values in a structured ARRAY column are returned as semi-structured ARRAY values to the client application.
- The values in a structured OBJECT or MAP column are returned as semi-structured OBJECT values to the client

Note

For client applications that use the JDBC driver, the `ResultSet.getArray()` method returns an error if the query results you want to retrieve contain a structured ARRAY value with NULL values.

To retrieve a string representation instead, use the `ResultSet.getString()` method:

```
String result = resultSet.getString(1);
```

Using structured types with user-defined functions (UDFs) and stored procedures

When you create a user-defined function (UDF), user-defined table function (UDTF), or stored procedure in SQL, Snowflake Scripting, Java, Python, or Scala, you can use structured types in the arguments and return values. For example:

```
CREATE OR REPLACE FUNCTION my_udf(
    location OBJECT(city VARCHAR, zipcode NUMBER, val ARRAY(BOOLEAN)))
  RETURNS VARCHAR
  AS
  $$
    ...
  $$;
```

```
CREATE OR REPLACE FUNCTION my_udtf(check BOOLEAN)
  RETURNS TABLE(col1 ARRAY(VARCHAR))
  AS
  $$
  ...
  $$;
```

```
CREATE OR REPLACE PROCEDURE my_procedure(values ARRAY(INTEGER))
  RETURNS ARRAY(INTEGER)
  LANGUAGE SQL
  AS
  $$
    ...
  $$;
```

```
CREATE OR REPLACE FUNCTION my_function(values ARRAY(INTEGER))
  RETURNS ARRAY(INTEGER)
  LANGUAGE PYTHON
  RUNTIME_VERSION=3.10
  AS
  $$
    ...
  $$;
```

Note

Viewing information about structured types

The following sections describe the views and commands that you can use to view information about structured types:

- Using the SHOW COLUMNS command to view structured type information
- Using the DESCRIBE and other SHOW commands to view structured type information
- Viewing information about the structured types used in a database

Using the SHOW COLUMNS command to view structured type information

In the output of the SHOW COLUMNS command, the `data_type` column includes information about the types of elements, keys, and values.

Using the DESCRIBE and other SHOW commands to view structured type information

The output of the following commands includes information about structured types:

- DESCRIBE TABLE
- DESCRIBE RESULT
- DESCRIBE FUNCTION
- DESCRIBE PROCEDURE

For example, in the DESCRIBE RESULT output, the row for a MAP(VARCHAR, VARCHAR) column contains the following value in the `type` column:

```
map(VARCHAR(16777216), VARCHAR(16777216))
```

The row for an ARRAY(NUMBER) column contains the following value in the `type` column:

```
ARRAY(NUMBER(38,0))
```

Viewing information about the structured types used in a database

For columns of structured types, the INFORMATION_SCHEMA COLUMNS view only provides information about the basic data type of the column (ARRAY, OBJECT, or MAP).

For example, the `data_type` column just contains `ARRAY`, `OBJECT`, or `MAP`. The column doesn't include the types of the elements, keys, or values.

To view information about the types of elements, keys, and values, use the following views:

- For information about the types of elements in structured ARRAY types, query the ELEMENT_TYPES view in INFORMATION_SCHEMA or the ELEMENT_TYPES view in ACCOUNT_USAGE.
- For information about the types of keys and values in structured OBJECT and MAP types, query the FIELDS view in