On this page

Related content

Transforming Data During a Load

Snowflake supports transforming data while loading it into a table using the COPY INTO <table> command, dramatically simplifying your ETL pipeline for basic transformations. This feature helps you avoid the use of temporary tables to store pre-transformed data when reordering columns during a data load.

The COPY command supports:

- Column reordering, column omission, and casts using a SELECT statement. There is no requirement for your data files to have the same number and ordering of columns as your target table.
- The ENFORCE_LENGTH | TRUNCATECOLUMNS option, which can truncate text strings that exceed the target column length.

For general information about querying staged data files, see Querying Data in Staged Files.

Usage Notes

This section provides usage information for transforming staged data files during a load.

Supported File Formats

The following file format types are supported for COPY transformations:

- CSV
- JSON
- Avro
- ORC
- Parquet
- XML

To parse a staged data file, it is necessary to describe its file format:

CSV

The default format is character-delimited UTF-8 text. The default field delimiter is a comma character (`,`). The default record delimiter is the new line character. If the source data is in another format, specify the file format type and options.

When querying staged data files, the `ERROR_ON_COLUMN_COUNT_MISMATCH` option is ignored. There is no requirement for your data files to have the same number and ordering of columns as your target table.

JSON

To transform JSON data during a load operation, you must structure the data files in NDJSON ("Newline delimited JSON") standard format; otherwise, you might encounter the following error:

`Error parsing JSON: more than one document in the input`

To take advantage of error checking, set CSV as the format type (default value). Similar to CSV, with ndjson-compliant data, each line is a separate record. Snowflake parses each line as a valid JSON object or array.

Specify the following format type and options:

```
type = 'csv' field_delimiter = none record_delimiter = '\\n'
```

You could specify JSON as the format type, but any error in the transformation would stop the COPY operation, even if you set the ON_ERROR option to continue or skip the file.

All other file format types

Specify the format type and options that match your data files.

To explicitly specify file format options, set them in one of the following ways:

Querying staged data files using a SELECT statement:	• As file format options specified for a named file format or stage object. The named file format/stage object can then be referenced in the SELECT statement.
Loading columns from staged data files using a COPY INTO *<table>* statement:	• As file format options specified directly in the COPY INTO *<table>* statement. • As file format options specified for a named file format or stage object. The named file format/stage object can th in the COPY INTO *<table>* statement.

Supported Functions

Snowflake currently supports the following subset of functions for COPY transformations:

- ARRAY_CONSTRUCT
- ARRAY_SIZE
- ASCII
- CASE
- CAST , ::
- CEIL
- CHECK_JSON
- CHECK_XML
- CHR , CHAR
- CONCAT , ||
- CONVERT_TIMEZONE
- ENDSWITH
- EQUAL_NULL
- FLOOR
- GET
- GET_PATH , :
- HEX_DECODE_STRING
- HEX_ENCODE
- IFF
- IFNULL
- ILIKE
- [NOT] IN
- IS_ARRAY
- IS_BOOLEAN
- IS_DECIMAL
- IS_INTEGER
- IS_NULL_VALUE
- IS_OBJECT
- IS_TIME

- IS_TIMESTAMP_*
- LENGTH, LEN
- LIKE
- LPAD
- LTRIM
- MD5 , MD5_HEX
- NULLIF
- NVL
- NVL2
- OBJECT_CONSTRUCT
- PARSE_IP
- PARSE_JSON
- PARSE_URL
- PARSE_XML
- RANDOM
- REGEXP_REPLACE
- REGEXP_SUBSTR
- REPLACE
- REVERSE
- RPAD
- RTRIM
- SPLIT
- SPLIT_PART
- STARTSWITH
- SUBSTR , SUBSTRING
- TO_ARRAY
- TO_BINARY
- TO_BOOLEAN
- TO_CHAR , TO_VARCHAR
- TO_DATE , DATE

  Note that when this function is used to explicitly cast a value, neither the DATE_FORMAT file format option nor the DATE_INPUT_FORMAT parameter is applied.
- TO_DECIMAL , TO_NUMBER , TO_NUMERIC

- TO_DOUBLE
- TO_OBJECT
- TO_TIME , TIME

  Note that when this function is used to explicitly cast a value, neither the TIME_FORMAT file format option nor the TIME_INPUT_FORMAT parameter is applied.
- TO_TIMESTAMP / TO_TIMESTAMP_*

  Note that when this function is used to explicitly cast a value, neither the TIMESTAMP_FORMAT file format option nor the TIMESTAMP_INPUT_FORMAT parameter is applied.
- TO_VARIANT
- TRIM
- TRY_CAST
- TRY_HEX_DECODE_STRING
- TRY_TO_BINARY
- TRY_TO_BOOLEAN
- TRY_TO_DATE

  Note that the COPY INTO *<table>* command does not support the optional `<format>` argument for this function.
- TRY_TO_DECIMAL, TRY_TO_NUMBER, TRY_TO_NUMERIC
- TRY_TO_DOUBLE
- TRY_TO_TIME

  Note that the COPY INTO *<table>* command does not support the optional `<format>` argument for this function.
- TRY_TO_TIMESTAMP / TRY_TO_TIMESTAMP_*

  Note that the COPY INTO *<table>* command does not support the optional `<format>` argument for this function.
- UNICODE
- UUID_STRING
- XMLGET

Note in particular that the VALIDATE function ignores the SELECT list in a COPY INTO *<table>* statement. The function parses the files referenced in the statement and returns any parsing errors. This behavior can be surprising if you expect the function to evaluate the f
context of the COPY INTO *<table>* expressions.

Note that COPY transformations do ***not*** support the FLATTEN function, or JOIN or GROUP BY (aggregate) syntax.

The list of supported functions might expand over time.

The following categories of functions are also supported:

- Scalar SQL UDFs.

Filtering Results

Filtering the results of a FROM clause using a WHERE clause is not supported. The ORDER BY, LIMIT,FETCH,TOP keywords in SELECT statements are also not supported.

The DISTINCT keyword in SELECT statements is not fully supported. Specifying the keyword can lead to inconsistent or unexpected ON_ERROR copy option behavior.

VALIDATION_MODE Parameter

The VALIDATION_MODE parameter does not support COPY statements that transform data during a load.

CURRENT_TIME, CURRENT_TIMESTAMP Default Column Values

When loading data into a table that captures the load time in a column with a default value of either CURRENT_TIME() or CURRENT_TIMESTAMP(), all rows loaded using a specific COPY statement have the same timestamp value. The value records the time that the COPY statement started.

For example:

```
create or replace table mytable(
  c1 timestamp DEFAULT current_timestamp(),
  c2 number
);

copy into mytable(c2)
  from (select t.$1 from @mystage/myfile.csv.gz t);

+-------------------------------+----------+
| C1                            | C2       |
|-------------------------------+----------+
| 2018-09-05 08:58:28.718 -0700 | 1        |
...
```

```
| 2018-09-05 08:58:28.718 -0700 | 500       |
+-------------------------------+-----------+
```

Transforming CSV Data

Load a Subset of Table Data

Load a subset of data into a table. For any missing columns, Snowflake inserts the default values. The following example loads data from columns 1, 2, 6, and 7 of a staged CSV file:

```
copy into home_sales(city, zip, sale_date, price)
   from (select t.$1, t.$2, t.$6, t.$7 from @mystage/sales.csv.gz t)
   FILE_FORMAT = (FORMAT_NAME = mycsvformat);
```

Reorder CSV Columns During a Load

The following example reorders the column data from a staged CSV file before loading it into a table. Additionally, the COPY statement uses the SUBSTR , SUBSTRING function to remove the first few characters of a string before inserting it:

```
copy into home_sales(city, zip, sale_date, price)
   from (select SUBSTR(t.$2,4), t.$1, t.$5, t.$4 from @mystage t)
   FILE_FORMAT = (FORMAT_NAME = mycsvformat);
```

Convert Data Types During a Load

Convert staged data into other data types during a data load. All conversion functions are supported.

For example, convert strings as binary values, decimals, or timestamps using the TO_BINARY, TO_DECIMAL , TO_NUMBER , TO_NUMERIC, and TO_TIMESTAMP / TO_TIMESTAMP_* functions, respectively.

Sample CSV file:

```
snowflake,2.8,2016-10-5
warehouse,-12.3,2017-01-23
```


SQL statements:

```
-- Stage a data file in the internal user stage
PUT file:///tmp/datafile.csv @~;

-- Query the staged data file
select t.$1,t.$2,t.$3 from @~/datafile.csv.gz t;

-- Create the target table
create or replace table casttb (
  col1 binary,
  col2 decimal,
  col3 timestamp_ntz
  );

-- Convert the staged CSV column data to the specified data types before loading it into th
copy into casttb(col1, col2, col3)
from (
  select to_binary(t.$1, 'utf-8'),to_decimal(t.$2, '99.9', 9, 5),to_timestamp_ntz(t.$3)
  from @~/datafile.csv.gz t
)
file_format = (type = csv);

-- Query the target table
select * from casttb;

+--------------------+------+-------------------------+
| COL1               | COL2 | COL3                    |
|--------------------+------+-------------------------|
| 736E6F77666C616B65 |    3 | 2016-10-05 00:00:00.000 |
| 77617265686F757365 |  -12 | 2017-01-23 00:00:00.000 |
+--------------------+------+-------------------------+
```

Include Sequence Columns in Loaded Data

Create a sequence object using CREATE SEQUENCE. When loading data into a table using the COPY command, access the object using a `NEXTVAL` expression to sequence the data in a target number column. For more information about using sequences in queries, see Using Sequences.

```
-- Create a sequence
create sequence seq1;

-- Create the target table
create or replace table mytable (
  col1 number default seq1.nextval,
  col2 varchar,
  col3 varchar
  );
```

```
-- Stage a data file in the internal user stage
PUT file:///tmp/myfile.csv @~;

-- Query the staged data file
select $1, $2 from @~/myfile.csv.gz t;

+-----+-----+
| $1  | $2  |
|-----+-----|
| abc | def |
| ghi | jkl |
| mno | pqr |
| stu | vwx |
+-----+-----+

-- Include the sequence nextval expression in the COPY statement
copy into mytable (col1, col2, col3)
from (
  select seq1.nextval, $1, $2
  from @~/myfile.csv.gz t
)
;

select * from mytable;

+------+------+------+
| COL1 | COL2 | COL3 |
|------+------+------|
|    1 | abc  | def  |
|    2 | ghi  | jkl  |
|    3 | mno  | pqr  |
|    4 | stu  | vwx  |
+------+------+------+
```

Include AUTOINCREMENT / IDENTITY Columns in Loaded Data

Set the AUTOINCREMENT or IDENTITY default value for a number column. When loading data into a table using the COPY command, omit the column in the SELECT statement. The statement automatically populates the column.

```
-- Create the target table
create or replace table mytable (
  col1 number autoincrement start 1 increment 1,
  col2 varchar,
  col3 varchar
  );

-- Stage a data file in the internal user stage
PUT file:///tmp/myfile.csv @~;
```

```
-- Query the staged data file
select $1, $2 from @~/myfile.csv.gz t;

+-----+-----+
| $1  | $2  |
|-----+-----|
| abc | def |
| ghi | jkl |
| mno | pqr |
| stu | vwx |
+-----+-----+

-- Omit the sequence column in the COPY statement
copy into mytable (col2, col3)
from (
  select $1, $2
  from @~/myfile.csv.gz t
)
;

select * from mytable;

+------+------+------+
| COL1 | COL2 | COL3 |
|------+------+------|
|    1 | abc  | def  |
|    2 | ghi  | jkl  |
|    3 | mno  | pqr  |
|    4 | stu  | vwx  |
+------+------+------+
```

Transforming Semi-structured Data

The examples in this section apply to any semi-structured data type except where noted.

Load semi-structured Data into Separate Columns

The following example loads repeating elements from a staged semi-structured file into separate table columns with different data types.

This example loads the following semi-structured data into separate columns in a relational table, with the `location` object values loaded into a VARIANT column and the remaining values loaded into relational columns:

Language: English

```
-- Sample data:
{"location": {"city": "Lexington","zip": "40503"},"dimensions": {"sq_ft": "1000"},"type": "
```

```
{"location": {"city": "Belmont","zip": "02478"},"dimensions": {"sq_ft": "1103"},"type": "Re
{"location": {"city": "Winchester","zip": "01890"},"dimensions": {"sq_ft": "1122"},"type":
```

The following SQL statements load the file `sales.json` from the internal stage `mystage`:

Note

This example loads JSON data, but the SQL statements are similar when loading semi-structured data of other types (e.g. Avro, ORC, etc.).

For an additional example using Parquet data, see Load Parquet Data into Separate Columns (in this topic).

```
-- Create an internal stage with the file type set as JSON.
CREATE OR REPLACE STAGE mystage
  FILE_FORMAT = (TYPE = 'json');

-- Stage a JSON data file in the internal stage.
PUT file:///tmp/sales.json @mystage;

-- Query the staged data. The data file comprises three objects in NDJSON format.
SELECT t.$1 FROM @mystage/sales.json.gz t;

+------------------------------+
| $1                           |
|------------------------------|
| {                            |
|   "dimensions": {            |
|     "sq_ft": "1000"          |
|   },                         |
|   "location": {              |
|     "city": "Lexington",     |
|     "zip": "40503"           |
|   },                         |
|   "price": "75836",          |
|   "sale_date": "2022-08-25", |
|   "type": "Residential"      |
| }                            |
| {                            |
|   "dimensions": {            |
|     "sq_ft": "1103"          |
|   },                         |
|   "location": {              |
|     "city": "Belmont",       |
|     "zip": "02478"           |
|   },                         |
|   "price": "92567",          |
|   "sale_date": "2022-09-18", |
```

```
|   "type": "Residential"      |
| }                            |
| {                            |
|   "dimensions": {            |
|     "sq_ft": "1122"          |
|   },                         |
|   "location": {              |
|     "city": "Winchester",    |
|     "zip": "01890"           |
|   },                         |
|   "price": "89921",          |
|   "sale_date": "2022-09-23", |
|   "type": "Condo"            |
| }                            |
+------------------------------+
```

```
-- Create a target table for the data.
CREATE OR REPLACE TABLE home_sales (
  CITY VARCHAR,
  POSTAL_CODE VARCHAR,
  SQ_FT NUMBER,
  SALE_DATE DATE,
  PRICE NUMBER
);

-- Copy elements from the staged file into the target table.
COPY INTO home_sales(city, postal_code, sq_ft, sale_date, price)
FROM (select
$1:location.city::varchar,
$1:location.zip::varchar,
$1:dimensions.sq_ft::number,
$1:sale_date::date,
$1:price::number
FROM @mystage/sales.json.gz t);

-- Query the target table.
SELECT * from home_sales;
```

```
+------------+-------------+-------+------------+-------+
| CITY       | POSTAL_CODE | SQ_FT | SALE_DATE  | PRICE |
|------------+-------------+-------+------------+-------|
| Lexington  | 40503       |  1000 | 2022-08-25 | 75836 |
| Belmont    | 02478       |  1103 | 2022-09-18 | 92567 |
| Winchester | 01890       |  1122 | 2022-09-23 | 89921 |
+------------+-------------+-------+------------+-------+
```

Load Parquet Data into Separate Columns

Similar to the previous example, but loads semi-structured data from a file in the Parquet format. This example is provided for users who are familiar with Apache Parquet:

```
-- Create a file format object that sets the file format type. Accept the default options.
create or replace file format my_parquet_format
  type = 'parquet';

-- Create an internal stage and specify the new file format
create or replace temporary stage mystage
  file_format = my_parquet_format;

-- Create a target table for the data.
create or replace table parquet_col (
  custKey number default NULL,
  orderDate date default NULL,
  orderStatus varchar(100) default NULL,
  price varchar(255)
);

-- Stage a data file in the internal stage
put file:///tmp/mydata.parquet @mystage;

-- Copy data from elements in the staged Parquet file into separate columns
-- in the target table.
-- Note that all Parquet data is stored in a single column ($1)
-- SELECT list items correspond to element names in the Parquet file
-- Cast element values to the target column data type
copy into parquet_col
  from (select
  $1:o_custkey::number,
  $1:o_orderdate::date,
  $1:o_orderstatus::varchar,
  $1:o_totalprice::varchar
  from @mystage/mydata.parquet);

-- Query the target table
SELECT * from parquet_col;

+---------+------------+-------------+-----------+
| CUSTKEY | ORDERDATE  | ORDERSTATUS | PRICE     |
|---------+------------+-------------+-----------|
|   27676 | 1996-09-04 | O           | 83243.94  |
|  140252 | 1994-01-09 | F           | 198402.97 |
...
+---------+------------+-------------+-----------+
```

Flatten Semi-structured Data

FLATTEN is a table function that produces a lateral view of a VARIANT, OBJECT, or ARRAY column. Using the sample data from Load semi-structured Data into Separate Col table with a separate row for each element in the objects.

```
-- Create an internal stage with the file delimiter set as none and the record delimiter se
create or replace stage mystage
  file_format = (type = 'json');

-- Stage a JSON data file in the internal stage with the default values
put file:///tmp/sales.json @mystage;

-- Create a table composed of the output from the FLATTEN function
create or replace table flattened_source
(seq string, key string, path string, index string, value variant, element variant)
as
  select
    seq::string
  , key::string
  , path::string
  , index::string
  , value::variant
  , this::variant
  from @mystage/sales.json.gz
    , table(flatten(input => parse_json($1)));

  select * from flattened_source;
```

```
+-----+-----------+-----------+-------+--------------------------+--------------------------
| SEQ | KEY       | PATH      | INDEX | VALUE                    | ELEMENT
|-----+-----------+-----------+-------+--------------------------+--------------------------
| 1   | location  | location  | NULL  | {                        | {
|     |           |           |       |   "city": "Lexington",   |   "location": {
|     |           |           |       |   "zip": "40503"         |     "city": "Lexington",
|     |           |           |       | }                        |     "zip": "40503"
|     |           |           |       |                          |   },
|     |           |           |       |                          |   "price": "75836",
|     |           |           |       |                          |   "sale_date": "2017-3-5"
|     |           |           |       |                          |   "sq__ft": "1000",
|     |           |           |       |                          |   "type": "Residential"
|     |           |           |       |                          | }
...
| 3   | type      | type      | NULL  | "Condo"                  | {
|     |           |           |       |                          |   "location": {
|     |           |           |       |                          |     "city": "Winchester",
|     |           |           |       |                          |     "zip": "01890"
|     |           |           |       |                          |   },
|     |           |           |       |                          |   "price": "89921",
|     |           |           |       |                          |   "sale_date": "2017-3-21
|     |           |           |       |                          |   "sq__ft": "1122",
|     |           |           |       |                          |   "type": "Condo"
|     |           |           |       |                          | }
+-----+-----------+-----------+-------+--------------------------+--------------------------
```

Split Semi-structured Elements and Load as VARIANT Values into Separate Columns

Following the instructions in Load semi-structured Data into Separate Columns, you can load individual elements from semi-structured data into different columns in your target table. Additionally, using the SPLIT function, you can split element values that contain a separator and load them as an array.

For example, split IP addresses on the dot separator in repeating elements. Load the IP addresses as arrays in separate columns:

```
-- Create an internal stage with the file delimiter set as none and the record delimiter se
create or replace stage mystage
  file_format = (type = 'json');

-- Stage a semi-structured data file in the internal stage
put file:///tmp/ipaddress.json @mystage auto_compress=true;

-- Query the staged data
select t.$1 from @mystage/ipaddress.json.gz t;

+--------------------------------------------------------------------------+
| $1                                                                       |
|--------------------------------------------------------------------------|
| {"ip_address": {"router1": "192.168.1.1","router2": "192.168.0.1"}}, |
| {"ip_address": {"router1": "192.168.2.1","router2": "192.168.3.1"}}  |
+--------------------------------------------------------------------------+

-- Create a target table for the semi-structured data
create or replace table splitjson (
  col1 array,
  col2 array
  );

-- Split the elements into individual arrays using the SPLIT function and load them into se
-- Note that all JSON data is stored in a single column ($1)
copy into splitjson(col1, col2)
from (
  select split($1:ip_address.router1, '.'),split($1:ip_address.router2, '.')
  from @mystage/ipaddress.json.gz t
);

-- Query the target table
select * from splitjson;

+----------+----------+
| COL1     | COL2     |
|----------+----------|
```

```
| [         | [         |
|   "192",  |   "192",  |
|   "168",  |   "168",  |
|   "1",    |   "0",    |
|   "1"     |   "1"     |
| ]         | ]         |
| [         | [         |
|   "192",  |   "192",  |
|   "168",  |   "168",  |
|   "2",    |   "3",    |
|   "1"     |   "1"     |
| ]         | ]         |
+-----------+-----------+
```

Was this page helpful? Yes No

 Visit Snowflake

 Having problems? Get support

 Join the conversation in our community

 Read the latest on our blog

 Develop with Snowflake

 Get your Snowflake certification

Language: English