

Steven Evans

SRE220000

BUAN 6335

Snowflake Lab 6

6.1.2 Creating and querying the Customers Table using JSON format

The screenshot shows the Snowflake interface with the URL <https://app.snowflake.com/sfedu02/yzb42236/wuOmOQSe8X#query>. The left sidebar lists various databases and worksheets. The main area shows a code editor with the following SQL script:

```
52 --     parse_json function to create the JSON data. It also uses the $1
53 --     notation to refer to the first column in the table, and $2 to refer
54 --     to the second column.
55
56 CREATE OR REPLACE TABLE customers AS
57     SELECT
58         $1 AS id,
59         parse_json($2) AS info
60     FROM
61         VALUES
62             ('12712555, {"name": {"first": "John", "last": "Smith"}},',
63             '9812771, {"name": {"first": "Jane", "last": "Doe"}},'
64
65 -- 6.1.3 Open a row of the table you just created to see its structure.
66
67
68 | SELECT * FROM customers;
```

Below the code editor is a results table:

ID	INFO
1	{ "name": { "first": "John", "last": "Smith" } }
2	{ "name": { "first": "Jane", "last": "Doe" } }

Query Details: Query duration 296ms, Rows 2, Query ID 01b8a42a-0004-3fd0-0...

6.1.4 Extracting the name data from customers using dot syntax

The screenshot shows the Snowflake interface with the same URL as the previous screenshot. The code editor contains the following SQL script:

```
71 -- Click on a row in the info column to display its structure on the
72 -- right. Notice that the top-level key is called name, and its value is
73 -- an object (a collection of key-value pairs).
74
75 -- 6.1.4 Extract the name data from your table using dot notation syntax.
76 -- One way to extract values from JSON data is with dot notation syntax.
77 -- Dot Notation Syntax:
78 -- In your SELECT statement, you would simply reference the column, add
79 -- a colon, then reference the elements one-by-one separated by dots:
80 -- <column>.<element>.<element>.<element>
81 -- Run the SELECT statement below to see dot notation syntax at work:
82
83 | SELECT
84     id,
85     info.name.first AS first_name,
86     info.name.last AS last_name
87
88 | FROM
89 |     customers;
```

Below the code editor is a results table:

ID	FIRST_NAME	LAST_NAME
1	"John"	"Smith"
2	"Jane"	"Doe"

Query Details: Query duration 159ms, Rows 2, Query ID 01b8a42b-0004-3f5c-0...

6.1.5 Determining the data types of the results

The screenshot shows a Snowflake worksheet titled 'CHIPMUNK_DB.CHIPMUNK_SCHEMA'. The code editor contains a query to determine data types:

```
-- 6.1.5 Determine the data types in your result set.  
-- Below we'll use the command DESCRIBE RESULT and the function  
-- last_query_id() to fetch details about the query we just ran.  
  
DESCRIBE RESULT last_query_id();  
  
-- Notice that the name columns are still of data type VARIANT. Let's  
-- look at another way to determine if our data is of data type VARIANT.  
  
-- 6.1.6 Rerun the query examine the output  
  
SELECT  
    id,  
    info:name.first AS first_name,  
    info:name.last AS last_name  
FROM  
    customers;  
  
-- You'll notice that the first_name and last_name column values are  
-- enclosed in double-quotes. Seeing string values in double-quotes is
```

The results pane shows a table of columns:

	name	type	kind	null?	default	primary key	unique key	check	exp
1	ID	NUMBER(8,0)	COLUMN	Y	null	N	N	null	null
2	FIRST_NAME	VARIANT	COLUMN	Y	null	N	N	null	null
3	LAST_NAME	VARIANT	COLUMN	Y	null	N	N	null	null

Query Details:

- Query duration: 82ms
- Rows: 3
- Query ID: 01b8a42b-0004-3fdd-0...

Buttons: Results, Chart, Ask Copilot.

6.1.7 Querying the customers table again and casting the names to VARCHAR

The screenshot shows the Snowflake Worksheet interface. On the left, a sidebar lists various databases and worksheets. The main area displays a query script and its execution results.

Query Script:

```
116 -- would be if you had cast the VARIANT columns to standard SQL data
117 --
118 --
119 --
120 --
121 -- 6.1.7 Extract the name data and cast it to type VARCHAR.
122
123 SELECT
124     id,
125     info:name.first::VARCHAR AS first_name,
126     info:name.last::VARCHAR AS last_name
127 FROM
128     customers;
129
130 DESCRIBE RESULT last_query_id();
131
132 --
133 --
134 --
135 --
136 -- 6.1.8 Create a table from your query
137
```

Results:

	name	type	kind	null?	default	primary key	unique key	check
1	ID	NUMBER(8,0)	COLUMN	Y	null	N	N	null
2	FIRST_NAME	VARCHAR(16777216)	COLUMN	Y	null	N	N	null
3	LAST_NAME	VARCHAR(16777216)	COLUMN	Y	null	N	N	null

Query Details:

- Query duration: 76ms
- Rows: 3
- Query ID: 01b8a42d-0004-3f5c-0...

Buttons: Results, Chart, Ask Copilot

6.1.8 Creating the structured version of the customers table

The screenshot shows the Snowflake interface with the URL app.snowflake.com/sfedu02/yzb42236/wuOmOQSe8X#query. The query editor displays a script to create a table named 'customers_structured'. The script includes a SELECT statement with columns 'id', 'first_name', and 'last_name' from the 'customers' table. The results pane shows two rows of data: ID 12712555 with first name John and last name Smith, and ID 98127771 with first name Jane and last name Doe.

```
-- As you can see, your first_name and last_name columns are now VARCHAR values. This allows you to sort them, apply string functions to them, or do anything you could with structured data.  
-- 6.1.8 Create a table from your query  
CREATE OR REPLACE TABLE customers_structured  
AS  
SELECT  
    id,  
    info:name.first::VARCHAR AS first_name,  
    info:name.last::VARCHAR AS last_name  
FROM  
    customers;  
  
SELECT * FROM customers_structured;  
  
-- You now have a table that can be used for analysis by your colleagues who do not know how to directly query semi-structured data.  
-- 6.2.0 Query Nested JSON Data with Dot Notation  
There are two ways to query nested JSON data: one is with dot notation
```

ID	FIRST_NAME	LAST_NAME
1	John	Smith
2	Jane	Doe

Query Details: Query duration 98ms, Rows 2, Query ID 01b8a42e-0004-3f5c-0...

6.2.2 Creating the customer table from nested json data

The screenshot shows the Snowflake interface with the URL app.snowflake.com/sfedu02/yzb42236/wuOmOQSe8X#query. The query editor displays a script to create a table named 'customers'. The script includes a SELECT statement with column 'INFO'. The results pane shows two rows of data: ID 12712555 with INFO containing a contact object, and ID 98127771 with INFO containing a contact object. The contact object contains business and personal contact details.

```
};  
-- Notice that the value for key contact is enclosed in square brackets.  
-- This is because the value is an array of key value pairs, each of  
-- which contains two key-value pairs. Whenever you see those square  
-- brackets, you will need to do something extra with your dot notation  
-- in order to fetch the values you want.  
-- 6.2.2 Select from your table and click a cell in the INFO column to view  
-- the structure.  
SELECT * FROM customers;  
  
-- Notice that there are two people: John Smith and Jane Doe. Each has a  
-- key-value called contact, with business and personal contact details  
-- nested within. Let's query the table to fetch both people and their  
-- business and personal phone numbers using the same dot notation  
-- syntax we've been using.  
-- 6.2.3 Query the table to fetch business and personal phone numbers
```

ID	INFO
1	{ "contact": [{ "business": { "email": "j.smith@company.com", "phone": "303-555-1234" }, "personal": { "email": "j.smith@company.com", "phone": "303-555-1234" } }] }
2	{ "contact": [{ "business": { "email": "jg_doe@company2.com", "phone": "303-555-1234" }, "personal": { "email": "jg_doe@company2.com", "phone": "303-555-1234" } }] }

Query Details: Query duration 126ms, Rows 2, Query ID 01b8a430-0004-3ffd-0...

6.2.3 Querying the customers table with dot notation and casting

The screenshot shows a Snowflake worksheet with the following details:

- Databases:** CHIPMUNK_DB.CHIPMUNK_SCHEMA
- Worksheet Name:** 02-visualizations-in-snow...
- Query ID:** 01b8a430-0004-3fd0-0...
- Results:** 2 rows returned in 63ms.

```

204 --      syntax we've been using.
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
-- 6.2.3 Query the table to fetch business and personal phone numbers
SELECT
  ID,
  info:name.first::VARCHAR AS first_name,
  info:name.last::VARCHAR AS last_name,
  info:contact.business.phone::VARCHAR AS business_phone,
  info:contact.personal.phone::VARCHAR AS personal_phone
FROM
  customers;
-- Notice that while our query fetched the full names of both John and
-- Jane, the business phone and personal phone columns are null.
-- The key contact has a value consisting of a zero-based array of
-- elements, each of which is a key-value pair. Because the keys
-- business and personal, along with their values are nested within an
-- array, we can't simply query info:contact.business and expect to get
-- the business phone number. That's because we haven't told the query

```

	ID	FIRST_NAME	LAST_NAME	BUSINESS_PHONE	PERSONAL_PHONE
1	12712555	John	Smith	null	null
2	98127771	Jane	Doe	null	null

Requerying to index the correct parts of the JSON data to pull the business and personal phone numbers.

The screenshot shows a Snowflake worksheet with the following details:

- Databases:** CHIPMUNK_DB.CHIPMUNK_SCHEMA
- Worksheet Name:** 02-visualizations-in-snow...
- Query ID:** 01b8a430-0004-3fd0-0...
- Results:** 2 rows returned in 63ms.

```

229 --      for business and 1 for personal. Remember, the first position is 0
230 --      because the array is zero-based.
231 --      Run the SELECT statement below to see this in action.
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
-- 6.2.4 Fetch all data and put it in structured format:
CREATE OR REPLACE TABLE customers_staged AS

```

	ID	FIRST_NAME	LAST_NAME	BUSINESS_PHONE	PERSONAL_PHONE
1	12712555	John	Smith	null	null
2	98127771	Jane	Doe	null	null

6.2.4 Using dot notation on the entire set of JSON data to make customers_dot

The screenshot shows the Snowflake web interface. In the top navigation bar, the URL is `app.snowflake.com/sfedu02/yzb42236/wuOmQSe8X#query`. The left sidebar lists various databases under the `CHIPMUNK_DB.CHIMPUNK_SCHEMA` node. The main query editor window contains the following code:

```

-- 6.2.4 Fetch all data and put it in structured format:
CREATE OR REPLACE TABLE customers_dot
AS
SELECT
    ID,
    info.name.first::VARCHAR AS first_name,
    info.name.last::VARCHAR AS last_name,
    info:contact[0].business.phone::VARCHAR AS business_phone,
    info:contact[0].business.email::VARCHAR AS business_email,
    info:contact[1].personal.phone::VARCHAR AS personal_phone,
    info:contact[1].personal.email::VARCHAR AS personal_email
FROM
    customers;
SELECT * FROM customers_dot;
-- As you can see, we've now converted our semi-structured data to a
-- structured format so it can be used by colleagues that need to join
-- it with other structured data, or that don't know how to work with
-- semi-structured data.

```

The results pane shows the following table:

ID	FIRST_NAME	LAST_NAME	BUSINESS_PHONE	BUSINESS_EMAIL	PERSONAL_PHONE
1	12712555	John	Smith	303-555-1234	j.smith@company.com
2	98127771	Jane	Doe	303-638-4887	jg_doe@company2.com

Query Details: Query duration 101ms, Rows 2, Query ID 01b8a432-0004-3fdd-0...

6.3.1 Flattening to view the top layer keys from customers

The screenshot shows the Snowflake web interface. In the top navigation bar, the URL is `app.snowflake.com/sfedu02/yzb42236/wuOmQSe8X#query`. The left sidebar lists various databases under the `CHIPMUNK_DB.CHIMPUNK_SCHEMA` node. The main query editor window contains the following code:

```

-- ADDITIONAL OUTPUT STUFF THAT YOU CAN USE TO FIGURE OUT HOW YOUR DATA
-- IS STRUCTURED AND TO FINE-TUNE YOUR QUERY TO GET THE RESULTS THAT YOU
-- WANT.
-- 6.3.1 Use the FLATTEN FUNCTION TO VIEW ALL THE TOP-LEVEL KEYS FROM YOUR
-- TABLE.
-- The top-level keys are the first level of key-value pairs that the
-- query engine finds when parsing the semi-structured data indicated in
-- your query.
-- Run the query below to find those keys.
SELECT
    *
FROM
    customers,
    LATERAL FLATTEN(input=>info);
-- When we use the FLATTEN function, its output is in essence joined to
-- the table customers and its columns can be selected just as you would
-- with columns from a table of structured data.
-- First, notice that the input of the FLATTEN function is
-- (<input>\><column>). For '<column>' you can use just the column name

```

The results pane shows the following table:

ID	INFO	SEQ	KEY
1	{ "contact": [{ "business": { "email": "j.smith@company.com", "phone": "303-555-1234" } }] }	1	cont
2	{ "contact": [{ "business": { "email": "j.smith@company.com", "phone": "303-638-4887" } }] }	1	name
3	{ "contact": [{ "business": { "email": "jg_doe@company2.com", "phone": "303-678-6789" } }] }	2	cont

Query Details: Query duration 119ms, Rows 4, Query ID 01b8a433-0004-3f5c-0...

Pulling the specified keys and values from the customers table using input=>info

```

304 --      Lists all of the top-level keys in the INFO column.
305 --      This might be a little easier to see if you just pull out the key and
306 --      value columns:
307
308 | SELECT
309 |   key,
310 |   value
311 | FROM
312 |   customers,
313 |   LATERAL FLATTEN(input=>info);
314
315 --      You get four rows of output: the first two rows are the top-level
316 --      keys (contact and name) from John Smith's record, and the second two
317 --      rows are the same keys from Jane Doe's record. But what if you want
318 --      to see all the keys, even those nested in other structures?
319
320 -- 6.3.2  Use the RECURSIVE option to view all the keys from your table.
321 --      This time you will include the index:
322
323 | SELECT
324 |   key, index, value
  
```

Results

KEY	VALUE
1 contact	[{ "business": { "email": "j.smith@company.com", "phone": "303-555-1234" } }, { "first": "John", "last": "Smith" }]
2 name	{ "first": "Jane", "last": "Doe" }
3 contact	[{ "business": { "email": "jg_doe@company2.com", "phone": "303-638-4887" } }, { "first": "Jane", "last": "Doe" }]

Query Details

- Query duration: 94ms
- Rows: 4
- Query ID: 01b8a434-0004-3fdd-0...

6.3.2 Selecting the key, value and index to separate the business from the personal data

```

318 -- 6.3.2  Use the RECURSIVE option to view all the keys from your table.
319 --      This time you will include the index:
320
321
322 | SELECT
323 |   key, index, value
324 | FROM
325 |   customers,
326 |   LATERAL FLATTEN(input=>info, RECURSIVE=>true);
327
328 --      Note that you have gone from four rows of output to 24. When you
329 --      include the RECURSIVE option, any time a value is either an object or
330 --      an array (as opposed to a simple value), then that object or array is
331 --      also flattened.
332
333 --      Note that when the index column shows 0, the value column shows the
334 --      business details. When the index column shows 1, the value column
335 --      shows the personal details. This is consistent with what we saw in
336 --      the earlier dot notation exercise.
337
338 -- 6.3.3  Show only the keys that have simple values.
339 --      When you query semi-structured data, you are typically accessing keys
340 --      that have simple values. For example, you're looking for the business
  
```

Results

KEY	INDEX	VALUE
1 contact	null	[{ "business": { "email": "j.smith@company.com", "phone": "303-555-1234" } }]
2 null	0	{ "business": { "email": "j.smith@company.com", "phone": "303-555-1234" } }
3 business	null	{ "email": "j.smith@company.com", "phone": "303-555-1234" }
4 email	null	"j.smith@company.com"
5 phone	null	"303-555-1234"
6 null	1	{ "personal": { "email": "jsmith332@gmail.com", "phone": "303-421-8322" } }

Query Details

- Query duration: 77ms
- Rows: 24
- Query ID: 01b8a435-0004-3f5c-0...

Show more ▾

Ask Copilot

6.3.3 Excluding any column that is not an object or arry from the query using recursion

```

354    -- notice that the index column contains nothing but null values. This
355    -- is because the index column tells you the position of values in an
356    -- array - but you filtered out all of the arrays with your WHERE
357    -- clause. So there's no point in including the index column in the
358    -- query.
359
360    SELECT
361        key, value
362    FROM
363        customers,
364    LATERAL FLATTEN(input=>info, RECURSIVE=>true)
365    WHERE TYPEOF(value) NOT IN ('OBJECT', 'ARRAY');
366
367    -- Now you know all the simple key:value pairs in each row, but you may
368    -- still have some questions about the data. For example, several of the
369    -- rows have a key of phone - but is that a business phone, or a
370    -- personal phone? And is it for John or for Jane? The columns this,
371    -- seq, and path from the FLATTEN function can help answer these
372    -- questions.
373
374    -- 6.3.4 Include the column this in your query.
375
  
```

Results

KEY	VALUE
1	"j.smith@company.com"
2	"303-555-1234"
3	"jsmith332@gmail.com"
4	"303-421-8322"
5	"John"
6	"Smith"

Query Details

- Query duration: 64ms
- Rows: 12
- Query ID: 01b8a438-0004-3fc5c-0...

Ask Copilot

6.3.4 Including the this column in the query to show the full parts the value was pulled from

```

364    LATERAL FLATTEN(input=>info, RECURSIVE=>true)
365    WHERE TYPEOF(value) NOT IN ('OBJECT', 'ARRAY');
366
367    -- Now you know all the simple key:value pairs in each row, but you may
368    -- still have some questions about the data. For example, several of the
369    -- rows have a key of phone - but is that a business phone, or a
370    -- personal phone? And is it for John or for Jane? The columns this,
371    -- seq, and path from the FLATTEN function can help answer these
372    -- questions.
373
374    -- 6.3.4 Include the column this in your query.
375
376    SELECT
377        key, value, this
378    FROM
379        customers,
380    LATERAL FLATTEN(input=>info, RECURSIVE=>true)
381    WHERE TYPEOF(value) NOT IN ('OBJECT', 'ARRAY');
382
383    -- The column named this shows you what is being flattened for each row.
384    -- This really only has value if you are doing a recursive flatten
  
```

Results

KEY	VALUE	THIS
1	"j.smith@company.com"	{ "email": "j.smith@company.com", "phone": "303-555-1234"}
2	"303-555-1234"	{ "email": "j.smith@company.com", "phone": "303-555-1234"}
3	"jsmith332@gmail.com"	{ "email": "jsmith332@gmail.com", "phone": "303-421-8322"}
4	"303-421-8322"	{ "email": "jsmith332@gmail.com", "phone": "303-421-8322"}
5	"John"	{ "first": "John", "last": "Smith"}
6	"Smith"	{ "first": "John", "last": "Smith"}

Query Details

- Query duration: 61ms
- Rows: 12
- Query ID: 01b8a439-0004-3fd0-0...

Ask Copilot

6.3.5 Showing the path where the value was pulled from. (Dot notation essentially)

The screenshot shows a Snowflake worksheet titled "CHIPMUNK_DB.CHIMPUNK_SCHEMA". The code in the editor highlights the "path" column, which is described as being essentially dot notation syntax. The results table shows columns: KEY, VALUE, and PATH. The PATH column contains JSON-style paths like "contact[0].business.email" and "name.first".

```

383 -- The column named this shows you what is being flattened for each row.
384 -- This really only has value if you are doing a recursive flatten
385 -- (otherwise, this will always show the input to the FLATTEN function).
386 -- If you look at this column across from each phone number, you
387 -- have a good idea of who the phone number is for based on their email
388 -- address. But email addresses do not always include someone's name...so
389 -- that is not the best indicator.
390
391 -- 6.3.5 Include the path column in your query.
392
393 SELECT
394   key, value, path
395 FROM
396   customers,
397   LATERAL FLATTEN(input=>info, RECURSIVE=>true)
398 WHERE TYPEOF(value) NOT IN ('OBJECT', 'ARRAY');
399
400 -- The path column is essentially the dot notation syntax representation
401 -- of how you got down to the value displayed. For a key of phone, you
402 -- can see that the path tells you whether it is a personal phone or a
403 -- business phone. But that still doesn't solve the problem of who that
404 -- phone number belongs to.
  
```

KEY	VALUE	PATH
1	"j.smith@company.com"	contact[0].business.email
2	"303-555-1234"	contact[0].business.phone
3	"jsmith332@gmail.com"	contact[1].personal.email
4	"303-421-8322"	contact[1].personal.phone
5	"John"	name.first
6	"Smith"	name.last

6.3.6 Including the seq column in the query to show which object of the json format the data was pulled from.

The screenshot shows a Snowflake worksheet titled "CHIPMUNK_DB.CHIMPUNK_SCHEMA". The code in the editor highlights the "seq" column, which is described as being assigned a sequence number for each row of JSON data. The results table shows columns: SEQ, KEY, and VALUE. The SEQ column values range from 1 to 6, corresponding to the rows in the previous screenshot.

```

404 -- phone number belongs to.
405
406
407
408
409
410
411
412
413
414
415 -- Each row of JSON data is assigned a sequence number. So when you see
416 -- multiple rows of output, you know that all rows with the same
417 -- sequence value came out of the same row of data. Now it's easy to
418 -- match up phone numbers with the owner - you just look for the first
419 -- and last name with the same sequence number as the phone. And the
420 -- path will tell you whether it is a personal phone, or a business
421 -- phone.
422 -- In your output, it is very likely that the first six rows have a
423 -- sequence value of 1, and the last six rows have a sequence value of
424 -- 2. However, sequence numbers are not guaranteed to be displayed in
  
```

SEQ	KEY	VALUE
1	1	"j.smith@company.com"
2	1	"303-555-1234"
3	1	"jsmith332@gmail.com"
4	1	"303-421-8322"
5	1	"John"
6	1	"Smith"

6.3.7 Pulling the info prior to exploding the data, we need to know what needs to be flattened

The screenshot shows a database interface with a sidebar listing databases and worksheets. The main area displays a query in the CHIPMUNK_DB.CHIMPUNK_SCHEMA schema. The query uses LATERAL FLATTEN to extract contact information from the 'info' column of the 'customers' table. The results table shows four rows with columns SEQ, KEY, PATH, INDEX, and VALUE. The VALUE column contains JSON objects representing business and personal contact details.

```

438 --      data to a structured format.
439
440 -- 6.3.7 Explode the data
441 -- Now we want to list ID, first name, last name, business phone number
442 -- and email, and personal phone number and email. We need to explode
443 -- the data to do that. But first we need to figure out the input into
444 -- our FLATTEN function.
445 -- Start by running the query below:
446
447 SELECT
448   info.*
449   FROM
450     customers,
451   LATERAL FLATTEN(input=>info, RECURSIVE=>true) info
452   WHERE index IS NOT NULL;
453
454 --      Using what we learned earlier, we exploded all of the data but
455 -- filtered to only find the rows where index is not null. This gives us
456 -- all of the positions in the contact array.
457 -- If we look in the path column, we find contact[0] and contact[1].
458 -- Those are the values we want to input into our FLATTEN function.
  
```

SEQ	KEY	PATH	INDEX	VALUE
1	1	contact[0]	0	{ "business": { "email": "j.smith@company.com", "phone": "303-555-4211" }}
2	1	contact[1]	1	{ "personal": { "email": "jsmith332@gmail.com", "phone": "303-421-5555" }}
3	2	contact[0]	0	{ "business": { "email": "jg_doe@company2.com", "phone": "303-678-9999" }}
4	2	contact[1]	1	{ "personal": { "email": "happyjane@gmail.com", "phone": "303-678-9999" }}

6.3.8 Pulling only the business info from the data by using indexing in the flatten statement

The screenshot shows a database interface with a sidebar listing databases and worksheets. The main area displays a query in the CHIPMUNK_DB.CHIMPUNK_SCHEMA schema. The query uses LATERAL FLATTEN to extract business contact information from the 'info' column of the 'customers' table, specifically targeting contact[0]. The results table shows two rows with columns ID, FIRST_NAME, LAST_NAME, BUSINESS_PHONE, and BUSINESS_EMAIL. The BUSINESS_PHONE and BUSINESS_EMAIL columns contain the extracted contact details.

```

498 --      Those are the values we want to input into our FLATTEN function.
499
500 -- 6.3.8 Fetch ID, first name, last name and business details
501 -- Run the statement below:
502
503 SELECT
504   ID,
505   info.name.first::VARCHAR AS first_name,
506   info.name.last::VARCHAR AS last_name,
507   business.value:phone::VARCHAR as business_phone,
508   business.value:email::VARCHAR as business_email
509   FROM
510     customers,
511   LATERAL FLATTEN(input=>info:contact[0]) business;
512
513 --      Fetching the ID, first and last names was simple. For the business
514 -- details, we've inputted contact[0] into the FLATTEN function to
515 -- access only the business details.
516
517 -- 6.3.9 Fetch ID, first name, last name and personal details
518 -- Run the statement below:
  
```

ID	FIRST_NAME	LAST_NAME	BUSINESS_PHONE	BUSINESS_EMAIL
1	John	Smith	303-555-1234	j.smith@company.com
2	Jane	Doe	303-638-4887	jg_doe@company2.com

6.3.9 Pulling only the personal info by pulling from the second index in flatten

```

SELECT
    ID,
    info.name.first::VARCHAR AS first_name,
    info.name.last::VARCHAR AS last_name,
    personal.value:phone::VARCHAR AS personal_phone,
    personal.value:email::VARCHAR AS personal_email
FROM
    customers,
    LATERAL FLATTEN(input=>info:contact[1]) personal;
-- Here we've fetched the personal details exactly as we fetched the
-- business details. Now let's put it all together.

-- 6.3.10 Fetch all contact details

SELECT
    ID,
    info.name.first::VARCHAR AS first_name,
    info.name.last::VARCHAR AS last_name,
    business.value:email::VARCHAR AS business_email,
    business.value:phone::VARCHAR AS business_phone,
    personal.value:email::VARCHAR AS personal_email,
    personal.value:phone::VARCHAR AS personal_phone
FROM
    customers,
    LATERAL FLATTEN(input=>info:contact[1]) business,
    LATERAL FLATTEN(input=>info:contact[1]) personal;
-- Here we simply used the FLATTEN function twice for each potential
-- path. Our semi-structured data is now structured data.

-- 6.4.0 Try it out!
-- In this section, you're going to query semi-structured data on your
-- own. First you'll run a statement to create a short set of JSON data.

```

	ID	FIRST_NAME	LAST_NAME	PERSONAL_PHONE	PERSONAL_EMAIL
1	12712555	John	Smith	303-421-8322	jsmith332@gmail.com
2	98127771	Jane	Doe	303-678-6789	happyjane@gmail.com

6.3.10 Pulling personal and business info by including both indices in flatten

```

-- 6.3.10 Fetch all contact details

SELECT
    ID,
    info.name.first::VARCHAR AS first_name,
    info.name.last::VARCHAR AS last_name,
    business.value:email::VARCHAR AS business_email,
    business.value:phone::VARCHAR AS business_phone,
    personal.value:email::VARCHAR AS personal_email,
    personal.value:phone::VARCHAR AS personal_phone
FROM
    customers,
    LATERAL FLATTEN(input=>info:contact[0]) business,
    LATERAL FLATTEN(input=>info:contact[1]) personal;
-- Here we simply used the FLATTEN function twice for each potential
-- path. Our semi-structured data is now structured data.

-- 6.4.0 Try it out!
-- In this section, you're going to query semi-structured data on your
-- own. First you'll run a statement to create a short set of JSON data.

```

	.AST_NAME	BUSINESS_EMAIL	BUSINESS_PHONE	PERSONAL_EMAIL	PERSONAL_PHONE
1	Smith	j.smith@company.com	303-555-1234	jsmith332@gmail.com	303-421-8322
2	Doe	jg_doe@company2.com	303-638-4887	happyjane@gmail.com	303-678-6789

6.4.2 Creating the weather_data table from the weather schema

```
520 -- 6.4.2 Run the statement below to create the data you'll query
521
522 CREATE OR REPLACE SCHEMA WEATHER;
523 USE SCHEMA WEATHER;
524
525 -- 6.4.2 Run the statement below to create the data you'll query
526
527 CREATE OR REPLACE TABLE weather_data
528 AS
529 SELECT
530   parse_json($1) AS w
531   FROM VALUES
532     ('{
533       "data": {
534         "observations": [
535           {
536             "air": {
537               "dew-point": 8.2,
538               "dew-point-quality-code": "1",
539               "temp": 29.8,
540               "temp-quality-code": "1"
541             }
542           }
543         ]
544       }
545     }')
```

status

1 Table WEATHER_DATA successfully created.

Query Details

Query duration 675ms

Rows 1

Query ID 01b8a43f-0004-3f5c-0...

Show more

Ask Copilot

6.4.4 Selecting the USAF, WBAN, country, elev, id and name

```
-- 6.4.4 Select USAF, WBAN, country, elev, id and name from station using dot
-- notation.
SELECT
  wd.w.station.USAF::VARCHAR AS USAF,
  wd.w.station.WBAN::VARCHAR AS WBAN,
  wd.w.station.country::VARCHAR AS country,
  wd.w.station.elev::VARCHAR AS elev,
  wd.w.station.id::VARCHAR AS id,
  wd.w.station.name::VARCHAR AS name|
FROM weather_data as wd;
```

USAF WBAN COUNTRY ELEV ID NAME

1 942340 99999 AS 211 94234099999 DALY WATERS AWS

Query Details

Query duration 47ms

Rows 1

Query ID 01b8a456-0004-3fd0-0...

Show more

Ask Copilot

6.4.5 Selecting the average, max, and min air temp using lateral flatten

The screenshot shows the Snowflake interface with the following details:

- Databases:** CHIPMUNK_DB.WEATHER
- Code:**

```

1016    wd.w.station.USAF::VARCHAR AS USAF,
1017    wd.w.station.WBAN::VARCHAR AS WBAN,
1018    wd.w.station.country::VARCHAR AS country,
1019    wd.w.station.elev::VARCHAR AS elev,
1020    wd.w.station.id::VARCHAR AS id,
1021    wd.w.station.name::VARCHAR AS name
FROM weather_data as wd;
-- 6.4.5  Select average, max and min air temperature using LATERAL FLATTEN.
1024    SELECT
1025      AVG(f.value:air.temp)::NUMBER(38,1) AS avg_t,
1026      MAX(f.value:air.temp) AS max_t,
1027      MIN(f.value:air.temp) AS min_t
1028
1029      FROM weather_data,
1030      LATERAL FLATTEN(w:data.observations) f;
-- 6.4.6  Select max and min atmospheric pressure using LATERAL FLATTEN.
1034    SELECT
1035      MAX(f.value:atmospheric.pressure)::NUMBER(38,1) AS max_pressure,
1036      MIN(f.value:atmospheric.pressure)::NUMBER(38,1) AS min_pressure
1037

```
- Results:** A table showing the results of the query.

	Avg_T	Max_T	Min_T
1	22.4	32.2	13.3

- Query Details:**
 - Query duration: 82ms
 - Rows: 1
 - Query ID: 01b8a456-0004-3f5c-0...
- Buttons:** Results, Chart, Ask Copilot

6.4.6 Selecting the max and min atmospheric pressure using lateral flatten

The screenshot shows the Snowflake interface with the following details:

- Databases:** CHIPMUNK_DB.WEATHER
- Code:**

```

1029
1030      FROM weather_data,
1031      LATERAL FLATTEN(w:data.observations) f;
-- 6.4.6  Select max and min atmospheric pressure using LATERAL FLATTEN.
1034    SELECT
1035      MAX(f.value:atmospheric.pressure)::NUMBER(38,1) AS max_pressure,
1036      MIN(f.value:atmospheric.pressure)::NUMBER(38,1) AS min_pressure
1037
1038      FROM weather_data wd,
1039      LATERAL FLATTEN(input=> w:data.observations) f;
-- 6.4.7  Use either dot notation or FLATTEN LATERAL to fetch the dew point,
-- atmospheric pressure, and wind speed rate for the 15th observation.
1041    SELECT
1042      wd.w.data.observations[15].air."dew-point":NUMBER(38,1) AS dew_point,
1043      wd.w.data.observations[15].atmospheric.pressure:NUMBER(38,1) as pressure,
1044      wd.w.data.observations[15].wind."speed-rate":NUMBER(38, 1) as wind_speed
1045
1046      from weather_data wd;

```
- Results:** A table showing the results of the query.

	MAX_PRESSURE	MIN_PRESSURE
1	10190.0	10123.0

- Query Details:**
 - Query duration: 64ms
 - Rows: 1
 - Query ID: 01b8a457-0004-3fd0-0...
- Buttons:** Results, Chart, Ask Copilot

6.4.8 Selecting the dewpoint, atmospheric pressure and wind rate using dot notation.

The screenshot shows a Snowflake interface with the following details:

- Databases:** Shows a list of databases including ADMIN, BADGER_DB, BEETLE_DB, BISON_DB, BLUEJAY_DB, BOA_DB, CAMEL_DB, CATFISH_DB, CAT_DB, CHEETAH_DB, CHIPMUNK_DB, COBRA_DB, COYOTE_DB, CRICKET_DB, DOG_DB, DOLPHIN_DB, DRAGON_DB, EAGLE_DB, FALCON_DB, FERRET_DB, FINCH_DB, and FLAMINGO_DB.
- Worksheet Title:** CHIPMUNK_DB.WEATHER
- Query:**

```
1040
1041 -- 6.4.7  Use either dot notation or FLATTEN LATERAL to fetch the dew point,
1042 --      atmospheric pressure, and wind speed rate for the 15th observation.
1043
1044 SELECT
1045   wd.w$data.observations[15].air."dew-point":NUMBER(38,1) AS dew_point,
1046   wd.w$data.observations[15].atmospheric.pressure:NUMBER(38,1) as pressure,
1047   wd.w$data.observations[15].wind."speed-rate":NUMBER(38, 1) as wind_speed
1048
1049   from weather_data wd;
1050
1051 -- 6.4.8  Check the solution just after the Key Takeaways if you need help.
1052 --      Otherwise congratulations! You've just completed this lab.
1053
1054 -- 6.5.0  Key Takeaways
1055 --      - JSON data, is stored in a column of data type VARIANT. This is true
1056 --      for all other forms of semi-structured data capable of being stored
1057 --      in Snowflake.
1058 --      - While you can use VARIANT data types in comparisons or aggregations
1059 --      with other data types, your queries won't be as performant as they
1060 --      would be if you had cast the VARIANT columns to standard SQL data
1061 --      types.
```
- Results:** A table with three rows of data:

	DEW_POINT	PRESSURE	WIND_SPEED
1	9.4	10190.0	21.0
2	10.0	10180.0	22.0
3	10.5	10170.0	23.0
- Query Details:**
 - Query duration: 58ms
 - Rows: 1
 - Query ID: 01b8a457-0004-3f5c-0...
- Buttons:** Results, Chart, Ask Copilot