Joining in Snowflake

INTRODUCTION TO SNOWFLAKE SQL

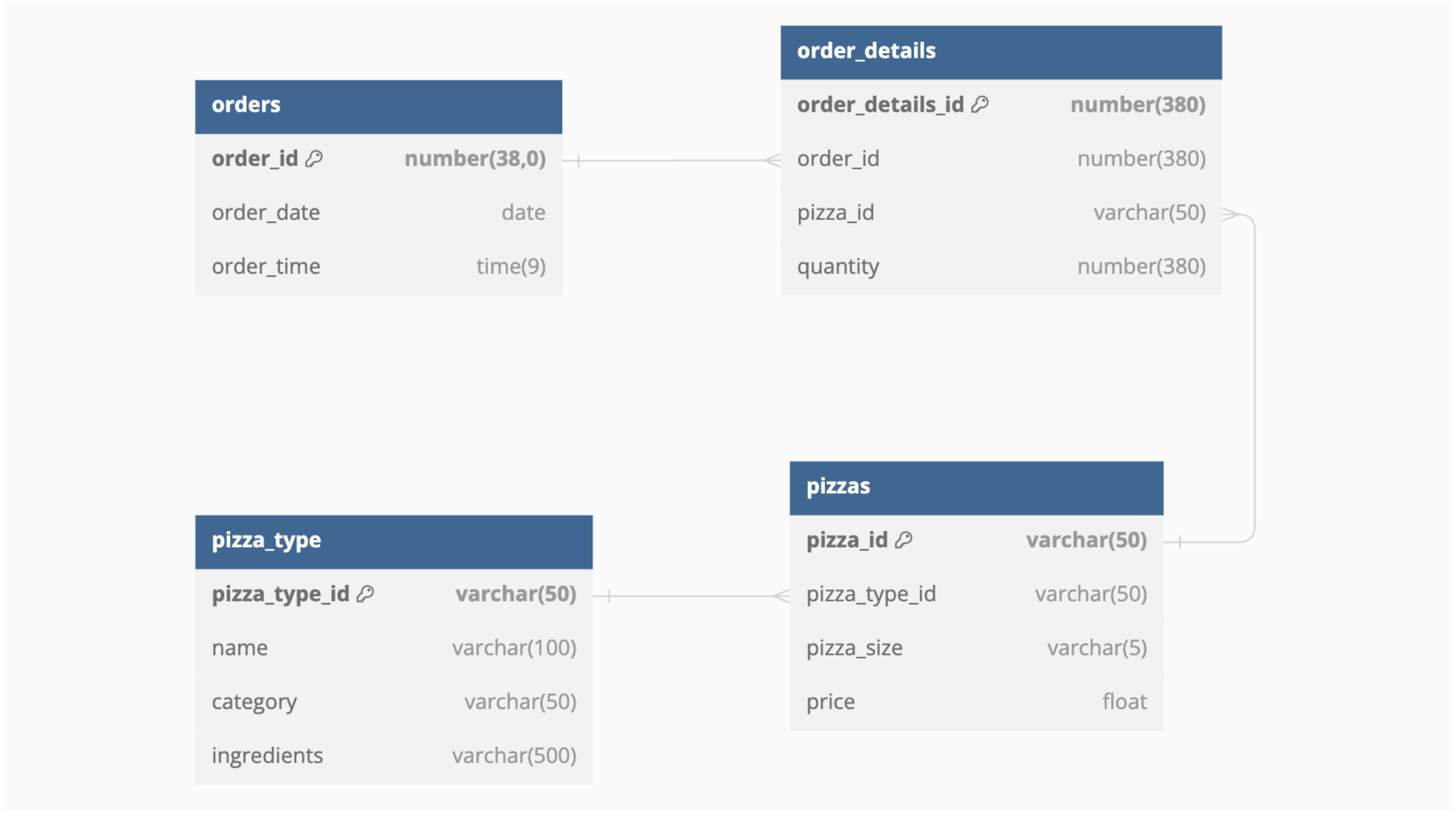
George Boorman
Senior Curriculum Manager, DataCamp

datacamp

JOINS

- INNER JOIN
- OUTER JOINS
  - LEFT OUTER JOIN or LEFT JOIN
  - RIGHT OUTER JOIN or RIGHT JOIN
  - FULL OUTER JOIN or FULL JOIN
- CROSS JOINS
- SELF JOINS
- NATURAL JOIN
- LATERAL JOIN

Pizza dataset



NATURAL JOIN

- `NATURAL JOIN` automatically match columns and eliminate duplicated ones

Syntax:

```
SELECT ...
FROM <table_one> [
                    {
                      | NATURAL [ { LEFT | RIGHT | FULL } [ OUTER ] ]
                    }
                  ]
                  JOIN <table_two>
[ ... ]
```

NATURAL JOIN

Without `NATURAL JOIN`

```
SELECT *
FROM pizzas AS p
JOIN  pizza_type AS t
    ON t.pizza_type_id = p.pizza_type_id
```

PIZZA_ID	PIZZA_TYPE_ID	PIZZA_SIZE	PRICE	PIZZA_TYPE_ID_2
bbq_ckn_s	bbq_ckn	S	12.75	bbq_ckn
bbq_ckn_m	bbq_ckn	M	16.75	bbq_ckn
bbq_ckn_l	bbq_ckn	L	20.75	bbq_ckn
cali_ckn_s	cali_ckn	S	12.75	cali_ckn
cali_ckn_m	cali_ckn	M	16.75	cali_ckn

With `NATURAL JOIN`

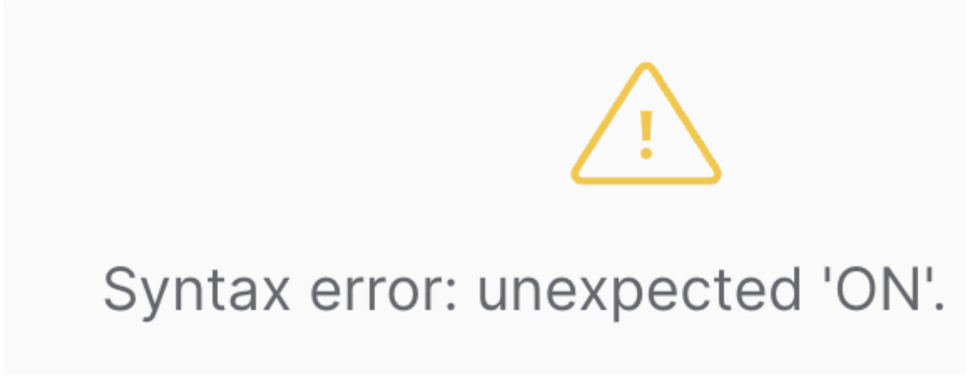
```
SELECT *
FROM pizzas AS p
NATURAL JOIN pizza_type AS t
```

PIZZA_TYPE_ID	PIZZA_ID	PIZZA_SIZE	PRICE	NAME
bbq_ckn	bbq_ckn_s	S	12.75	The Barbecue Chicken Pizza
bbq_ckn	bbq_ckn_m	M	16.75	The Barbecue Chicken Pizza
bbq_ckn	bbq_ckn_l	L	20.75	The Barbecue Chicken Pizza
cali_ckn	cali_ckn_s	S	12.75	The California Chicken Pizza
cali_ckn	cali_ckn_m	M	16.75	The California Chicken Pizza

NATURAL JOIN

NOT ALLOWED

```
select *
FROM pizzas AS p
NATURAL JOIN pizza_type AS t
    ON  t.pizza_type_id = p.pizza_type_id
```



NATURAL JOIN

ALLOWED

- `WHERE` clause

```
SELECT *
FROM pizzas AS p
NATURAL JOIN pizza_type AS t
WHERE pizza_type_id = 'bbq_ckn'
```

LATERAL JOIN

- `LATERAL JOIN` : lets a subquery in `FROM` reference columns from preceding tables or views.

Syntax:

```
SELECT ...
FROM <left_hand_expression> , --
LATERAL
(<right_hand_expression>)
```

- `left_hand_expression` - Table, view, or subquery
- `right_hand_expression` - Inline view or subquery

LATERAL JOIN with a subquery

```
SELECT
    p.pizza_id,
    lat.name,
    lat.category
FROM pizzas AS p,
LATERAL -- Keyword LATERAL
        ( SELECT *
          FROM pizza_type AS t
         -- Referencing outer query column: p.pizza_type_id
          WHERE p.pizza_type_id = t.pizza_type_id
        ) AS lat
```

Why LATERAL JOIN?

```
SELECT
    *
FROM orders AS o,
LATERAL (
   -- Subquery calculating total_spent
    SELECT
        SUM(p.price * od.quantity) AS total_spent
    FROM order_details AS od
    JOIN pizzas AS p
          ON od.pizza_id = p.pizza_id
    WHERE o.order_id = od.order_id
) AS t
ORDER BY o.order_id
```

Let's practice!

INTRODUCTION TO SNOWFLAKE SQL

Subquerying and Common Table Expressions

INTRODUCTION TO SNOWFLAKE SQL

George Boorman
Senior Curriculum Manager, DataCamp

datacamp

Subquerying

- Nested queries
- Used in `FROM`, `WHERE`, `HAVING` or `SELECT` clauses
- Example:

```
SELECT column1
FROM table1
WHERE column1 = (SELECT column2 FROM table2 WHERE condition)
```

- Types: Correlated and uncorrelated subqueries

Uncorrelated subquery

```
-- Main query returns pizzas priced at the maximum value found in the subquery
SELECT pizza_id
FROM pizzas
-- Uncorrelated subquery that identifies the highest pizza price
WHERE price = (
    SELECT MAX(price)
    FROM pizzas
)
```

- Subquery doesn't interact with the main query

Correlated subquery

- Subquery references columns from the main query

```
SELECT pt.name,
       pz.price,
       pt.category
FROM pizzas AS pz
JOIN pizza_type AS pt
    ON pz.pizza_type_id = pt.pizza_type_id
WHERE pz.price = (
  -- Identifies max price for each pizza category
    SELECT MAX(p2.price) -- Max price
    FROM pizzas AS p2
    WHERE -- Correlated: uses outer query column
      p2.pizza_type_id = pz.pizza_type_id
)
```

Common Table Expressions

General Syntax:

```
-- WITH keyword
WITH cte1 AS ( -- CTE name
        SELECT col_1, col_2
            FROM table1
    )
    ...
SELECT ...
FROM cte1 -- Query CTE
;
```

Common Table Expressions

```
WITH max_price AS ( -- CTE called max_price
    SELECT pizza_type_id,
           MAX(price) AS max_price
    FROM pizzas
    GROUP BY pizza_type_id
)
-- Main query
SELECT pt.name,
       pz.price,
       pt.category
FROM pizzas AS pz
JOIN pizza_type AS pt ON pz.pizza_type_id = pt.pizza_type_id
JOIN max_price AS mp  -- Joining with CTE max_price
     ON pt.pizza_type_id = mp.pizza_type_id
WHERE pz.price < mp.max_price -- Compare the price with max_price CTE column
```

Multiple CTEs

```
-- Define multiple CTEs separated by commas
WITH cte1 AS (
    SELECT ...
    FROM ...
),
cte2 AS (
    SELECT ...
    FROM ...
)
-- Main query combining both CTEs
SELECT ...
FROM cte1
JOIN cte2 ON ...
WHERE ...
```

Why Use CTEs?

- Managing complex operations
- Modular
- Readable
- Reusable

Let's practice!

INTRODUCTION TO SNOWFLAKE SQL

Snowflake Query Optimization

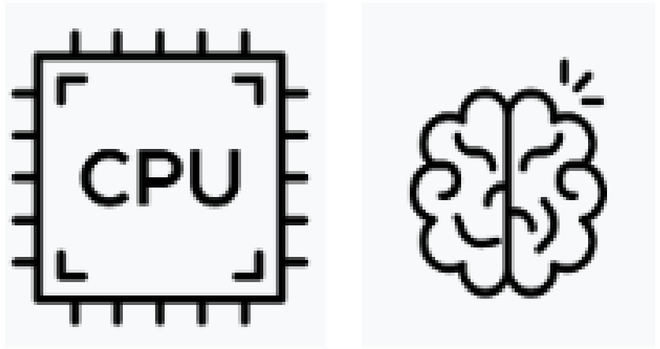
INTRODUCTION TO SNOWFLAKE SQL

George Boorman
Senior Curriculum Manager, DataCamp

datacamp

Why Optimize Queries in Snowflake?

- Achieve faster results
- Cost efficiency
  - Shorter query times consumes fewer resources like CPU and memory.

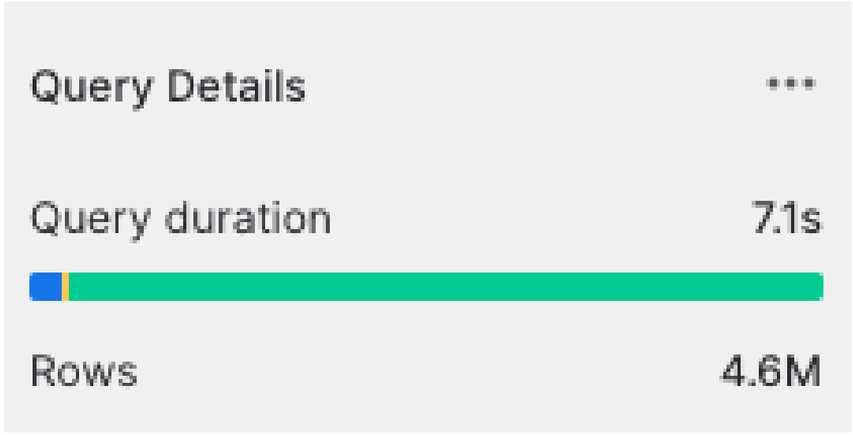


Common query problems

- Exploding Joins: Be cautious!

Incorrect

```
SELECT *
FROM order_details AS od
JOIN pizzas AS p -- Missing ON condition leading to exploding joins
```

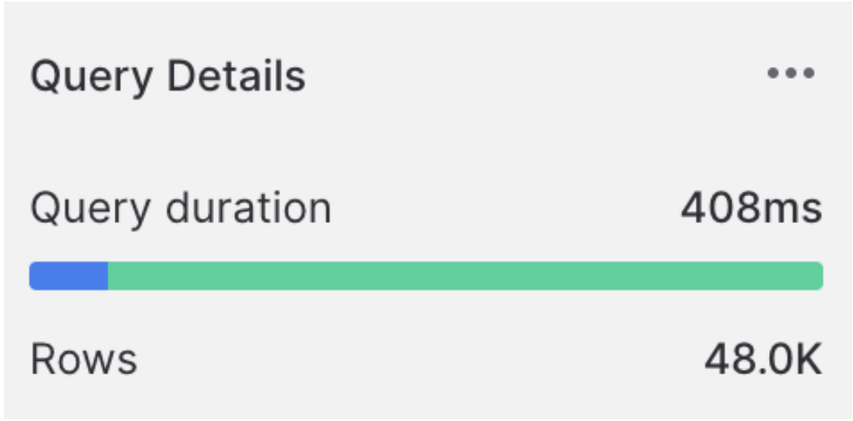


Common query problems

- Exploding Joins: Be cautious!

Correct

```
SELECT *
FROM order_details AS od
JOIN pizzas AS p
ON od.pizza_id = p.pizza_id
```



Common query problems

- `UNION` or `UNION ALL` : Know the difference
  - `UNION` removes duplicates, slows down the query
  - `UNION ALL` is faster if no duplicates
- Handling big data
  - Use filters to narrow down data
  - Apply limits for quicker results

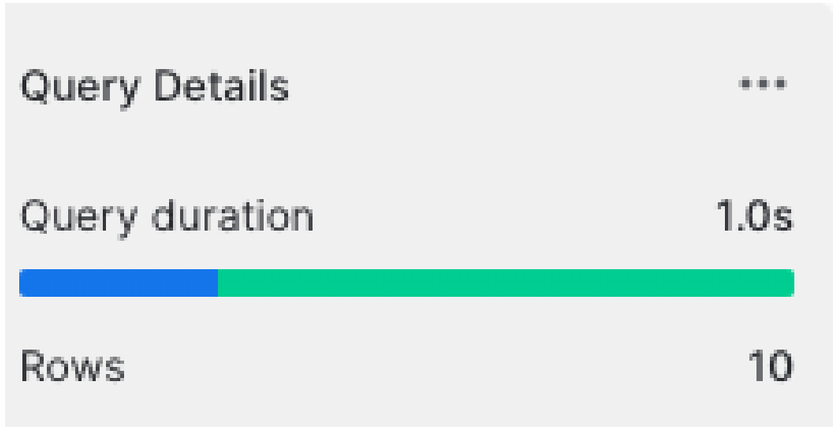
How to optimize queries?

SELECT *

```
SELECT
    *
FROM SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.ORDERS
```

SELECT LIMIT 10* ⚡

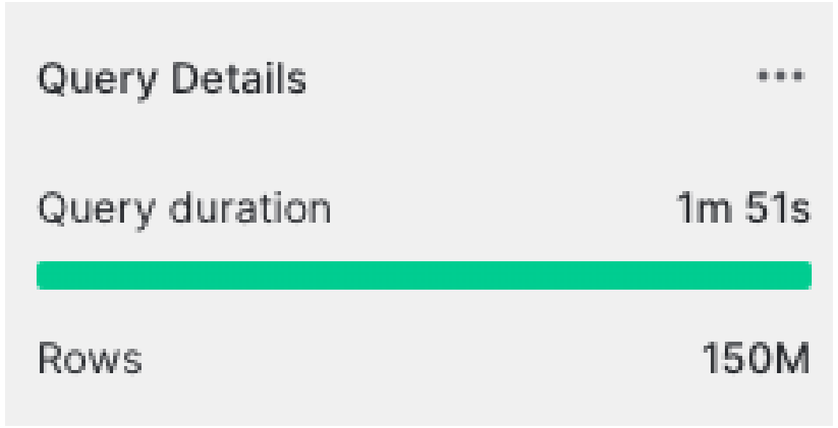
```
SELECT *
FROM SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.ORDERS
LIMIT 10
```



How to optimize queries?

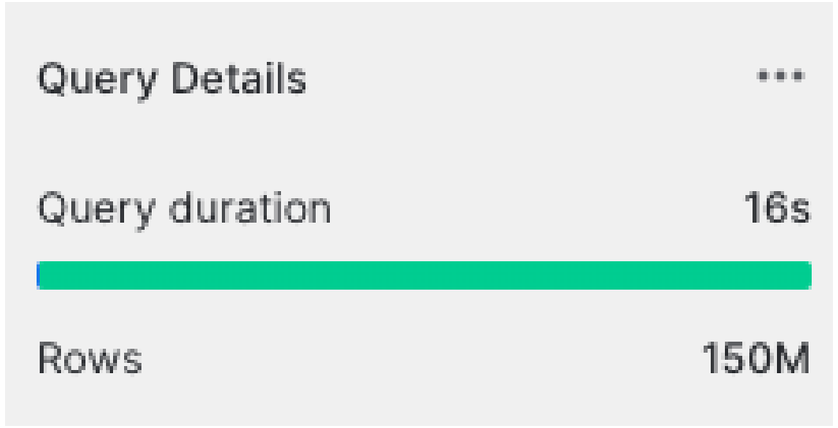
Using `SELECT *`

```
SELECT
    *
FROM SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.ORDERS
```



Avoid `SELECT *` ⚡

```
SELECT o_orderdate,
    o_orderstatus
FROM SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.ORDERS
```



How to optimize queries?

Filter Early

- Use `WHERE` Clause Early On
- Apply filters before `JOIN` s
  - `JOIN` will process fewer rows

Without early filtering

```
SELECT orders.order_id,
       orders.order_date,
       pizza_type.name,
       pizzas.pizza_size
FROM orders
JOIN order_details
ON orders.order_id = order_details.order_id
JOIN pizzas
ON order_details.pizza_id = pizzas.pizza_id
JOIN pizza_type
ON pizzas.pizza_type_id = pizza_type.pizza_type_id
WHERE orders.order_date = '2015-01-01';  -- Filtering after JOIN
```

With early filtering

```
WITH filtered_orders AS (
  SELECT *
  FROM orders
  WHERE order_date = '2015-01-01'  -- Filtering in CTE before JOIN
)
SELECT filtered_orders.order_id,
       filtered_orders.order_date,
       pizza_type.name,
       pizzas.pizza_size
FROM filtered_orders -- Joining with CTE
JOIN order_details
ON filtered_orders.order_id = order_details.order_id
JOIN pizzas
ON order_details.pizza_id = pizzas.pizza_id
```

Query history

- Query History
  - `snowflake.account_usage.query_history`
  - Query History provides different metrics such as execution time

```
SELECT query_text, start_time, end_time, execution_time
FROM
   snowflake.account_usage.query_history
WHERE query_text ILIKE '%order_details%'
```

QUERY_TEXT ...	START_TIME	END_TIME	EXECUTION_TIME
SELECT * FROM order_details AS od JOIN pizzas AS p ON od.pizza_id = p.pizza_id	2023-09-01 03:44:37.233 -0700	2023-09-01 03:44:38.309 -0700	529
SELECT * FROM order_details AS od JOIN pizzas AS p;	2023-09-01 03:43:37.899 -0700	2023-09-01 03:43:47.369 -0700	8,747

- `ILIKE` : Case-insensitive string-matching

Query history

- Spot slow or frequently running queries

```
SELECT query_text,
    start_time,
    end_time,
    execution_time
FROM
   snowflake.account_usage.query_history
WHERE
   execution_time > 1000
```

QUERY_TEXT ···	START_TIME	END_TIME	EXECUTION_TIME
select * from customer	2023-08-08 10:21:24.128 -0700	2023-08-08 10:21:53.826 -0700	29,176

Let's practice!

INTRODUCTION TO SNOWFLAKE SQL

Handling semi-structured data

INTRODUCTION TO SNOWFLAKE SQL

George Boorman
Senior Curriculum Manager, DataCamp

datacamp

Structured versus semi-structured

Example of structured data

cust_id	cust_name	cust_age	cust_email
1	cust1	40	cust1***@gmail.com
2	cust2	35	cust2***@gmail.com
3	cust3	42	cust3***@gmail.com

Example of semi-structured data

```
{
  "cust_id": 1,
  "cust_name": "cust1",
  "cust_age": 40,
  "cust_email": "cust1***@gmail.com"
},
{
  "cust_id": 2,
  "cust_name": "cust2",
  "cust_age": 35,
  "cust_email": [
    "cust2***@gmail.com",
    "cust2_alternate***@gmail.com"
  ]
}
```

Introducing JSON

- JavaScript Object Notation
- **Common use cases:** Web APIs and Config files
- **JSON data structure:**
  - Key-Value Pairs, e.g., `cust_id: 1`

```
{
  "cust_id": 1,
  "cust_name": "cust1",
  "cust_age": 40,
  "cust_email": "cust1***@gmail.com"
}
```

JSON in Snowflake

- Native JSON support
- Flexible for evolving schemas

Comparisons:

- Postgres: Uses `JSONB`
- Snowflake: Uses `VARIANT`

How Snowflake stores JSON data

- `VARIANT` supports OBJECT and ARRAY data types
  - OBJECT: `{ "key": "value"}`
  - ARRAY: `["list", "of", "values"]`
- Creating a Snowflake Table to handle JSON data

```
CREATE TABLE cust_info_json_data (
  customer_id INT,
  customer_info VARIANT -- VARIANT data type
);
```

Semi-structured data functions

- `PARSE_JSON`
  - `expr` : JSON data in string format
  - Returns: `VARIANT` type, valid JSON object

PARSE_JSON

Example:

```
SELECT PARSE_JSON(
  -- Enclosed in strings
  '{
  "cust_id": 1,
  "cust_name": "cust1",
  "cust_age": 40,
  "cust_email":"cust1***@gmail.com"
  }
  '-- Enclosed in strings
) AS customer_info_json
```

CUSTOMER_INFO_JSON
{ "cust_age": 40, "cust_email": "cust1***@gmail.com", "cust_id": 1, "cust_name": "cust1" }

OBJECT_CONSTRUCT

- `OBJECT_CONSTRUCT`
  - Syntax: `OBJECT_CONSTRUCT( [<key1>, <value1> [, <keyN>, <valueN> ...]] )`
  - Returns: JSON object

```
SELECT OBJECT_CONSTRUCT(
  -- Comma separated values rather than : notation
      'cust_id', 1,
      'cust_name', 'cust1',
      'cust_age', 40,
      'cust_email', 'cust1***@gmail.com'
)
```

ID	CUSTOMER_INFO
2	{ "cust_age": 40, "cust_email": "cust1***@gmail.com", "cust_id": 1, "cust_name": "cust1" }

Querying JSON data in Snowflake

Simple JSON

- `:`

```
SELECT
  customer_info:cust_age, -- Use colon to access cust_age from column
  customer_info:cust_name,
  customer_info:cust_email,
FROM
  cust_info_json_data;
```

CUSTOMER_INFO:CUST_AGE	CUSTOMER_INFO:CUST_NAME	CUSTOMER_INFO:CUST_EMAIL
40	"cust1"	"cust1***@gmail.com"

Querying nested JSON Data in Snowflake

Example of nested JSON

```
{
  "cust_id": 1,
  "cust_name": "cust1",
  "cust_age": 40,
  "cust_email": "cust1***@gmail.com",
  "address": {
    "street": "St1",
    "city": "Cityv",
    "state": "Statev"
  }
}
```

- Colon: `:`
- Dot: `.`

Querying nested JSON using colon/dot notations

Accessing values using colon notation

```
<column>:<level1_element>:
<level2_element>:<level3_element>
```

```
SELECT
    customer_info:address:street AS street_name
FROM
    cust_info_json_data
```

STREET_NAME
"St1"

Accessing values using dot notation

```
<column>:<level1_element>.
<level2_element>.<level3_element>
```

```
SELECT
    customer_info:address.street AS street_name
FROM
    cust_info_json_data
```

STREET_NAME
"St1"

Let's practice!

INTRODUCTION TO SNOWFLAKE SQL

Wrap-up

INTRODUCTION TO SNOWFLAKE SQL

George Boorman
Senior Curriculum Manager, DataCamp

This is just the beginning!

INTRODUCTION TO SNOWFLAKE SQL