

Joining in Snowflake

INTRODUCTION TO SNOWFLAKE



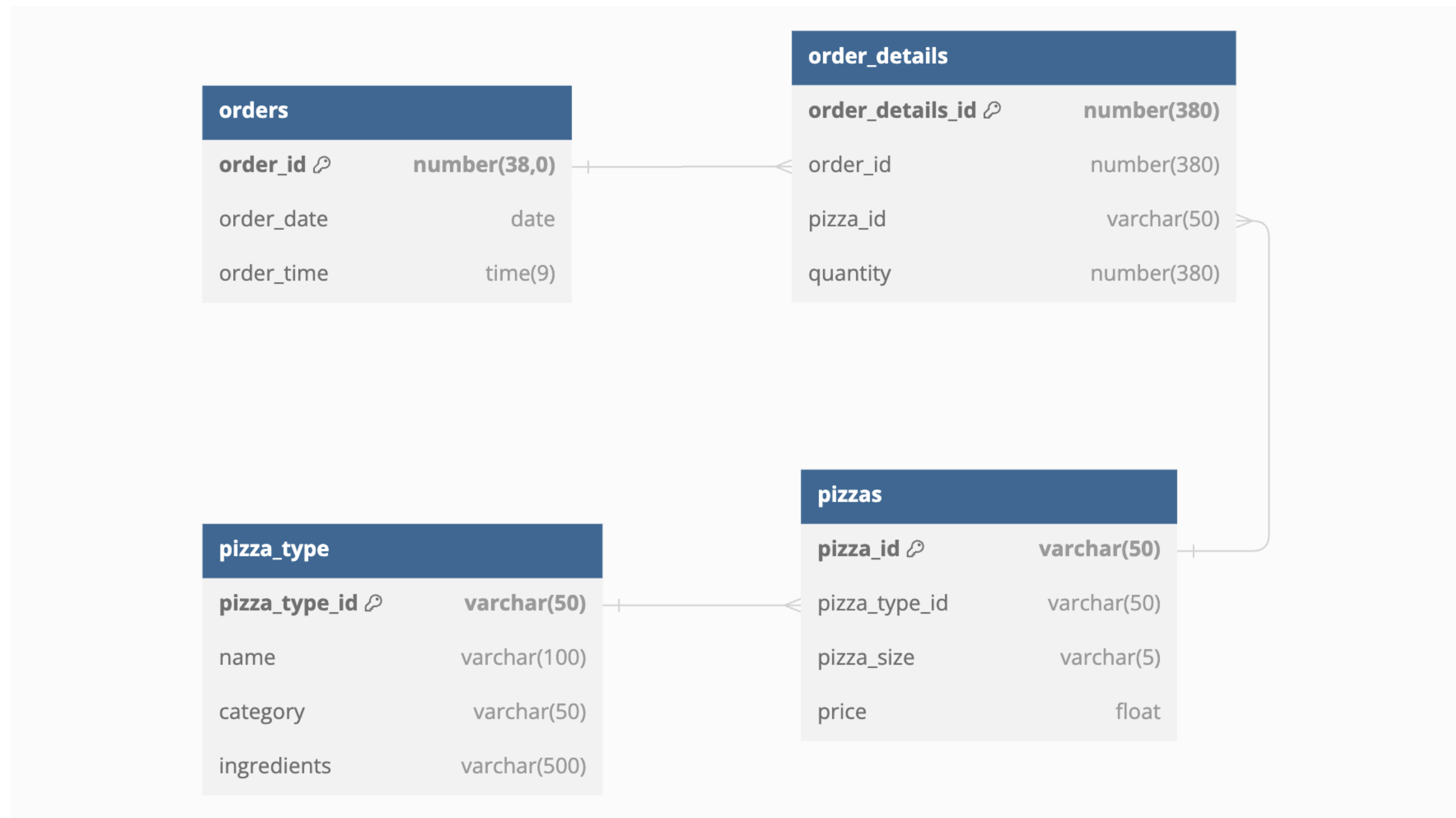
Palak Raina

Senior Data Engineer

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



Similarities with PostgreSQL - INNER JOIN

SELECT

```
pt.name AS pizza_type_name,  
pt.category,  
pt.ingredients,  
p.size,  
p.price
```

FROM

```
pizza_type AS pt
```

INNER JOIN -- Using INNER JOIN

```
pizzas AS p
```

ON

```
pt.pizza_type_id = p.pizza_type_id
```

PIZZA_TYPE_NAME	CATEGORY	INGREDIENTS	SIZE	...	PRICE
The Barbecue Chicken Pizza	Chicken	Barbecued Chicken, Red Pe	S		12.75
The Barbecue Chicken Pizza	Chicken	Barbecued Chicken, Red Pe	M		16.75
The Barbecue Chicken Pizza	Chicken	Barbecued Chicken, Red Pe	L		20.75
The California Chicken Pizza	Chicken	Chicken, Artichoke, Spinach	S		12.75
The California Chicken Pizza	Chicken	Chicken, Artichoke, Spinach	M		16.75

NATURAL JOIN

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
```



Syntax error: unexpected 'ON'.

NATURAL JOIN

ALLOWED

- WHERE clause

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


LATERAL JOIN

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

Subquerying and Common Table Expressions

INTRODUCTION TO SNOWFLAKE



Palak Raina
Senior Data Engineer

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

Correlated subquery

- References columns from the outer 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 highest 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  
)
```

Uncorrelated subquery

- No reference to outer or main query

```
SELECT order_id
FROM order_details AS od
WHERE pizza_id = (
    -- Uncorrelated: standalone subquery
    -- Not referencing to outer query columns
    SELECT pizza_id
    FROM pizzas
    ORDER BY price DESC
    LIMIT 1
)
```


Correlated subquery limitations

- Can't use LIMIT with correlated subquery

```
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 < (  
  SELECT p2.price -- Get price  
  FROM pizzas AS p2  
  WHERE p2.pizza_type_id = pz.pizza_type_id -- Correlated to outer query  
  ORDER BY p2.price DESC -- Order with max price first  
  LIMIT 1 -- LIMIT 1 fetches only the top record, i.e., the max price  
)
```

Correlated subquery limitations

Result:



Unsupported subquery type cannot be evaluated

Using LIMIT in uncorrelated subquery

```
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 < (  
  SELECT p2.price  
  FROM pizzas AS p2 -- No where clause  
  ORDER BY p2.price DESC  
  -- Can use limit here  
  LIMIT 1  
)
```

Result:

NAME	PRICE	CATEGORY
The Barbecue Chicken Pizza	12.75	Chicken
The Barbecue Chicken Pizza	16.75	Chicken
The Barbecue Chicken Pizza	20.75	Chicken
The California Chicken Pizza	12.75	Chicken
The California Chicken Pizza	16.75	Chicken
The California Chicken Pizza	20.75	Chicken
The Chicken Alfredo Pizza	12.75	Chicken
The Chicken Alfredo Pizza	16.75	Chicken

Common Table Expressions

Basic 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: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
- Readable
- Modular
- Reusable

Let's practice!
INTRODUCTION TO SNOWFLAKE

Snowflake Query Optimization

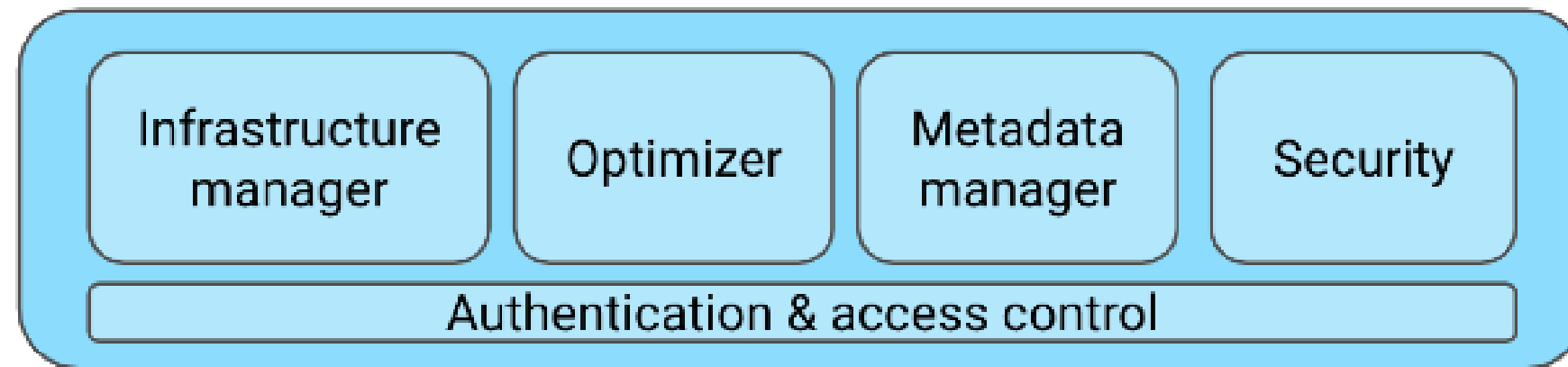
INTRODUCTION TO SNOWFLAKE



Palak Raina
Senior Data Engineer

What's Snowflake Query Optimization?

- Transforming into more efficient queries
- Snowflake's Cloud Services Layer

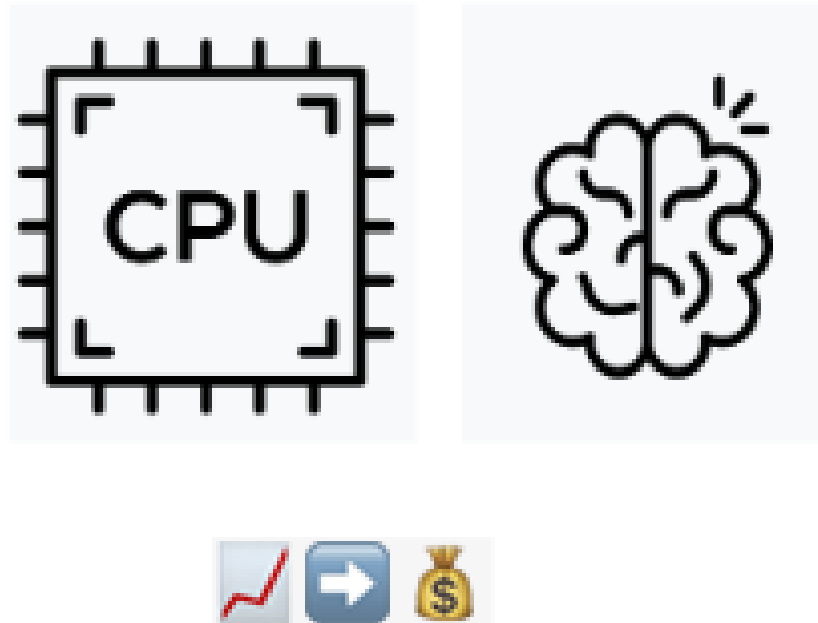


Cloud Services Layer



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
```


Query Details	...
Query duration	7.1s
	
Rows	4.6M

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
```

Query Details	...
Query duration	408ms
	
Rows	48.0K

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
```

Query Details

...

Query duration

1m 51s



Rows

150M

SELECT TOP 10*

```
SELECT
```

```
  TOP 10*
```

```
FROM SNOWFLAKE_SAMPLE_DATA.TPCH_SF100.ORDERS
```

Query Details

...

Query duration

1.0s



Rows

10

How to optimize queries?

SELECT *

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

Query Details	...
Query duration	1m 51s
Rows	150M


LIMIT

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

Query Details	...
Query duration	1.7s
Rows	10


How to optimize queries?

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

Query Details	...
Query duration	1m 51s
	
Rows	150M

Avoid `SELECT *`

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

Query Details	...
Query duration	16s
	
Rows	150M

How to optimize queries?

Filter Early

- Use `WHERE` Clause Early On.
- Apply filters before `JOIN` s

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
```

Query history

- Query History
 - `snowflake.account_usage.query_history`

```
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

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

Handling semi-structured data

INTRODUCTION TO SNOWFLAKE



Palak Raina
Senior Data Engineer

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, Mobile Apps, 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(  
    'cust_id', 1,  
    'cust_name', 'cust1',  
    'cust_age', 40,  
    'cust_email', 'cust1***@gmail.com'  
)
```

- Customer info data stored in table

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 Data

- :

SELECT

```
customer_info:cust_age, -- using colon to access data 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 object

```
{
  "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

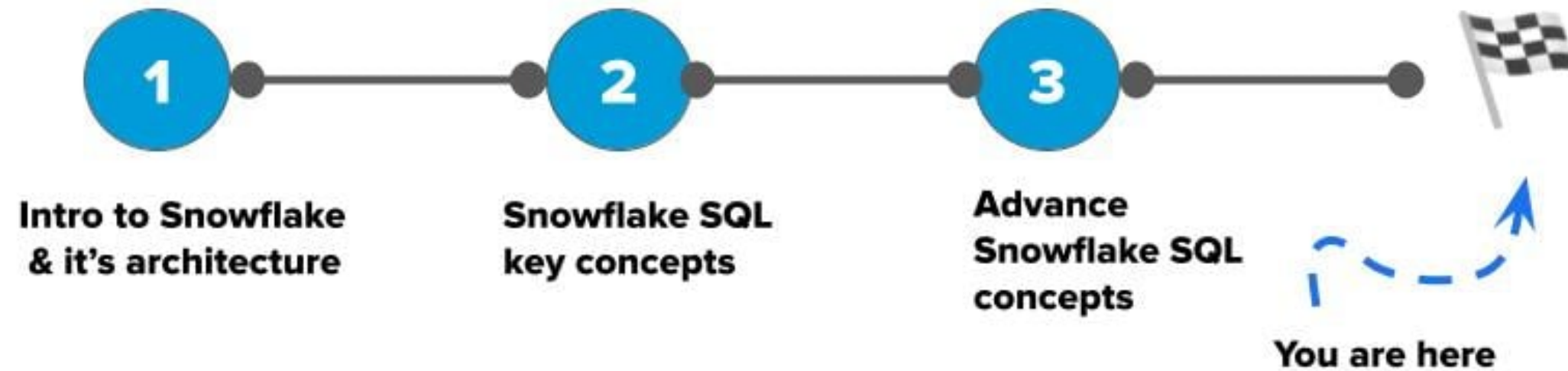
Wrap-up

INTRODUCTION TO SNOWFLAKE

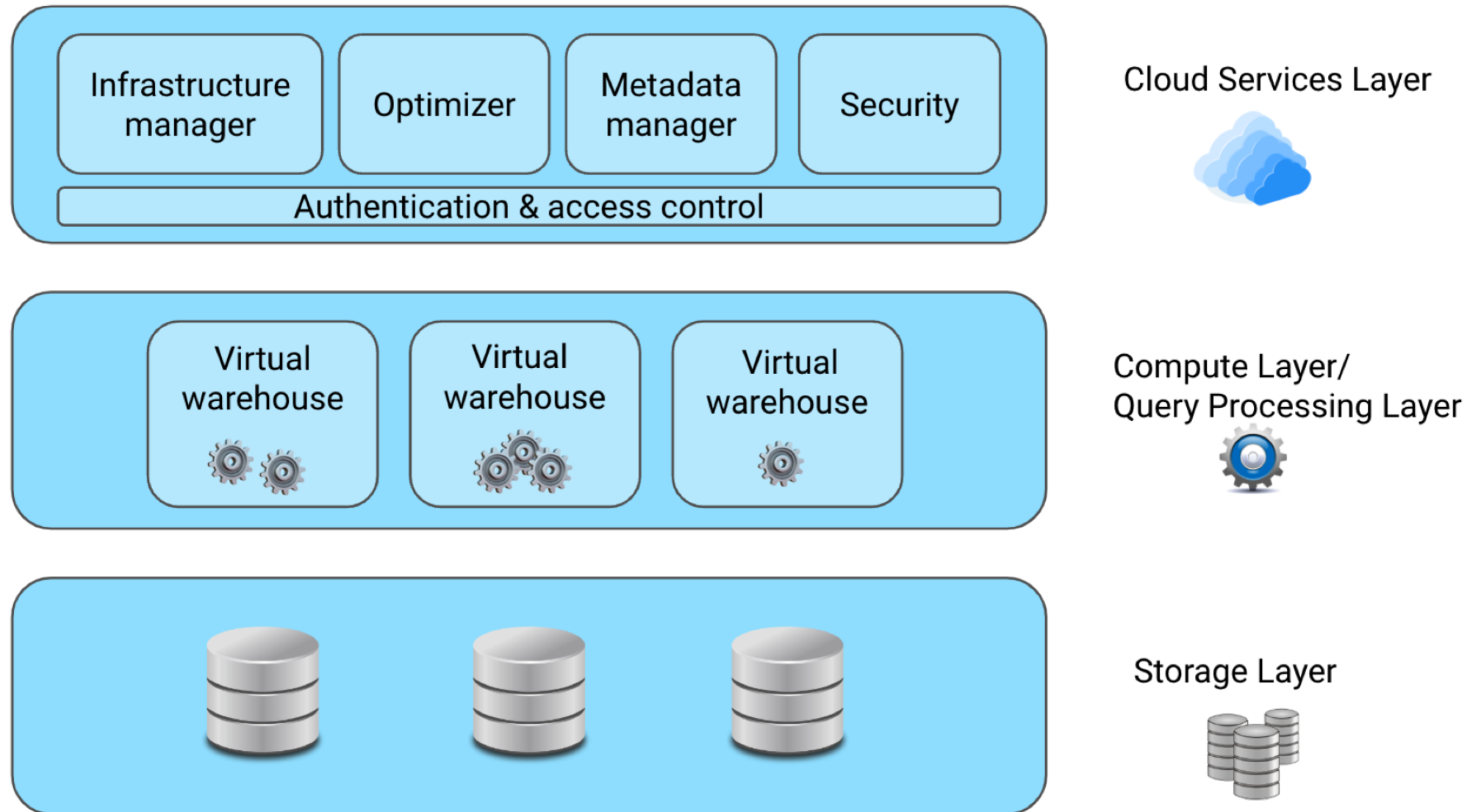


Palak Raina
Senior Data Engineer

Journey



Chapter 1: Architecture, Competitors, and SnowflakeSQL



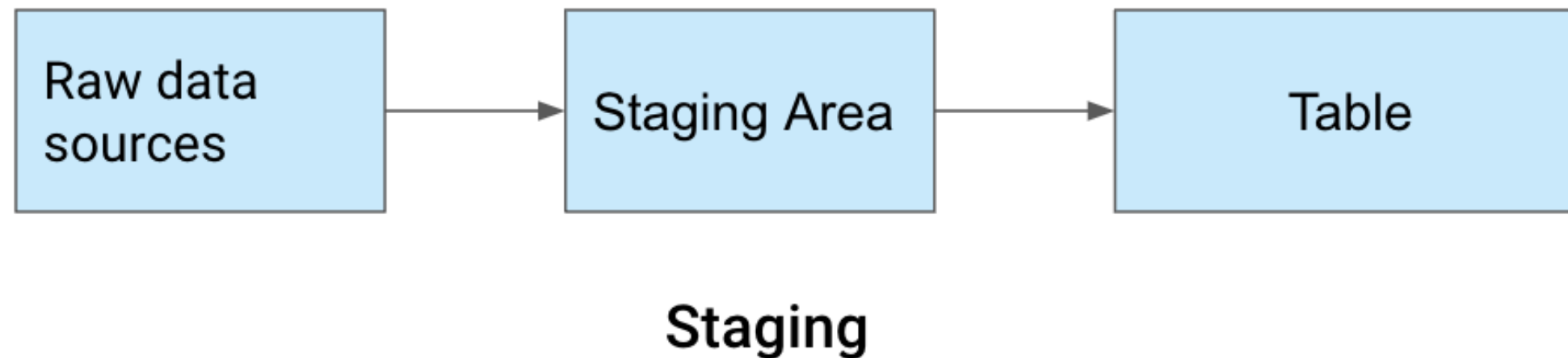
Chapter 1: Architecture, Competitors, and Snowflake SQL



Chapter2: Snowflake SQL and key concepts

Connecting to Snowflake

- WEB UI
- Drivers & Connectors
- SnowSQL



Chapter2: Snowflake SQL and key concepts

Data Definition Language (DDL)

- CREATE
- ALTER
- DROP
- RENAME
- COMMENT

Database structures and DML commands

- SHOW
- DESCRIBE
- INSERT
- UPDATE
- MERGE
- COPY

Chapter2: Snowflake SQL and key concepts

- `VARCHAR`
- `NUMERIC`
- `INT`
- `DATE`
- `TIME`
- `TIMESTAMP`
- `VARIANT` -> Semi-structured data

Data Type conversion - What, Why, How?

Conversion Functions: `TO_VARCHAR` , `TO_DATE`

STRING functions: `CONCAT` , `UPPER` , `LOWER`

DATE & TIME functions: `CURRENT_DATE` ,
`CURRENT_TIME`

EXTRACT functions: `GROUP BY ALL`

Chapter 3: Advance Snowflake SQL Concepts

JOINS

- NATURAL JOIN
- LATERAL JOIN

Subquerying

CTEs

Chapter 3: Advance Snowflake SQL Concepts

Snowflake Query Optimization

- Common query problems: Exploding Joins, `UNION` vs `UNION ALL`
- Rewriting queries: `TOP` , `LIMIT` , Early filtering, Avoid `Select *`

Semi structured data

- `PARSE_JSON` , `OBJECT_CONSTRUCT`
- Querying JSON data in Snowflake

Is this all?

- Much more to unfold.

Not addressed

- Setting context
- Roles, Users
- Setting up Virtual Warehouses
- Window functions
- Query profiling
- Materialized Views
- Clustering
- ...

Useful resources

- Snowflake documentation: <https://docs.snowflake.com/>
- Snowflake forums: <https://community.snowflake.com/s/forum>

**This is just the
beginning!**

INTRODUCTION TO SNOWFLAKE