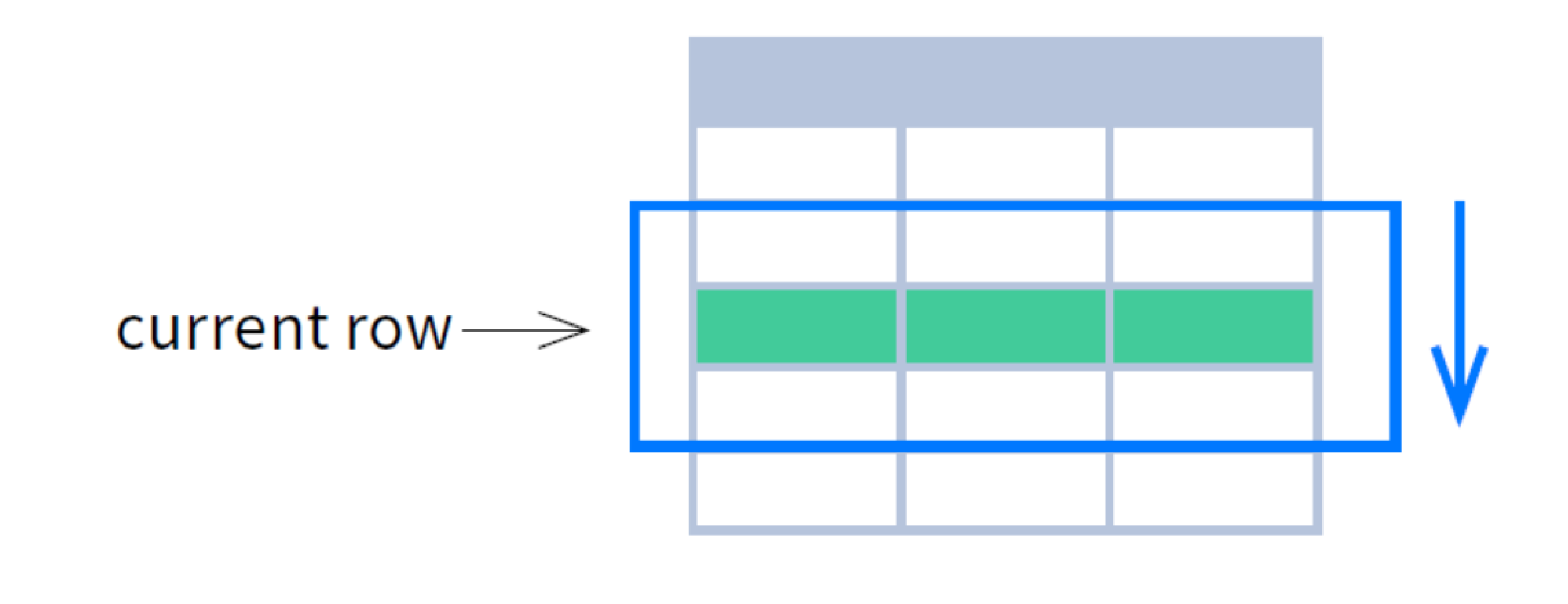
10Alytics

INTERMEDIATE SQL

CHU NGWOKE

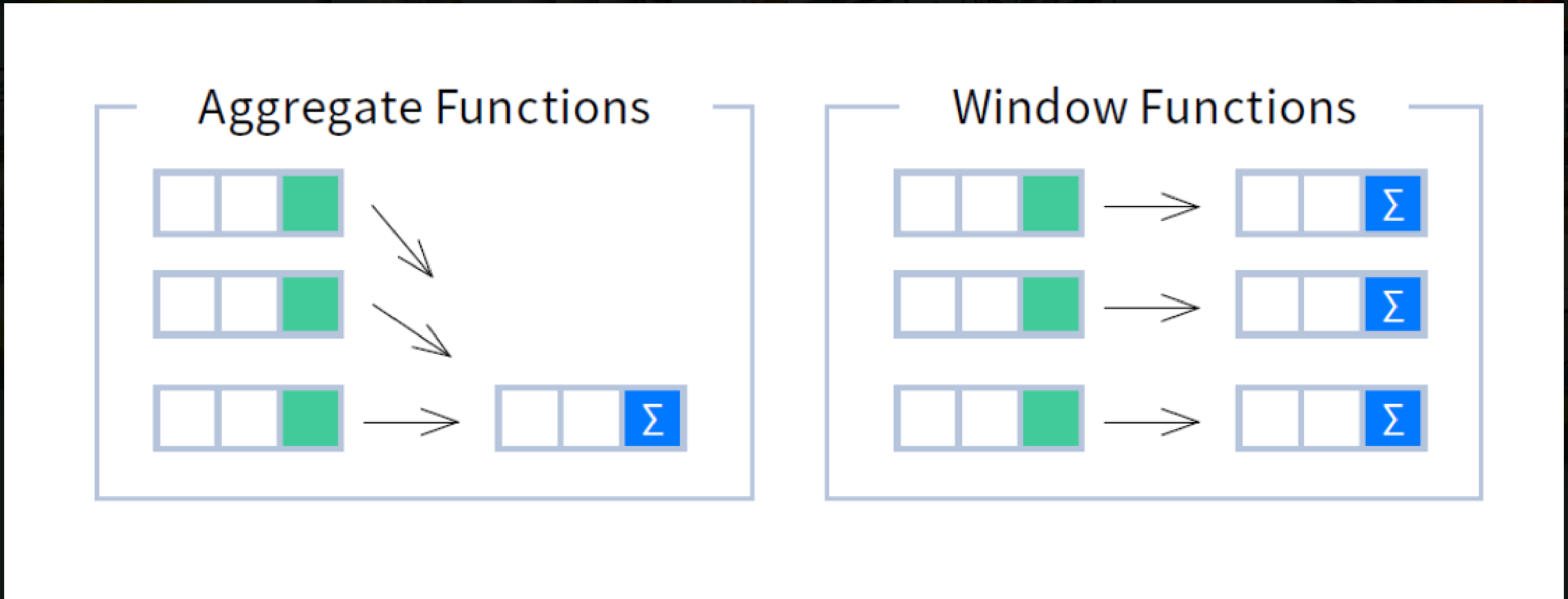
SQL WINDOW FUNCTIONS

Window functions in SQL are a type of function that perform calculations across a set of rows related to the current row within a query result set.



SQL WINDOW FUNCTIONS

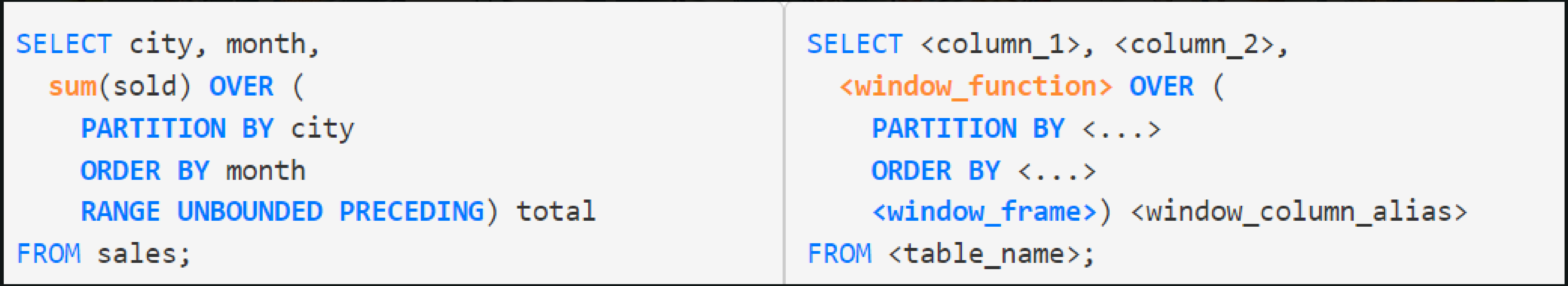
Unlike regular aggregate functions (group by), window functions do not collapse multiple rows into a single output; instead, they maintain the individual rows in the result set while applying the specified function to each row in relation to its "window" of neighboring rows.



SQL WINDOW FUNCTIONS

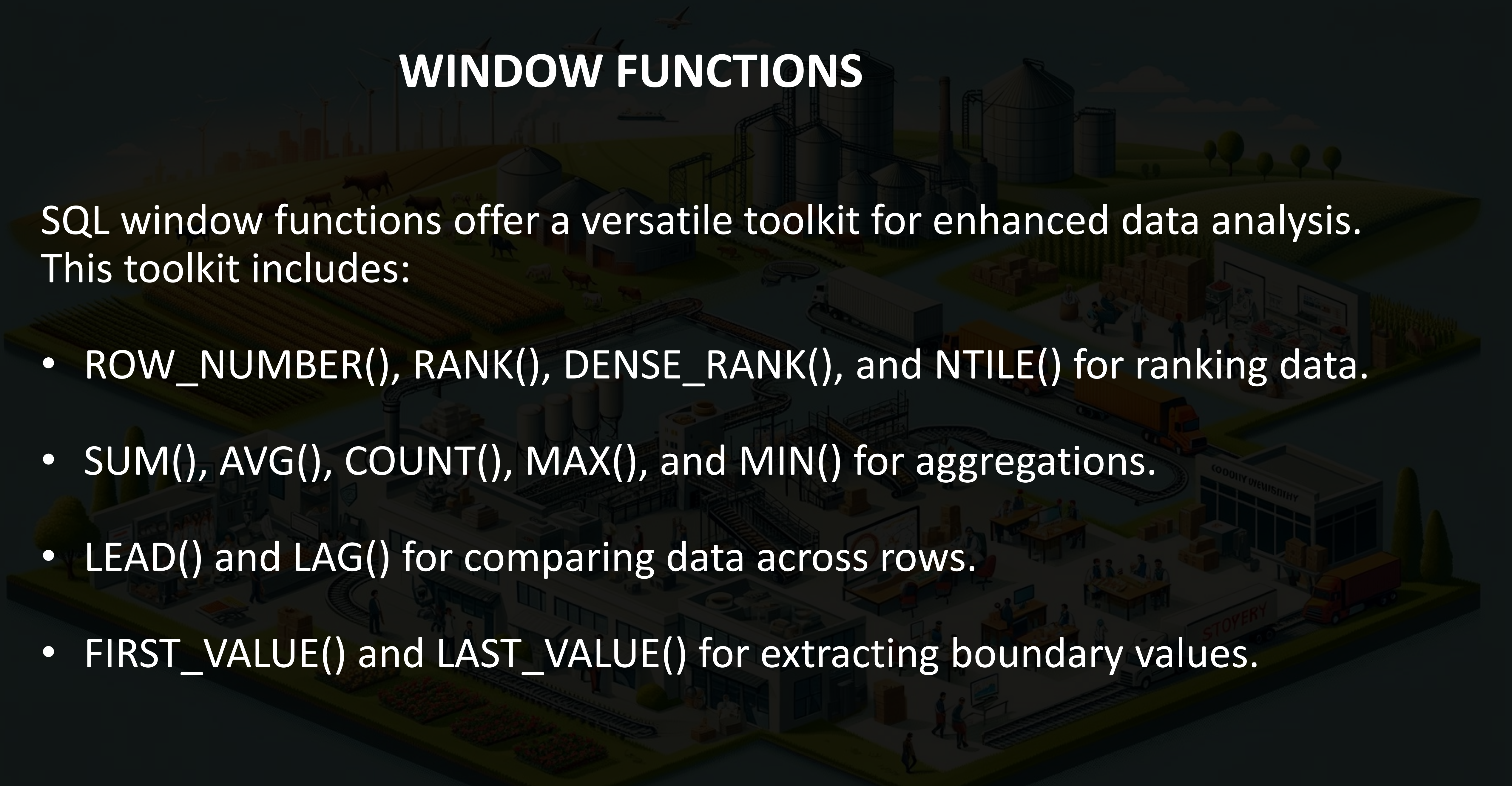
The syntax of a window functions consists of the following keywords/parts

- Window function
- Over()
- Partition by
- Order by
- Window frame



```
SELECT city, month,
  sum(sold) OVER (
    PARTITION BY city
    ORDER BY month
    RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
  <window_function> OVER (
    PARTITION BY <...>
    ORDER BY <...>
    <window_frame>) <window_column_alias>
FROM <table_name>;
```



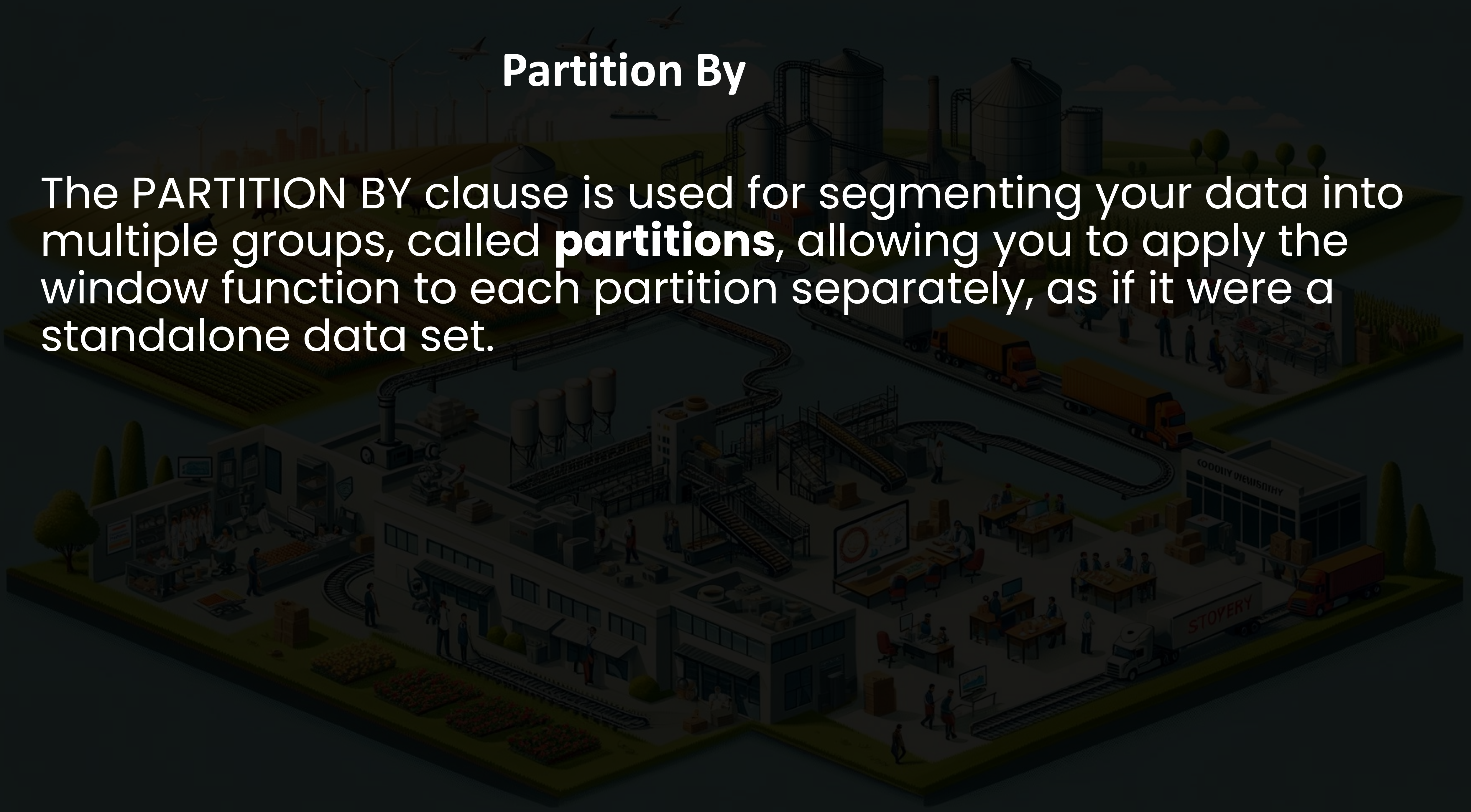
WINDOW FUNCTIONS

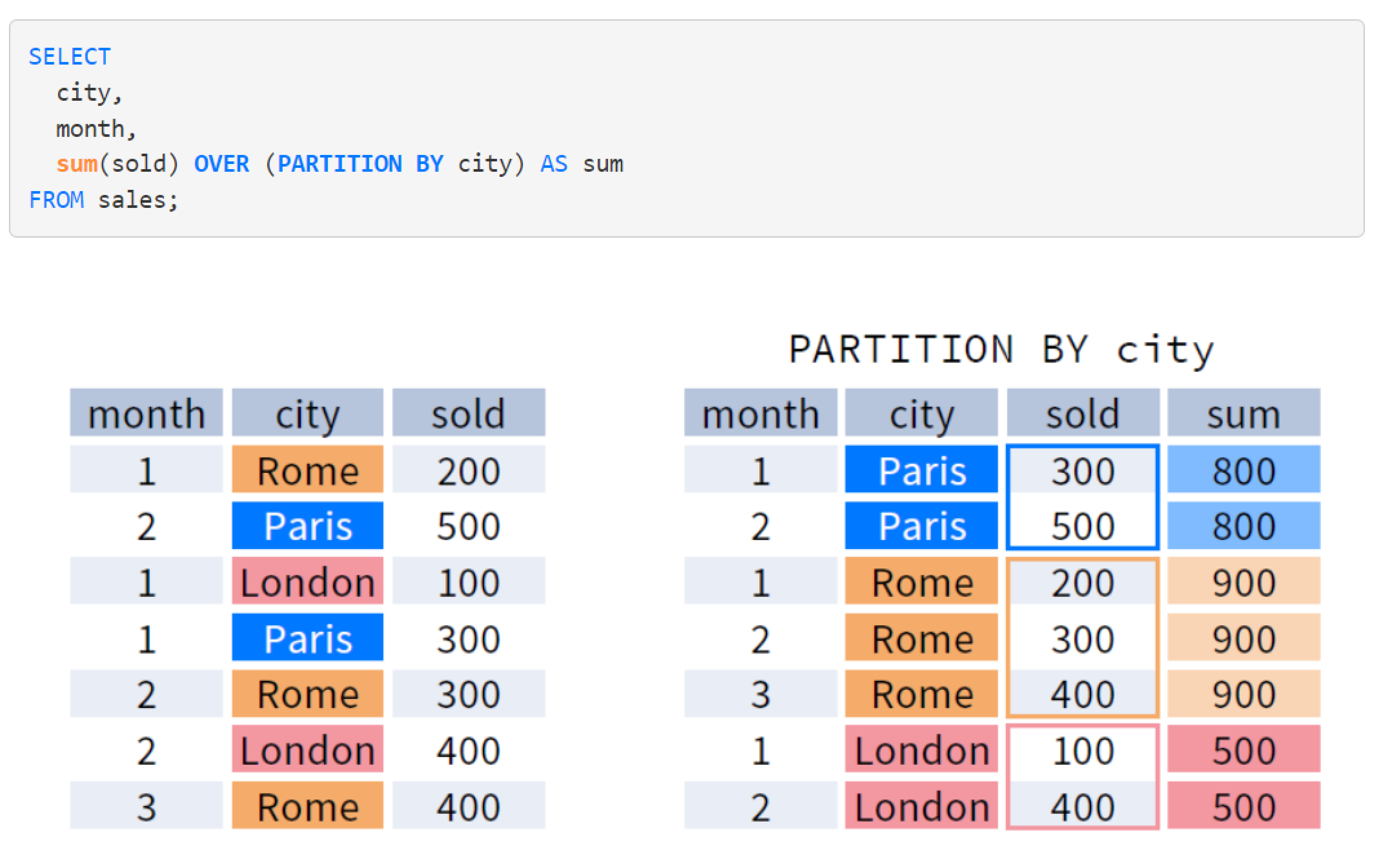
SQL window functions offer a versatile toolkit for enhanced data analysis. This toolkit includes:

- ROW_NUMBER(), RANK(), DENSE_RANK(), and NTILE() for ranking data.
- SUM(), AVG(), COUNT(), MAX(), and MIN() for aggregations.
- LEAD() and LAG() for comparing data across rows.
- FIRST_VALUE() and LAST_VALUE() for extracting boundary values.

Partition By

The PARTITION BY clause is used for segmenting your data into multiple groups, called **partitions**, allowing you to apply the window function to each partition separately, as if it were a standalone data set.



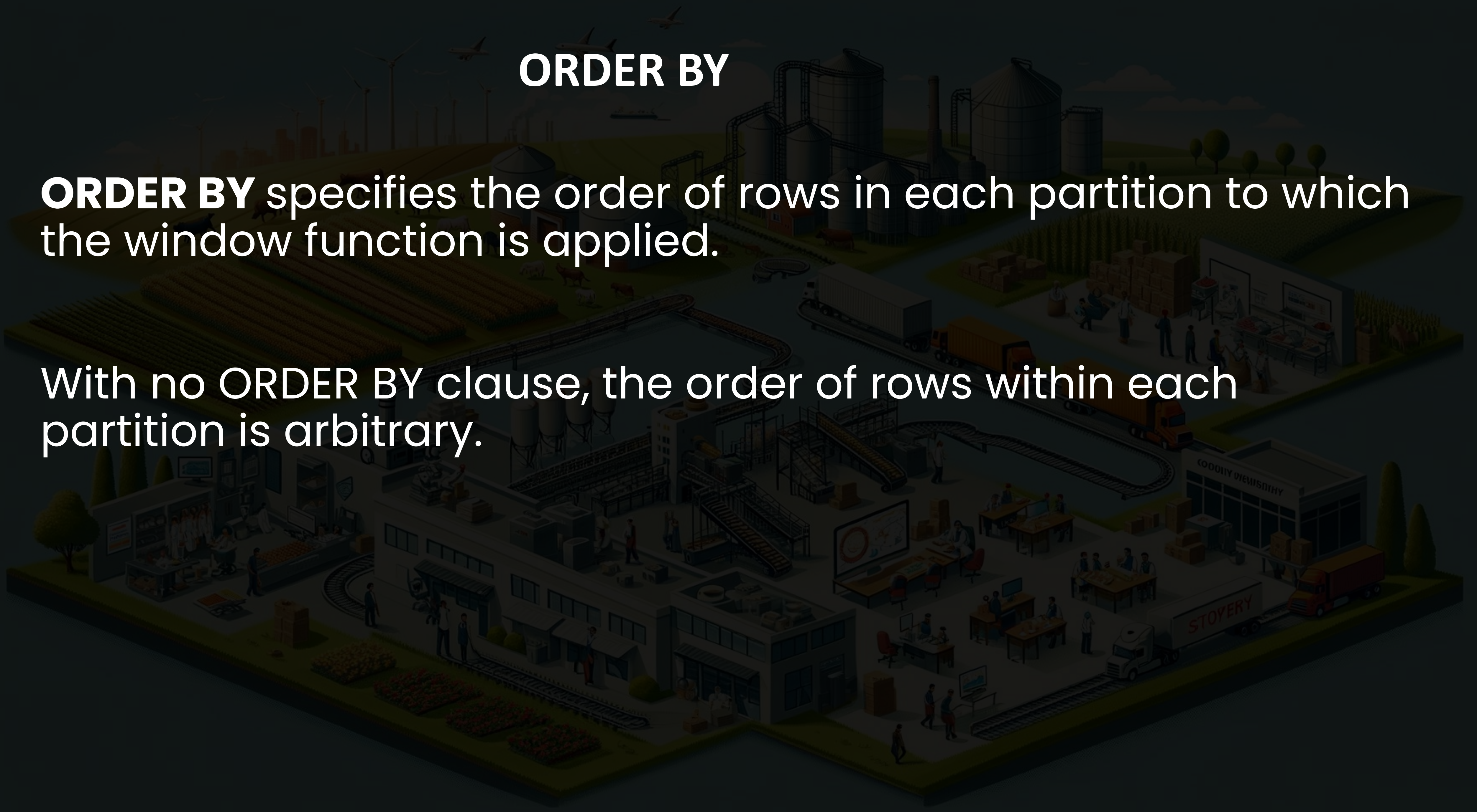


```
SELECT
  city,
  month,
  sum(sold) OVER (PARTITION BY city) AS sum
FROM sales;
```

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

PARTITION BY city

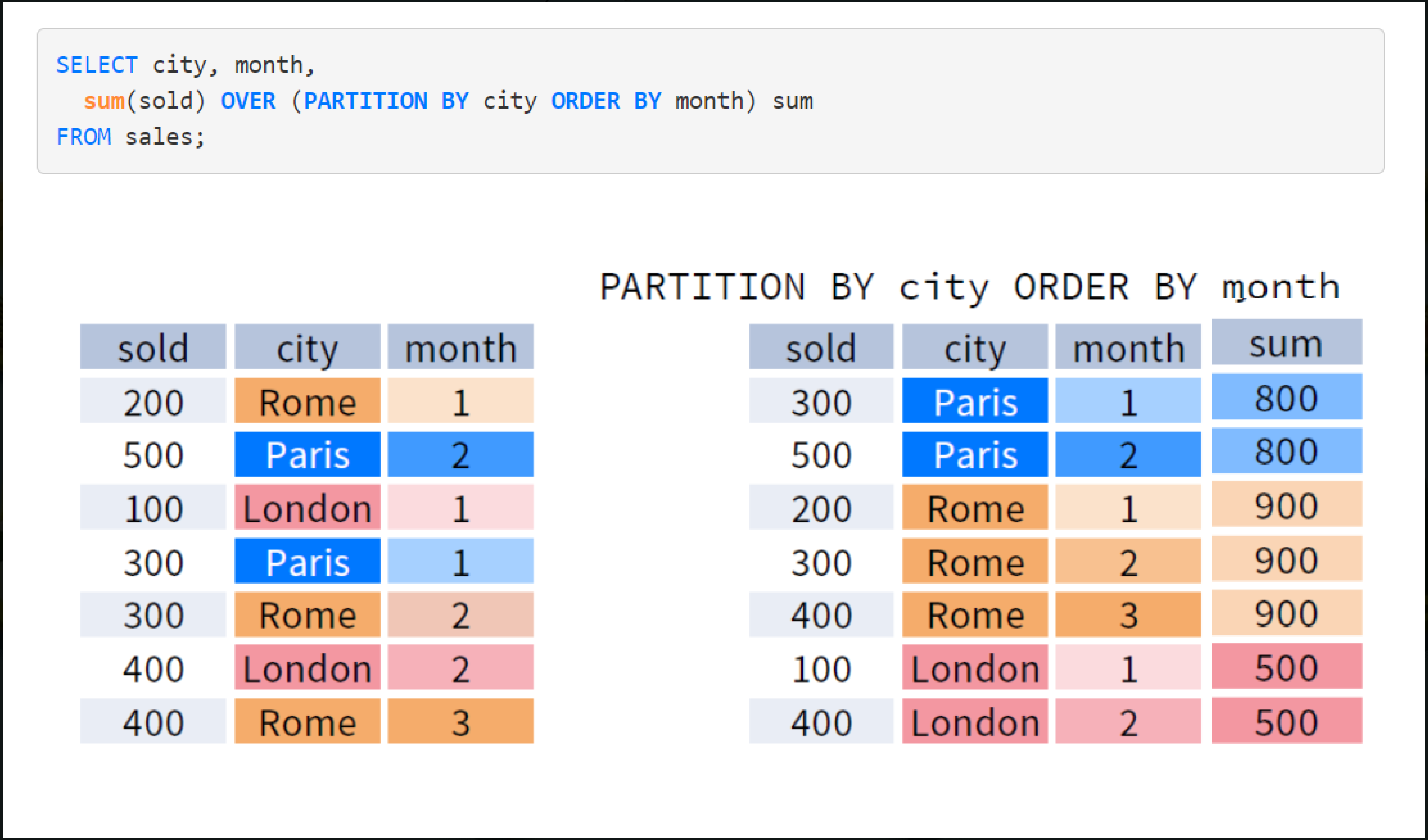
month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500



ORDER BY

ORDER BY specifies the order of rows in each partition to which the window function is applied.

With no ORDER BY clause, the order of rows within each partition is arbitrary.



```
SELECT city, month,
  sum(sold) OVER (PARTITION BY city ORDER BY month) sum
FROM sales;
```

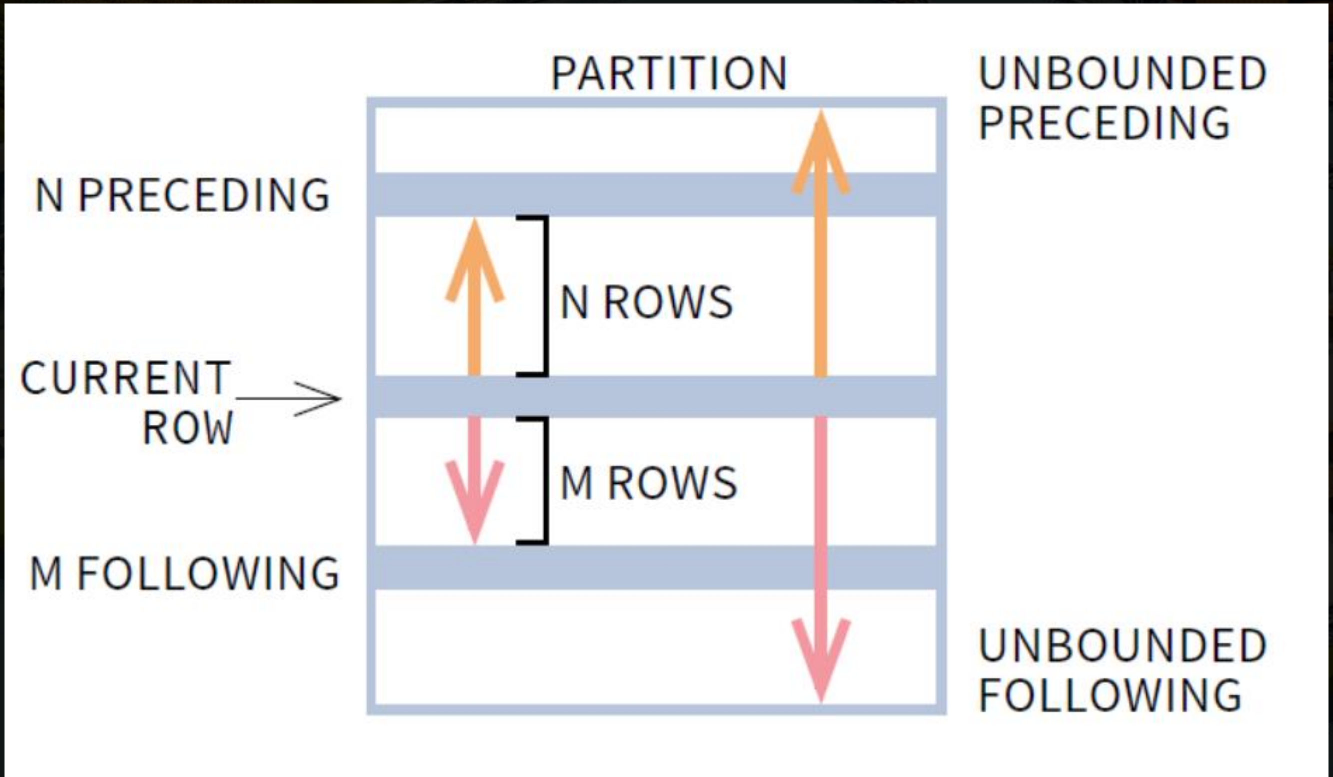
sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

PARTITION BY city ORDER BY month

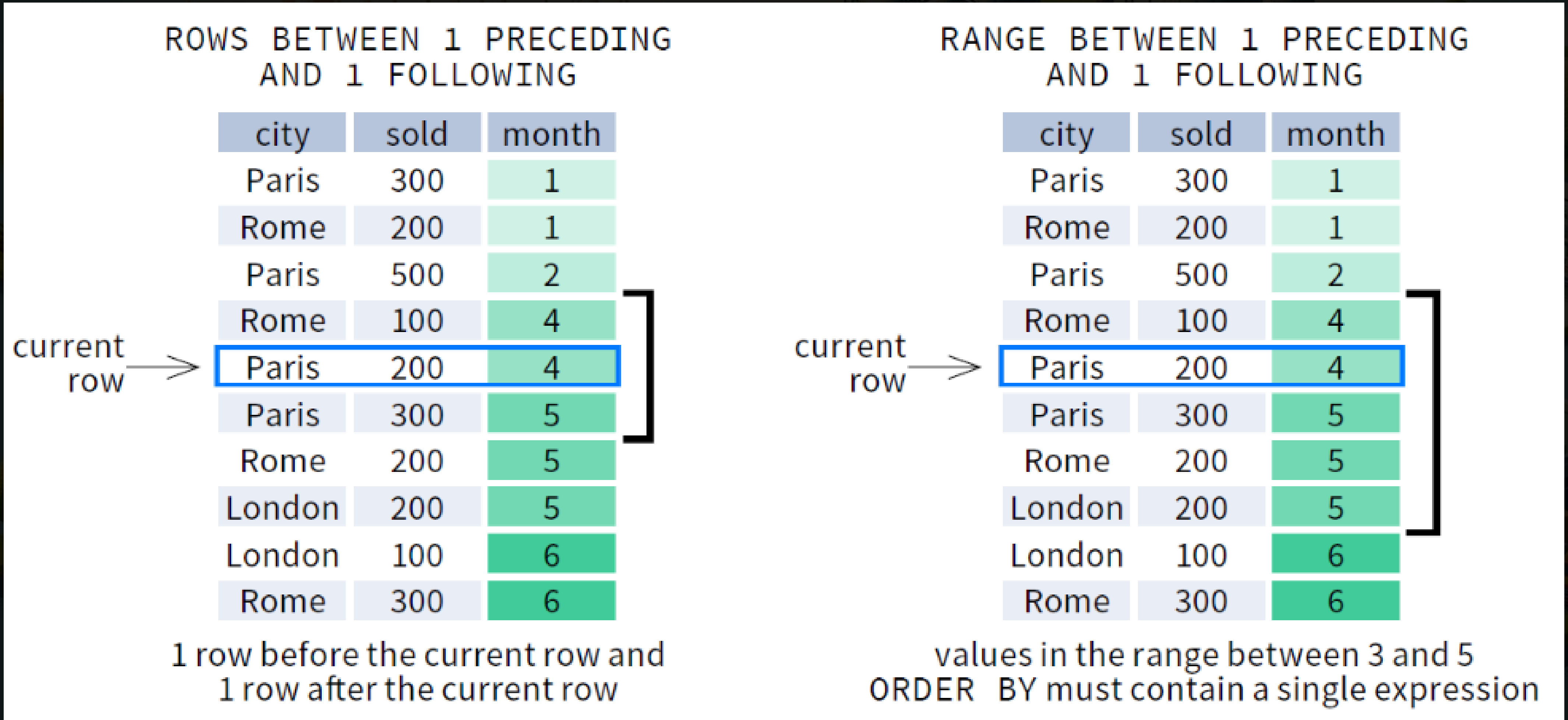
sold	city	month	sum
300	Paris	1	800
500	Paris	2	800
200	Rome	1	900
300	Rome	2	900
400	Rome	3	900
100	London	1	500
400	London	2	500

WINDOW FRAME

A **window frame** is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.



WINDOW FRAME



ROWS BETWEEN 1 PRECEDING
AND 1 FOLLOWING

city	sold	month
Paris	300	1
Rome	200	1
Paris	500	2
Rome	100	4
Paris	200	4
Paris	300	5
Rome	200	5
London	200	5
London	100	6
Rome	300	6

current
row

1 row before the current row and
1 row after the current row

RANGE BETWEEN 1 PRECEDING
AND 1 FOLLOWING

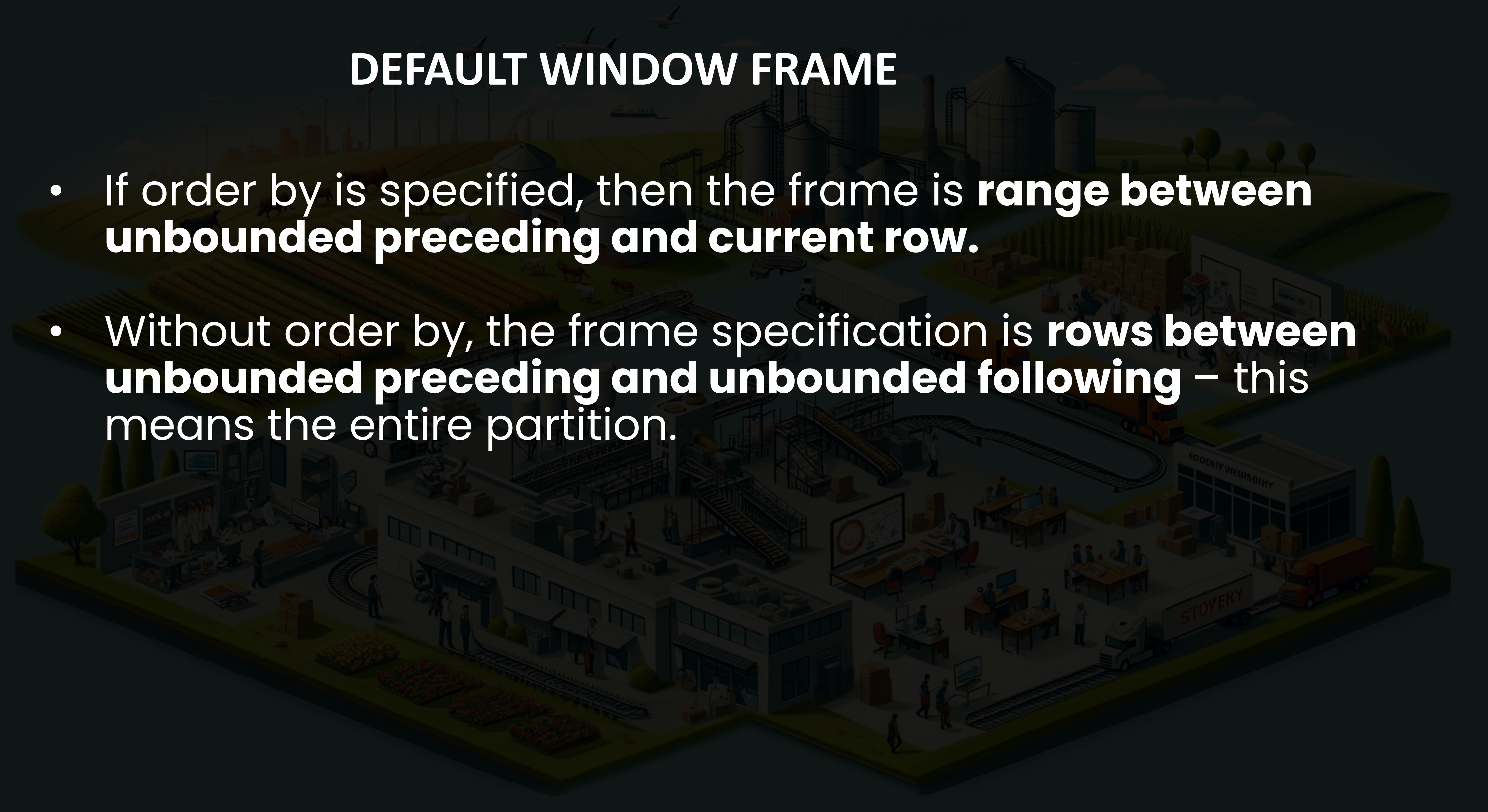
city	sold	month
Paris	300	1
Rome	200	1
Paris	500	2
Rome	100	4
Paris	200	4
Paris	300	5
Rome	200	5
London	200	5
London	100	6
Rome	300	6

values in the range between 3 and 5
ORDER BY must contain a single expression

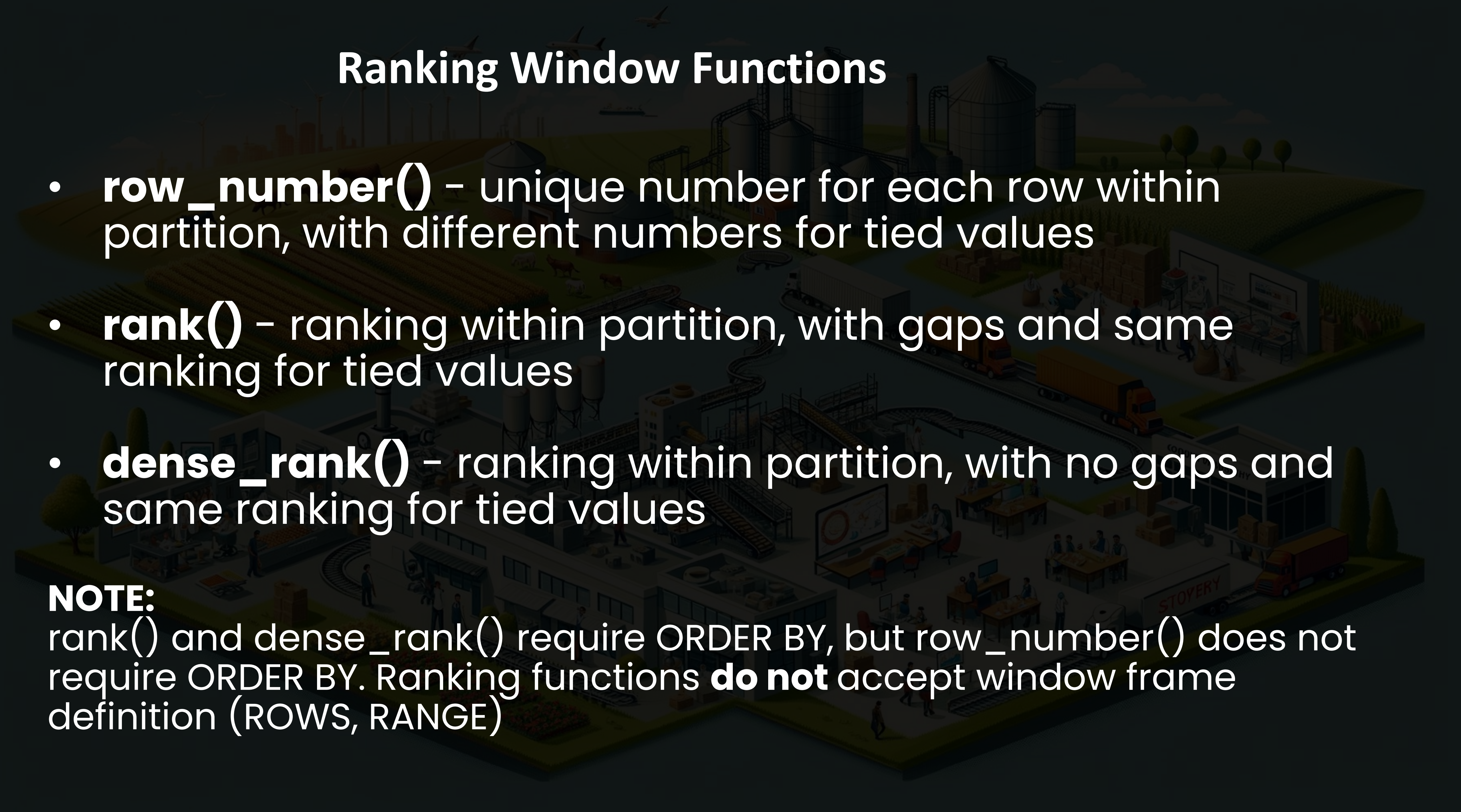
DEFAULT WINDOW FRAME

- If order by is specified, then the frame is **range between unbounded preceding and current row.**

- Without order by, the frame specification is **rows between unbounded preceding and unbounded following** – this means the entire partition.



Ranking Window Functions



- **row_number()** - unique number for each row within partition, with different numbers for tied values

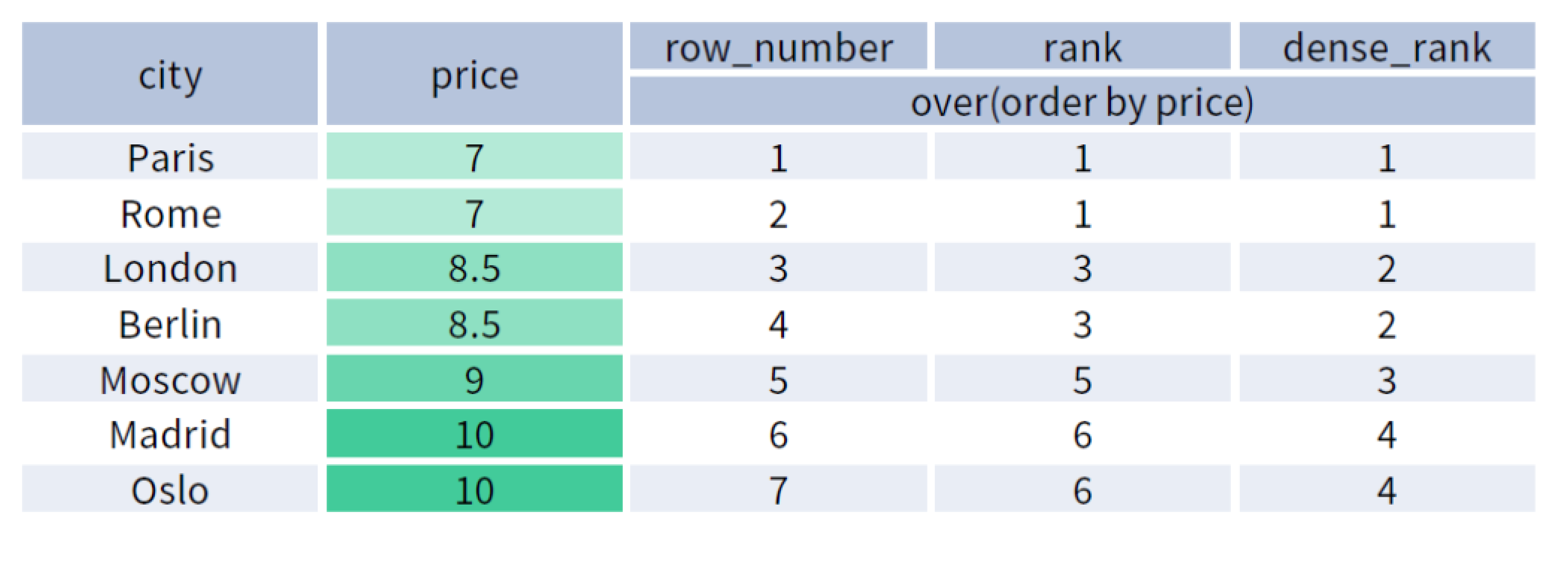
- **rank()** - ranking within partition, with gaps and same ranking for tied values

- **dense_rank()** - ranking within partition, with no gaps and same ranking for tied values

NOTE:
rank() and dense_rank() require ORDER BY, but row_number() does not require ORDER BY. Ranking functions **do not** accept window frame definition (ROWS, RANGE)

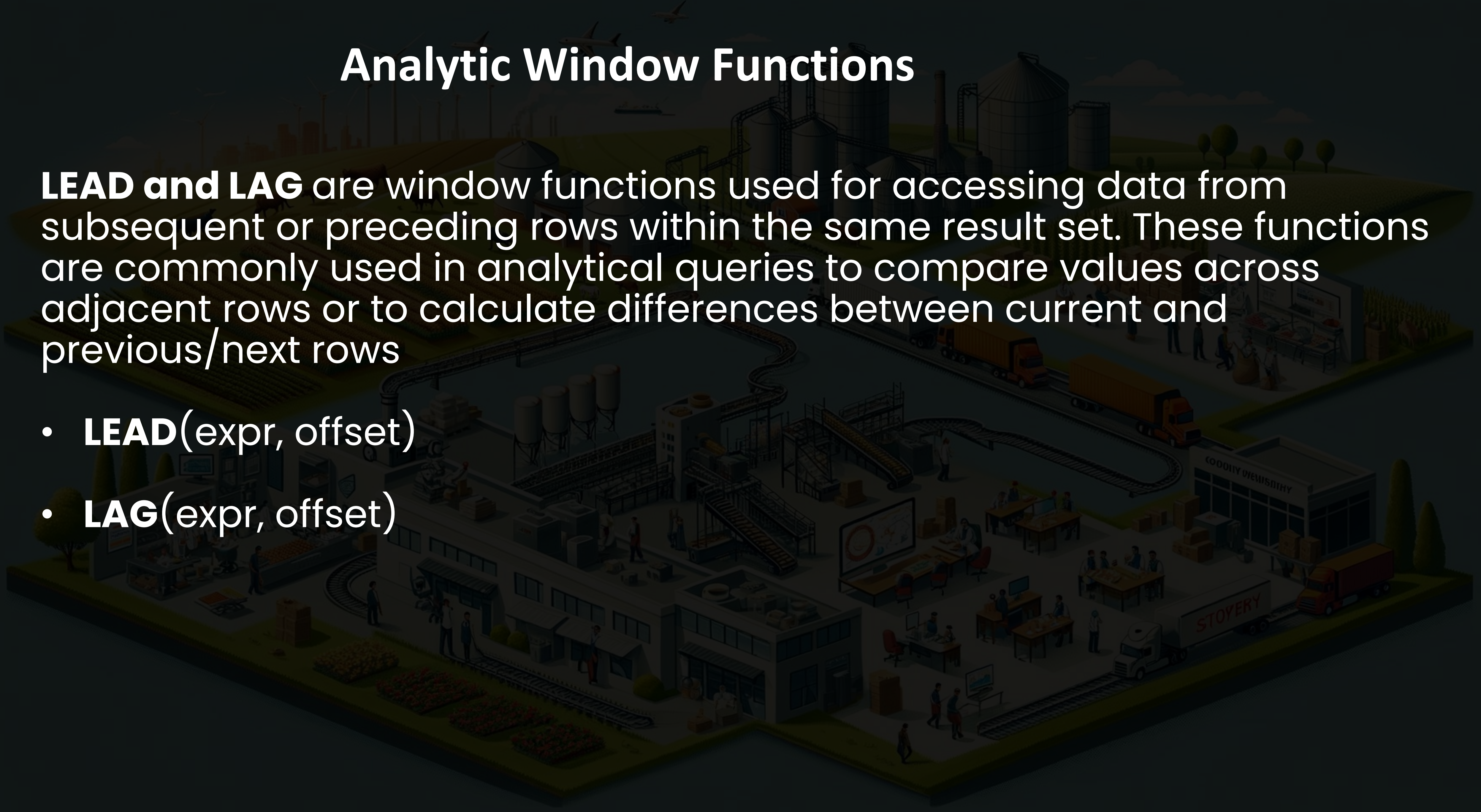
Ranking Window Functions

```
SELECT city,
        price,
        row_number() OVER (ORDER BY price),
        rank() OVER (ORDER BY price),
        dense_rank() OVER (ORDER BY price)
```



city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

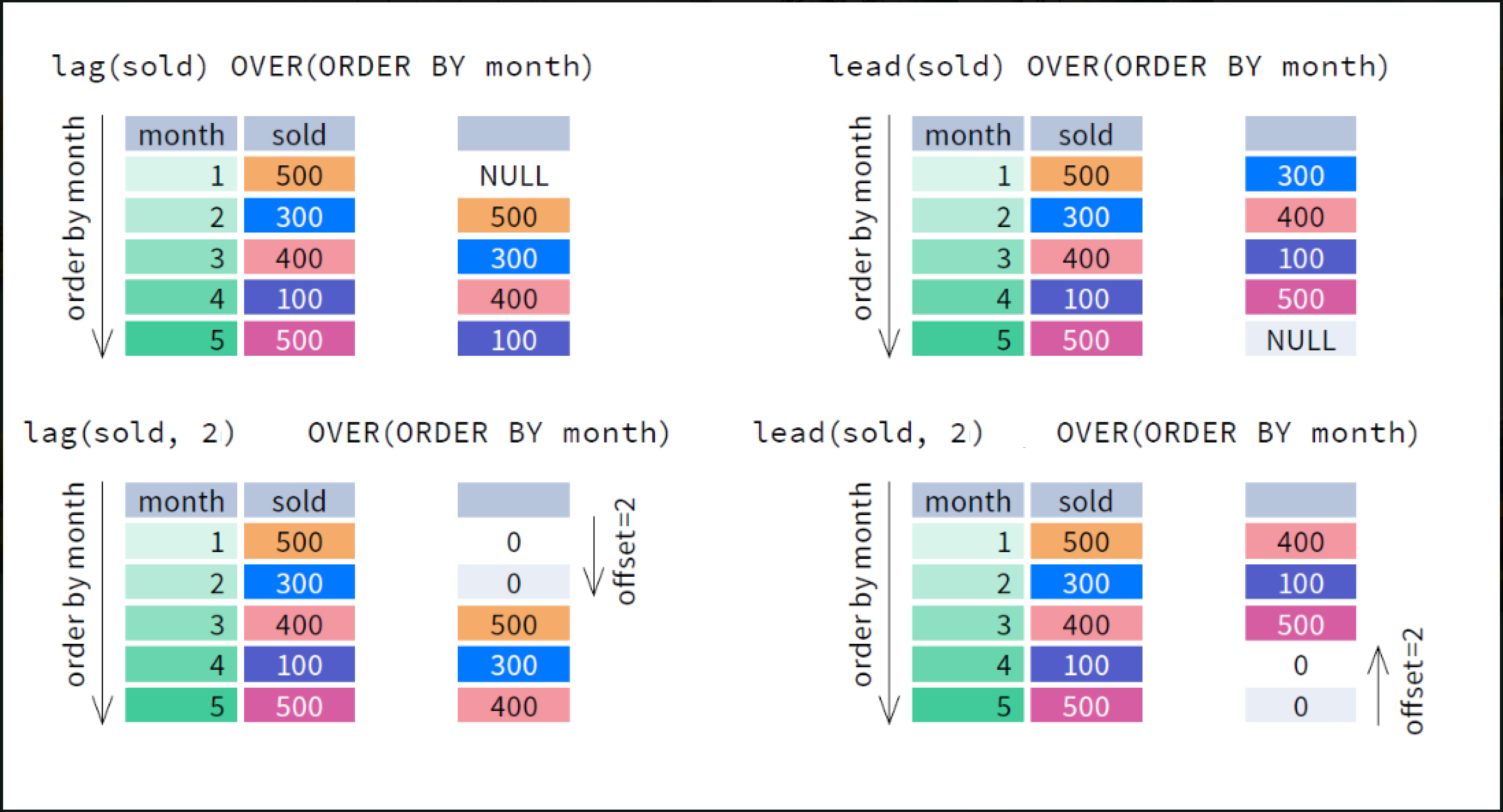
Analytic Window Functions



LEAD and LAG are window functions used for accessing data from subsequent or preceding rows within the same result set. These functions are commonly used in analytical queries to compare values across adjacent rows or to calculate differences between current and previous/next rows

- **LEAD**(expr, offset)
- **LAG**(expr, offset)

Lead() and Lag()



```
lag(sold) OVER(ORDER BY month)
```

month	sold	
1	500	NULL
2	300	500
3	400	300
4	100	400
5	500	100

```
lead(sold) OVER(ORDER BY month)
```

month	sold	
1	500	300
2	300	400
3	400	100
4	100	500
5	500	NULL

```
lag(sold, 2)    OVER(ORDER BY month)
```

month	sold	
1	500	0
2	300	0
3	400	500
4	100	300
5	500	400

```
lead(sold, 2)    OVER(ORDER BY month)
```

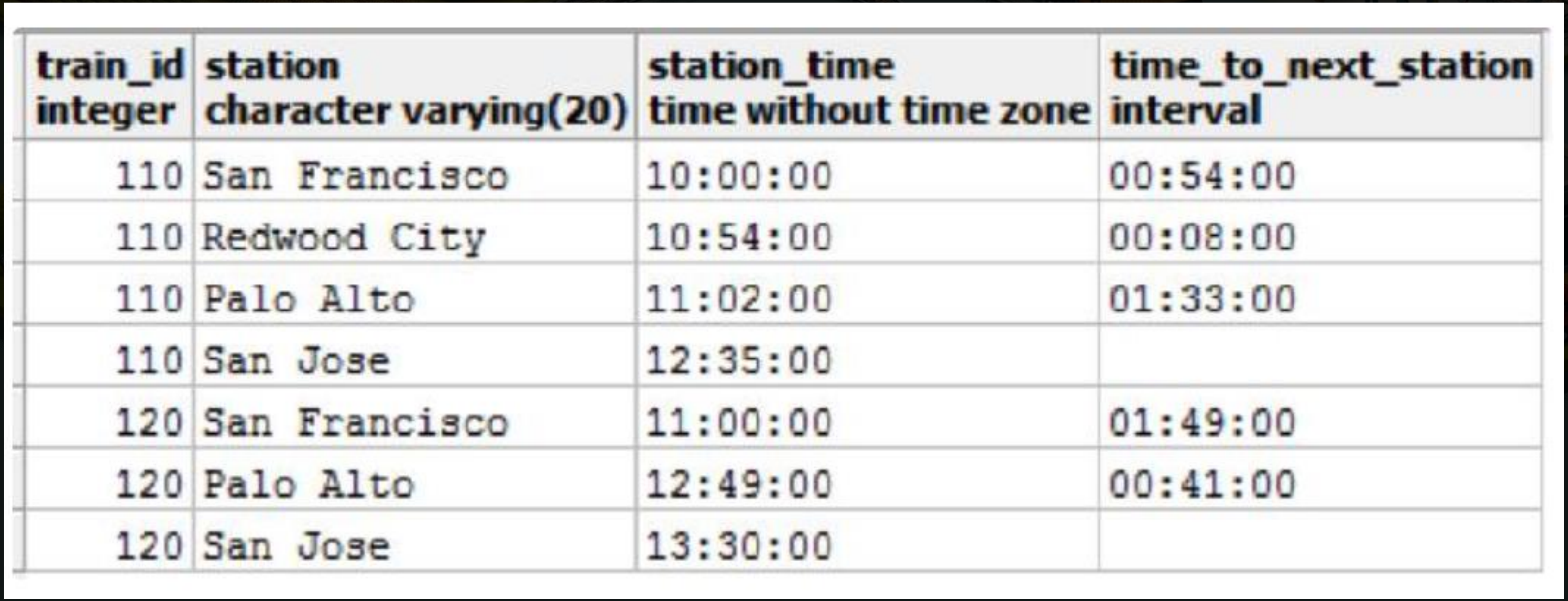
month	sold	
1	500	400
2	300	100
3	400	500
4	100	0
5	500	0

Lead() and Lag(): Example

Train_id	Station	Time
110	San Francisco	10:00:00
110	Redwood City	10:54:00
110	Palo Alto	11:02:00
110	San Jose	12:35:00
120	San Francisco	11:00:00
120	Redwood City	Non Stop
120	Palo Alto	12:49:00
120	San Jose	13:30:00

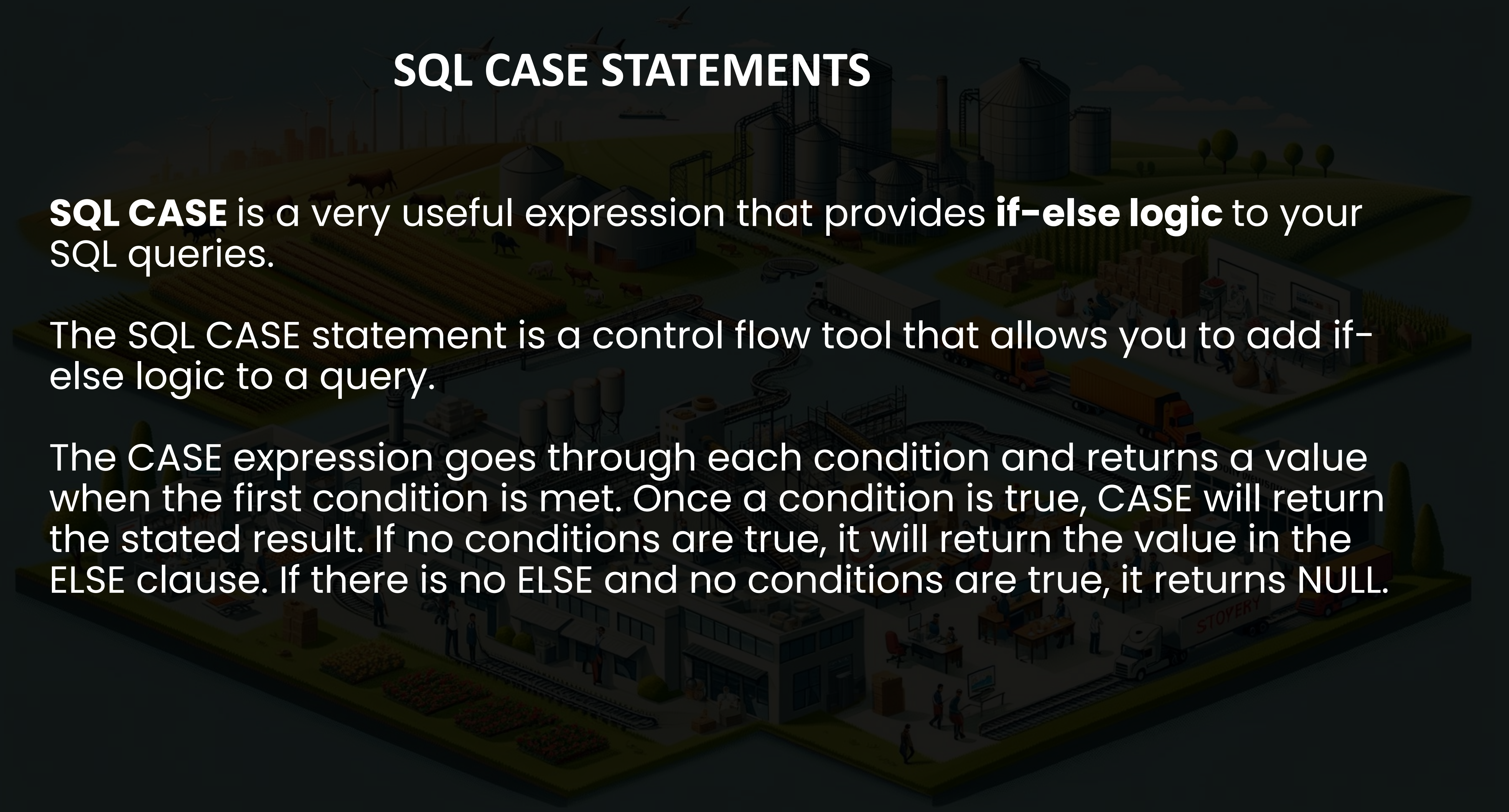
Window Functions – LAG() and LAG()

```
SELECT train_id,
        station,
        time as station_time,
        LEAD(time) OVER (PARTITON BY train_id ORDER BY time) – time AS time_to_next_station
```



train_id integer	station character varying(20)	station_time time without time zone	time_to_next_station interval
110	San Francisco	10:00:00	00:54:00
110	Redwood City	10:54:00	00:08:00
110	Palo Alto	11:02:00	01:33:00
110	San Jose	12:35:00	
120	San Francisco	11:00:00	01:49:00
120	Palo Alto	12:49:00	00:41:00
120	San Jose	13:30:00	

SQL CASE STATEMENTS

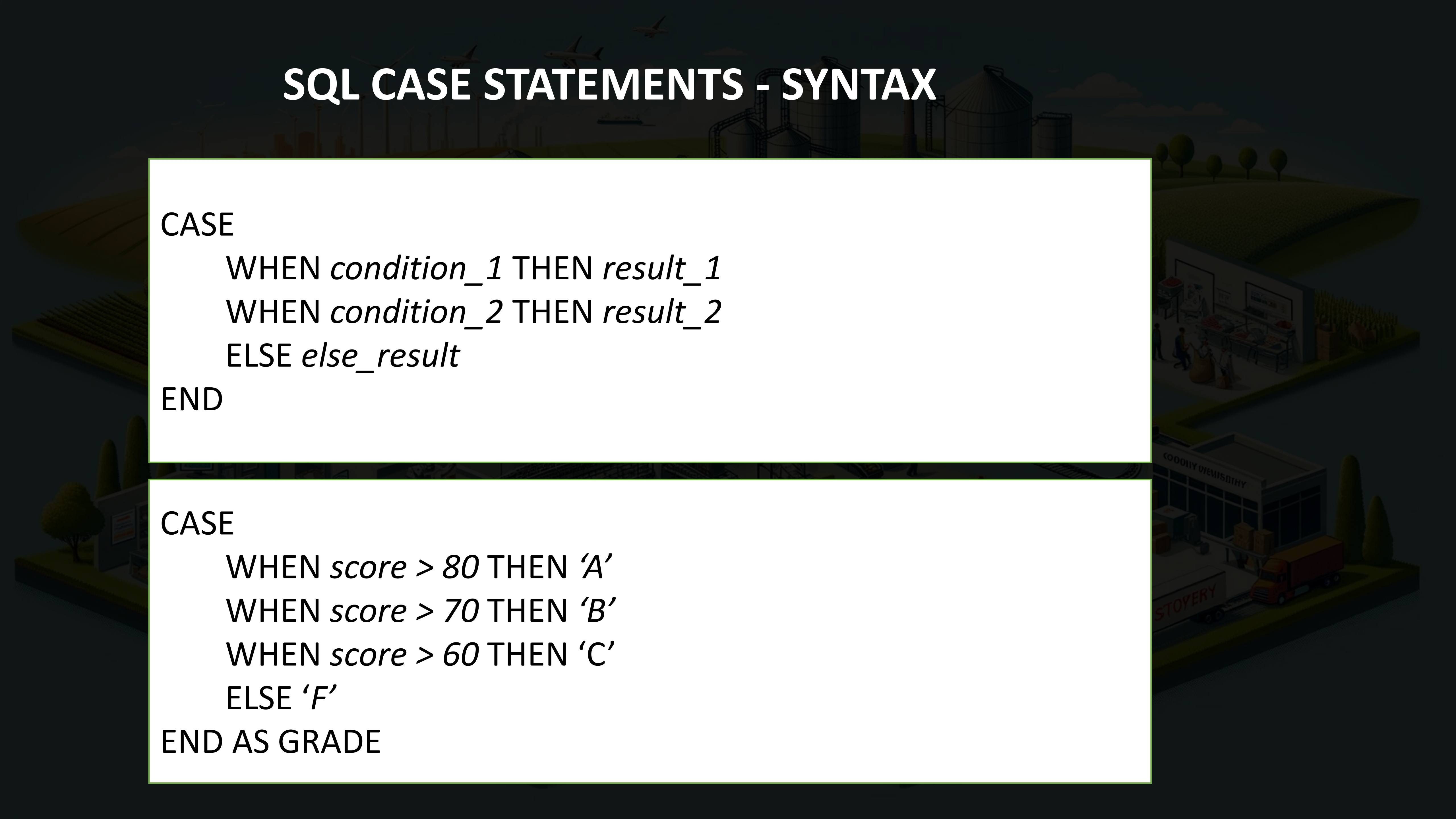


SQL CASE is a very useful expression that provides **if-else logic** to your SQL queries.

The SQL CASE statement is a control flow tool that allows you to add if-else logic to a query.

The CASE expression goes through each condition and returns a value when the first condition is met. Once a condition is true, CASE will return the stated result. If no conditions are true, it will return the value in the ELSE clause. If there is no ELSE and no conditions are true, it returns NULL.

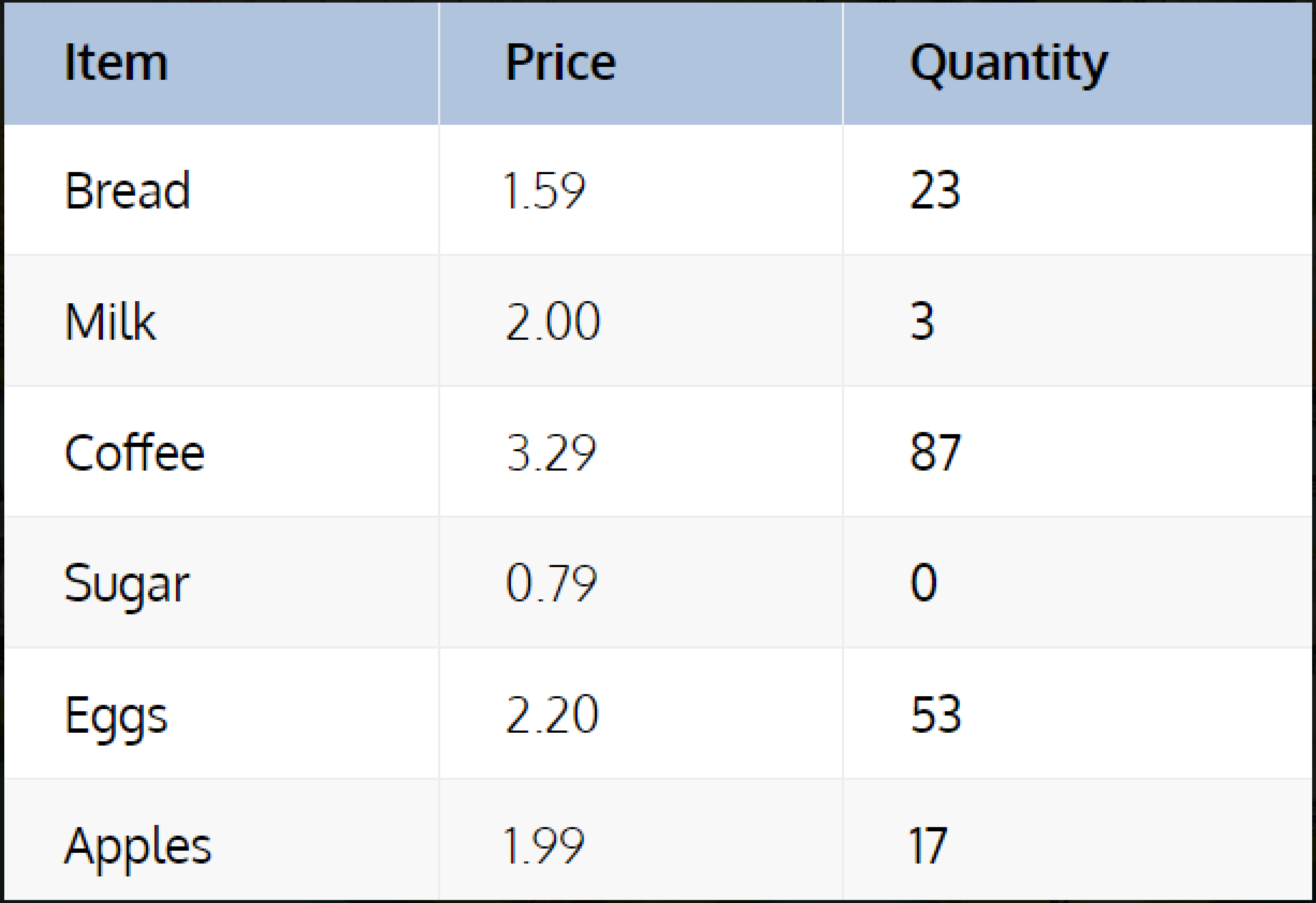
SQL CASE STATEMENTS - SYNTAX



```
CASE
    WHEN condition_1 THEN result_1
    WHEN condition_2 THEN result_2
    ELSE else_result
END
```

```
CASE
    WHEN score > 80 THEN ‘A’
    WHEN score > 70 THEN ‘B’
    WHEN score > 60 THEN ‘C’
    ELSE ‘F’
END AS GRADE
```

SQL CASE STATEMENTS - EXAMPLE

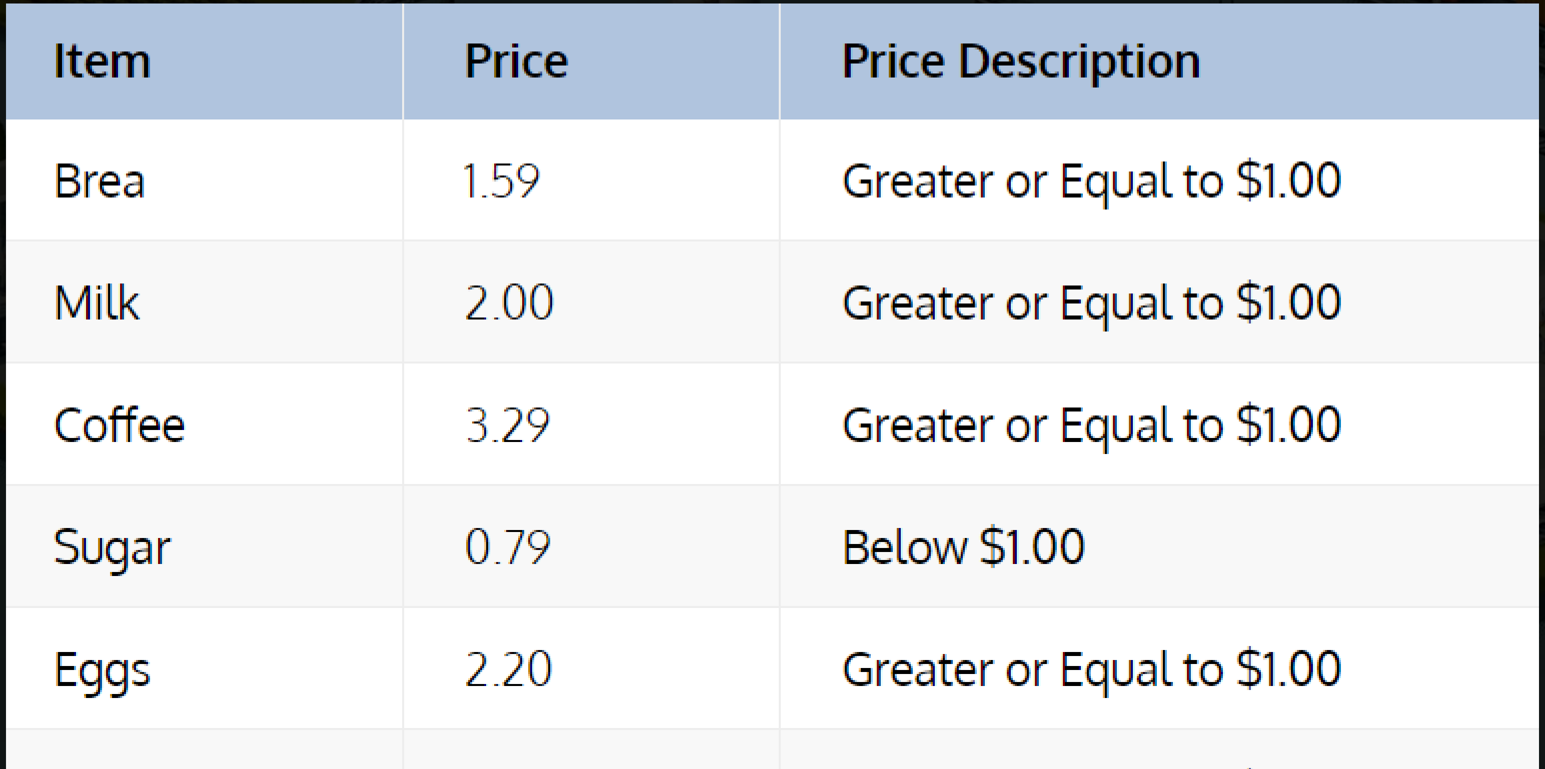


Item	Price	Quantity
Bread	1.59	23
Milk	2.00	3
Coffee	3.29	87
Sugar	0.79	0
Eggs	2.20	53
Apples	1.99	17

Q: Categorize the Price into above $1 and below $1

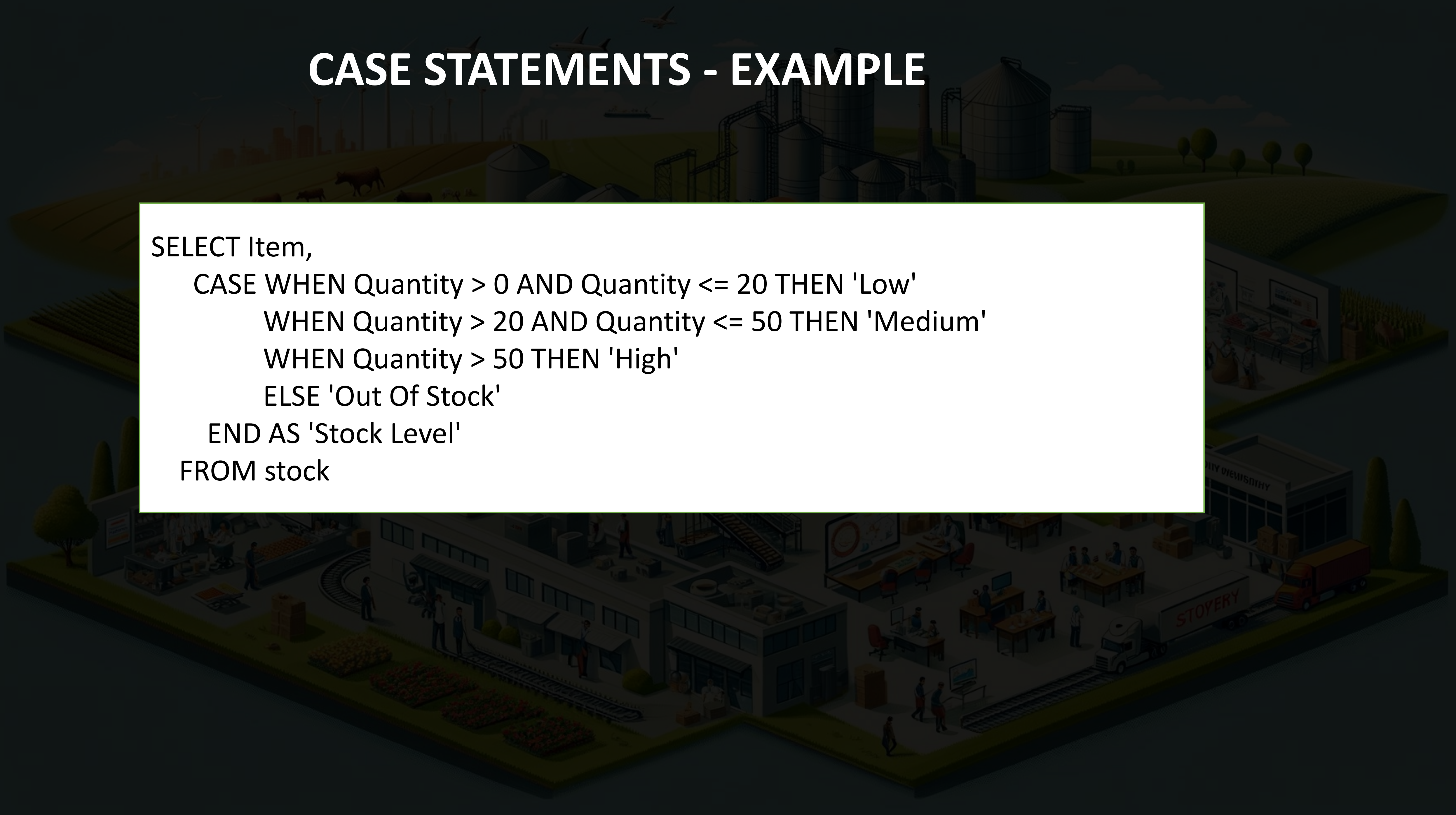
CASE STATEMENTS - EXAMPLE

```
SELECT item, price,
          CASE
             WHEN price < 1 THEN ‘Below $1’
             WHEN price >= 1 THEN ‘Greater than or Equal to $1’
          END AS ‘Price Description’
FROM Stock
```



Item	Price	Price Description
Brea	1.59	Greater or Equal to $1.00
Milk	2.00	Greater or Equal to $1.00
Coffee	3.29	Greater or Equal to $1.00
Sugar	0.79	Below $1.00
Eggs	2.20	Greater or Equal to $1.00

CASE STATEMENTS - EXAMPLE



```
SELECT Item,
      CASE WHEN Quantity > 0 AND Quantity <= 20 THEN 'Low'
                WHEN Quantity > 20 AND Quantity <= 50 THEN 'Medium'
                WHEN Quantity > 50 THEN 'High'
                ELSE 'Out Of Stock'
        END AS 'Stock Level'
    FROM stock
```

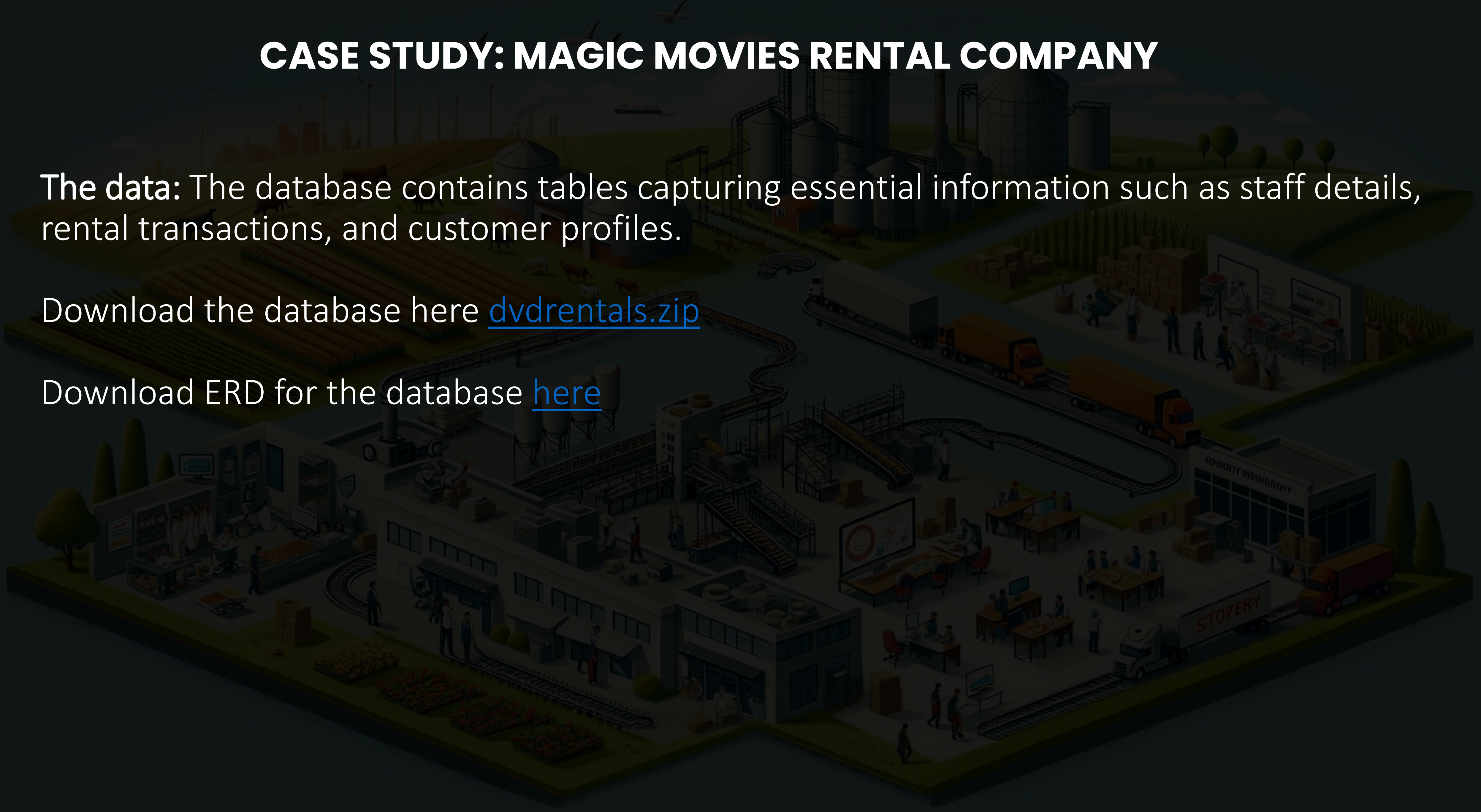
CASE STUDY: MAGIC MOVIES RENTAL COMPANY

Introduction: Movie Magic is a large movie rental company. Movie Magic is renowned for its vast collection of films catering to every taste and preference.

The Challenge: Evaluating Performance

In a quest to provide unparalleled customer experiences, MM face the challenge of evaluating staff performance and understanding customer behavior. With their rich database of rental transactions, they aim to uncover insights that will drive operational excellence and customer satisfaction.

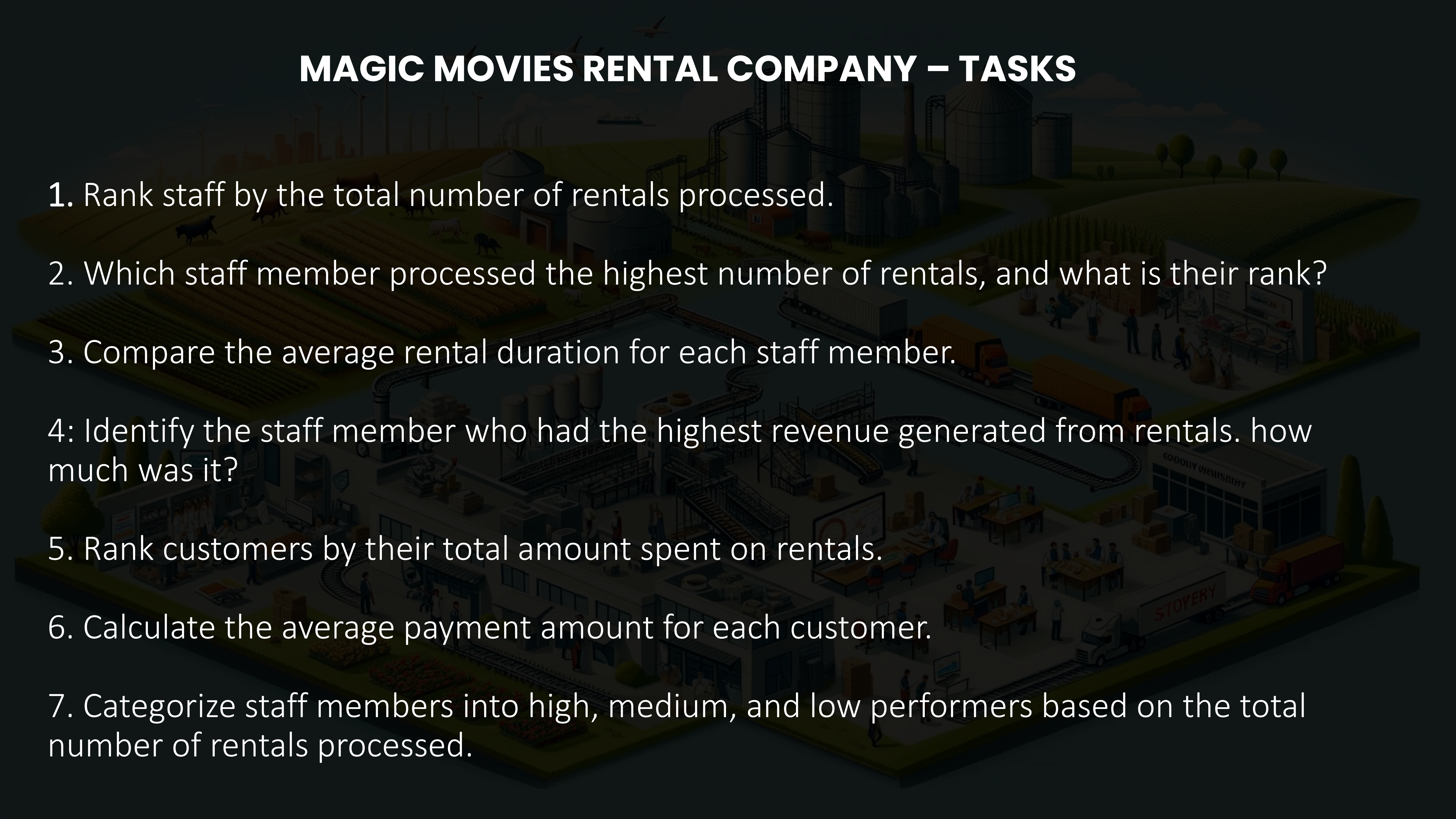
CASE STUDY: MAGIC MOVIES RENTAL COMPANY



The data: The database contains tables capturing essential information such as staff details, rental transactions, and customer profiles.

Download the database here [dvdrentals.zip]

Download ERD for the database [here]



MAGIC MOVIES RENTAL COMPANY – TASKS

1. Rank staff by the total number of rentals processed.

2. Which staff member processed the highest number of rentals, and what is their rank?

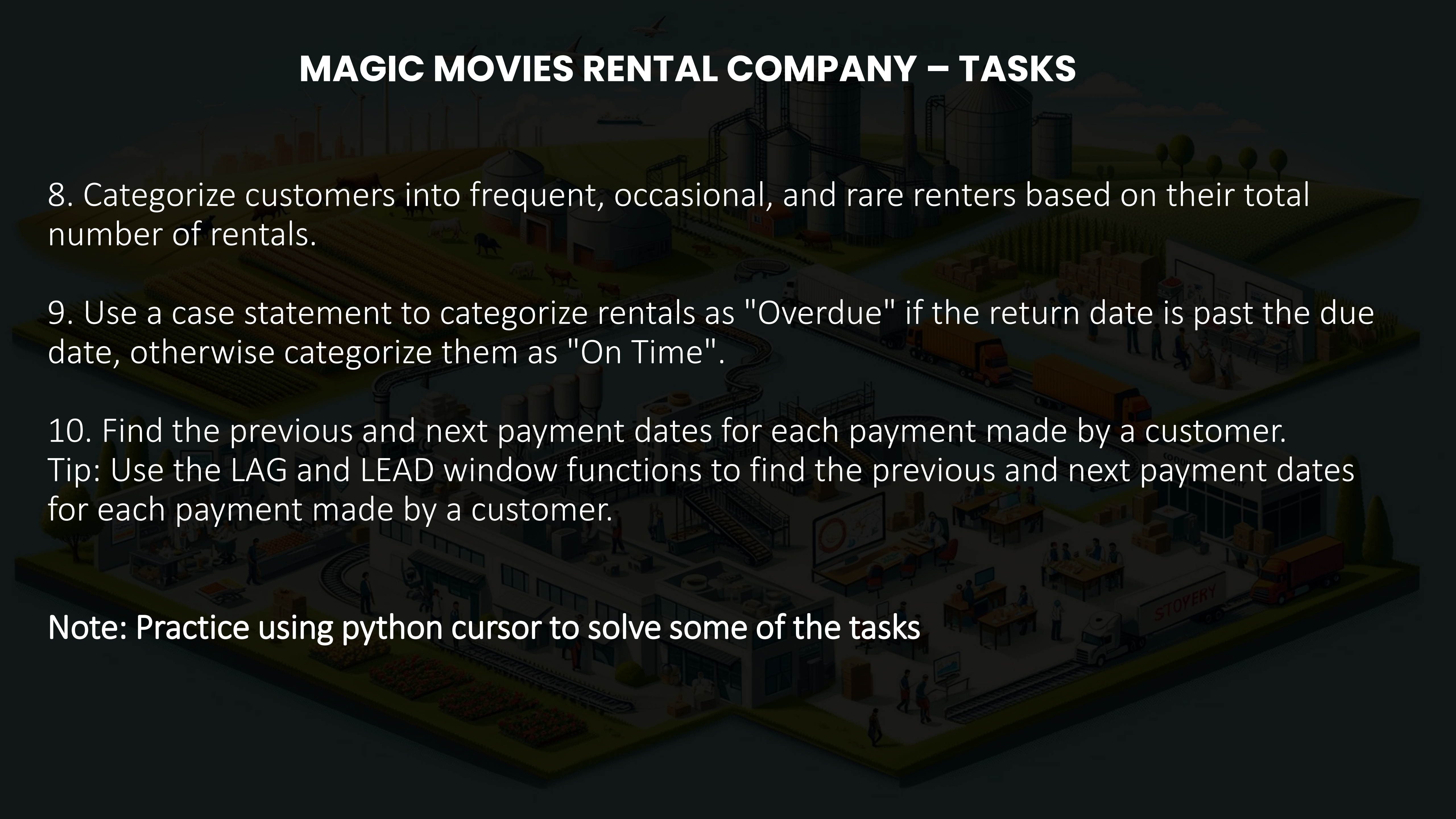
3. Compare the average rental duration for each staff member.

4: Identify the staff member who had the highest revenue generated from rentals. how much was it?

5. Rank customers by their total amount spent on rentals.

6. Calculate the average payment amount for each customer.

7. Categorize staff members into high, medium, and low performers based on the total number of rentals processed.



MAGIC MOVIES RENTAL COMPANY – TASKS

8. Categorize customers into frequent, occasional, and rare renters based on their total number of rentals.

9. Use a case statement to categorize rentals as "Overdue" if the return date is past the due date, otherwise categorize them as "On Time".

10. Find the previous and next payment dates for each payment made by a customer.
Tip: Use the LAG and LEAD window functions to find the previous and next payment dates for each payment made by a customer.

Note: Practice using python cursor to solve some of the tasks

GOODLUCK