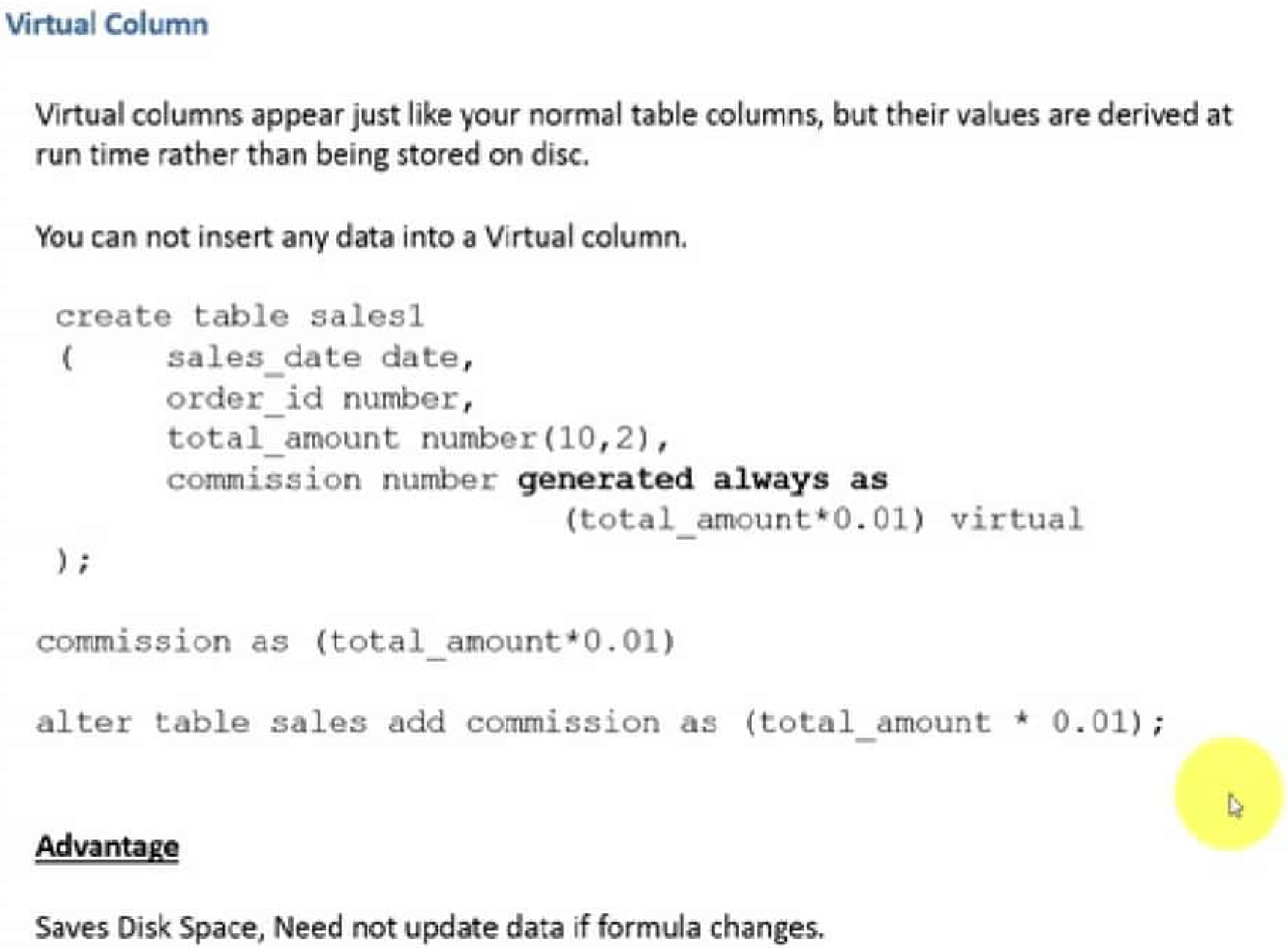


Virtual Column

Virtual columns appear just like your normal table columns, but their values are derived at run time rather than being stored on disc.

You can not insert any data into a Virtual column.

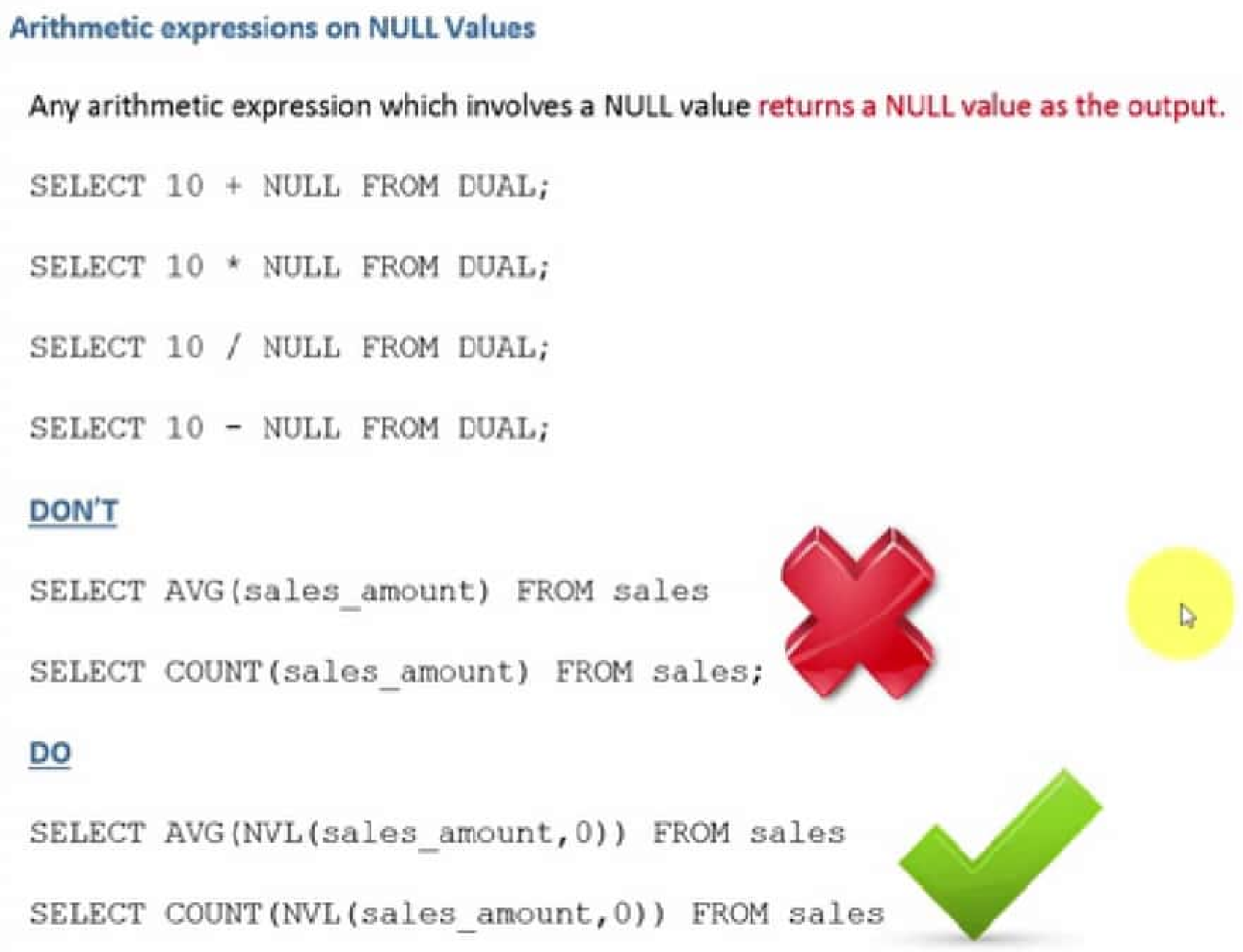
```
create table sales1
(     sales_date date,
      order_id number,
      total_amount number(10,2),
      commission number generated always as
                         (total_amount*0.01) virtual
);
```

```
commission as (total_amount*0.01)
```

```
alter table sales add commission as (total_amount * 0.01);
```

Advantage

Saves Disk Space, Need not update data if formula changes.



Arithmetic expressions on NULL Values

Any arithmetic expression which involves a NULL value returns a NULL value as the output.

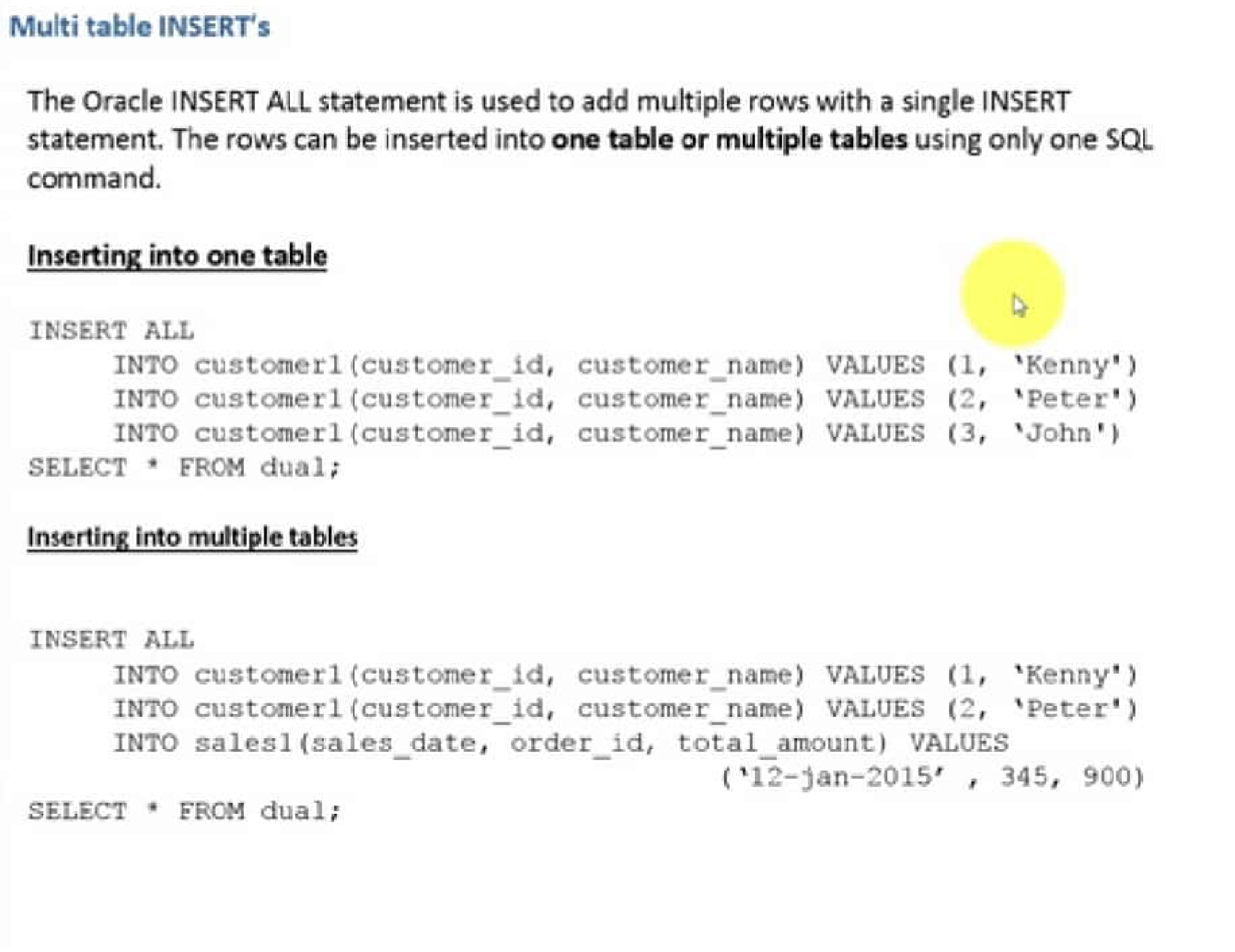
```
SELECT 10 + NULL FROM DUAL;

SELECT 10 * NULL FROM DUAL;

SELECT 10 / NULL FROM DUAL;

SELECT 10 - NULL FROM DUAL;
```

DON'T

```
SELECT AVG(sales_amount) FROM sales

SELECT COUNT(sales_amount) FROM sales;
```

DO

```
SELECT AVG(NVL(sales_amount,0)) FROM sales

SELECT COUNT(NVL(sales_amount,0)) FROM sales
```



Multi table INSERT's

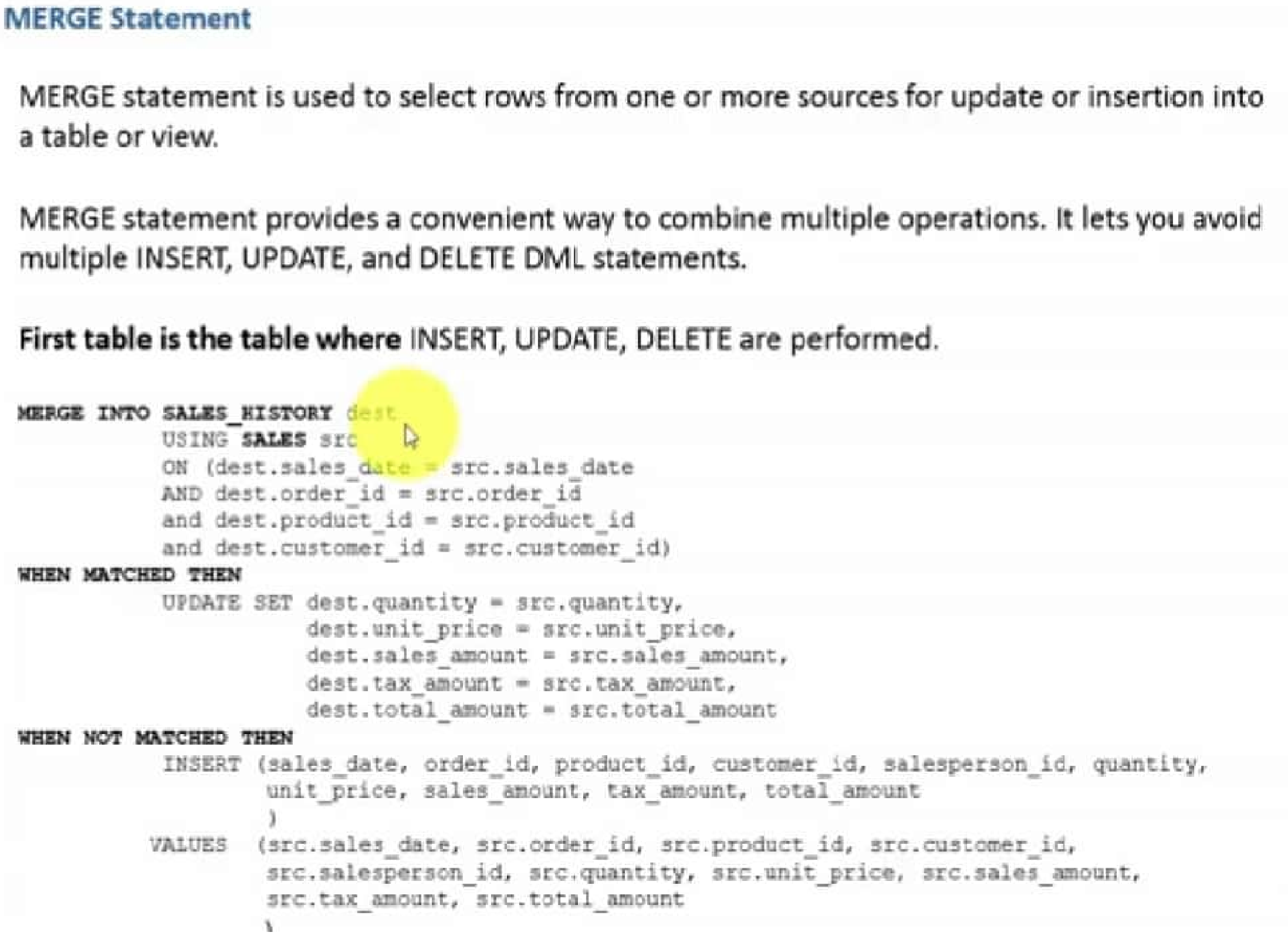
The Oracle INSERT ALL statement is used to add multiple rows with a single INSERT statement. The rows can be inserted into **one table or multiple tables** using only one SQL command.

Inserting into one table

```
INSERT ALL
     INTO customer1(customer_id, customer_name) VALUES (1, ‘Kenny')
     INTO customer1(customer_id, customer_name) VALUES (2, ‘Peter')
     INTO customer1(customer_id, customer_name) VALUES (3, ‘John')
SELECT * FROM dual;
```

Inserting into multiple tables

```
INSERT ALL
     INTO customer1(customer_id, customer_name) VALUES (1, ‘Kenny')
     INTO customer1(customer_id, customer_name) VALUES (2, ‘Peter')
     INTO sales1(sales_date, order_id, total_amount) VALUES
                                            (‘12-jan-2015’ , 345, 900)
SELECT * FROM dual;
```



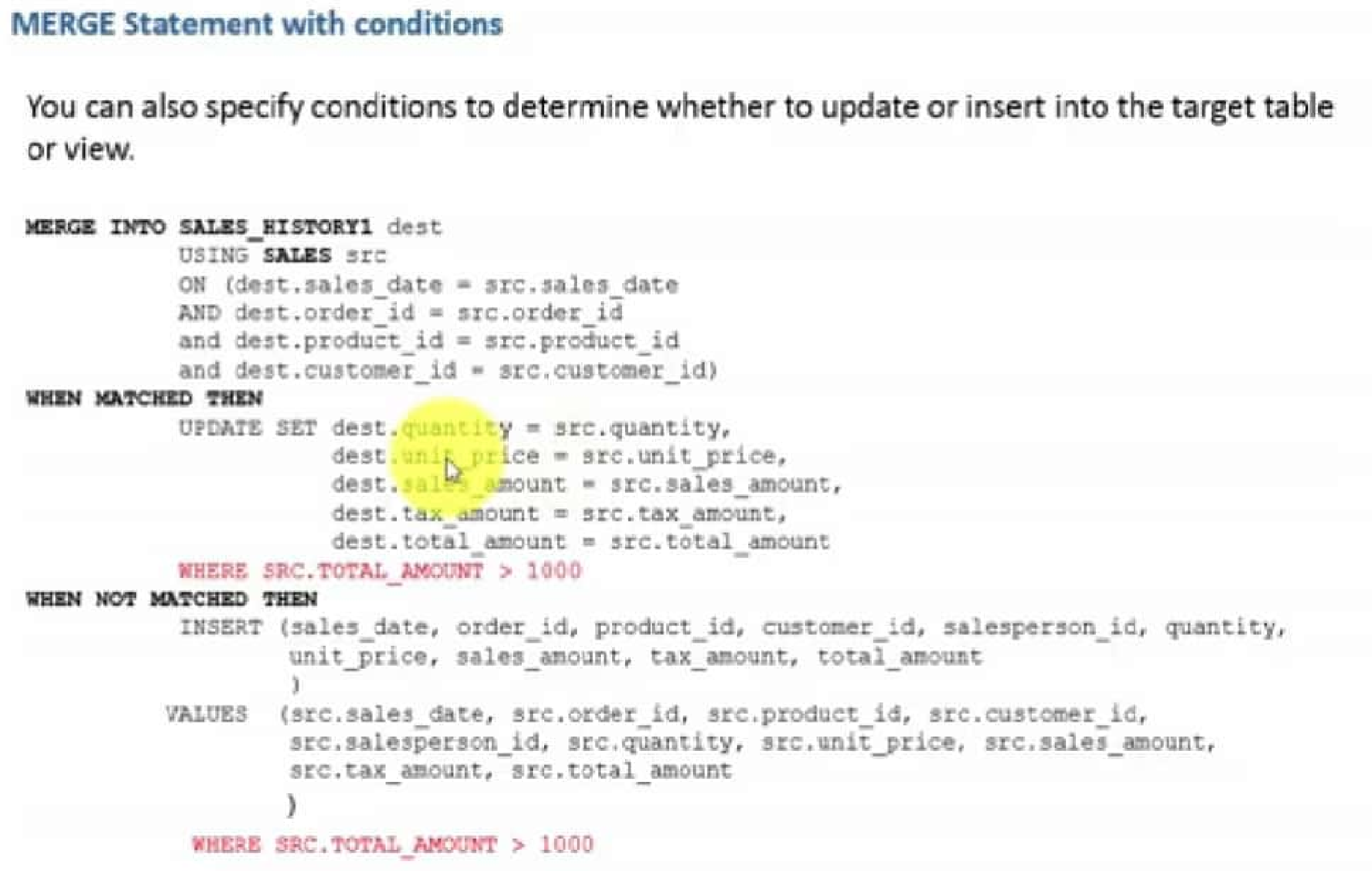
MERGE Statement

MERGE statement is used to select rows from one or more sources for update or insertion into a table or view.

MERGE statement provides a convenient way to combine multiple operations. It lets you avoid multiple INSERT, UPDATE, and DELETE DML statements.

First table is the table where INSERT, UPDATE, DELETE are performed.

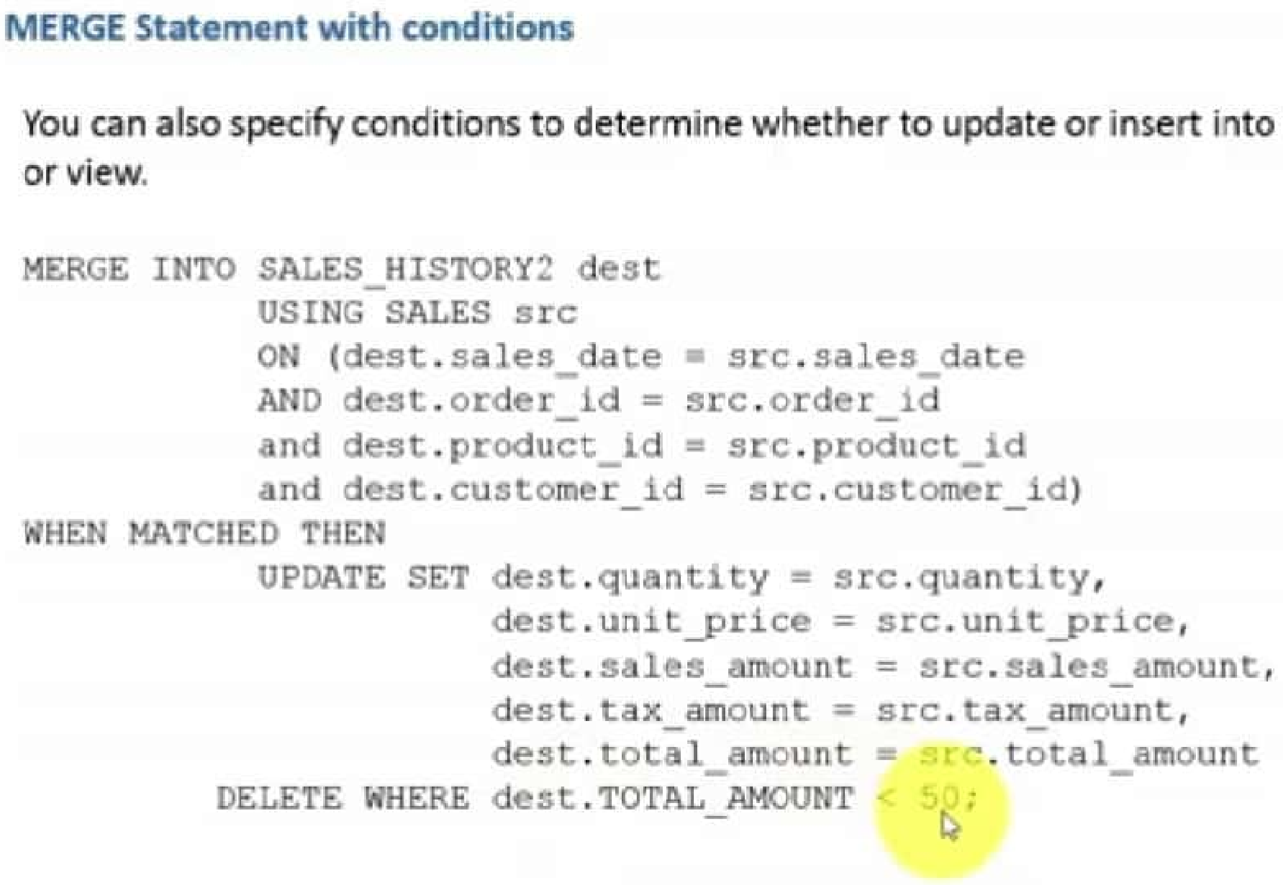
```
MERGE INTO SALES_HISTORY dest
           USING SALES src
           ON (dest.sales_date = src.sales_date
           AND dest.order_id = src.order_id
           and dest.product_id = src.product_id
           and dest.customer_id = src.customer_id)
WHEN MATCHED THEN
           UPDATE SET dest.quantity = src.quantity,
                      dest.unit_price = src.unit_price,
                      dest.sales_amount = src.sales_amount,
                      dest.tax_amount = src.tax_amount,
                      dest.total_amount = src.total_amount
WHEN NOT MATCHED THEN
           INSERT (sales_date, order_id, product_id, customer_id, salesperson_id, quantity,
                   unit_price, sales_amount, tax_amount, total_amount
                   )
          VALUES  (src.sales_date, src.order_id, src.product_id, src.customer_id,
                   src.salesperson_id, src.quantity, src.unit_price, src.sales_amount,
                   src.tax_amount, src.total_amount
                   )
```



MERGE Statement with conditions

You can also specify conditions to determine whether to update or insert into the target table or view.

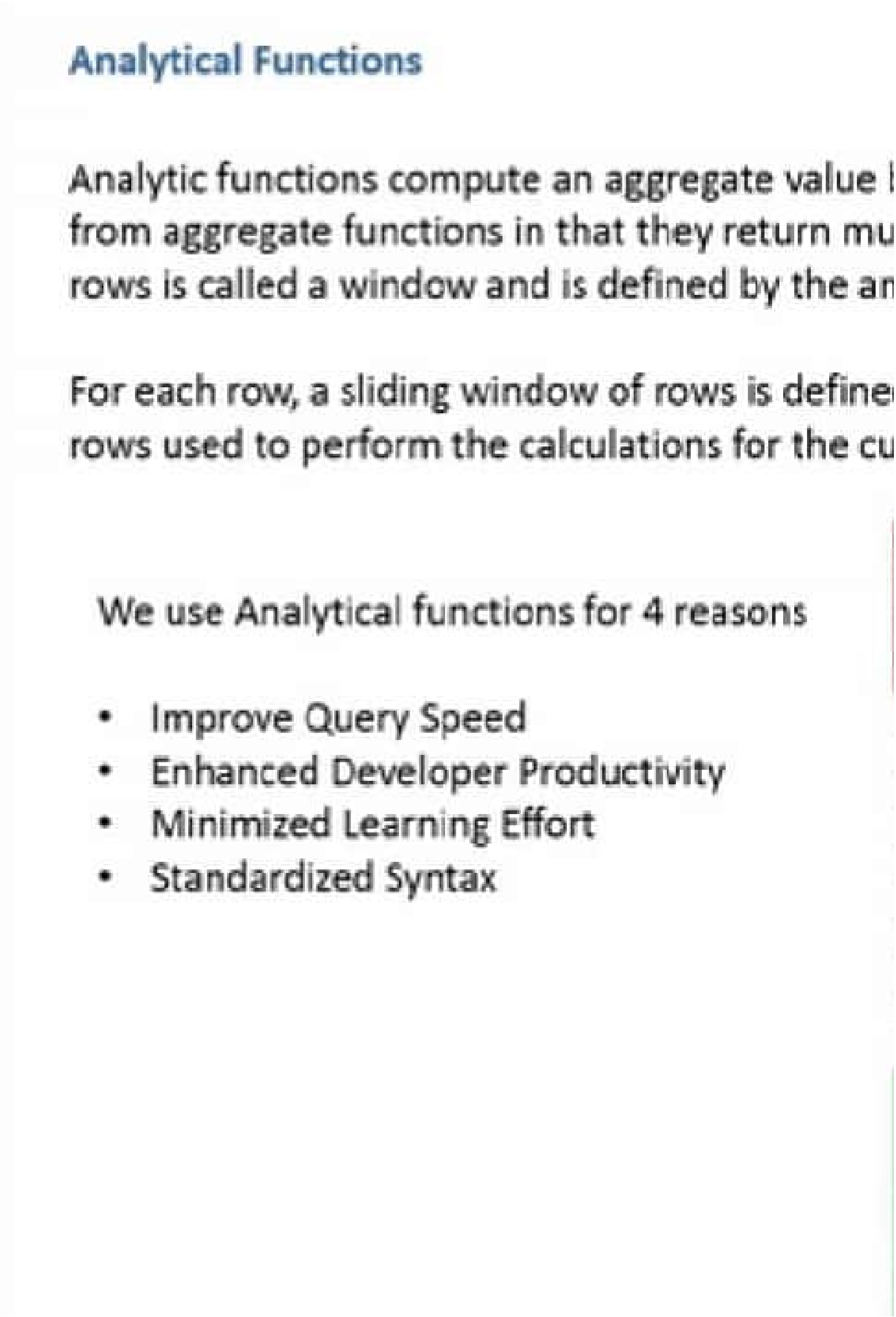
```
MERGE INTO SALES_HISTORY1 dest
           USING SALES src
           ON (dest.sales_date = src.sales_date
           AND dest.order_id = src.order_id
           and dest.product_id = src.product_id
           and dest.customer_id = src.customer_id)
WHEN MATCHED THEN
           UPDATE SET dest.quantity = src.quantity,
                      dest.unit_price = src.unit_price,
                      dest.sales_amount = src.sales_amount,
                      dest.tax_amount = src.tax_amount,
                      dest.total_amount = src.total_amount
           WHERE SRC.TOTAL_AMOUNT > 1000
WHEN NOT MATCHED THEN
           INSERT (sales_date, order_id, product_id, customer_id, salesperson_id, quantity,
                   unit_price, sales_amount, tax_amount, total_amount
                   )
          VALUES  (src.sales_date, src.order_id, src.product_id, src.customer_id,
                   src.salesperson_id, src.quantity, src.unit_price, src.sales_amount,
                   src.tax_amount, src.total_amount
                   )
            WHERE SRC.TOTAL_AMOUNT > 1000
```



MERGE Statement with conditions

You can also specify conditions to determine whether to update or insert into the target table or view.

```
MERGE INTO SALES_HISTORY2 dest
           USING SALES src
           ON (dest.sales_date = src.sales_date
           AND dest.order_id = src.order_id
           and dest.product_id = src.product_id
           and dest.customer_id = src.customer_id)
WHEN MATCHED THEN
           UPDATE SET dest.quantity = src.quantity,
                      dest.unit_price = src.unit_price,
                      dest.sales_amount = src.sales_amount,
                      dest.tax_amount = src.tax_amount,
                      dest.total_amount = src.total_amount
         DELETE WHERE dest.TOTAL_AMOUNT < 50;
```



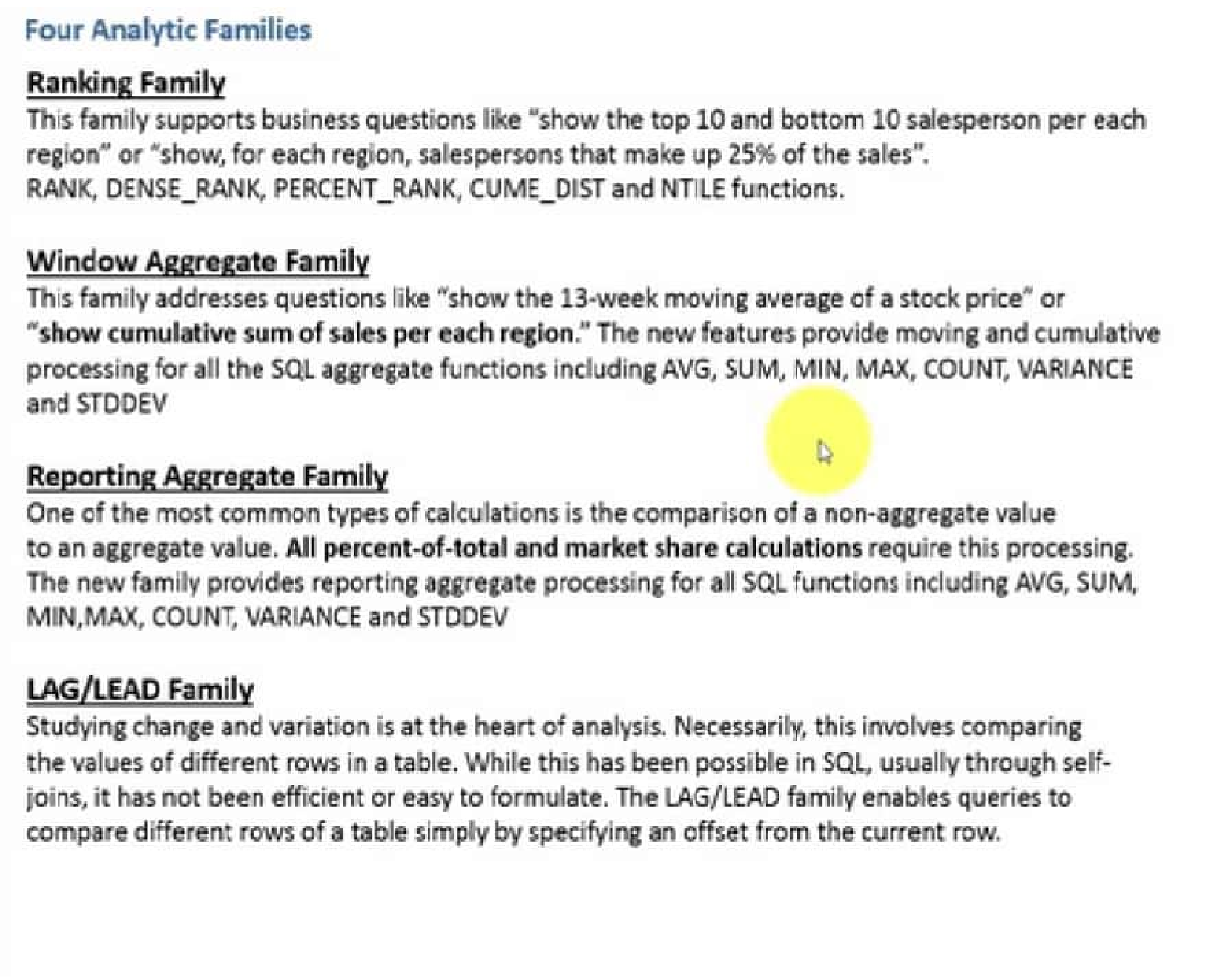
Analytical Functions

Analytic functions compute an aggregate value b
from aggregate functions in that they return mu
rows is called a window and is defined by the ar

For each row, a sliding window of rows is define
rows used to perform the calculations for the cu

We use Analytical functions for 4 reasons

- Improve Query Speed
- Enhanced Developer Productivity
- Minimized Learning Effort
- Standardized Syntax



Four Analytic Families

Ranking Family

This family supports business questions like "show the top 10 and bottom 10 salesperson per each region" or "show, for each region, salespersons that make up 25% of the sales".
RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST and NTILE functions.

Window Aggregate Family

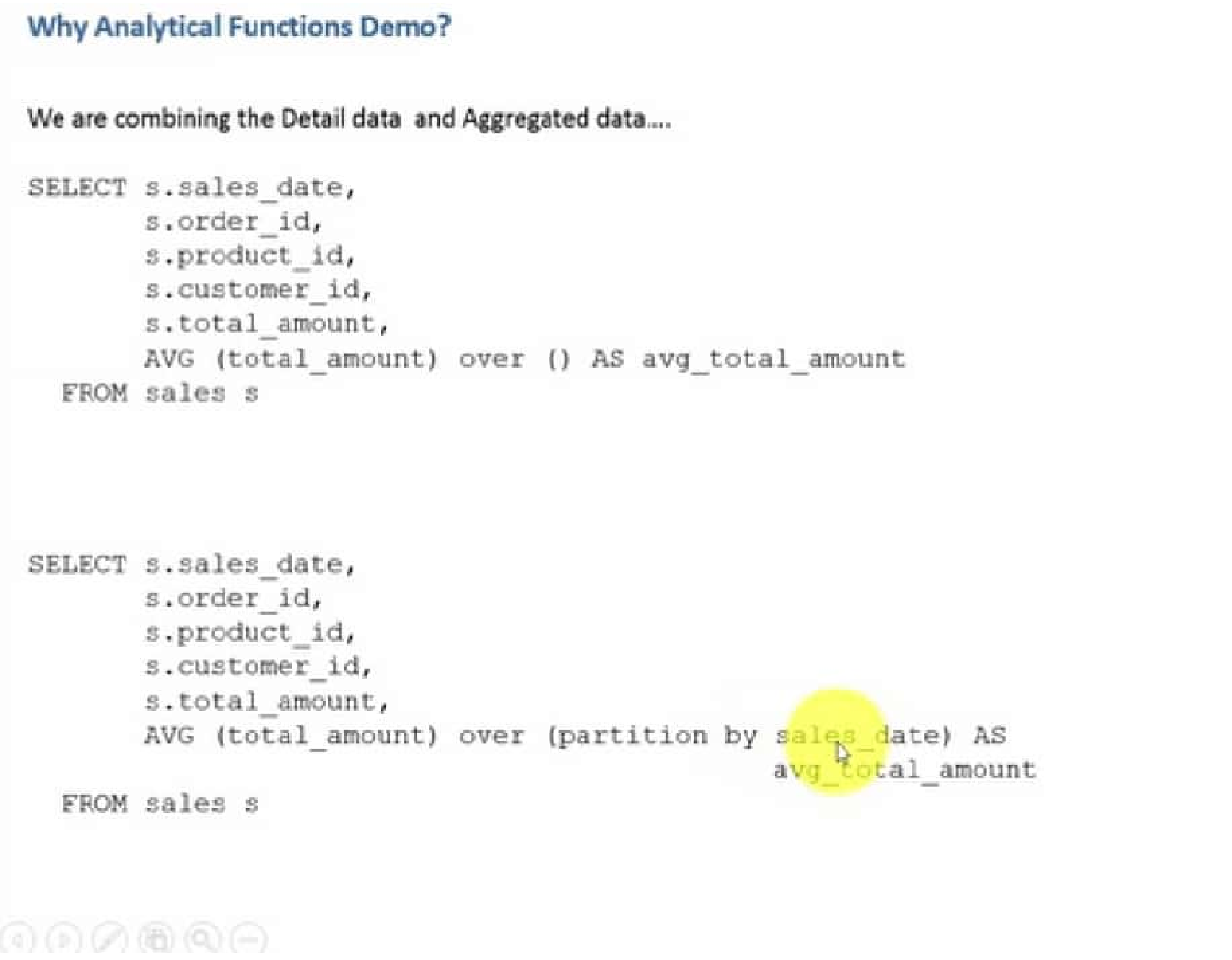
This family addresses questions like "show the 13-week moving average of a stock price" or **"show cumulative sum of sales per each region."** The new features provide moving and cumulative processing for all the SQL aggregate functions including AVG, SUM, MIN, MAX, COUNT, VARIANCE and STDDEV

Reporting Aggregate Family

One of the most common types of calculations is the comparison of a non-aggregate value to an aggregate value. **All percent-of-total and market share calculations** require this processing. The new family provides reporting aggregate processing for all SQL functions including AVG, SUM, MIN,MAX, COUNT, VARIANCE and STDDEV

LAG/LEAD Family

Studying change and variation is at the heart of analysis. Necessarily, this involves comparing the values of different rows in a table. While this has been possible in SQL, usually through self-joins, it has not been efficient or easy to formulate. The LAG/LEAD family enables queries to compare different rows of a table simply by specifying an offset from the current row.

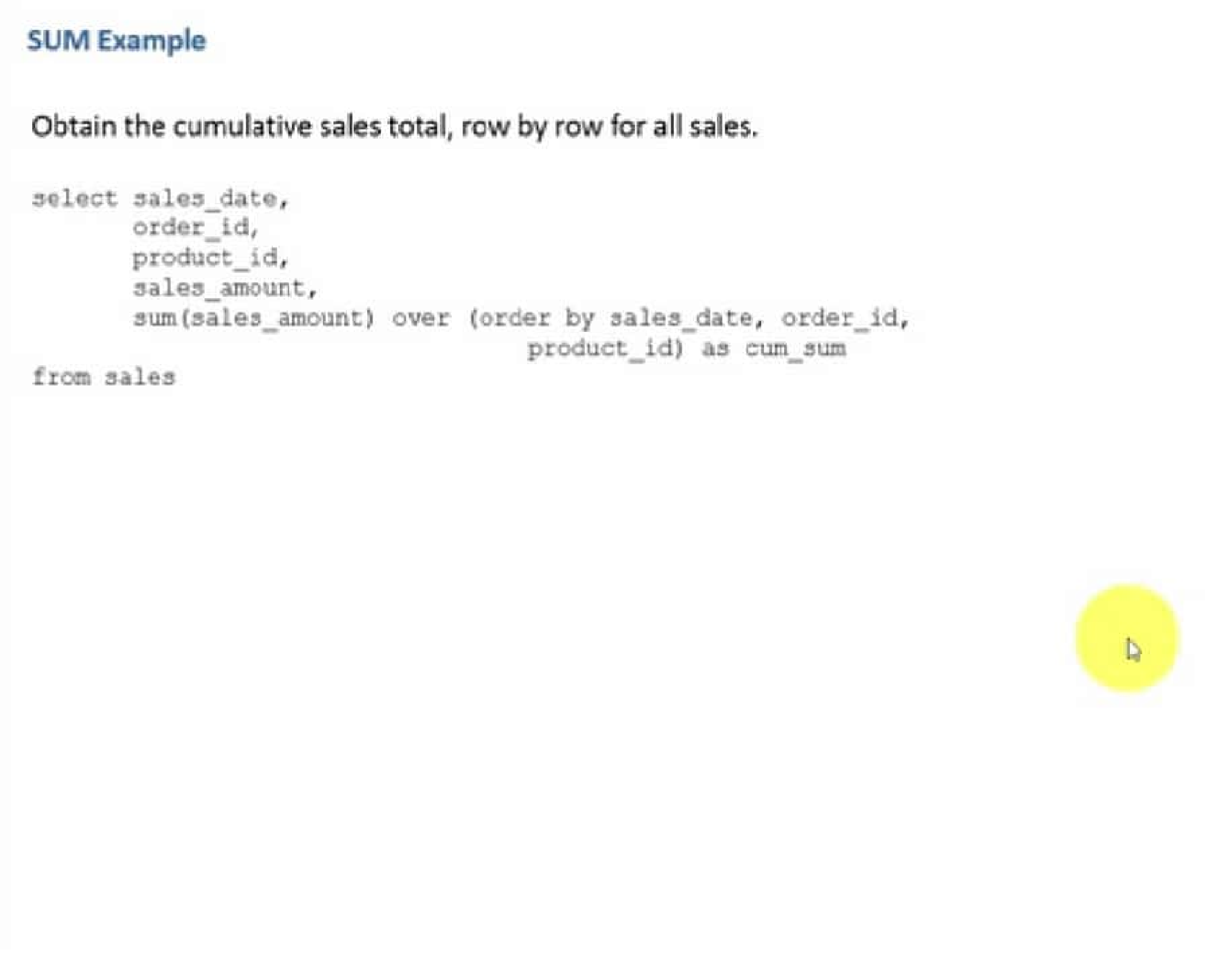


Why Analytical Functions Demo?

We are combining the Detail data and Aggregated data....

```
SELECT s.sales_date,
       s.order_id,
       s.product_id,
       s.customer_id,
       s.total_amount,
       AVG (total_amount) over () AS avg_total_amount
  FROM sales s
```

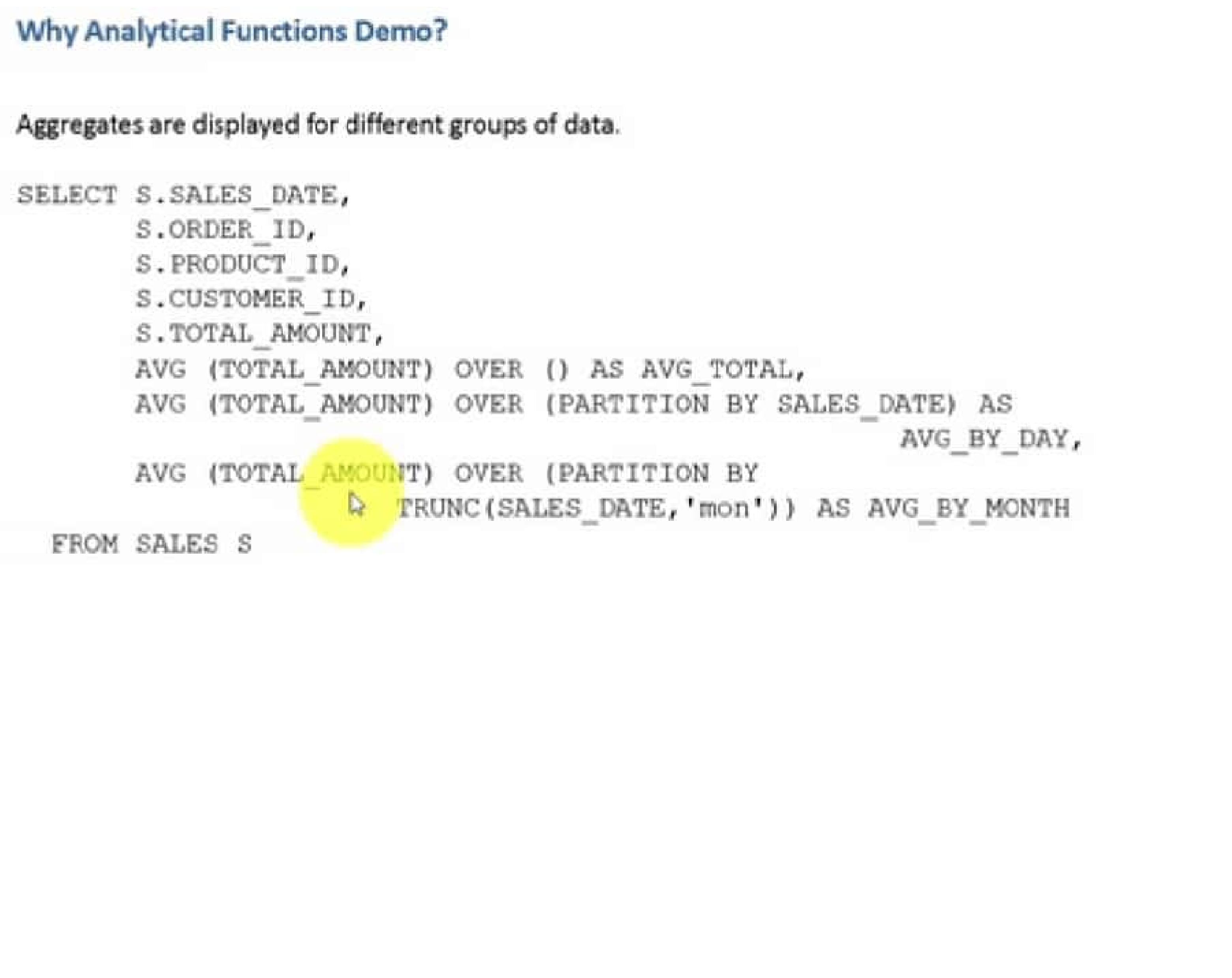
```
SELECT s.sales_date,
       s.order_id,
       s.product_id,
       s.customer_id,
       s.total_amount,
       AVG (total_amount) over (partition by sales_date) AS
                                            avg_total_amount
  FROM sales s
```



SUM Example

Obtain the cumulative sales total, row by row for all sales.

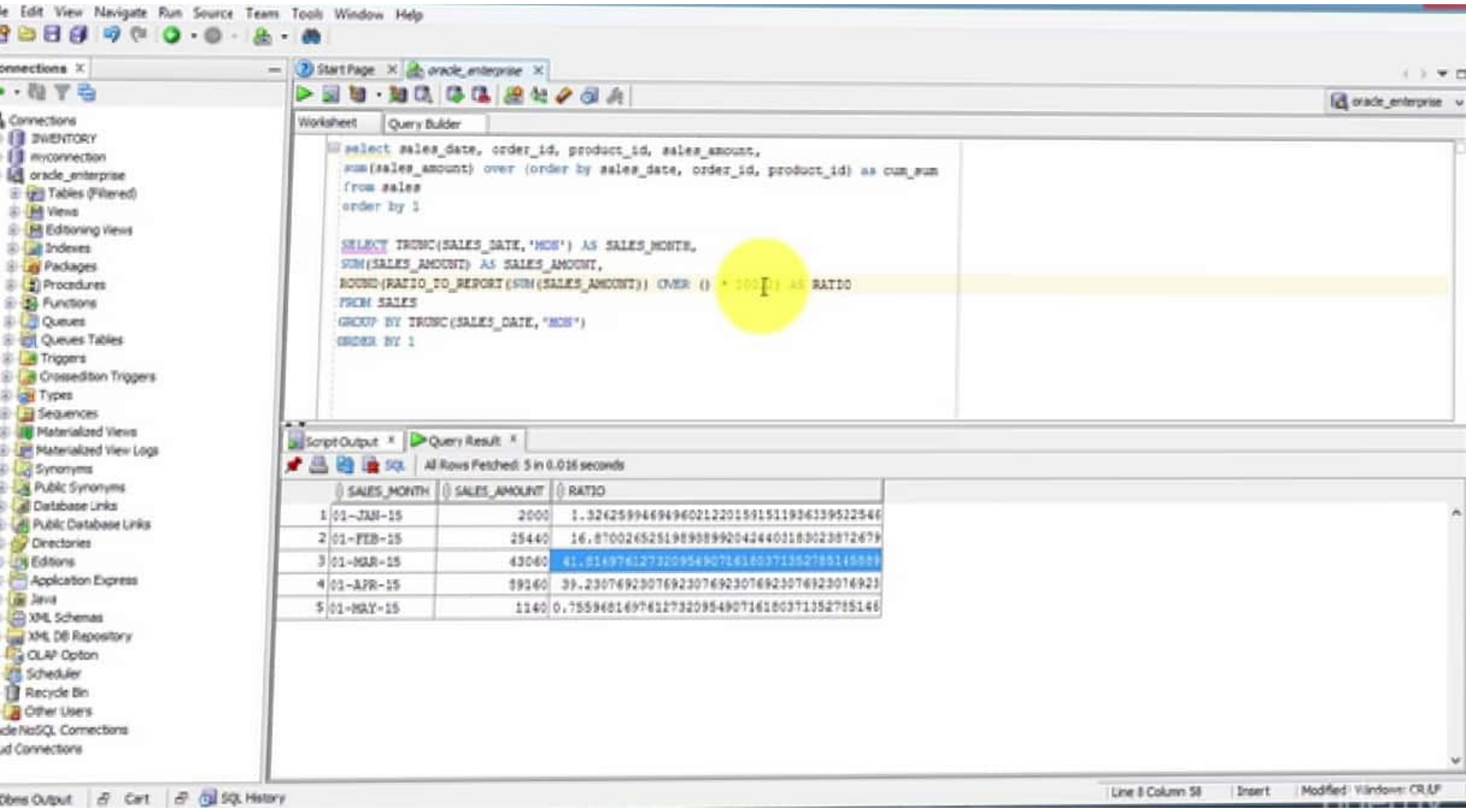
```
select sales_date,
       order_id,
       product_id,
       sales_amount,
       sum(sales_amount) over (order by sales_date, order_id,
                                        product_id) as cum_sum
from sales
```



Why Analytical Functions Demo?

Aggregates are displayed for different groups of data.

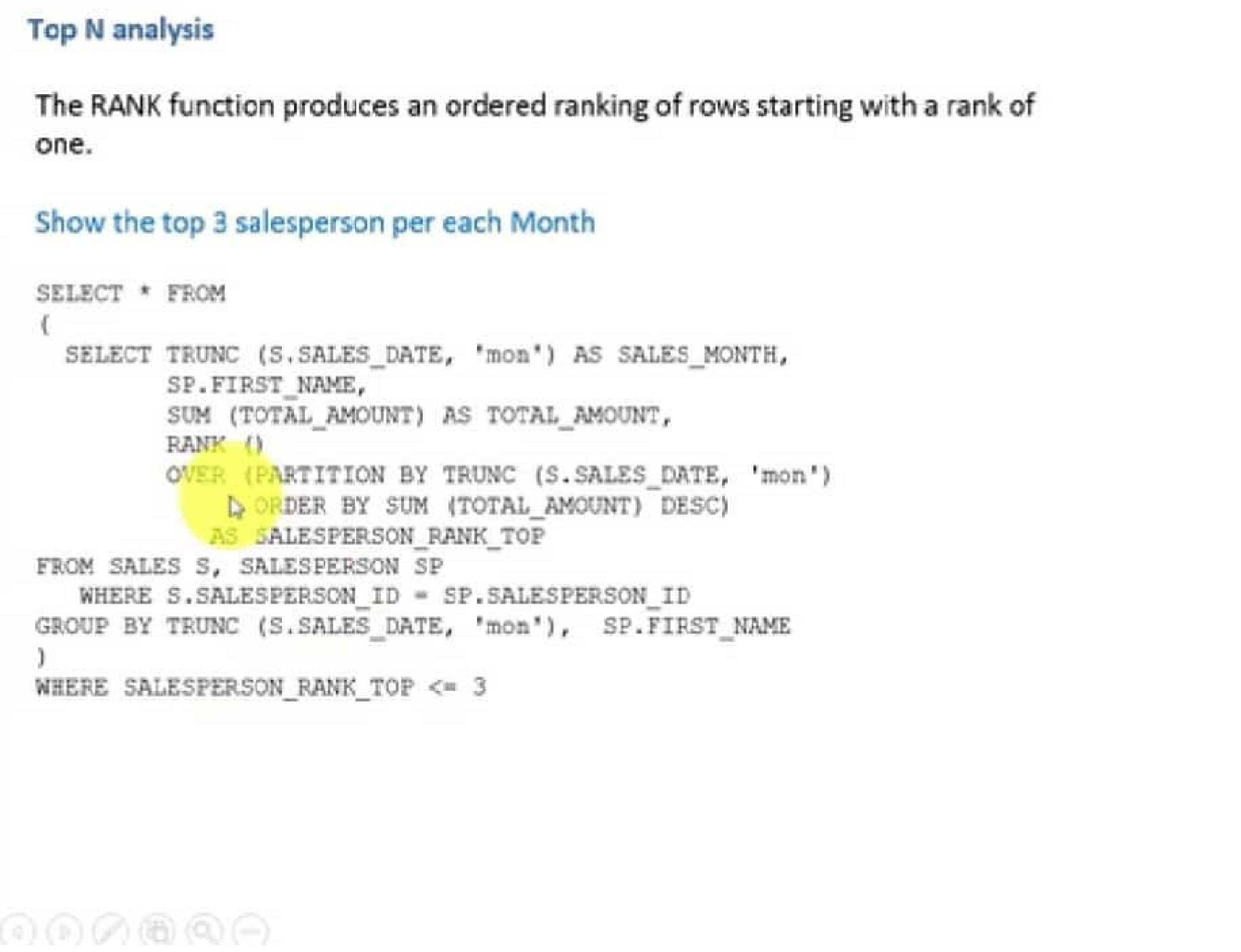
```
SELECT S.SALES_DATE,
       S.ORDER_ID,
       S.PRODUCT_ID,
       S.CUSTOMER_ID,
       S.TOTAL_AMOUNT,
       AVG (TOTAL_AMOUNT) OVER () AS AVG_TOTAL,
       AVG (TOTAL_AMOUNT) OVER (PARTITION BY SALES_DATE) AS
                                                     AVG_BY_DAY,
       AVG (TOTAL_AMOUNT) OVER (PARTITION BY
                    TRUNC(SALES_DATE,'mon')) AS AVG_BY_MONTH
  FROM SALES S
```



```
select sales_date, order_id, product_id, sales_amount,
sum(sales_amount) over (order by sales_date, order_id, product_id) as cum_sum
from sales
order by 1

SELECT TRUNC(SALES_DATE,'MON') AS SALES_MONTH,
SUM(SALES_AMOUNT) AS SALES_AMOUNT,
ROUND(RATIO_TO_REPORT(SUM(SALES_AMOUNT)) OVER () * 100) AS RATIO
FROM SALES
GROUP BY TRUNC(SALES_DATE,'MON')
ORDER BY 1
```

All Rows Fetched: 5 in 0.016 seconds

```
    SALES_MONTH  SALES_AMOUNT  RATIO
  1 01-JAN-15            2000    1.32625994694960212201591511936339522546
  2 01-FEB-15           25440    16.8700265251989389920424403183023872679
  3 01-MAR-15           63060   41.816976127320954907161803713527851458
  4 01-APR-15           59160   39.2307692307692307692307692307692307692
  5 01-MAY-15            1140 0.7559681697612732095490716180371352785146
```

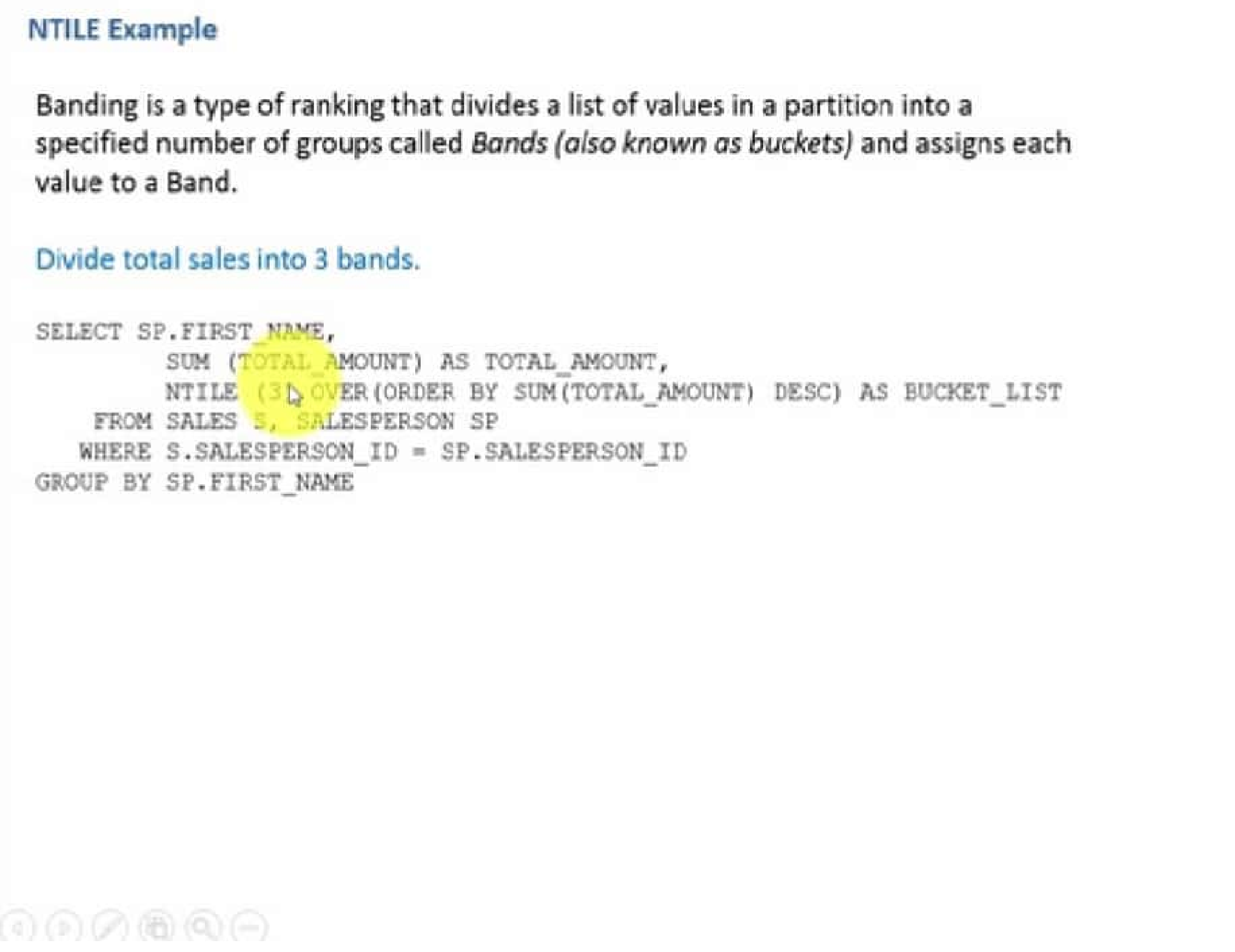


Top N analysis

The RANK function produces an ordered ranking of rows starting with a rank of one.

Show the top 3 salesperson per each Month

```
SELECT * FROM
(
  SELECT TRUNC (S.SALES_DATE, 'mon') AS SALES_MONTH,
         SP.FIRST_NAME,
         SUM (TOTAL_AMOUNT) AS TOTAL_AMOUNT,
         RANK ()
         OVER (PARTITION BY TRUNC (S.SALES_DATE, 'mon')
               ORDER BY SUM (TOTAL_AMOUNT) DESC)
            AS SALESPERSON_RANK_TOP
FROM SALES S, SALESPERSON SP
   WHERE S.SALESPERSON_ID = SP.SALESPERSON_ID
GROUP BY TRUNC (S.SALES_DATE, 'mon'),  SP.FIRST_NAME
)
WHERE SALESPERSON_RANK_TOP <= 3
```

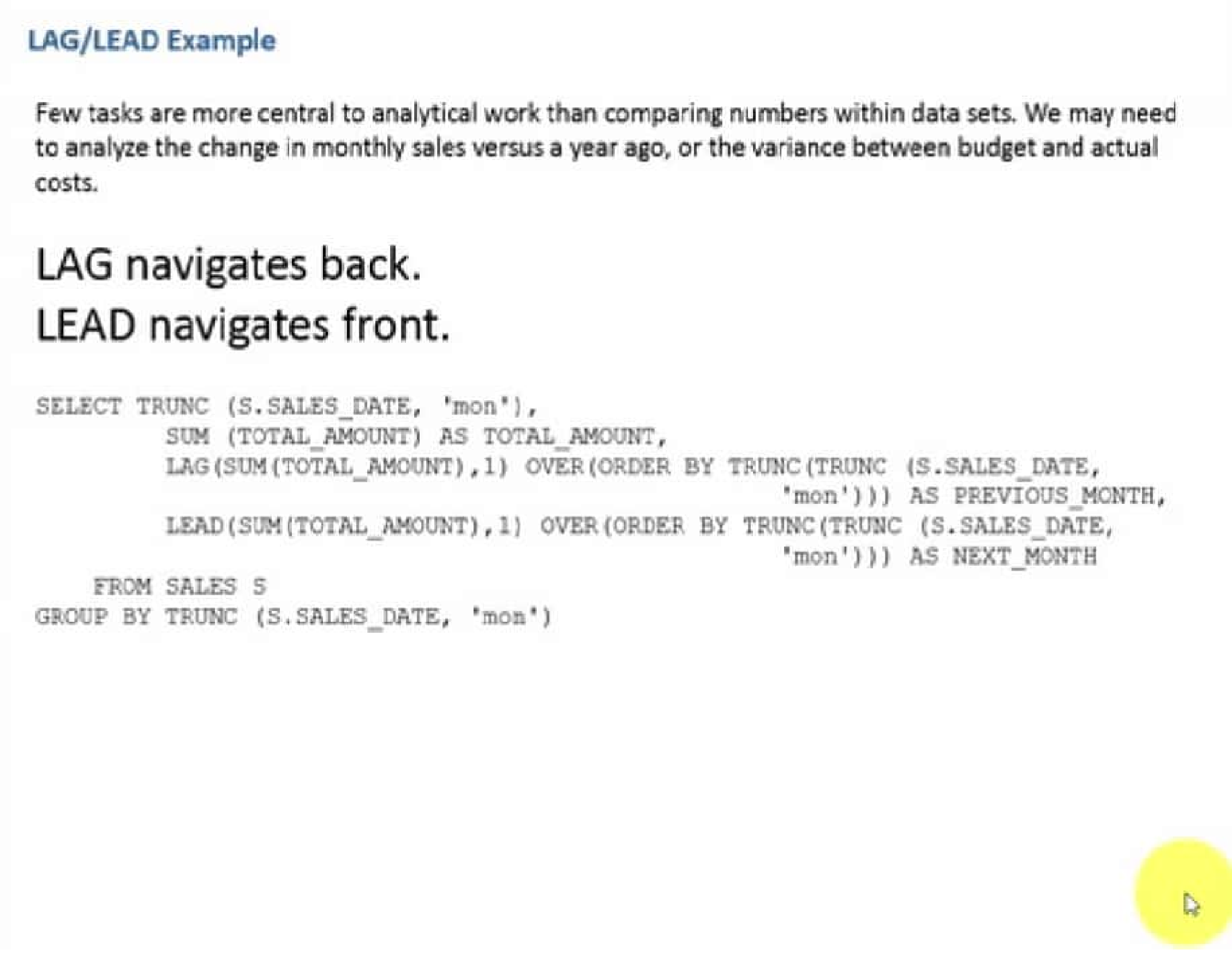


NTILE Example

Banding is a type of ranking that divides a list of values in a partition into a specified number of groups called *Bands (also known as buckets)* and assigns each value to a Band.

Divide total sales into 3 bands.

```
SELECT SP.FIRST_NAME,
         SUM (TOTAL_AMOUNT) AS TOTAL_AMOUNT,
         NTILE (3) OVER(ORDER BY SUM(TOTAL_AMOUNT) DESC) AS BUCKET_LIST
    FROM SALES S, SALESPERSON SP
   WHERE S.SALESPERSON_ID = SP.SALESPERSON_ID
GROUP BY SP.FIRST_NAME
```

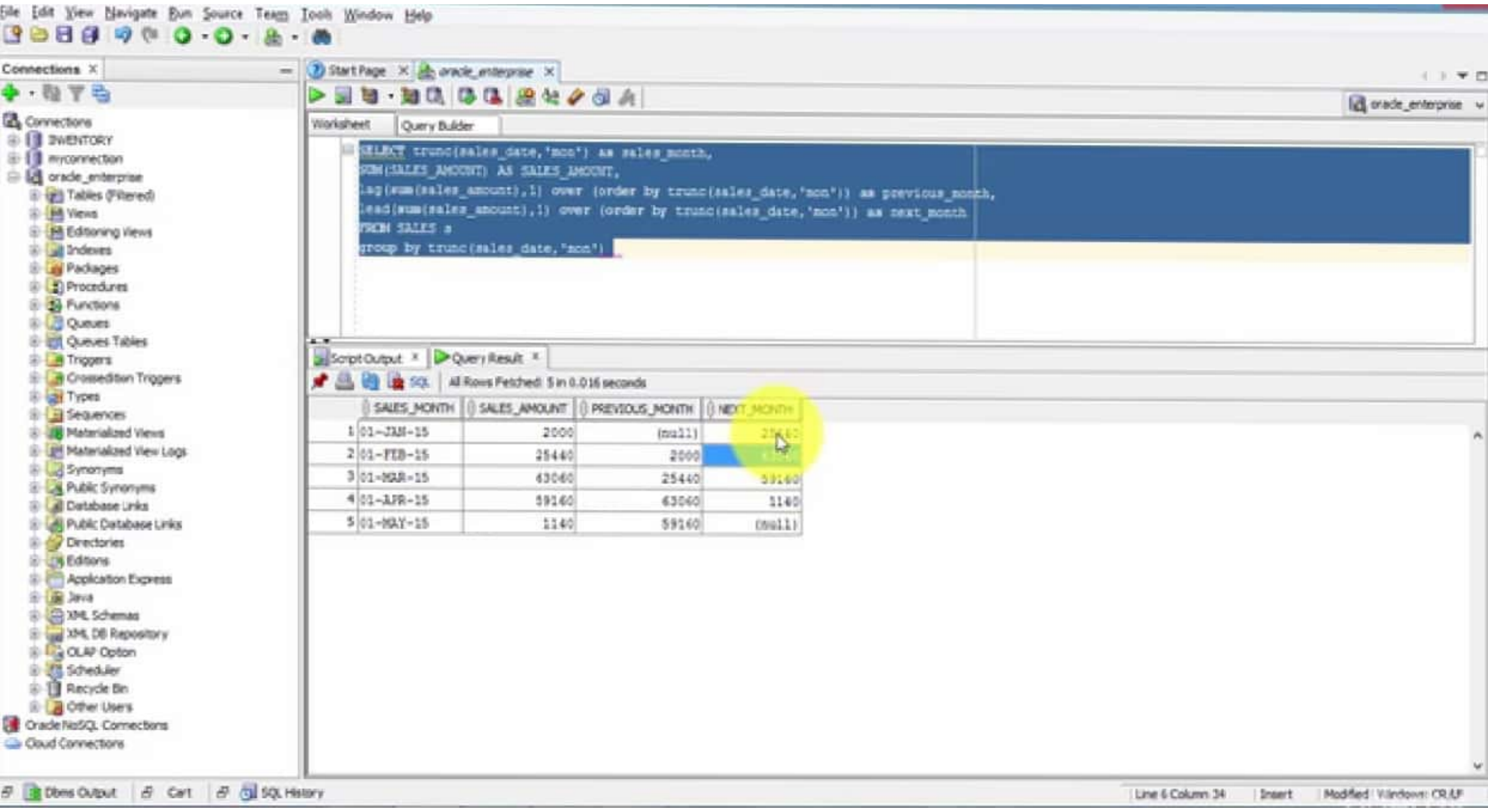


LAG/LEAD Example

Few tasks are more central to analytical work than comparing numbers within data sets. We may need to analyze the change in monthly sales versus a year ago, or the variance between budget and actual costs.

LAG navigates back.
LEAD navigates front.

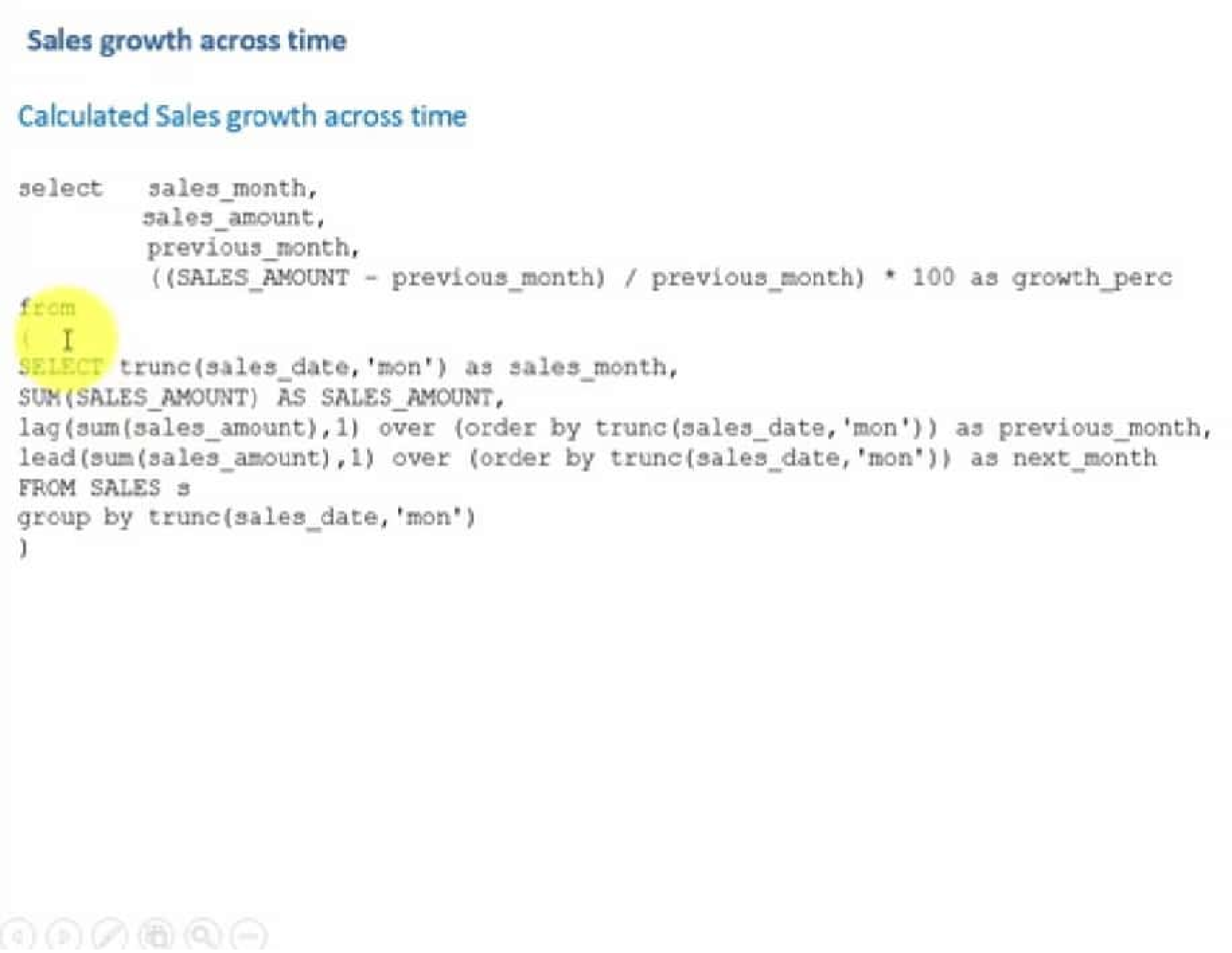
```
SELECT TRUNC (S.SALES_DATE, 'mon'),
         SUM (TOTAL_AMOUNT) AS TOTAL_AMOUNT,
         LAG(SUM(TOTAL_AMOUNT),1) OVER(ORDER BY TRUNC(TRUNC (S.SALES_DATE,
                                              'mon'))) AS PREVIOUS_MONTH,
         LEAD(SUM(TOTAL_AMOUNT),1) OVER(ORDER BY TRUNC(TRUNC (S.SALES_DATE,
                                              'mon'))) AS NEXT_MONTH
    FROM SALES S
GROUP BY TRUNC (S.SALES_DATE, 'mon')
```



```
SELECT trunc(sales_date,'mon') as sales_month,
SUM(SALES_AMOUNT) AS SALES_AMOUNT,
lag(sum(sales_amount),1) over (order by trunc(sales_date,'mon')) as previous_month,
lead(sum(sales_amount),1) over (order by trunc(sales_date,'mon')) as next_month
FROM SALES s
group by trunc(sales_date,'mon')
```

All Rows Fetched: 5 in 0.016 seconds

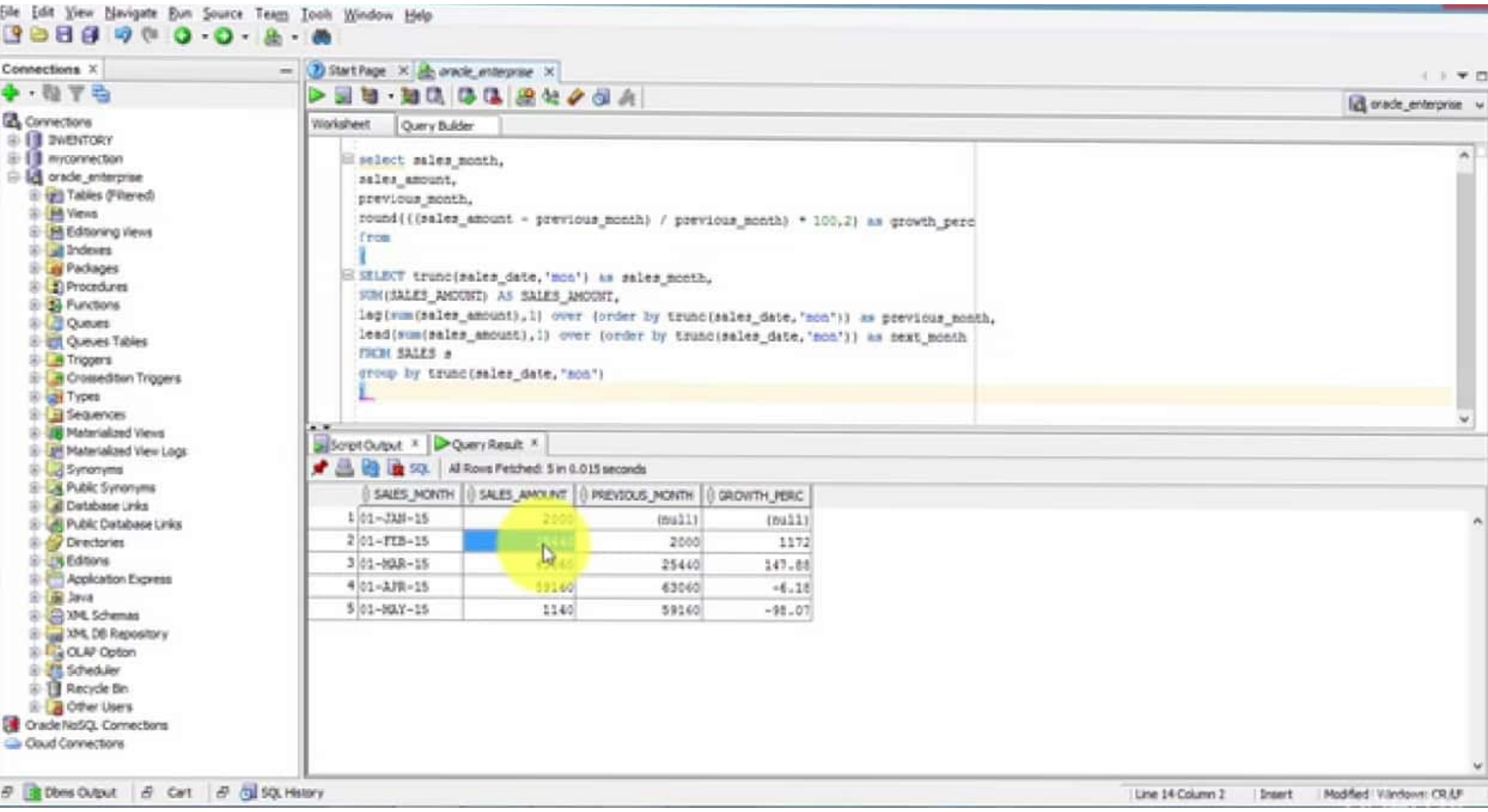
	SALES_MONTH	SALES_AMOUNT	PREVIOUS_MONTH	NEXT_MONTH
1	01-JAN-15	2000	(null)	25440
2	01-FEB-15	25440	2000	[illegible]
3	01-MAR-15	63060	25440	59160
4	01-APR-15	59160	63060	1140
5	01-MAY-15	1140	59160	(null)



Sales growth across time

Calculated Sales growth across time

```
select    sales_month,
         sales_amount,
          previous_month,
          ((SALES_AMOUNT - previous_month) / previous_month) * 100 as growth_perc
from
(
SELECT trunc(sales_date,'mon') as sales_month,
SUM(SALES_AMOUNT) AS SALES_AMOUNT,
lag(sum(sales_amount),1) over (order by trunc(sales_date,'mon')) as previous_month,
lead(sum(sales_amount),1) over (order by trunc(sales_date,'mon')) as next_month
FROM SALES s
group by trunc(sales_date,'mon')
)
```



```
select sales_month,
sales_amount,
previous_month,
round(((sales_amount - previous_month) / previous_month) * 100,2) as growth_perc
from
(
SELECT trunc(sales_date,'mon') as sales_month,
SUM(SALES_AMOUNT) AS SALES_AMOUNT,
lag(sum(sales_amount),1) over (order by trunc(sales_date,'mon')) as previous_month,
lead(sum(sales_amount),1) over (order by trunc(sales_date,'mon')) as next_month
FROM SALES s
group by trunc(sales_date,'mon')
)
```

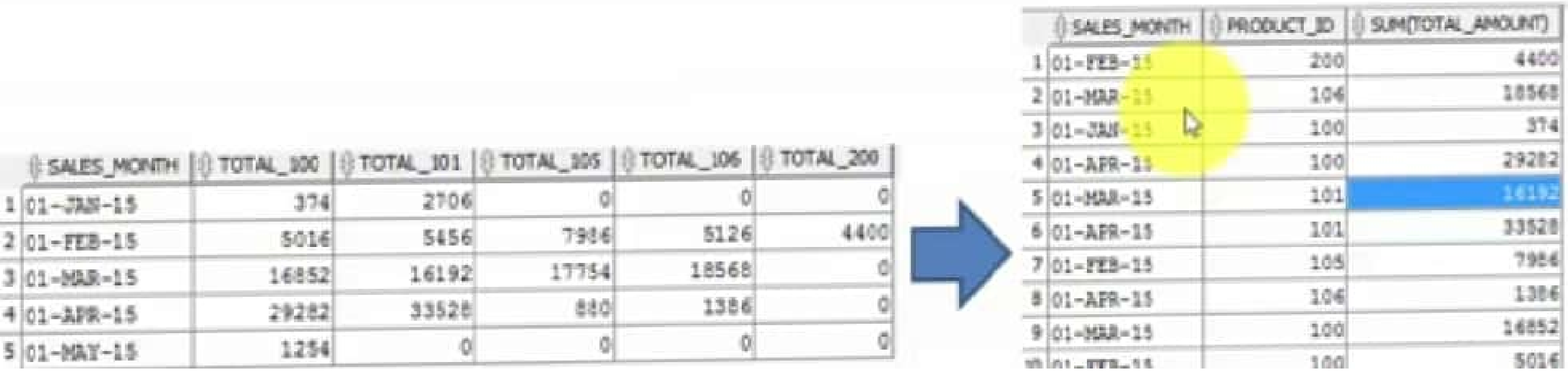
All Rows Fetched: 5 in 0.015 seconds

	SALES_MONTH	SALES_AMOUNT	PREVIOUS_MONTH	GROWTH_PERC
1	01-JAN-15	2000	(null)	(null)
2	01-FEB-15	[illegible]	2000	1172
3	01-MAR-15	[illegible]	25440	147.88
4	01-APR-15	59160	63060	-6.18
5	01-MAY-15	1140	59160	-98.07

Columns to Rows using UNION

You can convert the column level data to row level using UNION.

```
SELECT SALES_MONTH,100 AS PRODUCT_ID,TOTAL_100 AS TOTAL_AMOUNT FROM SALES_PIVOT
UNION ALL
SELECT SALES_MONTH,101 AS PRODUCT_ID,TOTAL_101 AS TOTAL_AMOUNT FROM SALES_PIVOT
UNION ALL
SELECT SALES_MONTH,105 AS PRODUCT_ID,TOTAL_105 AS TOTAL_AMOUNT FROM SALES_PIVOT
UNION ALL
SELECT SALES_MONTH,106 AS PRODUCT_ID,TOTAL_106 AS TOTAL_AMOUNT FROM SALES_PIVOT
UNION ALL
SELECT SALES_MONTH,200 AS PRODUCT_ID,TOTAL_200 AS TOTAL_AMOUNT FROM SALES_PIVOT
```



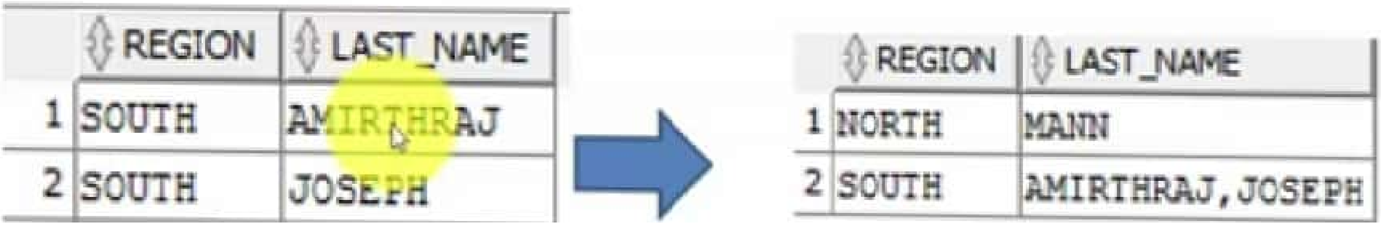
	SALES_MONTH	TOTAL_100	TOTAL_101	TOTAL_105	TOTAL_106	TOTAL_200
1	01-JAN-15	374	2706	0	0	0
2	01-FEB-15	5016	5456	7986	5126	4400
3	01-MAR-15	16852	16192	17754	18568	0
4	01-APR-15	29282	33528	880	1386	0
5	01-MAY-15	1254	0	0	0	0

	SALES_MONTH	PRODUCT_ID	SUM(TOTAL_AMOUNT)
1	01-FEB-15	200	4400
2	01-MAR-15	106	18568
3	01-JAN-15	100	374
4	01-APR-15	100	29282
5	01-MAR-15	101	16192
6	01-APR-15	101	33528
7	01-FEB-15	105	7986
8	01-APR-15	106	1386
9	01-MAR-15	100	16852
10	01-FEB-15	100	5016

Rows to columns using LISTAGG

You can convert the row level data to column level using LISTAGG Analytical function.

```
SELECT REGION,
          LISTAGG (LAST_NAME, ',') WITHIN GROUP (ORDER BY LAST_NAME) AS LAST_NAME
    FROM CUSTOMER
GROUP BY REGION
```



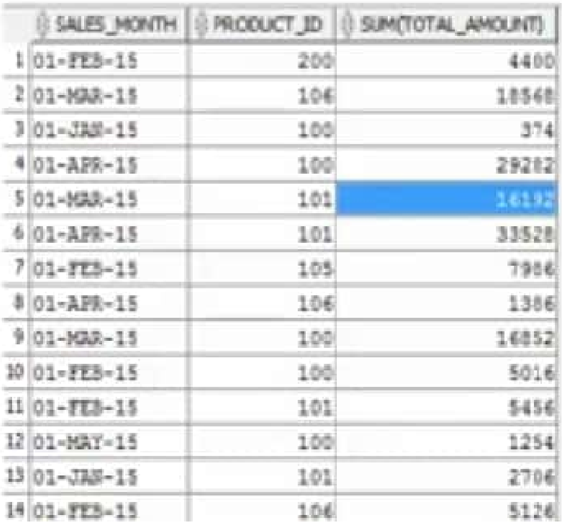
	REGION	LAST_NAME
1	SOUTH	AMIRTHRAJ
2	SOUTH	JOSEPH

	REGION	LAST_NAME
1	NORTH	MANN
2	SOUTH	AMIRTHRAJ,JOSEPH

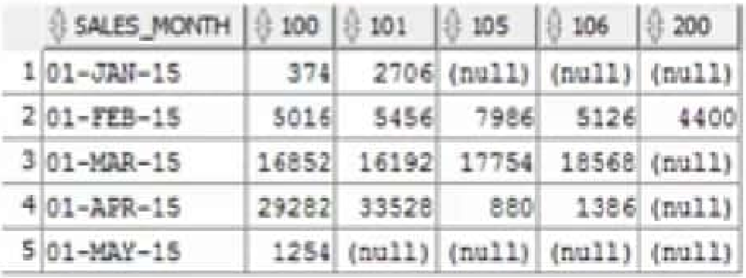
Rows to columns using PIVOT

You can convert the row level data to column level using PIVOT Analytical function.

```
SELECT *
FROM
 (
    SELECT TRUNC(SALES_DATE,'MON') AS SALES_MONTH, PRODUCT_ID, TOTAL_AMOUNT  FROM  SALES
 )
    PIVOT  (SUM(TOTAL_AMOUNT)  FOR (PRODUCT_ID) IN (100 , 101 , 105, 106, 200 )
)
ORDER BY SALES_MONTH
```



	SALES_MONTH	PRODUCT_ID	SUM(TOTAL_AMOUNT)
1	01-FEB-15	200	4400
2	01-MAR-15	106	18568
3	01-JAN-15	100	374
4	01-APR-15	100	29282
5	01-MAR-15	101	16192
6	01-APR-15	101	33528
7	01-FEB-15	105	7986
8	01-APR-15	106	1386
9	01-MAR-15	100	16852
10	01-FEB-15	100	5016
11	01-FEB-15	101	5456
12	01-MAY-15	100	1254
13	01-JAN-15	101	2706
14	01-FEB-15	106	5126

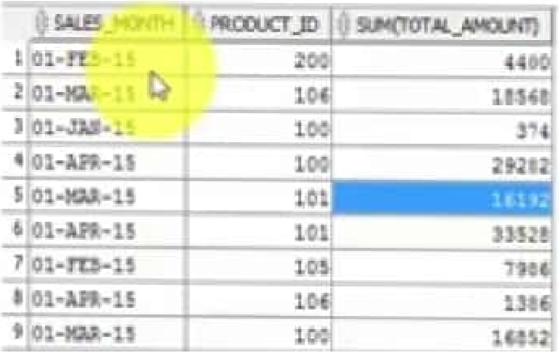


	SALES_MONTH	100	101	105	106	200
1	01-JAN-15	374	2706	(null)	(null)	(null)
2	01-FEB-15	5016	5456	7986	5126	4400
3	01-MAR-15	16852	16192	17754	18568	(null)
4	01-APR-15	29282	33528	880	1386	(null)
5	01-MAY-15	1254	(null)	(null)	(null)	(null)

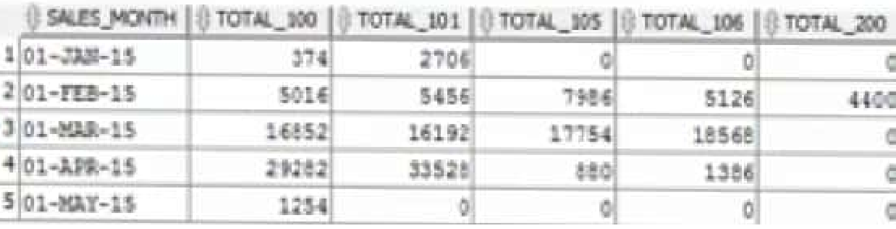
Rows to columns using Case

You can convert the row level data to column level using Case.

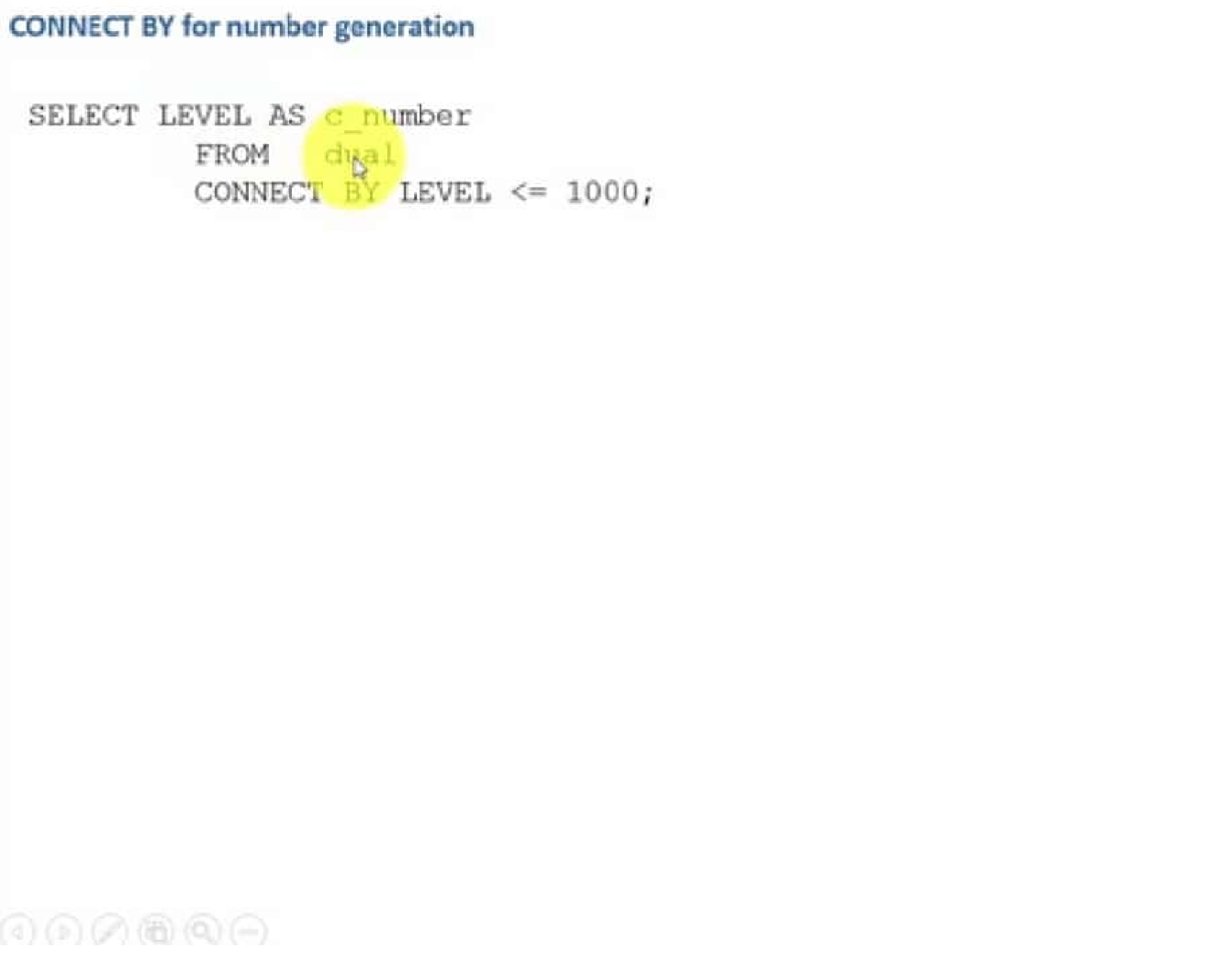
```
SELECT
   TRUNC(SALES_DATE,'MON') AS SALES_MONTH,
   SUM(CASE WHEN PRODUCT_ID = 100 THEN TOTAL_AMOUNT ELSE 0 END) AS TOTAL_100,
   SUM(CASE WHEN PRODUCT_ID = 101 THEN TOTAL_AMOUNT ELSE 0 END) AS TOTAL_101,
   SUM(CASE WHEN PRODUCT_ID = 105 THEN TOTAL_AMOUNT ELSE 0 END) AS TOTAL_105,
   SUM(CASE WHEN PRODUCT_ID = 106 THEN TOTAL_AMOUNT ELSE 0 END) AS TOTAL_106,
   SUM(CASE WHEN PRODUCT_ID = 200 THEN TOTAL_AMOUNT ELSE 0 END) AS TOTAL_200
FROM SALES
GROUP BY TRUNC(SALES_DATE,'MON')
ORDER BY TRUNC(SALES_DATE,'MON')
```



	SALES_MONTH	PRODUCT_ID	SUM(TOTAL_AMOUNT)
1	01-FEB-15	200	4400
2	01-MAR-15	106	18568
3	01-JAN-15	100	374
4	01-APR-15	100	29282
5	01-MAR-15	101	16192
6	01-APR-15	101	33528
7	01-FEB-15	105	7986
8	01-APR-15	106	1386
9	01-MAR-15	100	16852

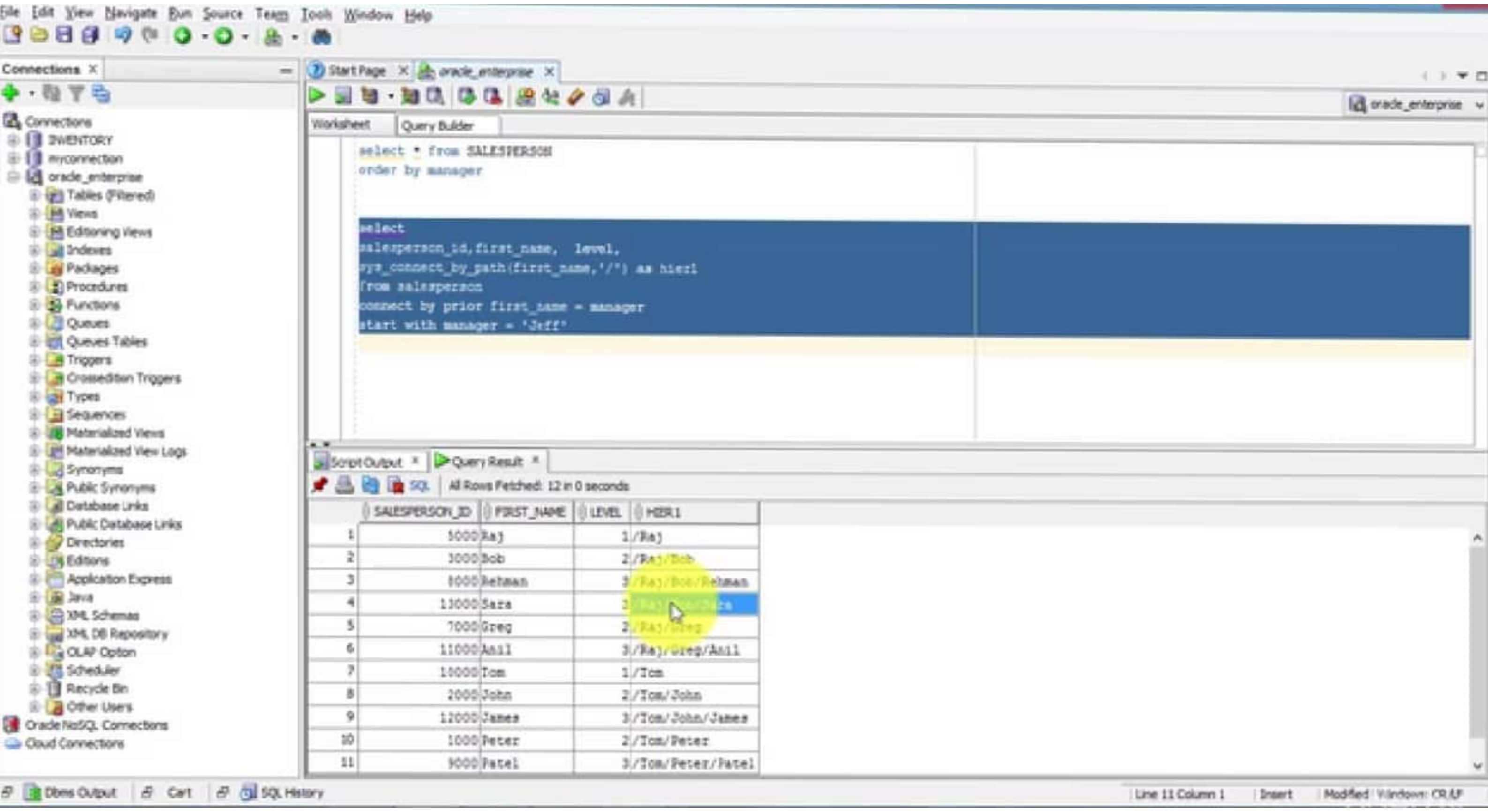


	SALES_MONTH	TOTAL_100	TOTAL_101	TOTAL_105	TOTAL_106	TOTAL_200
1	01-JAN-15	374	2706	0	0	0
2	01-FEB-15	5016	5456	7986	5126	4400
3	01-MAR-15	16852	16192	17754	18568	0
4	01-APR-15	29282	33528	880	1386	0
5	01-MAY-15	1254	0	0	0	0



CONNECT BY for number generation

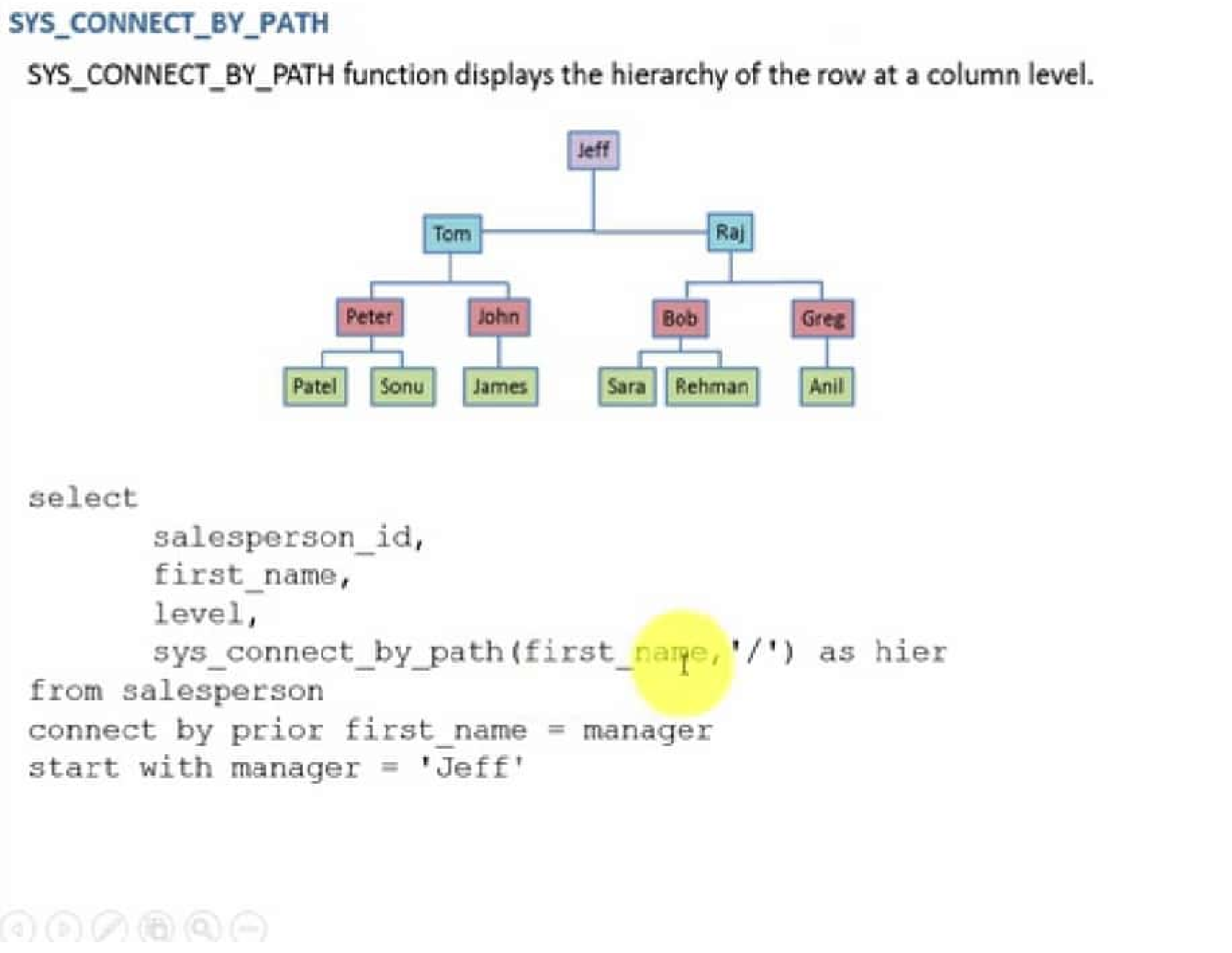
```
SELECT LEVEL AS c_number
          FROM   dual
          CONNECT BY LEVEL <= 1000;
```



```
select * from SALESPERSON
order by manager


select
salesperson_id,first_name,  level,
sys_connect_by_path(first_name,'/') as hier1
from salesperson
connect by prior first_name = manager
start with manager = 'Jeff'
```

All Rows Fetched: 12 in 0 seconds

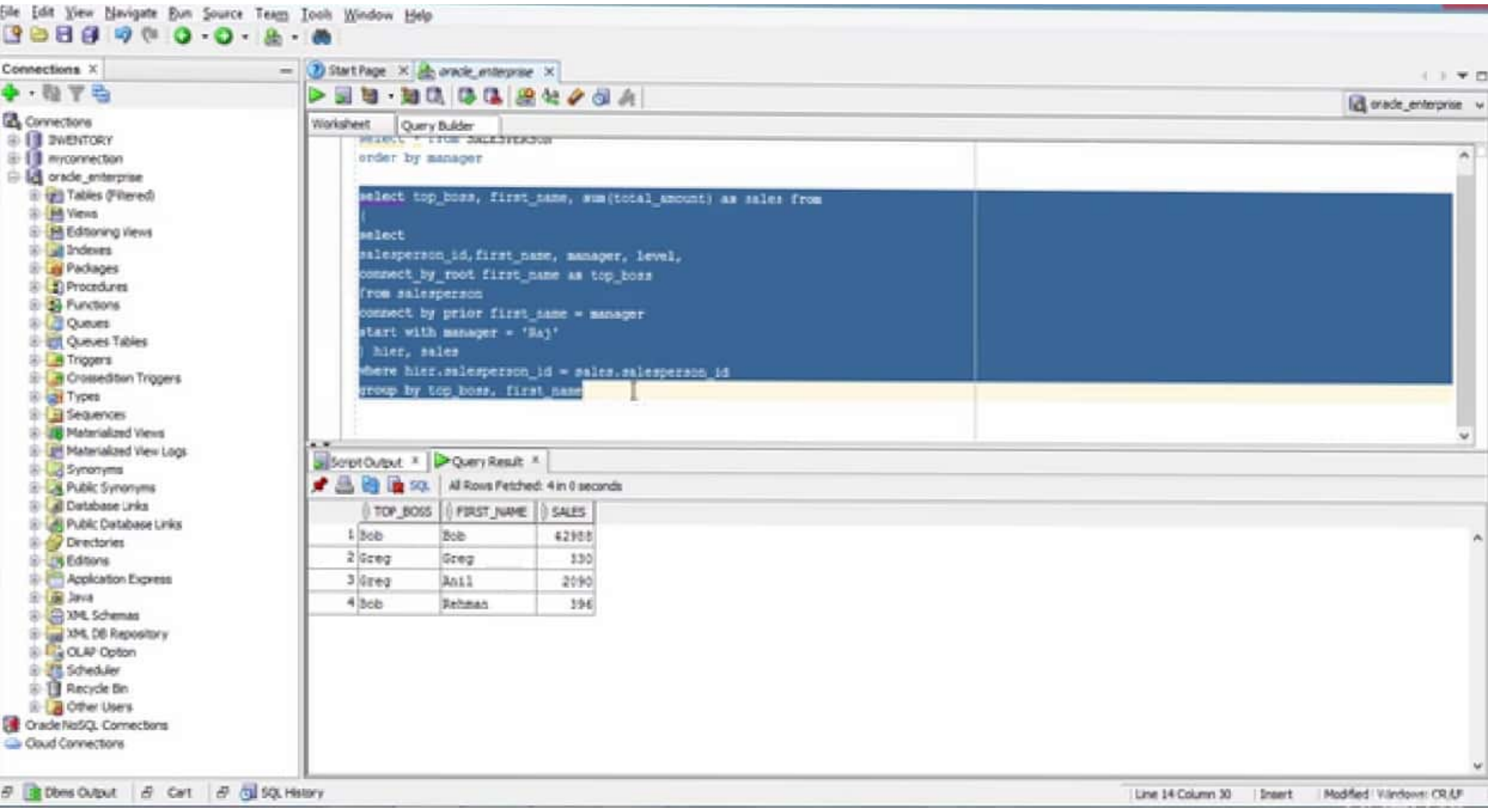
	SALESPERSON_ID	FIRST_NAME	LEVEL	HIER1
1	5000	Raj	1	/Raj
2	3000	Bob	2	/Raj/Bob
3	8000	Rehman	3	/Raj/Bob/Rehman
4	13000	Sara	3	/Raj/Bob/Sara
5	7000	Greg	2	/Raj/Greg
6	11000	Anil	3	/Raj/Greg/Anil
7	10000	Tom	1	/Tom
8	2000	John	2	/Tom/John
9	12000	James	3	/Tom/John/James
10	1000	Peter	2	/Tom/Peter
11	9000	Patel	3	/Tom/Peter/Patel



SYS_CONNECT_BY_PATH

SYS_CONNECT_BY_PATH function displays the hierarchy of the row at a column level.

```
select
       salesperson_id,
       first_name,
       level,
       sys_connect_by_path(first_name,'/') as hier
from salesperson
connect by prior first_name = manager
start with manager = 'Jeff'
```



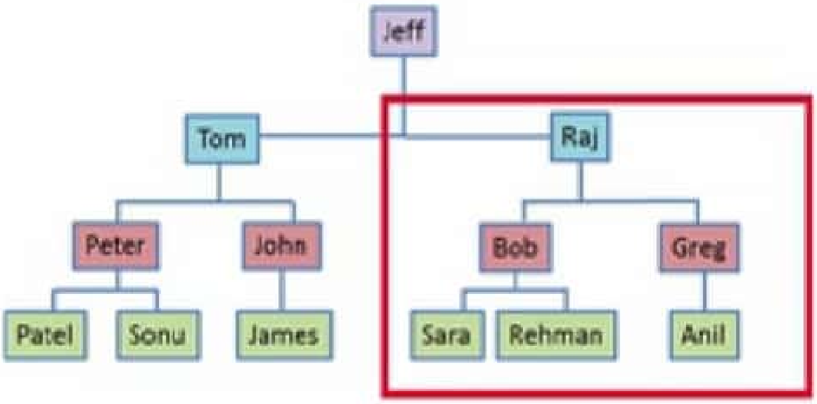
```
select * from SALESPERSON
order by manager

select top_boss, first_name, sum(total_amount) as sales from
(
select
salesperson_id,first_name, manager, level,
connect_by_root first_name as top_boss
from salesperson
connect by prior first_name = manager
start with manager = 'Raj'
) hier, sales
where hier.salesperson_id = sales.salesperson_id
group by top_boss, first_name
```

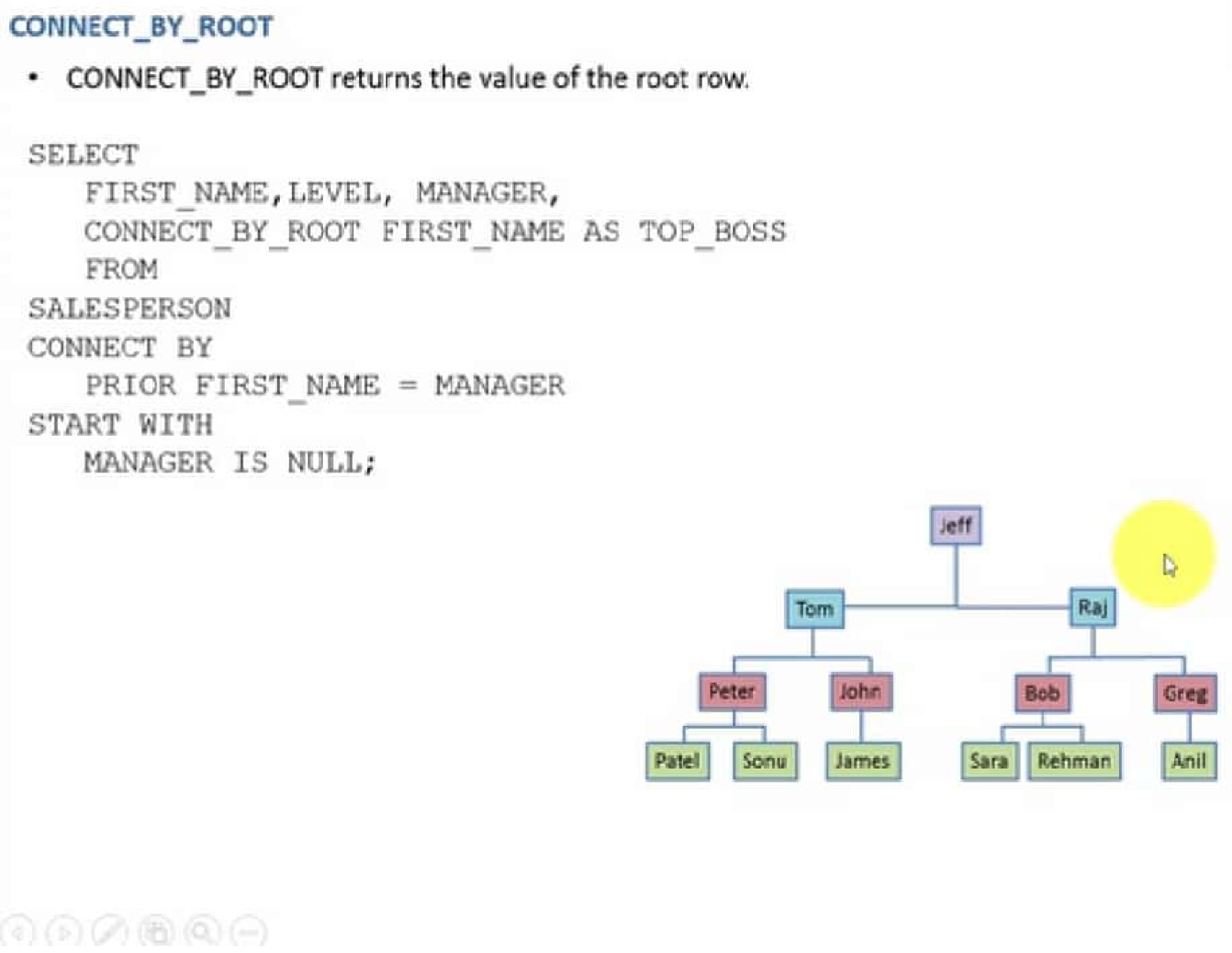
All Rows Fetched: 4 in 0 seconds

	TOP_BOSS	FIRST_NAME	SALES
1	Bob	Bob	62955
2	Greg	Greg	330
3	Greg	Anil	2090
4	Bob	Rehman	396

Example:

Create a report which returns the total sales of each salesperson for the Manager 'Raj' and all salespersons below that salespersons in the hierarchy.





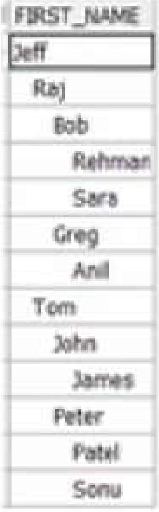
CONNECT_BY_ROOT

- CONNECT_BY_ROOT returns the value of the root row.

```
SELECT
   FIRST_NAME,LEVEL, MANAGER,
   CONNECT_BY_ROOT FIRST_NAME AS TOP_BOSS
   FROM
SALESPERSON
CONNECT BY
   PRIOR FIRST_NAME = MANAGER
START WITH
   MANAGER IS NULL;
```

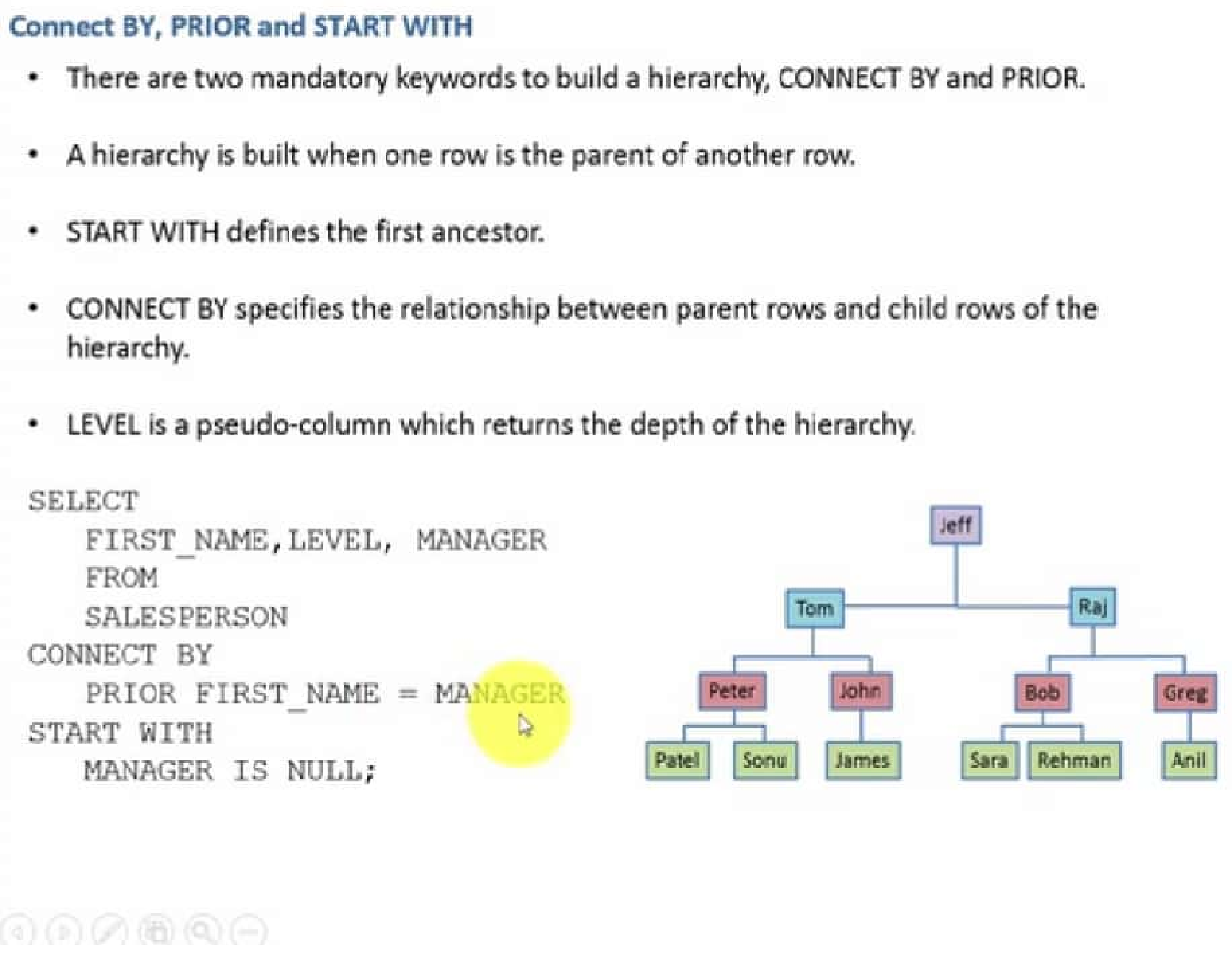
ORDER SIBLINGS BY

The rows in a hierarchical query are returned as a tree, the children following the parent. ORDER SIBLINGS BY preserves the hierarchy and orders the children of each parent.



```
FIRST_NAME
Jeff
   Raj
      Bob
         Rehman
         Sara
      Greg
         Anil
   Tom
      John
         James
      Peter
         Patel
         Sonu
```

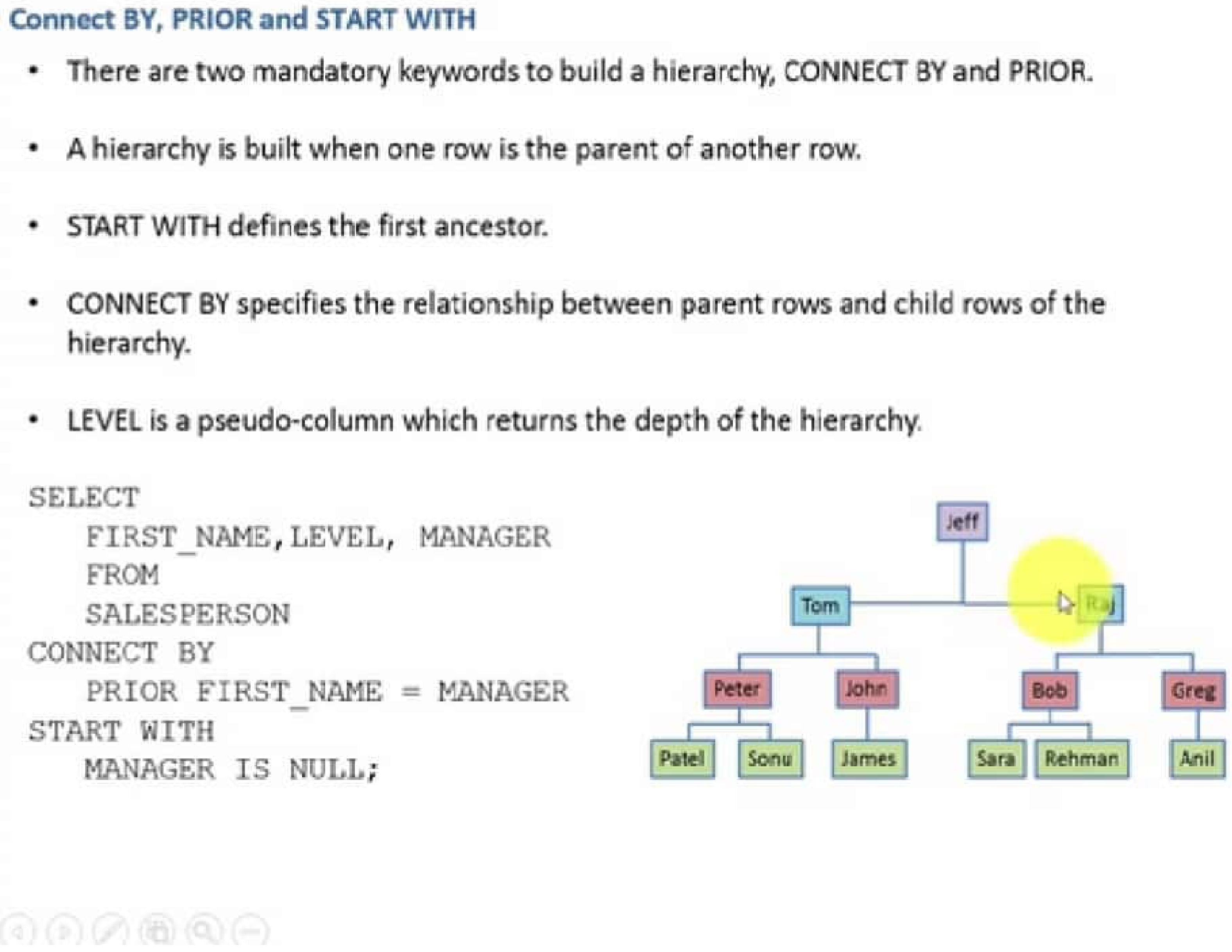
```
SELECT
   CONCAT ( LPAD  (' ', LEVEL*3-3), FIRST_NAME) AS FIRST_NAME
   FROM
   SALESPERSON
CONNECT BY
   PRIOR FIRST_NAME = MANAGER
START WITH
   MANAGER IS NULL
ORDER SIBLINGS BY  SALESPERSON.FIRST_NAME desc;
```



Connect BY, PRIOR and START WITH

- There are two mandatory keywords to build a hierarchy, CONNECT BY and PRIOR.
- A hierarchy is built when one row is the parent of another row.
- START WITH defines the first ancestor.
- CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy.
- LEVEL is a pseudo-column which returns the depth of the hierarchy.

```
SELECT
   FIRST_NAME,LEVEL, MANAGER
   FROM
   SALESPERSON
CONNECT BY
   PRIOR FIRST_NAME = MANAGER
START WITH
   MANAGER IS NULL;
```



Connect BY, PRIOR and START WITH

- There are two mandatory keywords to build a hierarchy, CONNECT BY and PRIOR.

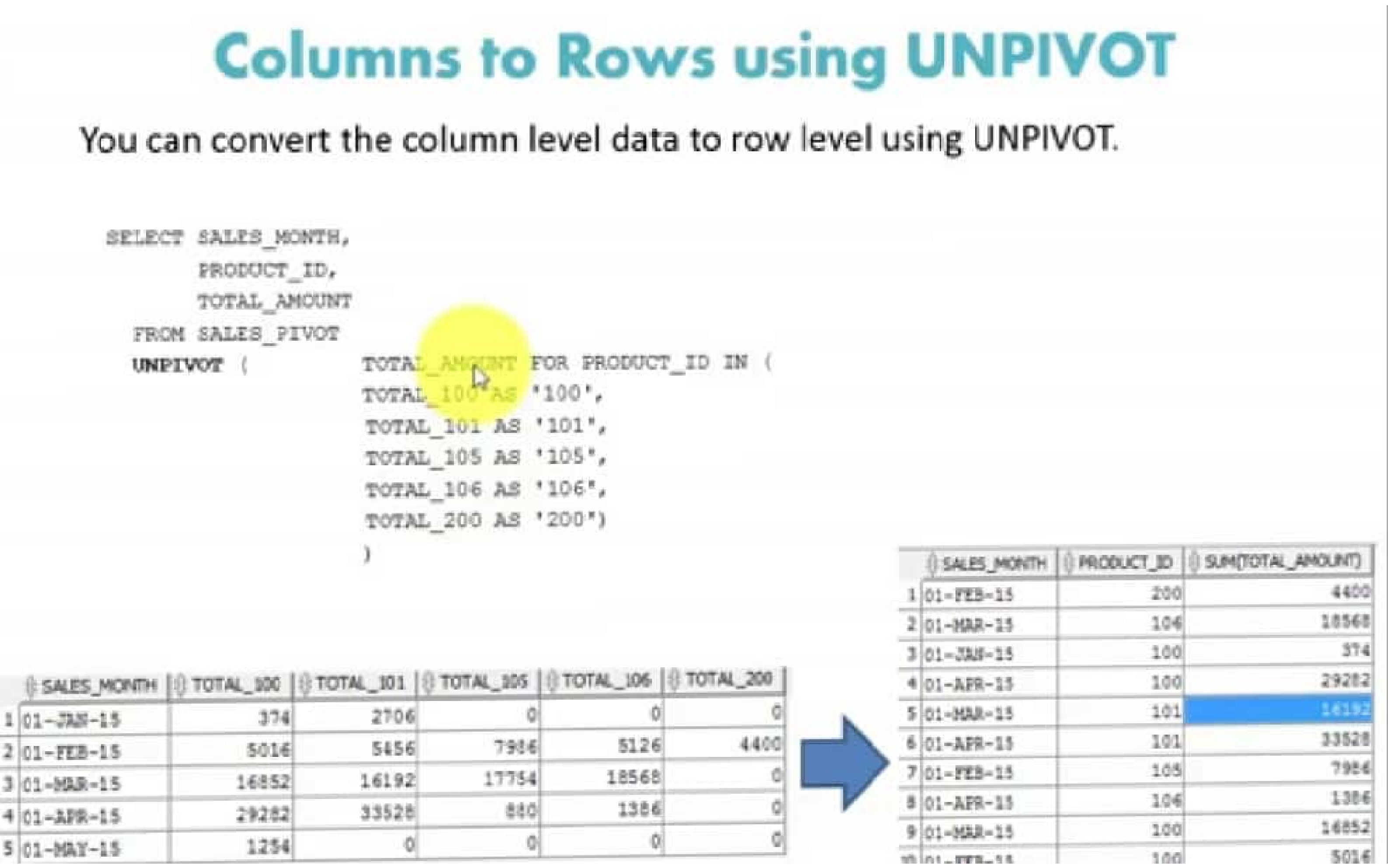
- A hierarchy is built when one row is the parent of another row.

- START WITH defines the first ancestor.

- CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy.

- LEVEL is a pseudo-column which returns the depth of the hierarchy.

```
SELECT
   FIRST_NAME,LEVEL, MANAGER
   FROM
   SALESPERSON
CONNECT BY
   PRIOR FIRST_NAME = MANAGER
START WITH
   MANAGER IS NULL;
```



Columns to Rows using UNPIVOT

You can convert the column level data to row level using UNPIVOT.

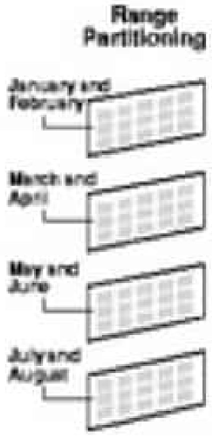
```
SELECT SALES_MONTH,
       PRODUCT_ID,
       TOTAL_AMOUNT
  FROM SALES_PIVOT
  UNPIVOT (         TOTAL_AMOUNT FOR PRODUCT_ID IN (
                    TOTAL_100 AS '100',
                    TOTAL_101 AS '101',
                    TOTAL_105 AS '105',
                    TOTAL_106 AS '106',
                    TOTAL_200 AS '200')
                    )
```

	SALES_MONTH	TOTAL_100	TOTAL_101	TOTAL_105	TOTAL_106	TOTAL_200
1	01-JAN-15	374	2706	0	0	0
2	01-FEB-15	5016	5456	7986	5126	4400
3	01-MAR-15	16852	16192	17754	18568	0
4	01-APR-15	29282	33528	880	1386	0
5	01-MAY-15	1254	0	0	0	0

	SALES_MONTH	PRODUCT_ID	SUM(TOTAL_AMOUNT)
1	01-FEB-15	200	4400
2	01-MAR-15	106	18568
3	01-JAN-15	100	374
4	01-APR-15	100	29282
5	01-MAR-15	101	16192
6	01-APR-15	101	33528
7	01-FEB-15	105	7986
8	01-APR-15	106	1386
9	01-MAR-15	100	16852
10	01-FEB-15	100	5016

Range Partition

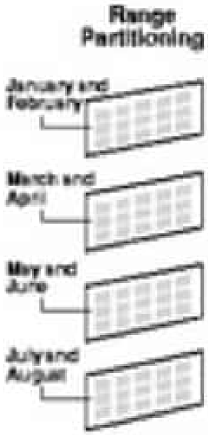
Range partitioning is a partitioning technique where data is stored separately in different sub-tables based on the data range.



```
CREATE TABLE SALES1
(
customer_id NUMBER,
order_date DATE,
order_amount number,
Region varchar2(10)
)
PARTITION BY RANGE (order_date)
(
PARTITION sales_p1507 VALUES LESS THAN (TO_DATE('2015-07-01', 'YYYY-MM-DD')),
PARTITION sales_p1508 VALUES LESS THAN (TO_DATE('2015-08-01', 'YYYY-MM-DD')),
PARTITION sales_p1509 VALUES LESS THAN (TO_DATE('2015-09-01', 'YYYY-MM-DD')),
PARTITION sales_pmax VALUES LESS THAN (MAXVALUE)
);
```

Range Partition

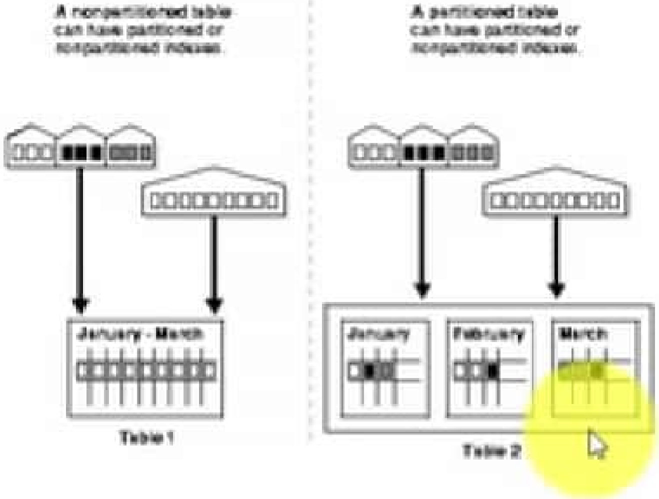
Range partitioning is a partitioning technique where data is stored separately in different sub-tables based on the data range.



```
CREATE TABLE SALES1
(
customer_id NUMBER,
order_date DATE,
order_amount number,
Region varchar2(10)
)
PARTITION BY RANGE (order_date)
(
PARTITION sales_p1507 VALUES LESS THAN (TO_DATE('2015-07-01', 'YYYY-MM-DD')),
PARTITION sales_p1508 VALUES LESS THAN (TO_DATE('2015-08-01', 'YYYY-MM-DD')),
PARTITION sales_p1509 VALUES LESS THAN (TO_DATE('2015-09-01', 'YYYY-MM-DD')),
PARTITION sales_pmax VALUES LESS THAN (MAXVALUE)
);
```

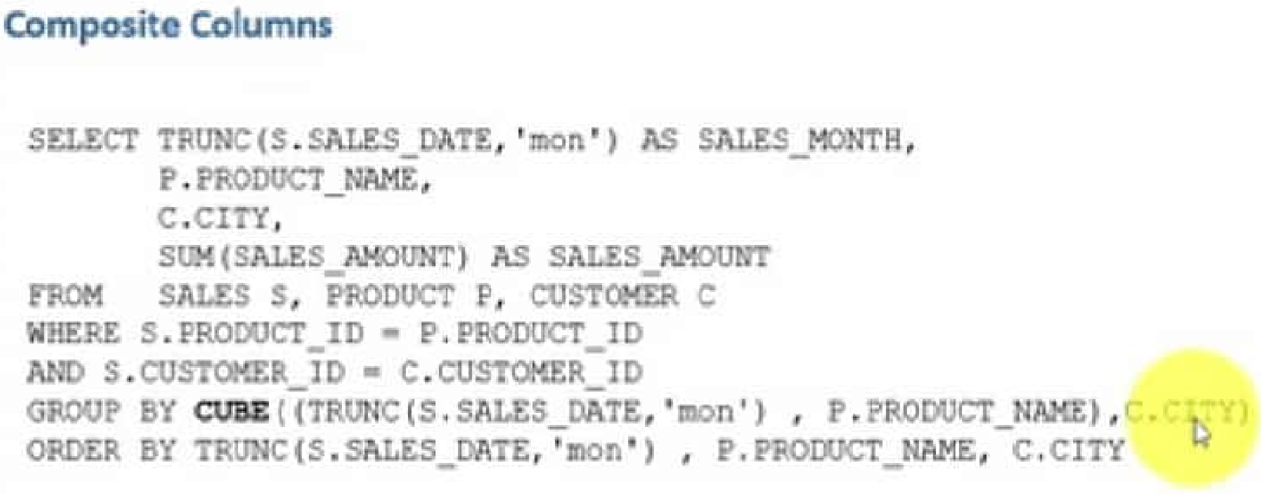
Table Partitions

Dividing table into multiple pieces is called Partitioning.


A nonpartitioned table can have partitioned or nonpartitioned indexes.
A partitioned table can have partitioned or nonpartitioned indexes.
January - March
Table 1
January
February
March
Table 2

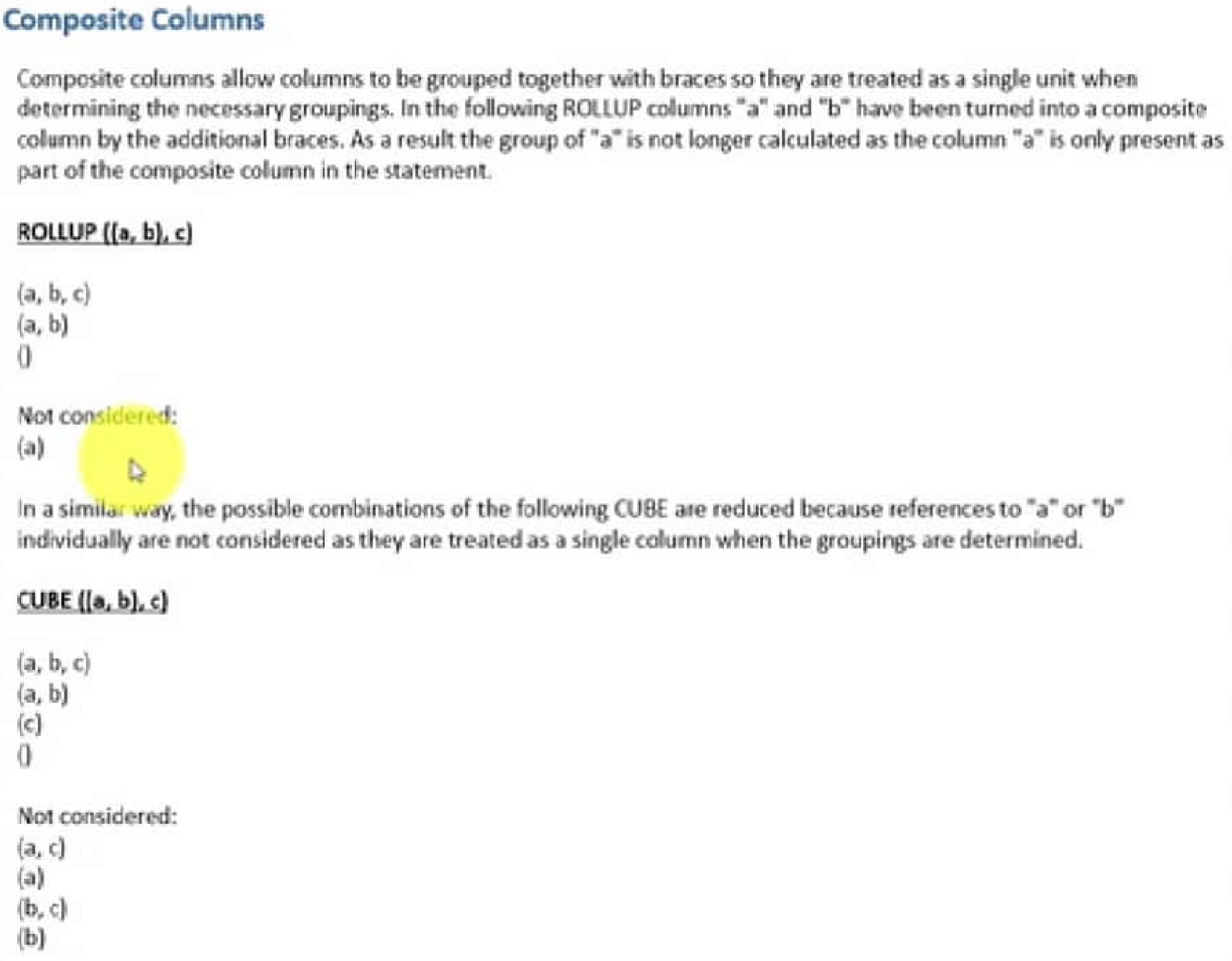
Why do we partition tables??

- Partitioning enables data management operations such data loads, index creation and rebuilding, and backup/recovery at the partition level, rather than on the entire table. This results in significantly reduced times for these operations.
- Partitioning improves query performance.
- Partition independence for partition maintenance operations lets you perform concurrent maintenance operations on different partitions of the same table or index.
- Partitioning increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures.
- Partitioning can be implemented without requiring any modifications to your applications.



Composite Columns

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
       P.PRODUCT_NAME,
       C.CITY,
       SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P, CUSTOMER C
WHERE S.PRODUCT_ID = P.PRODUCT_ID
AND S.CUSTOMER_ID = C.CUSTOMER_ID
GROUP BY CUBE((TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME),C.CITY)
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME, C.CITY
```



Composite Columns

Composite columns allow columns to be grouped together with braces so they are treated as a single unit when determining the necessary groupings. In the following ROLLUP columns "a" and "b" have been turned into a composite column by the additional braces. As a result the group of "a" is not longer calculated as the column "a" is only present as part of the composite column in the statement.

ROLLUP ((a, b), c)

(a, b, c)
(a, b)
()

Not considered:
(a)

In a similar way, the possible combinations of the following CUBE are reduced because references to "a" or "b" individually are not considered as they are treated as a single column when the groupings are determined.

CUBE ((a, b), c)

(a, b, c)
(a, b)
(c)
()

Not considered:
(a, c)
(a)
(b, c)
(b)

Composite Columns

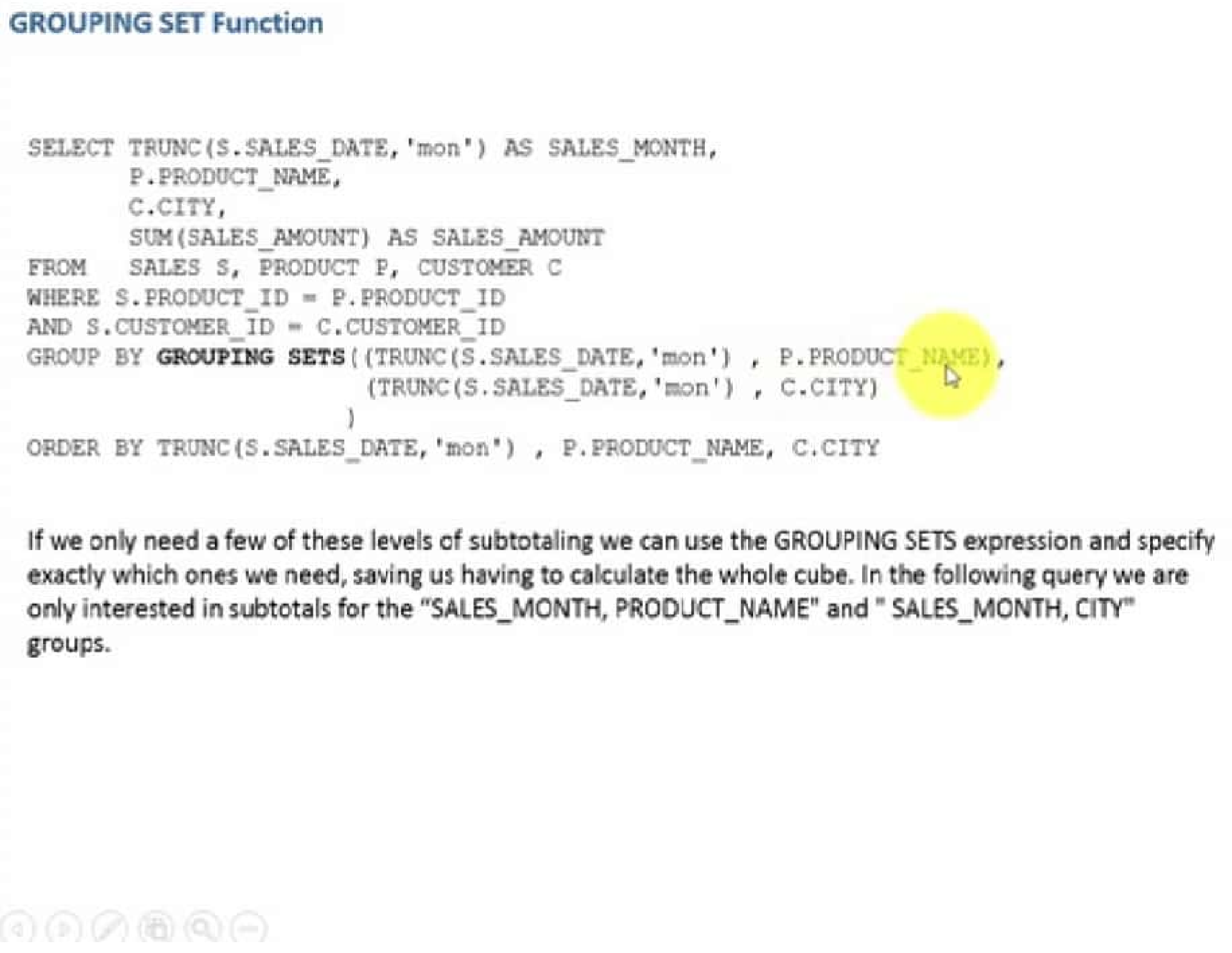
ROLLUP and CUBE consider each column independently when deciding which subtotals must be calculated.

ROLLUP (SALES_MONTH, PRODUCT_NAME, CITY)

(SALES_MONTH, PRODUCT_NAME, CITY)
(SALES_MONTH, PRODUCT_NAME)
(SALES_MONTH)
()

CUBE (SALES_MONTH, PRODUCT_NAME, CITY)

(SALES_MONTH, PRODUCT_NAME, CITY)
(SALES_MONTH, PRODUCT_NAME)
(SALES_MONTH, CITY)
(SALES_MONTH)
(PRODUCT_NAME, CITY)
(PRODUCT_NAME)
(CITY)
()



GROUPING SET Function

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
       P.PRODUCT_NAME,
       C.CITY,
       SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P, CUSTOMER C
WHERE S.PRODUCT_ID = P.PRODUCT_ID
AND S.CUSTOMER_ID = C.CUSTOMER_ID
GROUP BY GROUPING SETS((TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME),
                        (TRUNC(S.SALES_DATE,'mon') , C.CITY)
                       )
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME, C.CITY
```

If we only need a few of these levels of subtotaling we can use the GROUPING SETS expression and specify exactly which ones we need, saving us having to calculate the whole cube. In the following query we are only interested in subtotals for the "SALES_MONTH, PRODUCT_NAME" and " SALES_MONTH, CITY" groups.

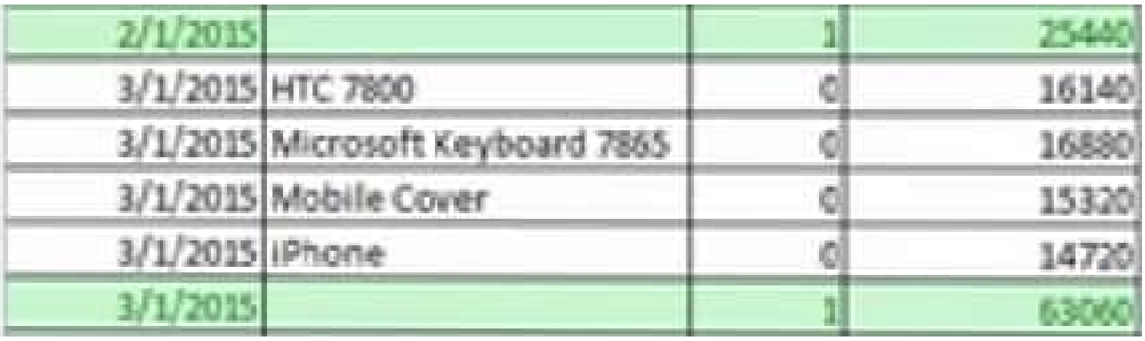
GROUPING SETS Function

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
       P.PRODUCT_NAME,
       C.CITY,
       SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P, CUSTOMER C
WHERE S.PRODUCT_ID = P.PRODUCT_ID
AND S.CUSTOMER_ID = C.CUSTOMER_ID
GROUP BY CUBE (TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME, C.CITY)
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME, C.CITY
```

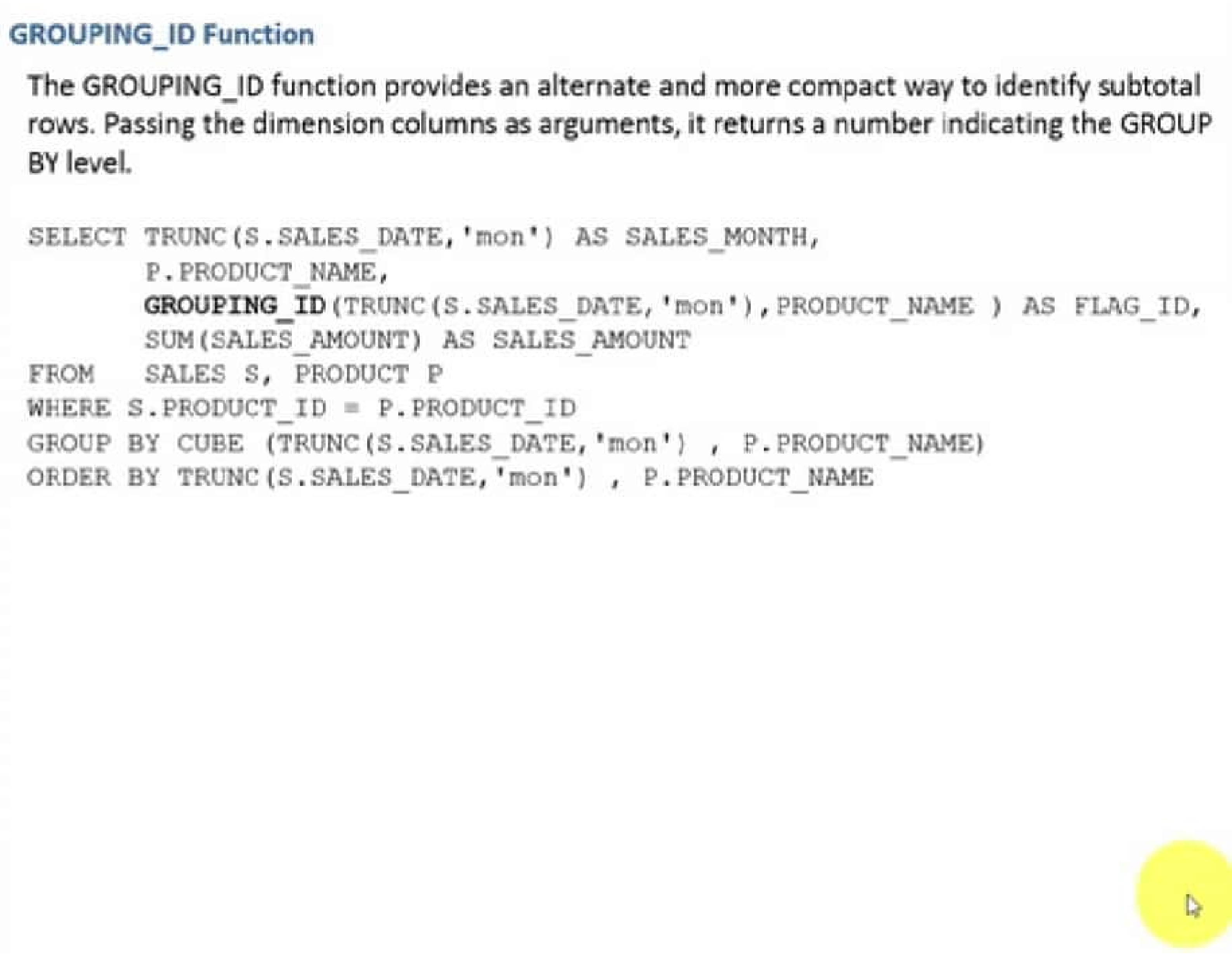
Calculating all possible subtotals in a cube, especially those with many dimensions, can be quite an intensive process. If you don't need all the subtotals, this can represent a considerable amount of wasted effort. The following cube with three dimensions gives 8 levels of subtotals

CUBE (SALES_MONTH, PRODUCT_NAME, CITY)

(SALES_MONTH, PRODUCT_NAME, CITY)
(SALES_MONTH, PRODUCT_NAME)
(SALES_MONTH, CITY)
(SALES_MONTH)
(PRODUCT_NAME, CITY)
(PRODUCT_NAME)
(CITY)
()



2/1/2015		1	25440
3/1/2015	HTC 7800	0	16140
3/1/2015	Microsoft Keyboard 7865	0	16880
3/1/2015	Mobile Cover	0	15320
3/1/2015	iPhone	0	14720
3/1/2015		1	63060



GROUPING_ID Function

The GROUPING_ID function provides an alternate and more compact way to identify subtotal rows. Passing the dimension columns as arguments, it returns a number indicating the GROUP BY level.

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
       P.PRODUCT_NAME,
       GROUPING_ID(TRUNC(S.SALES_DATE,'mon'),PRODUCT_NAME ) AS FLAG_ID,
       SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P
WHERE S.PRODUCT_ID = P.PRODUCT_ID
GROUP BY CUBE (TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME)
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME
```

GROUPING Function

From this we can see:

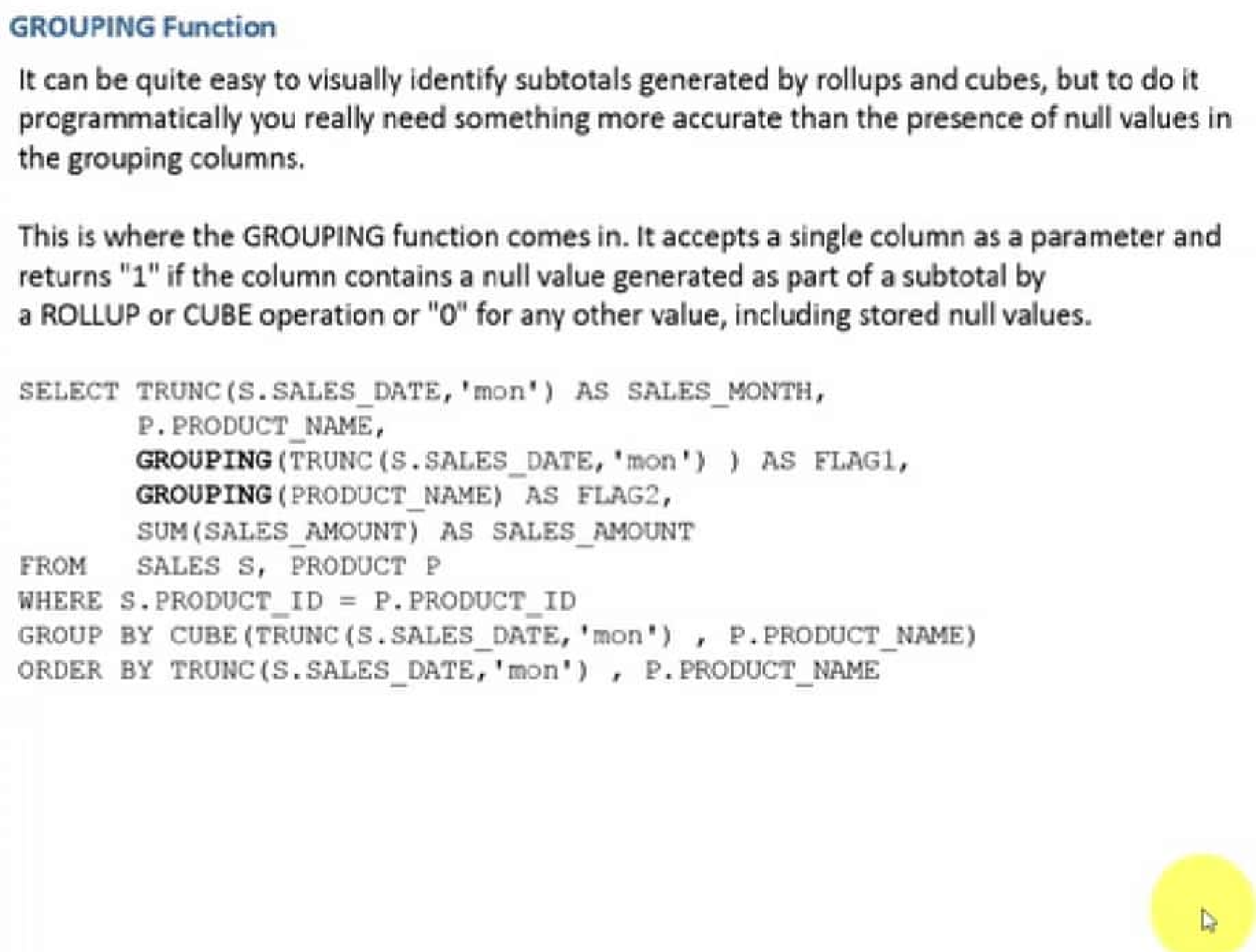
Flag1 = 0 and Flag2 = 0
Represents a row containing regular subtotal we would expect from a GROUP BY operation.

Flag1 = 0 and Flag2 = 1
Represents a row containing a subtotal for a distinct value of the SALES_MONTH column, as generated by ROLLUP and CUBE operations.

Flag1 = 1 and Flag2 = 0
Represents a row containing a subtotal for a distinct value of the PRODUCT_NAME column, which we would only see in a CUBE operation.

Flag1 = 1 and Flag2 = 1
Represents a row containing a grand total for the query, as generated by ROLLUP and CUBE operations.

SALES_MONTH	PRODUCT_NAME	FLAG1	FLAG2	SALES_AMOUNT
1/1/2015	Mobile Cover	0	0	340
1/1/2015	iPhone	0	0	60
1/1/2015		0	1	400
2/1/2015	HTC 7800	0	0	7260
2/1/2015	Microsoft Keyboard 7865	0	0	4660
2/1/2015	Mobile Cover	0	0	4560
2/1/2015	Samsung F7100	0	0	4000
2/1/2015	iPhone	0	0	4960
2/1/2015		0	1	25440
3/1/2015	HTC 7800	0	0	16140
3/1/2015	Microsoft Keyboard 7865	0	0	16880
3/1/2015	Mobile Cover	0	0	15320
3/1/2015	iPhone	0	0	14720
3/1/2015		0	1	63060
4/1/2015	HTC 7800	0	0	800
4/1/2015	Microsoft Keyboard 7865	0	0	1260
4/1/2015	Mobile Cover	0	0	26620
4/1/2015	iPhone	0	0	30480
4/1/2015		0	1	59160
5/1/2015	Mobile Cover	0	0	1140
5/1/2015		0	1	1140
	HTC 7800	1	0	24200
	Microsoft Keyboard 7865	1	0	22800
	Mobile Cover	1	0	47980
	Samsung F7100	1	0	4000
	iPhone	1	0	50220
		1	1	149200

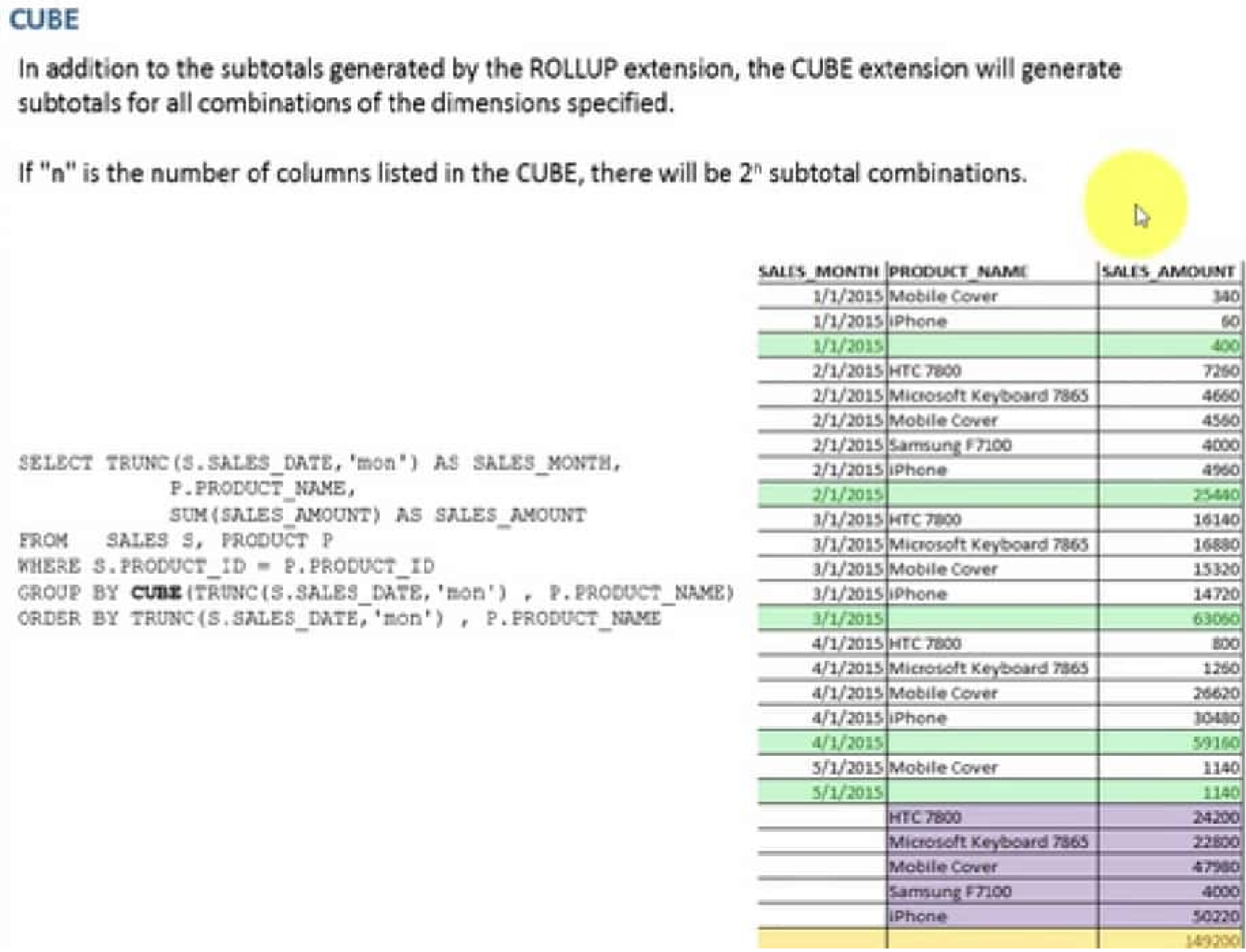


GROUPING Function

It can be quite easy to visually identify subtotals generated by rollups and cubes, but to do it programmatically you really need something more accurate than the presence of null values in the grouping columns.

This is where the GROUPING function comes in. It accepts a single column as a parameter and returns "1" if the column contains a null value generated as part of a subtotal by a ROLLUP or CUBE operation or "0" for any other value, including stored null values.

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
       P.PRODUCT_NAME,
       GROUPING(TRUNC(S.SALES_DATE,'mon') ) AS FLAG1,
       GROUPING(PRODUCT_NAME) AS FLAG2,
       SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P
WHERE S.PRODUCT_ID = P.PRODUCT_ID
GROUP BY CUBE(TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME)
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME
```



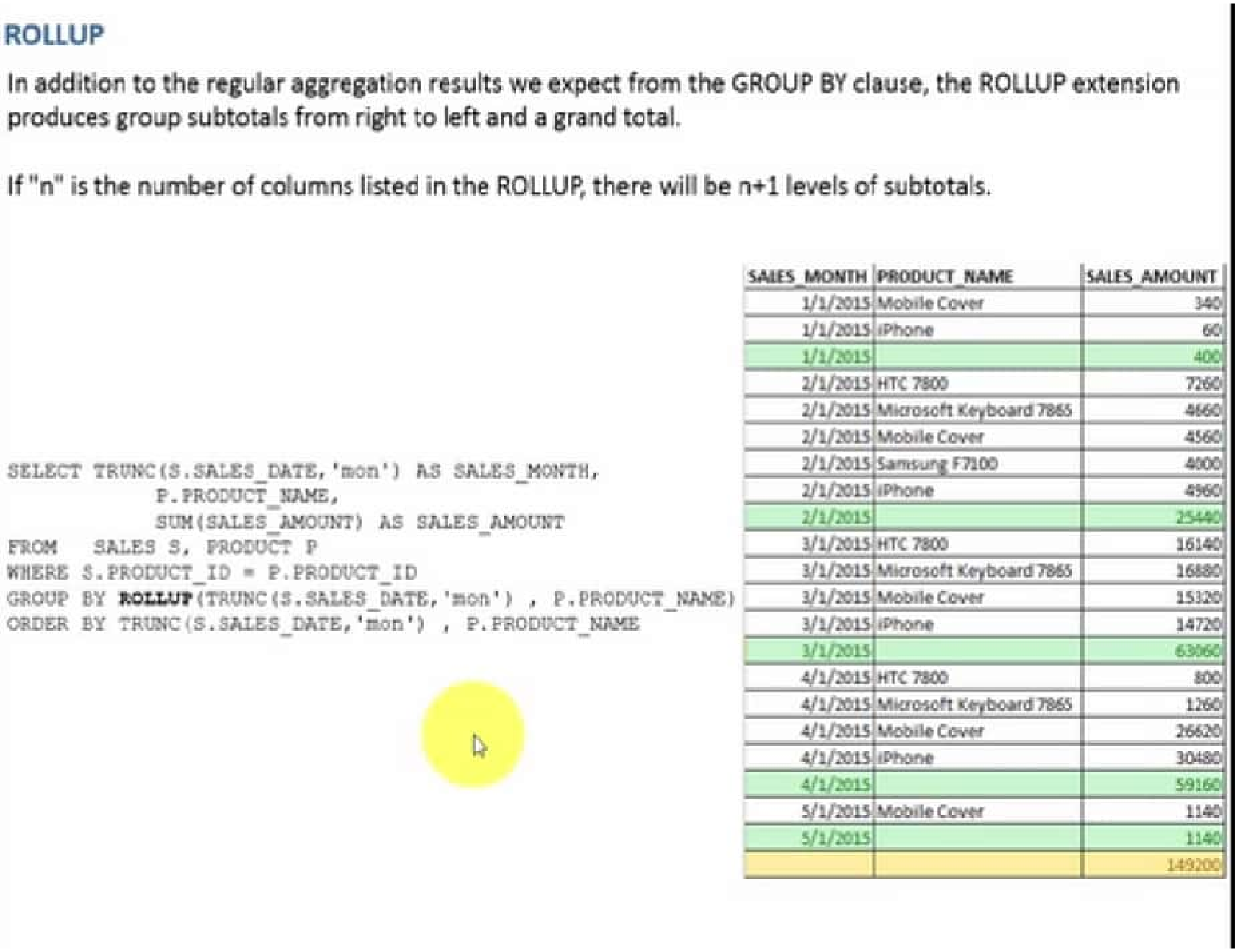
CUBE

In addition to the subtotals generated by the ROLLUP extension, the CUBE extension will generate subtotals for all combinations of the dimensions specified.

If "n" is the number of columns listed in the CUBE, there will be 2^n subtotal combinations.

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
            P.PRODUCT_NAME,
            SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P
WHERE S.PRODUCT_ID = P.PRODUCT_ID
GROUP BY CUBE(TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME)
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME
```

SALES_MONTH	PRODUCT_NAME	SALES_AMOUNT
1/1/2015	Mobile Cover	340
1/1/2015	iPhone	60
1/1/2015		400
2/1/2015	HTC 7800	7260
2/1/2015	Microsoft Keyboard 7865	4660
2/1/2015	Mobile Cover	4560
2/1/2015	Samsung F7100	4000
2/1/2015	iPhone	4960
2/1/2015		25440
3/1/2015	HTC 7800	16140
3/1/2015	Microsoft Keyboard 7865	16880
3/1/2015	Mobile Cover	15320
3/1/2015	iPhone	14720
3/1/2015		63060
4/1/2015	HTC 7800	800
4/1/2015	Microsoft Keyboard 7865	1260
4/1/2015	Mobile Cover	26620
4/1/2015	iPhone	30480
4/1/2015		59160
5/1/2015	Mobile Cover	1140
5/1/2015		1140
	HTC 7800	24200
	Microsoft Keyboard 7865	22800
	Mobile Cover	47980
	Samsung F7100	4000
	iPhone	50220
		149200



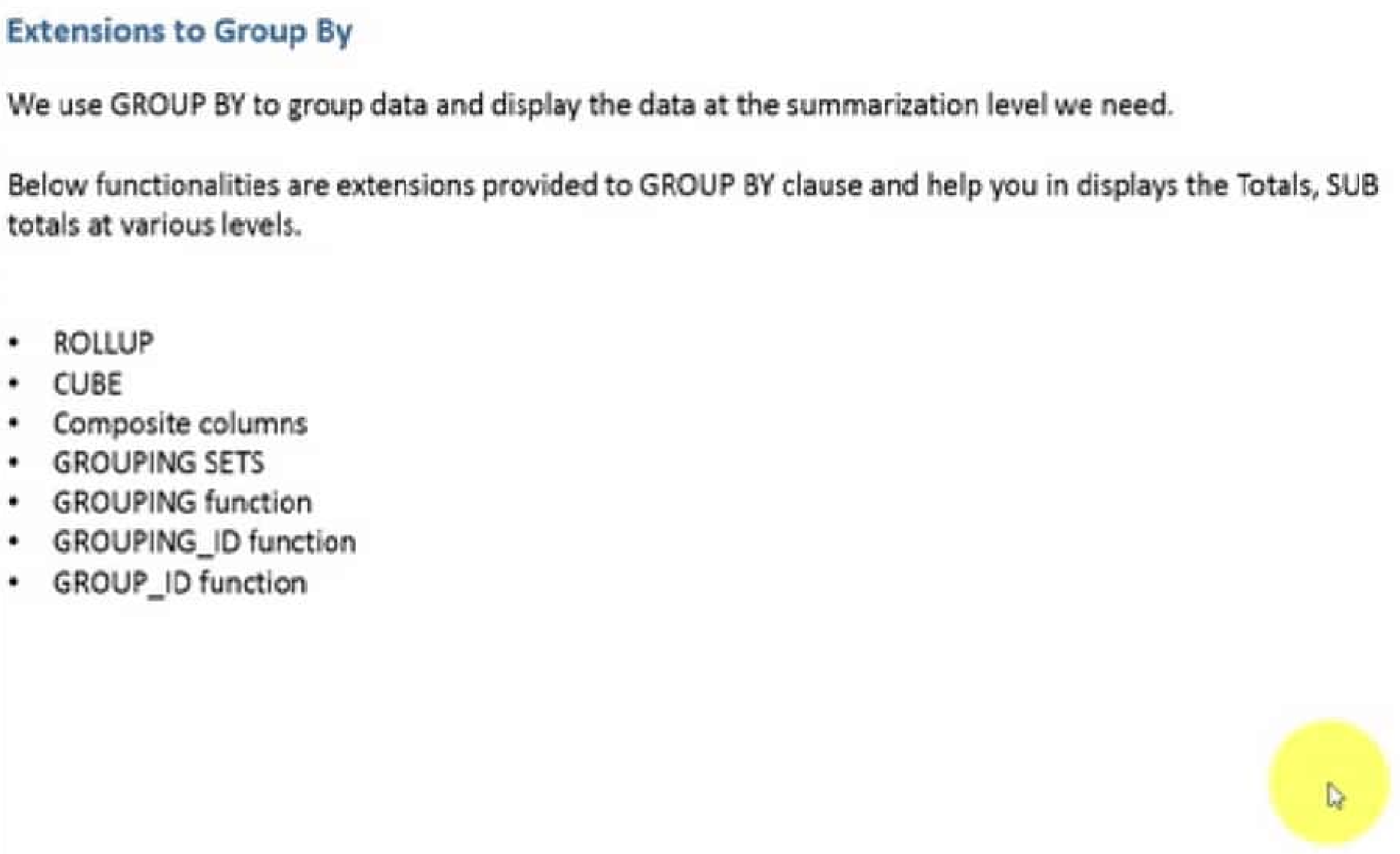
ROLLUP

In addition to the regular aggregation results we expect from the GROUP BY clause, the ROLLUP extension produces group subtotals from right to left and a grand total.

If "n" is the number of columns listed in the ROLLUP, there will be n+1 levels of subtotals.

```
SELECT TRUNC(S.SALES_DATE,'mon') AS SALES_MONTH,
            P.PRODUCT_NAME,
            SUM(SALES_AMOUNT) AS SALES_AMOUNT
FROM   SALES S, PRODUCT P
WHERE S.PRODUCT_ID = P.PRODUCT_ID
GROUP BY ROLLUP(TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME)
ORDER BY TRUNC(S.SALES_DATE,'mon') , P.PRODUCT_NAME
```

SALES_MONTH	PRODUCT_NAME	SALES_AMOUNT
1/1/2015	Mobile Cover	340
1/1/2015	iPhone	60
1/1/2015		400
2/1/2015	HTC 7800	7260
2/1/2015	Microsoft Keyboard 7865	4660
2/1/2015	Mobile Cover	4560
2/1/2015	Samsung F7100	4000
2/1/2015	iPhone	4960
2/1/2015		25440
3/1/2015	HTC 7800	16140
3/1/2015	Microsoft Keyboard 7865	16880
3/1/2015	Mobile Cover	15320
3/1/2015	iPhone	14720
3/1/2015		63060
4/1/2015	HTC 7800	800
4/1/2015	Microsoft Keyboard 7865	1260
4/1/2015	Mobile Cover	26620
4/1/2015	iPhone	30480
4/1/2015		59160
5/1/2015	Mobile Cover	1140
5/1/2015		1140
		149200

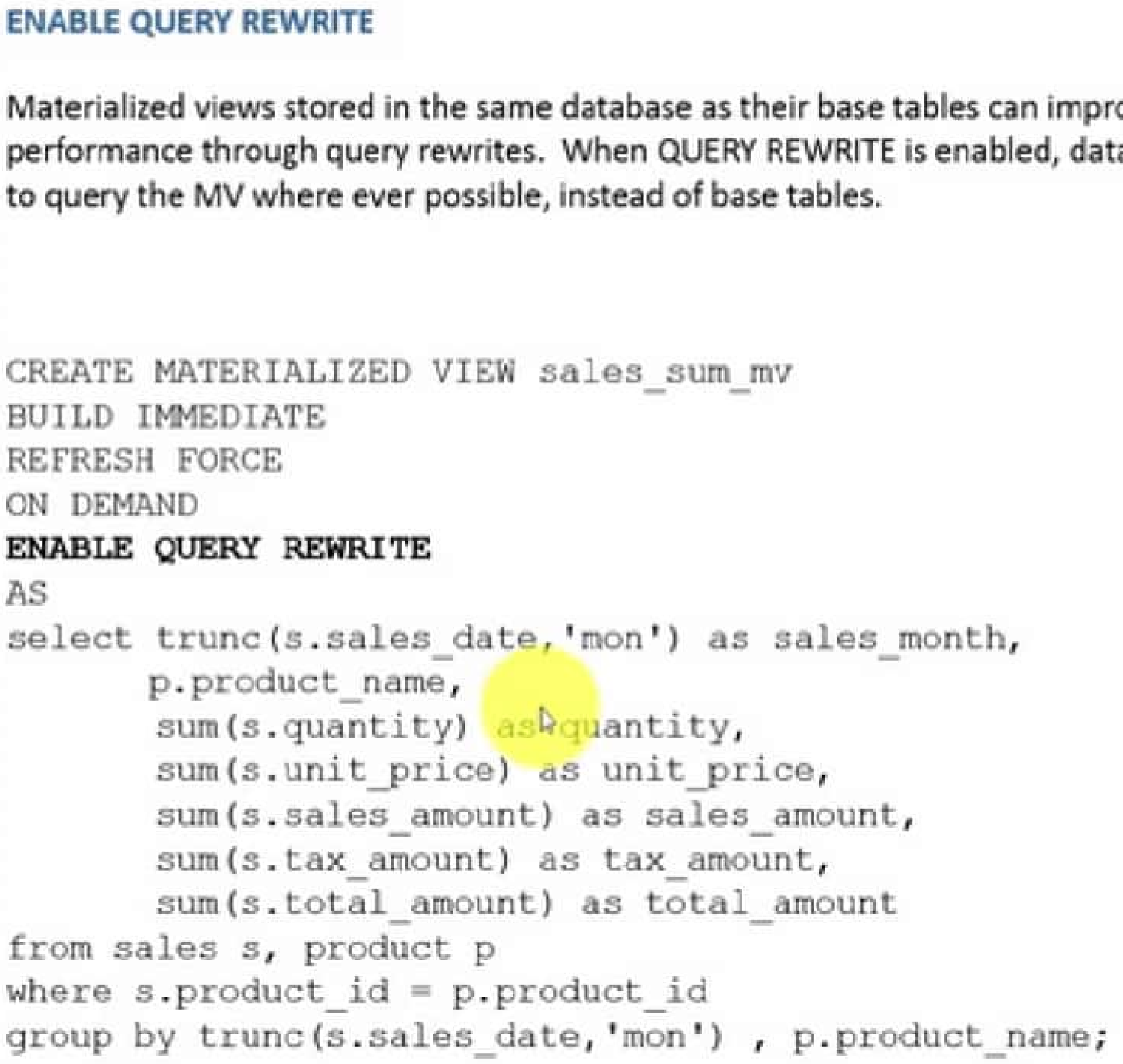


Extensions to Group By

We use GROUP BY to group data and display the data at the summarization level we need.

Below functionalities are extensions provided to GROUP BY clause and help you in displays the Totals, SUB totals at various levels.

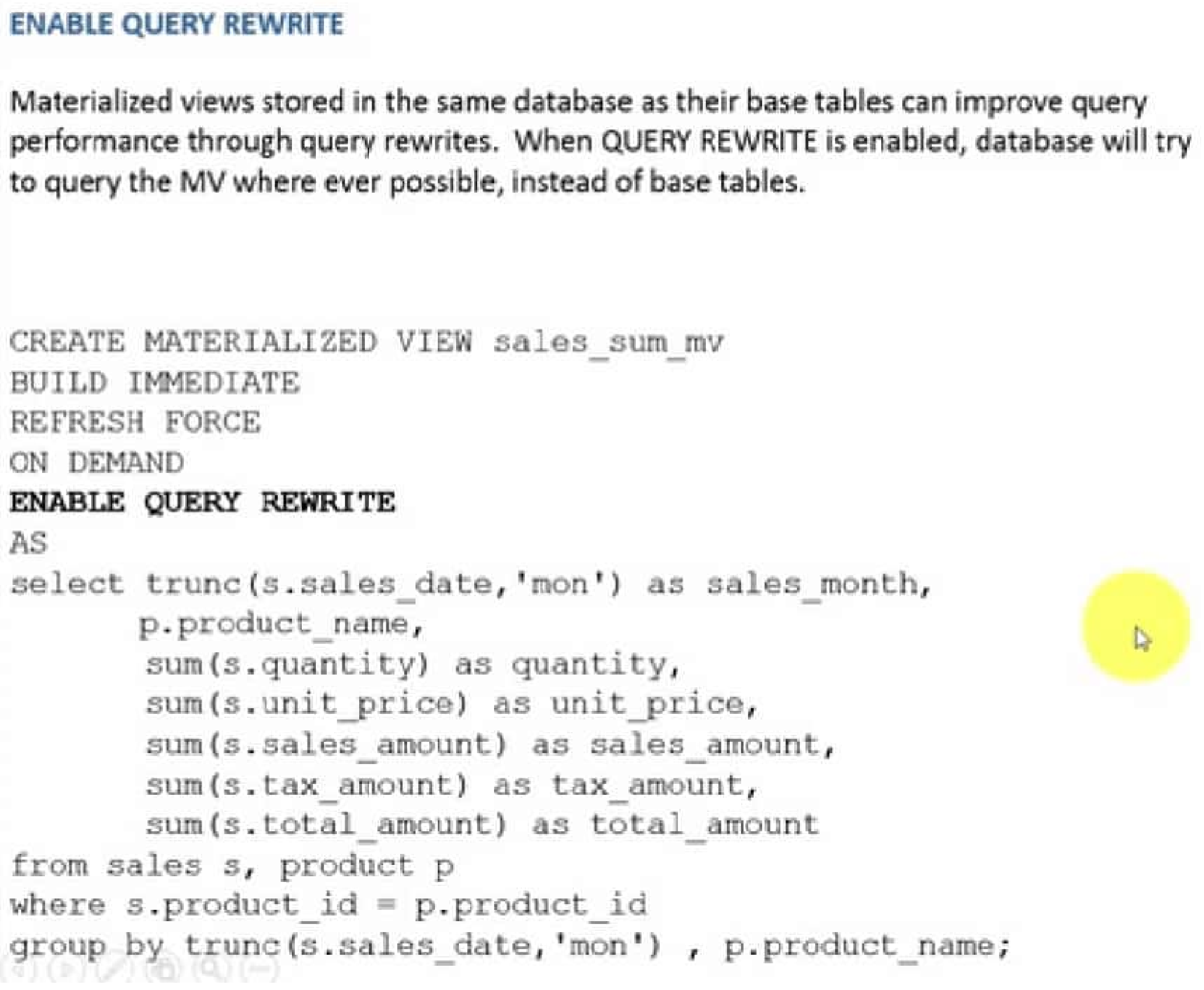
- ROLLUP
- CUBE
- Composite columns
- GROUPING SETS
- GROUPING function
- GROUPING_ID function
- GROUP_ID function



ENABLE QUERY REWRITE

Materialized views stored in the same database as their base tables can improve query performance through query rewrites. When QUERY REWRITE is enabled, database will try to query the MV where ever possible, instead of base tables.

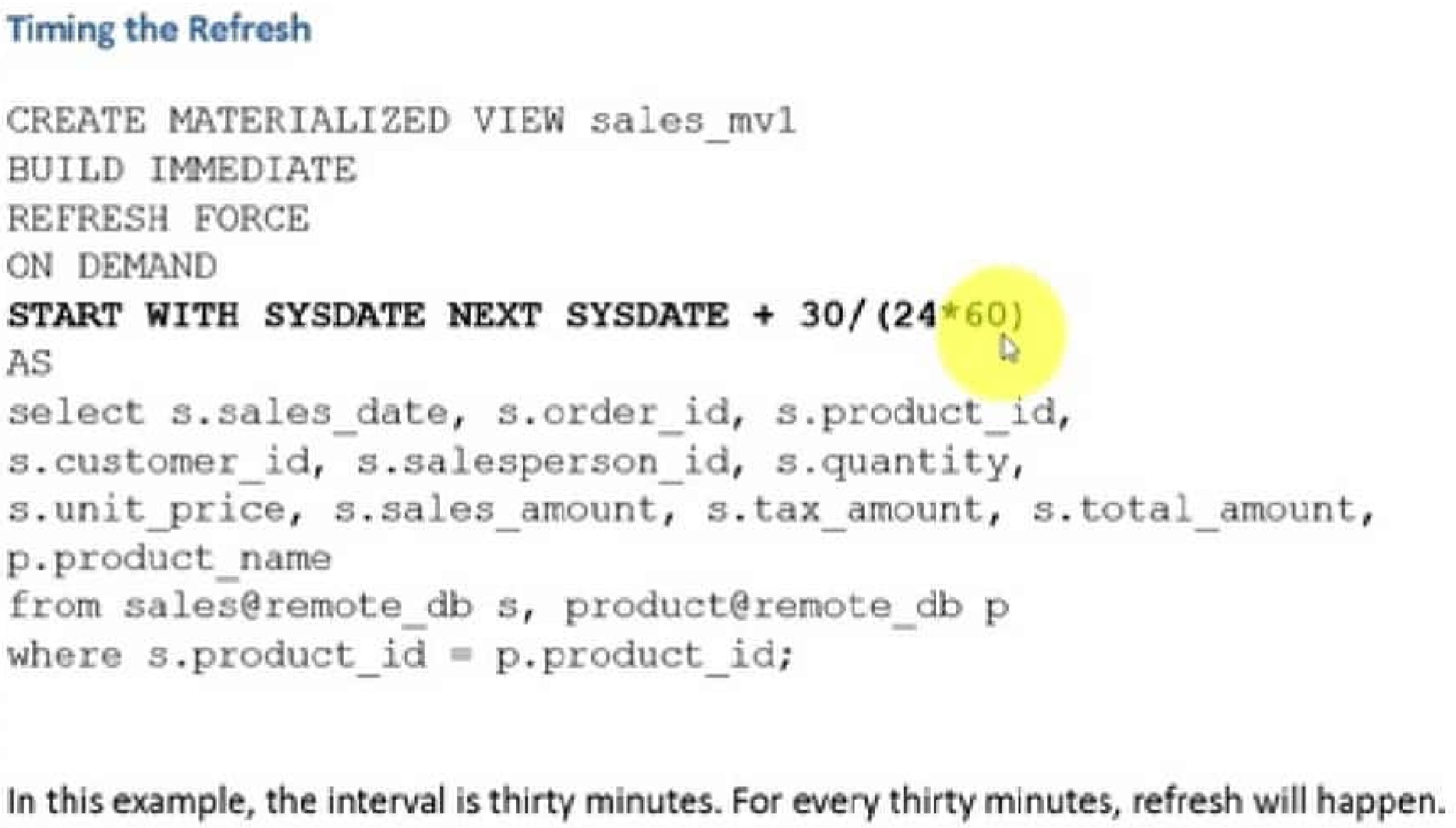
```
CREATE MATERIALIZED VIEW sales_sum_mv
BUILD IMMEDIATE
REFRESH FORCE
ON DEMAND
ENABLE QUERY REWRITE
AS
select trunc(s.sales_date,'mon') as sales_month,
      p.product_name,
       sum(s.quantity) as quantity,
       sum(s.unit_price) as unit_price,
       sum(s.sales_amount) as sales_amount,
       sum(s.tax_amount) as tax_amount,
       sum(s.total_amount) as total_amount
from sales s, product p
where s.product_id = p.product_id
group by trunc(s.sales_date,'mon') , p.product_name;
```



ENABLE QUERY REWRITE

Materialized views stored in the same database as their base tables can improve query performance through query rewrites. When QUERY REWRITE is enabled, database will try to query the MV where ever possible, instead of base tables.

```
CREATE MATERIALIZED VIEW sales_sum_mv
BUILD IMMEDIATE
REFRESH FORCE
ON DEMAND
ENABLE QUERY REWRITE
AS
select trunc(s.sales_date,'mon') as sales_month,
       p.product_name,
       sum(s.quantity) as quantity,
       sum(s.unit_price) as unit_price,
       sum(s.sales_amount) as sales_amount,
       sum(s.tax_amount) as tax_amount,
       sum(s.total_amount) as total_amount
from sales s, product p
where s.product_id = p.product_id
group by trunc(s.sales_date,'mon') , p.product_name;
```



Timing the Refresh

```
CREATE MATERIALIZED VIEW sales_mv1
BUILD IMMEDIATE
REFRESH FORCE
ON DEMAND
START WITH SYSDATE NEXT SYSDATE + 30/(24*60)
AS
select s.sales_date, s.order_id, s.product_id,
s.customer_id, s.salesperson_id, s.quantity,
s.unit_price, s.sales_amount, s.tax_amount, s.total_amount,
p.product_name
from sales@remote_db s, product@remote_db p
where s.product_id = p.product_id;
```

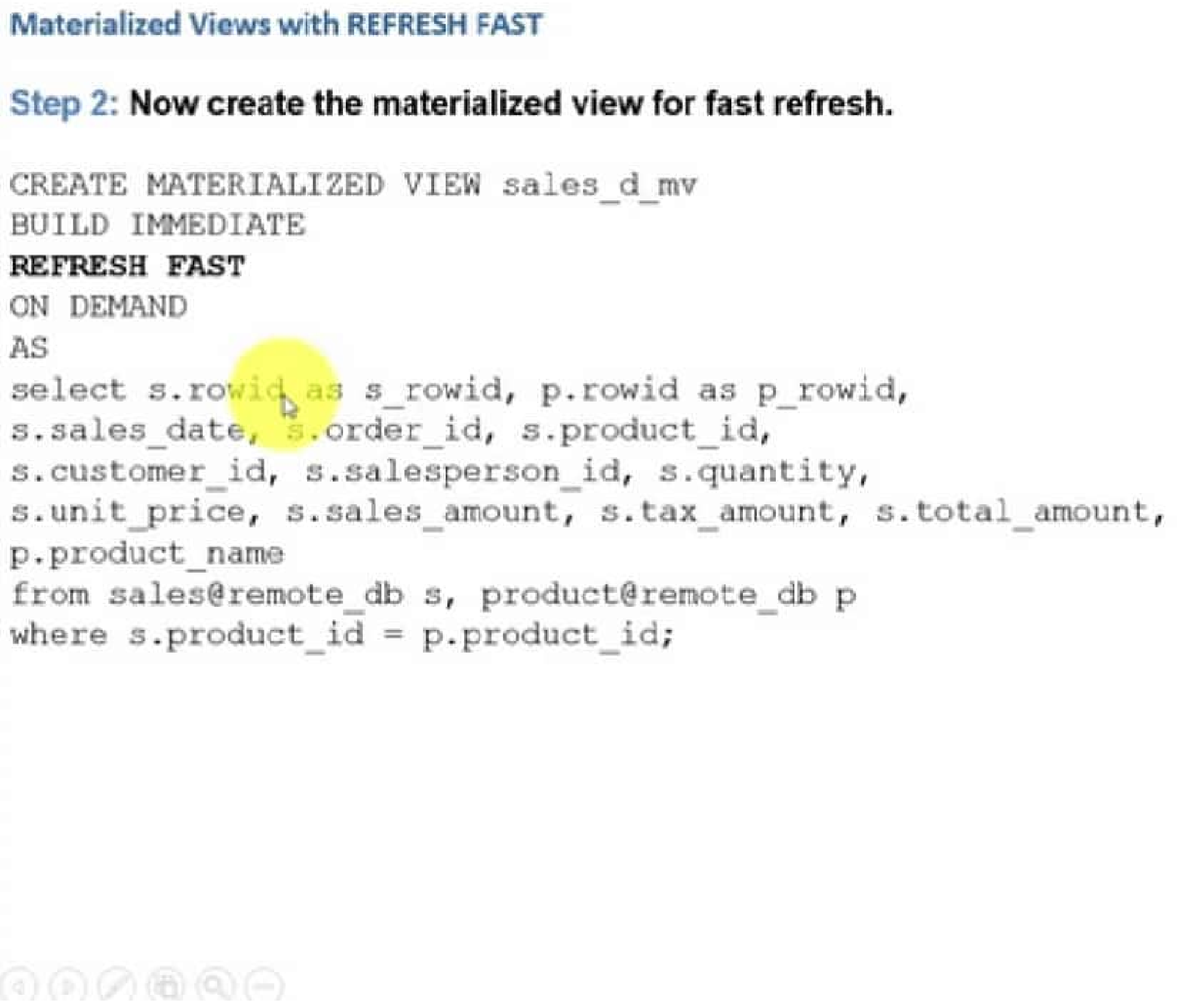
In this example, the interval is thirty minutes. For every thirty minutes, refresh will happen.

Timing the Refresh

The START WITH clause tells the database when to perform the first replication from the master table to the local base table. The NEXT clause specifies the interval between refreshes.

```
CREATE MATERIALIZED VIEW sales_mv1
BUILD IMMEDIATE
REFRESH FORCE
ON DEMAND
START WITH SYSDATE NEXT SYSDATE + 7
AS
select s.sales_date, s.order_id, s.product_id,
s.customer_id, s.salesperson_id, s.quantity,
s.unit_price, s.sales_amount, s.tax_amount, s.total_amount,
p.product_name
from sales@remote_db s, product@remote_db p
where s.product_id = p.product_id;
```

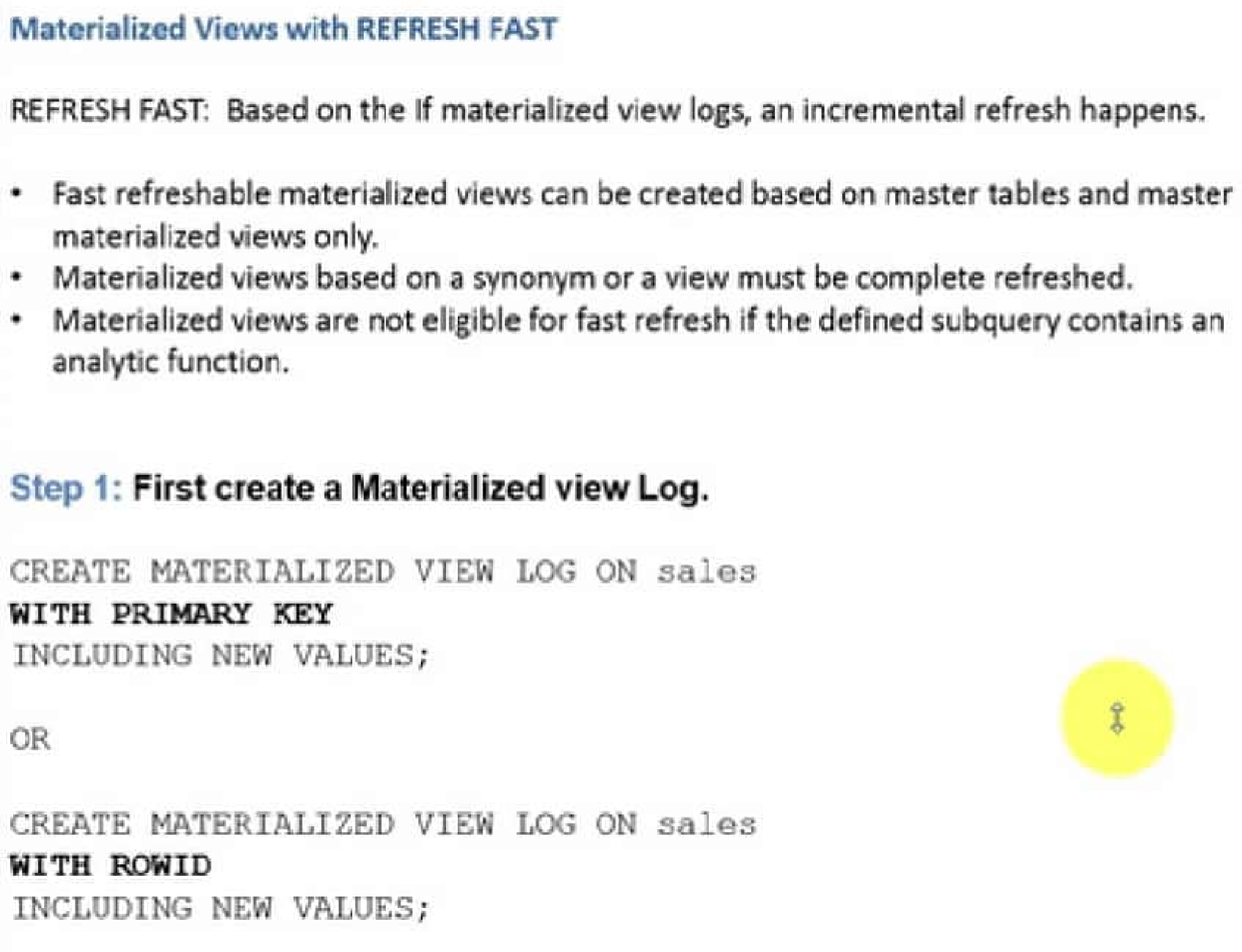
In the above example, the first copy of the materialized view is made at SYSDATE (immediately) and the interval at which the refresh has to be performed is every seven days.



Materialized Views with REFRESH FAST

Step 2: Now create the materialized view for fast refresh.

```
CREATE MATERIALIZED VIEW sales_d_mv
BUILD IMMEDIATE
REFRESH FAST
ON DEMAND
AS
select s.rowid as s_rowid, p.rowid as p_rowid,
s.sales_date, s.order_id, s.product_id,
s.customer_id, s.salesperson_id, s.quantity,
s.unit_price, s.sales_amount, s.tax_amount, s.total_amount,
p.product_name
from sales@remote_db s, product@remote_db p
where s.product_id = p.product_id;
```



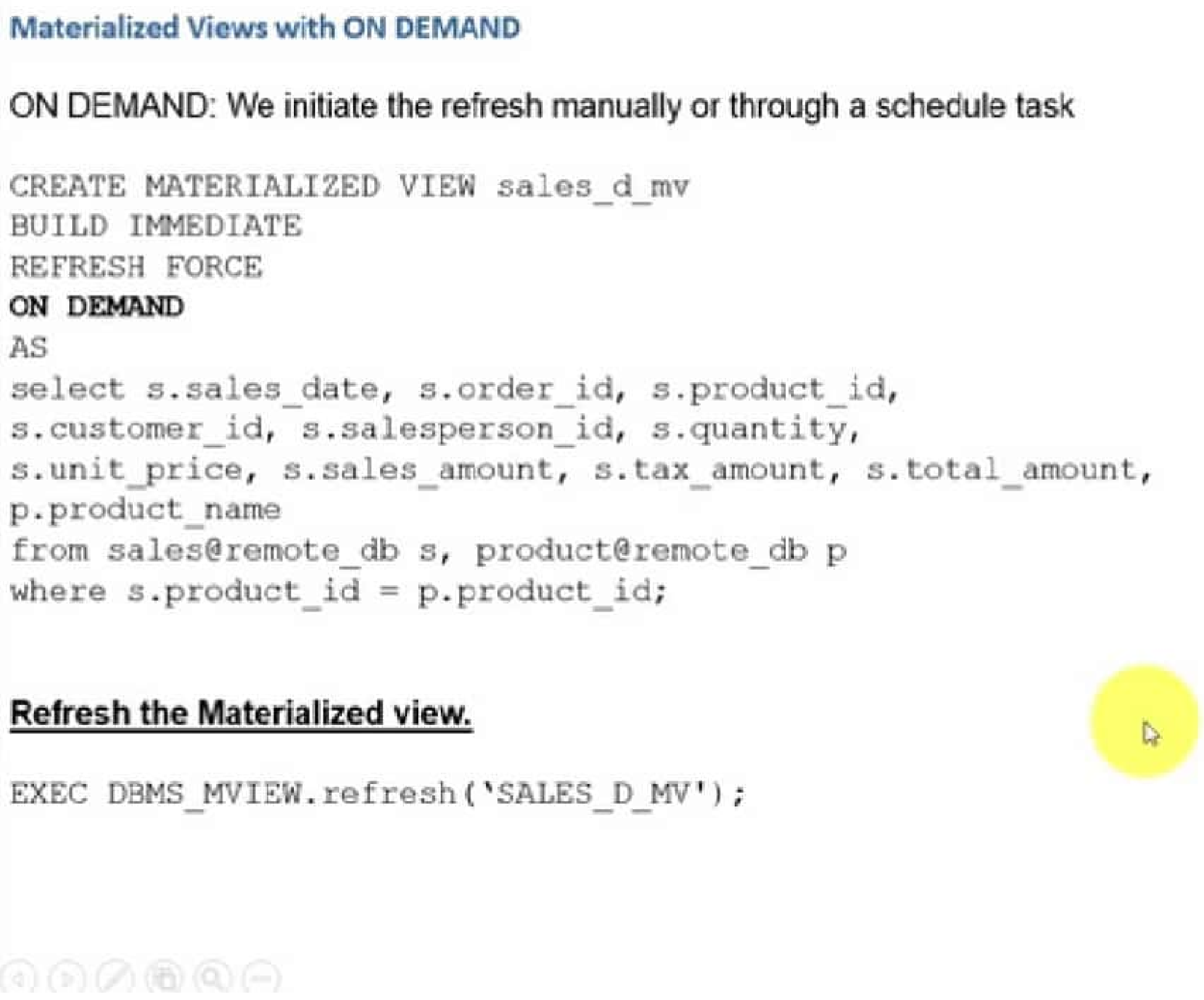
Materialized Views with REFRESH FAST

REFRESH FAST: Based on the If materialized view logs, an incremental refresh happens.

- Fast refreshable materialized views can be created based on master tables and master materialized views only.
- Materialized views based on a synonym or a view must be complete refreshed.
- Materialized views are not eligible for fast refresh if the defined subquery contains an analytic function.

Step 1: First create a Materialized view Log.

```
CREATE MATERIALIZED VIEW LOG ON sales
WITH PRIMARY KEY
INCLUDING NEW VALUES;

OR

CREATE MATERIALIZED VIEW LOG ON sales
WITH ROWID
INCLUDING NEW VALUES;
```



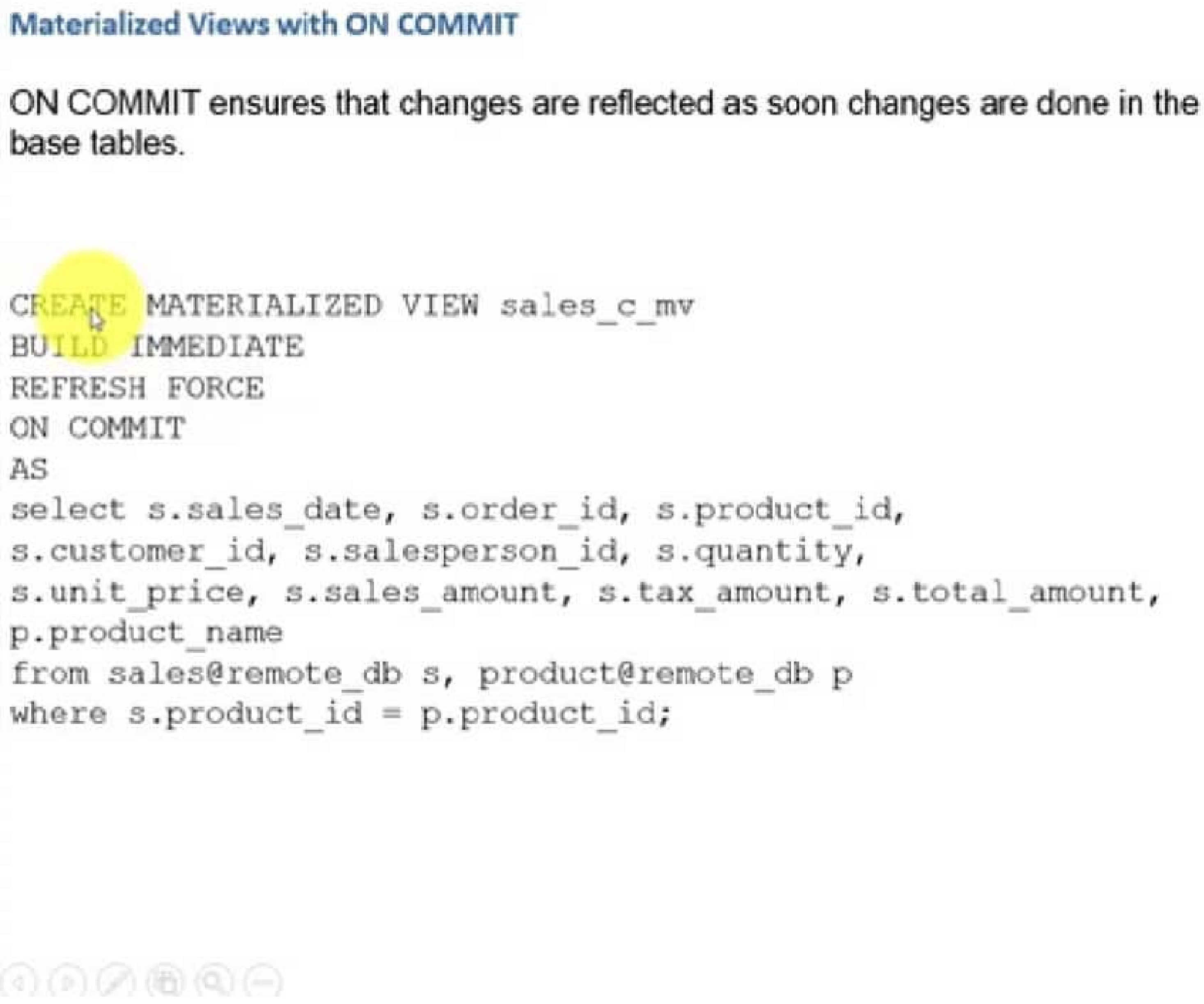
Materialized Views with ON DEMAND

ON DEMAND: We initiate the refresh manually or through a schedule task

```
CREATE MATERIALIZED VIEW sales_d_mv
BUILD IMMEDIATE
REFRESH FORCE
ON DEMAND
AS
select s.sales_date, s.order_id, s.product_id,
s.customer_id, s.salesperson_id, s.quantity,
s.unit_price, s.sales_amount, s.tax_amount, s.total_amount,
p.product_name
from sales@remote_db s, product@remote_db p
where s.product_id = p.product_id;
```

Refresh the Materialized view.

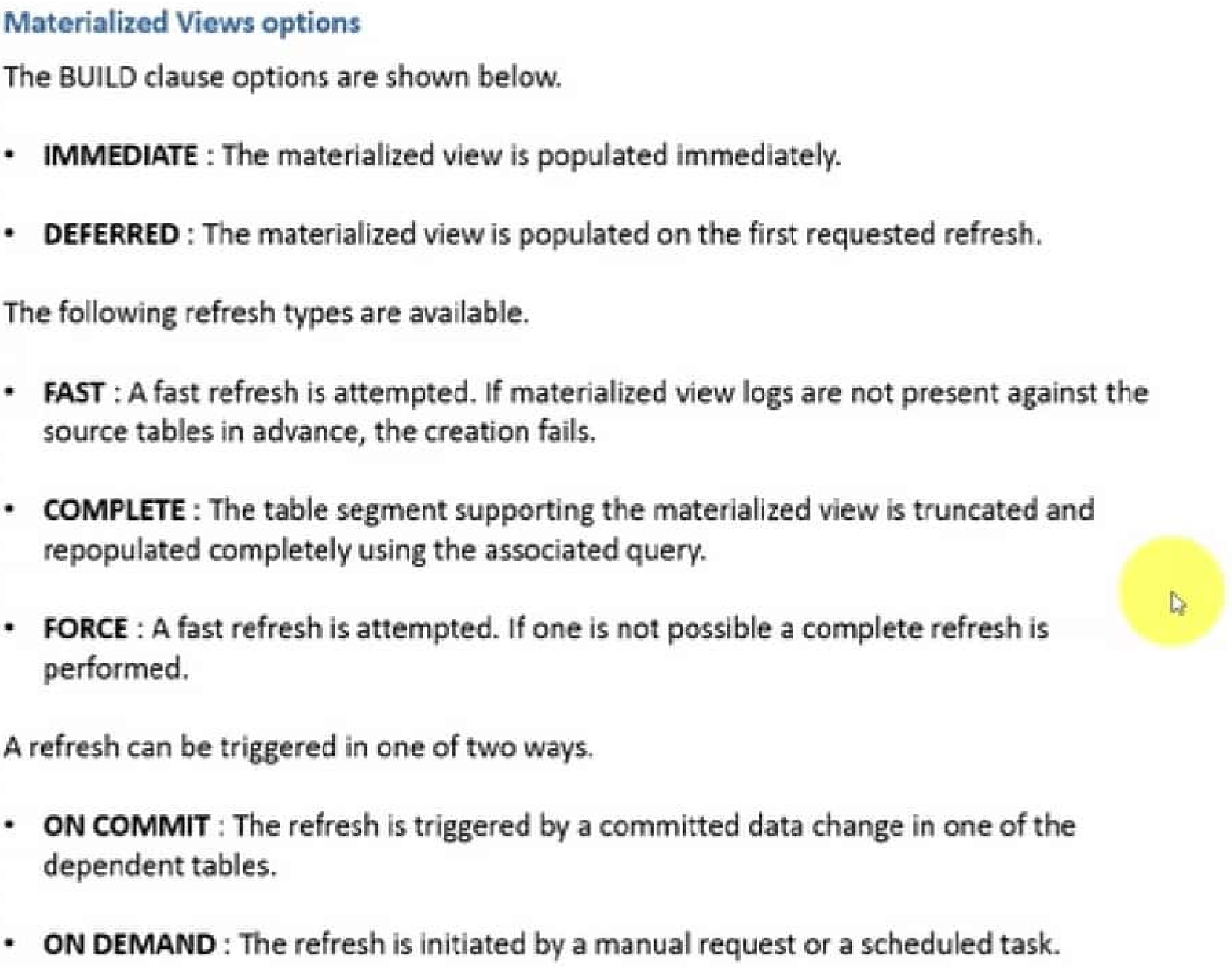
```
EXEC DBMS_MVIEW.refresh('SALES_D_MV');
```



Materialized Views with ON COMMIT

ON COMMIT ensures that changes are reflected as soon changes are done in the base tables.

```
CREATE MATERIALIZED VIEW sales_c_mv
BUILD IMMEDIATE
REFRESH FORCE
ON COMMIT
AS
select s.sales_date, s.order_id, s.product_id,
s.customer_id, s.salesperson_id, s.quantity,
s.unit_price, s.sales_amount, s.tax_amount, s.total_amount,
p.product_name
from sales@remote_db s, product@remote_db p
where s.product_id = p.product_id;
```



Materialized Views options

The BUILD clause options are shown below.

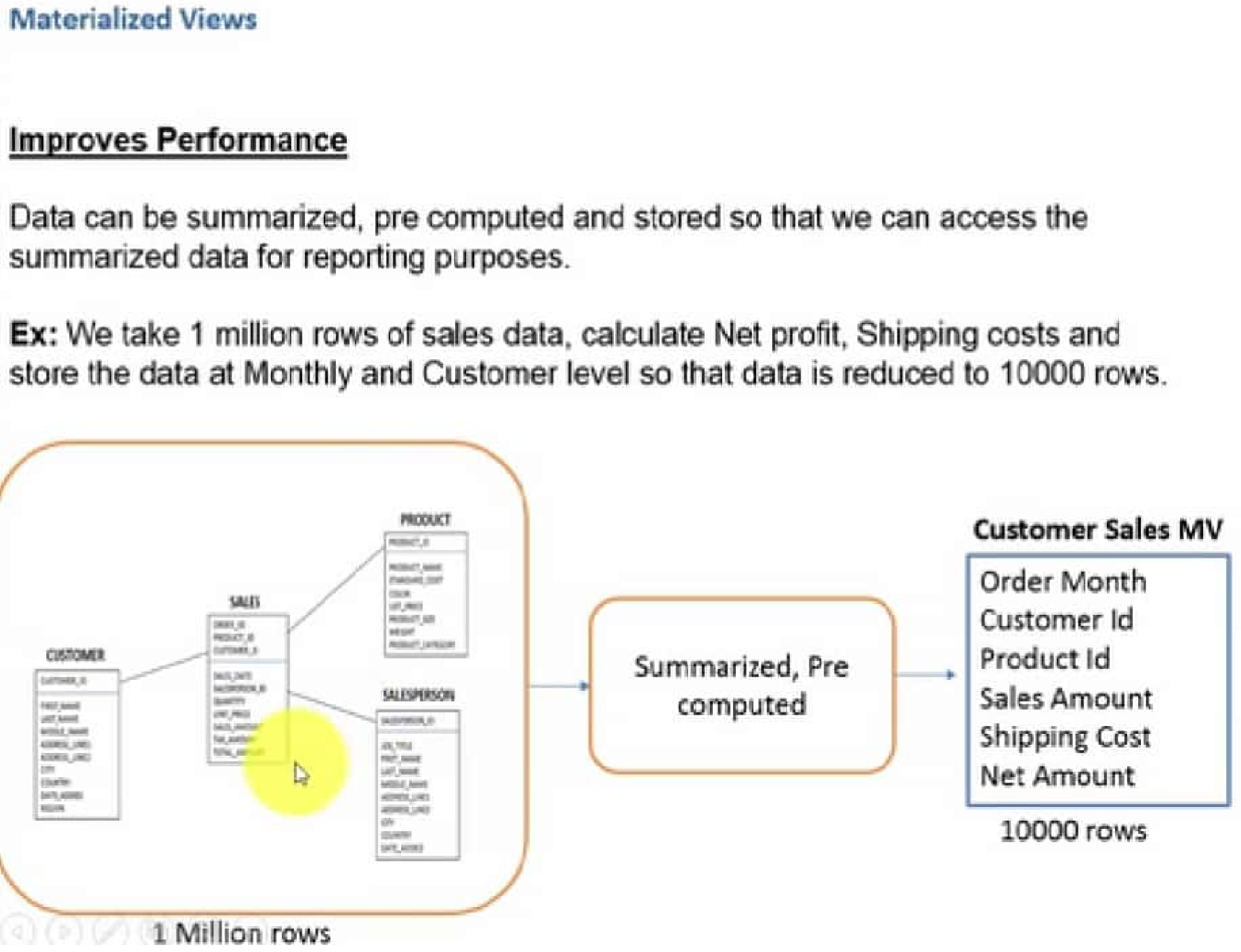
- **IMMEDIATE** : The materialized view is populated immediately.
- **DEFERRED** : The materialized view is populated on the first requested refresh.

The following refresh types are available.

- **FAST** : A fast refresh is attempted. If materialized view logs are not present against the source tables in advance, the creation fails.
- **COMPLETE** : The table segment supporting the materialized view is truncated and repopulated completely using the associated query.
- **FORCE** : A fast refresh is attempted. If one is not possible a complete refresh is performed.

A refresh can be triggered in one of two ways.

- **ON COMMIT** : The refresh is triggered by a committed data change in one of the dependent tables.
- **ON DEMAND** : The refresh is initiated by a manual request or a scheduled task.



Materialized Views

Improves Performance

Data can be summarized, pre computed and stored so that we can access the summarized data for reporting purposes.

Ex: We take 1 million rows of sales data, calculate Net profit, Shipping costs and store the data at Monthly and Customer level so that data is reduced to 10000 rows.

Customer Sales MV

Order Month
Customer Id
Product Id
Sales Amount
Shipping Cost
Net Amount

10000 rows

Materialized Views

A materialized view is a table segment whose contents are periodically refreshed based on a query, either against a local or remote table.

Data can be summarized, pre computed and distributed...

In early releases, they were knows as Snapshots.

- Replication of data between sites.

- Improves Performance