SQL Tutorial

SQL

Agenda

1 Corelated Subquery

2 Grouping Sets

3 ROLLUP

4 CUBE

Agenda

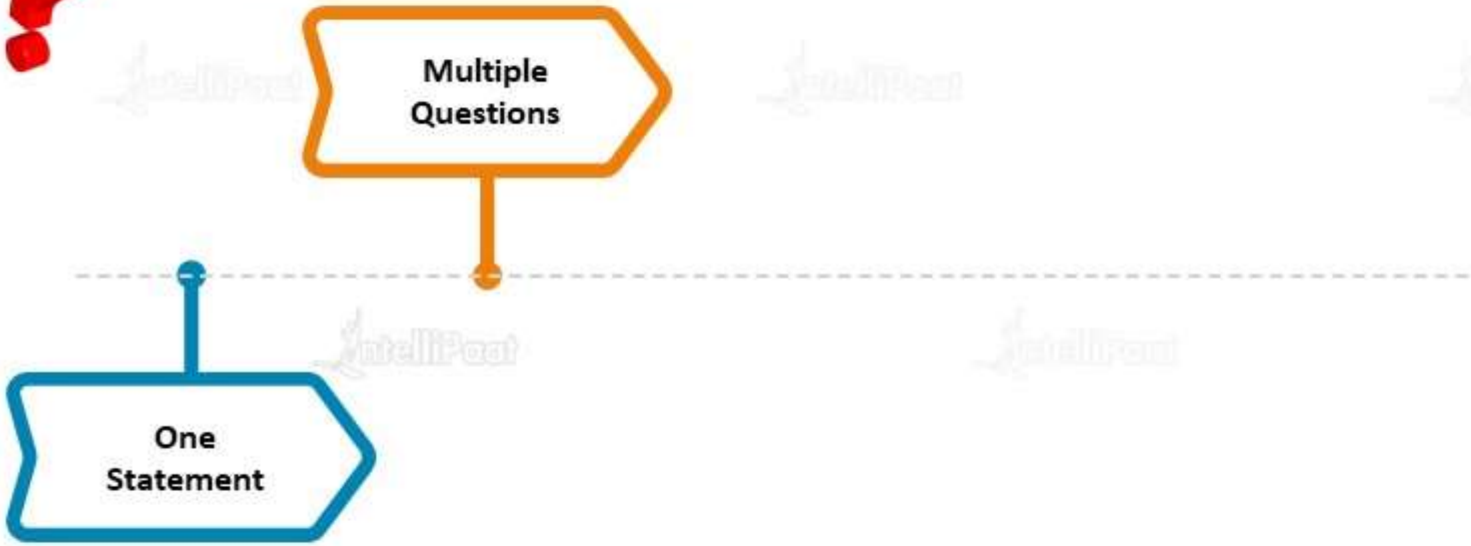
1 Corelated Subquery

Purpose of Corelated Subqueries

Sometimes you have to answer more than one question in one sentence

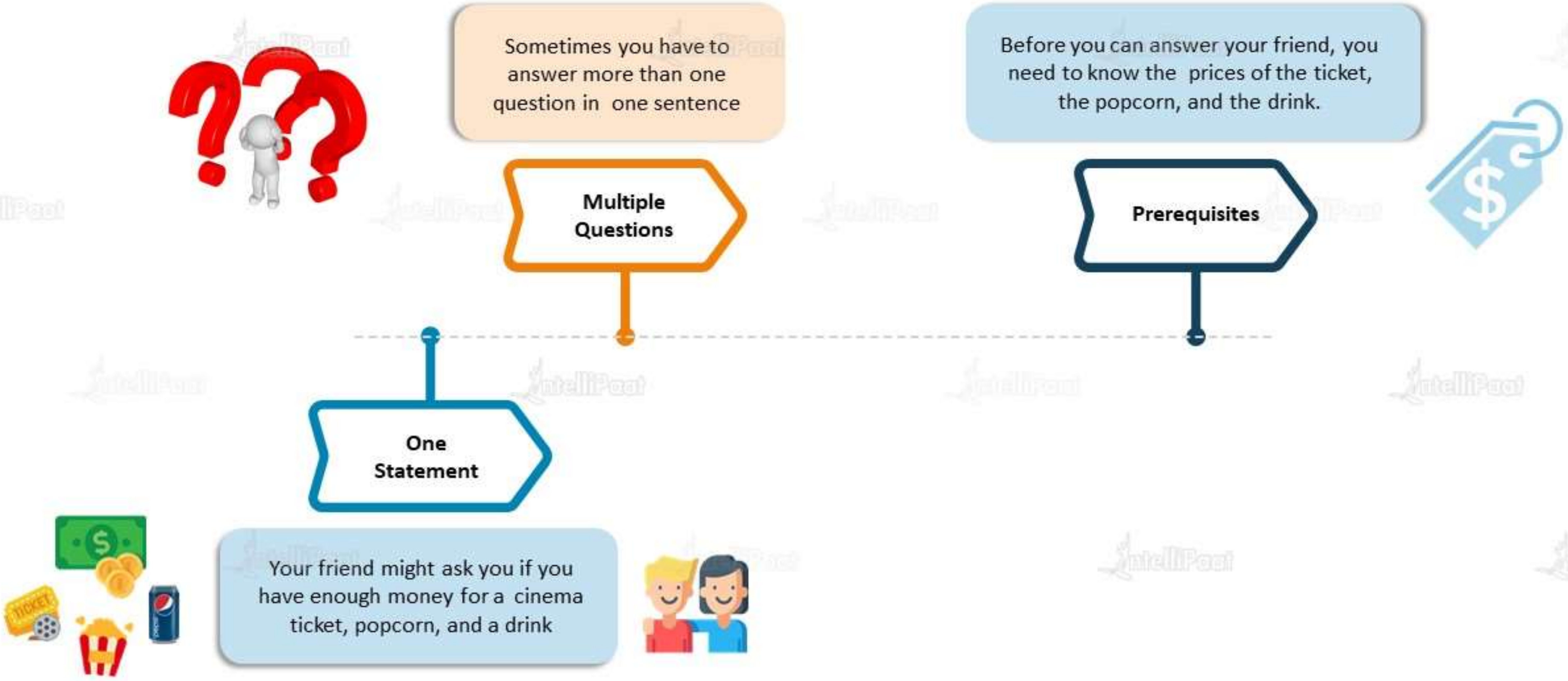
Purpose of Corelated Subqueries

Sometimes you have to answer more than one question in one sentence



Your friend might ask you if you have enough money for a cinema ticket, popcorn, and a drink

Purpose of Corelated Subqueries



One Statement

Your friend might ask you if you have enough money for a cinema ticket, popcorn, and a drink

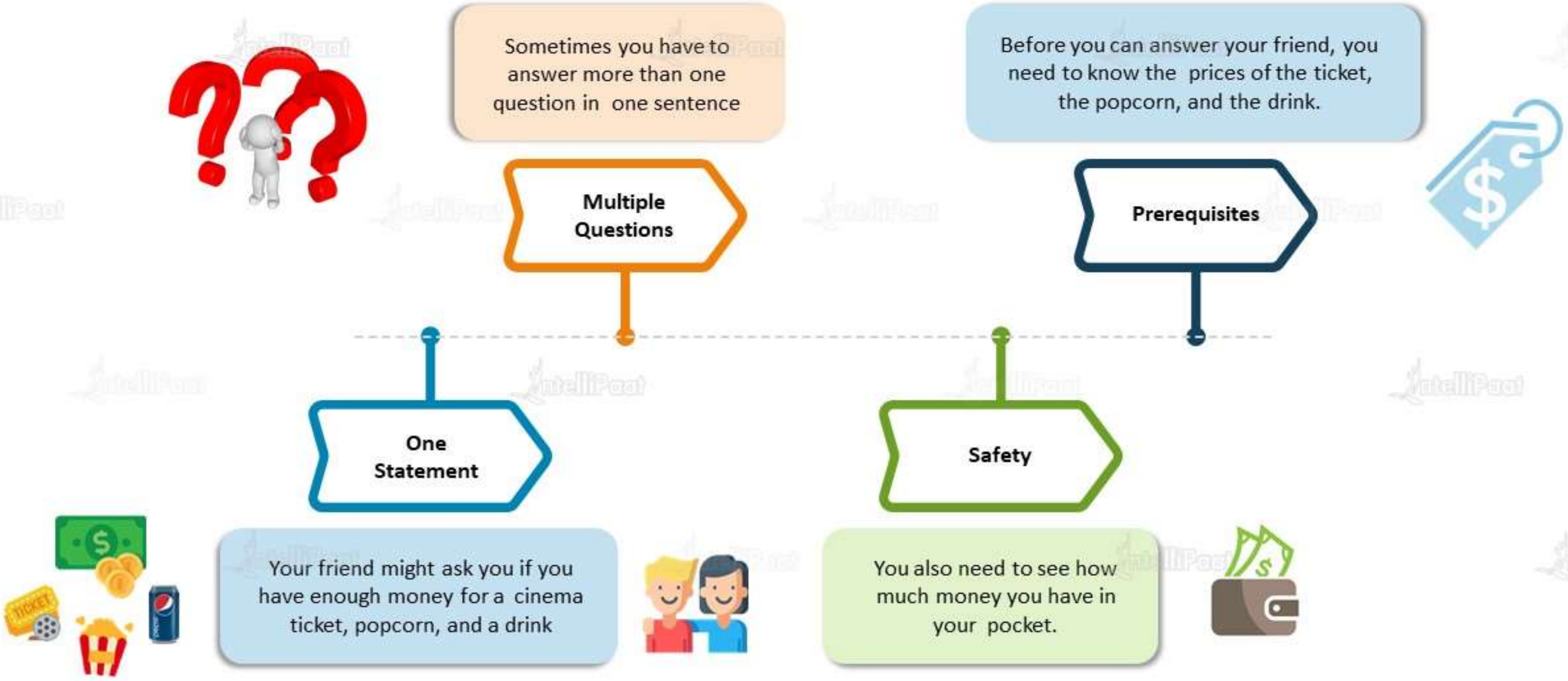
Multiple Questions

Sometimes you have to answer more than one question in one sentence

Prerequisites

Before you can answer your friend, you need to know the prices of the ticket, the popcorn, and the drink.

Purpose of Corelated Subqueries



One Statement

Your friend might ask you if you have enough money for a cinema ticket, popcorn, and a drink

Multiple Questions

Sometimes you have to answer more than one question in one sentence

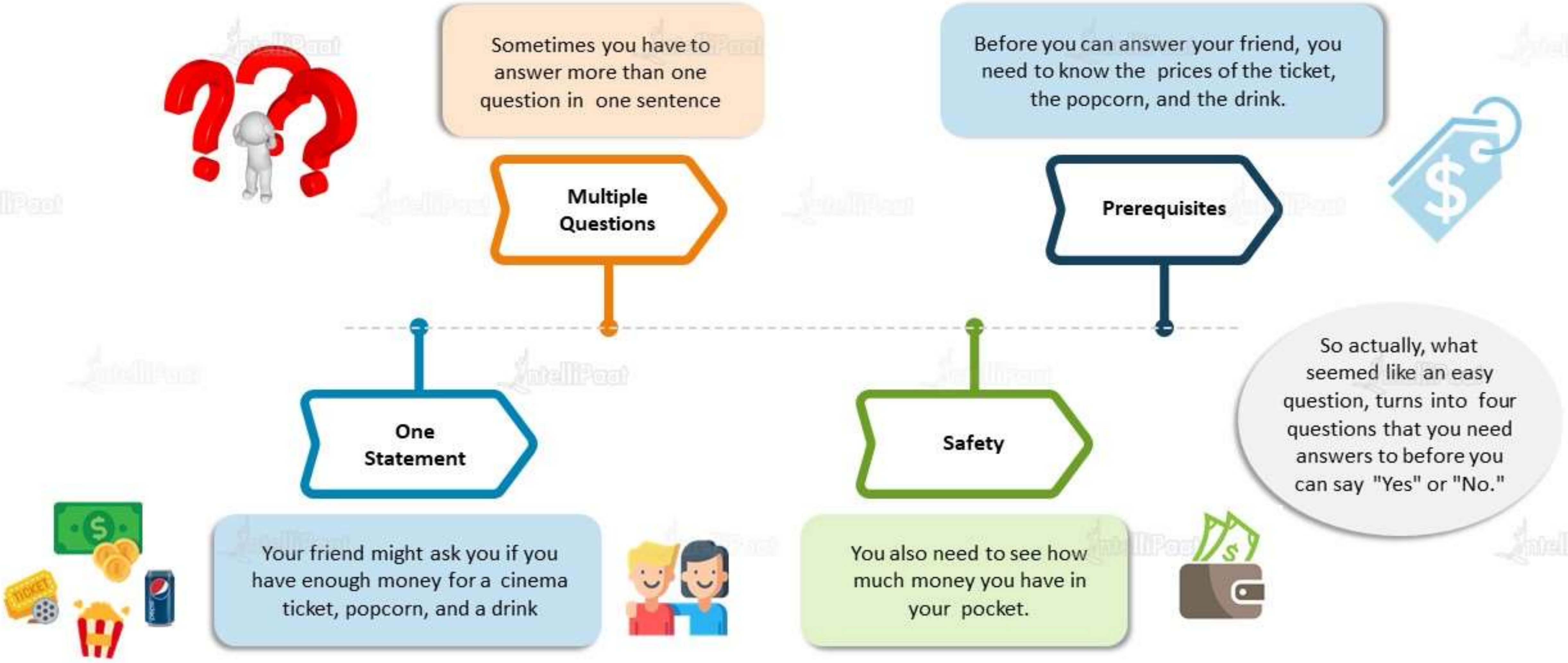
Safety

You also need to see how much money you have in your pocket.

Prerequisites

Before you can answer your friend, you need to know the prices of the ticket, the popcorn, and the drink.

Purpose of Corelated Subqueries



One Statement

Your friend might ask you if you have enough money for a cinema ticket, popcorn, and a drink

Multiple Questions

Sometimes you have to answer more than one question in one sentence

Safety

You also need to see how much money you have in your pocket.

Prerequisites

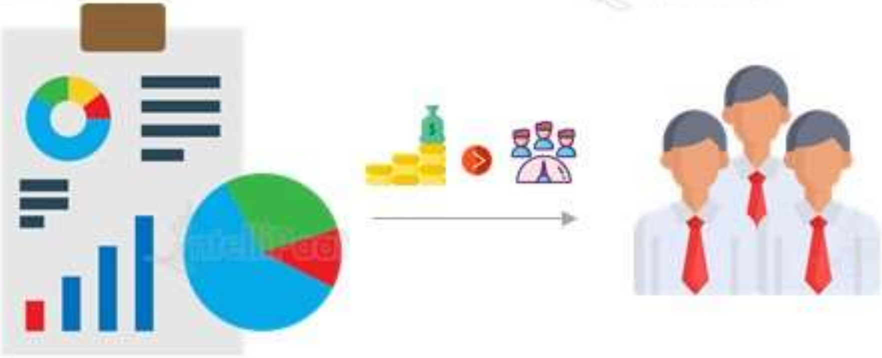
Before you can answer your friend, you need to know the prices of the ticket, the popcorn, and the drink.

So actually, what seemed like an easy question, turns into four questions that you need answers to before you can say "Yes" or "No."

Purpose of Corelated Subqueries

In business, you might get asked to produce a report of all employees earning more than the average salary for their departments.

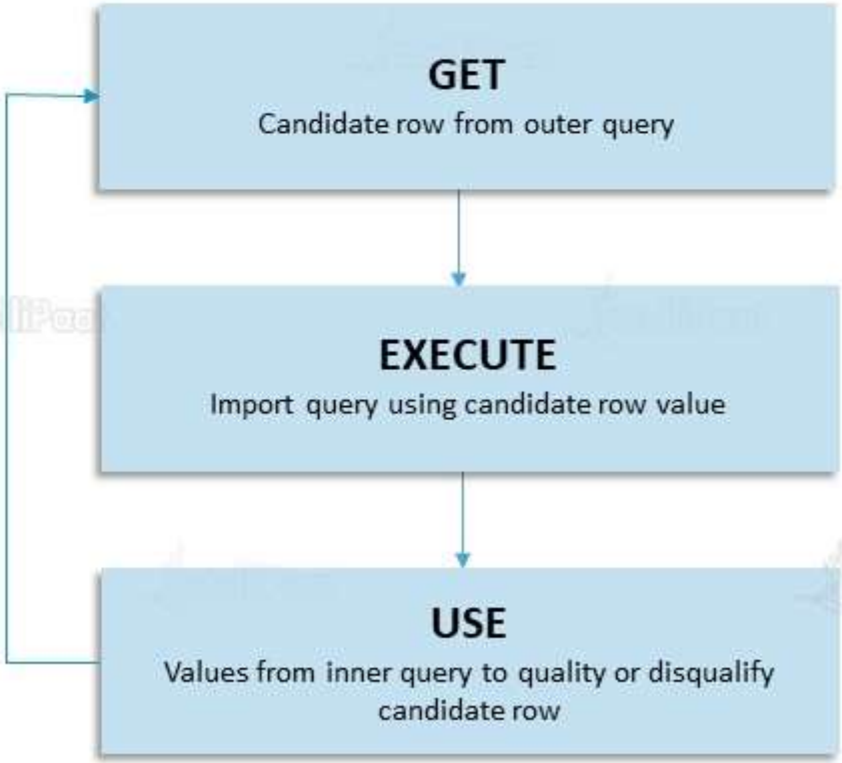
So here you first have to calculate the average salary per department, and then compare the salary for each employee to the average salary of that employee's department.



Corelated Subqueries

Corelated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



GET

Candidate row from outer query

EXECUTE

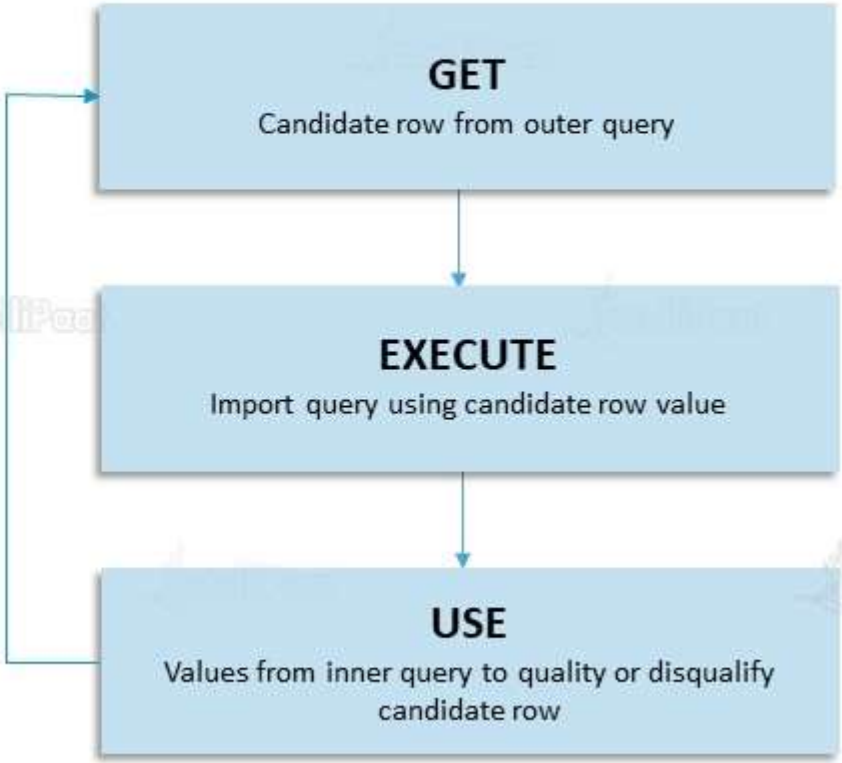
Import query using candidate row value

USE

Values from inner query to quality or disqualify candidate row

Corelated Subqueries

A correlated subquery is evaluated once for each row processed by the parent statement.



GET

Candidate row from outer query

EXECUTE

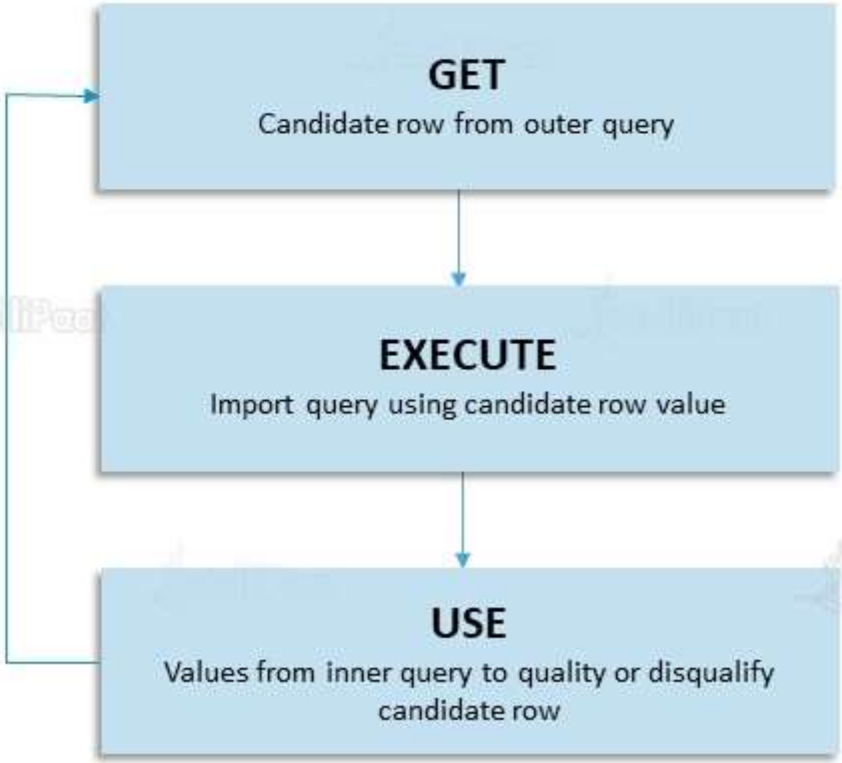
Import query using candidate row value

USE

Values from inner query to quality or disqualify candidate row

Corelated Subqueries

The parent statement can be a SELECT, UPDATE, or DELETE statement.

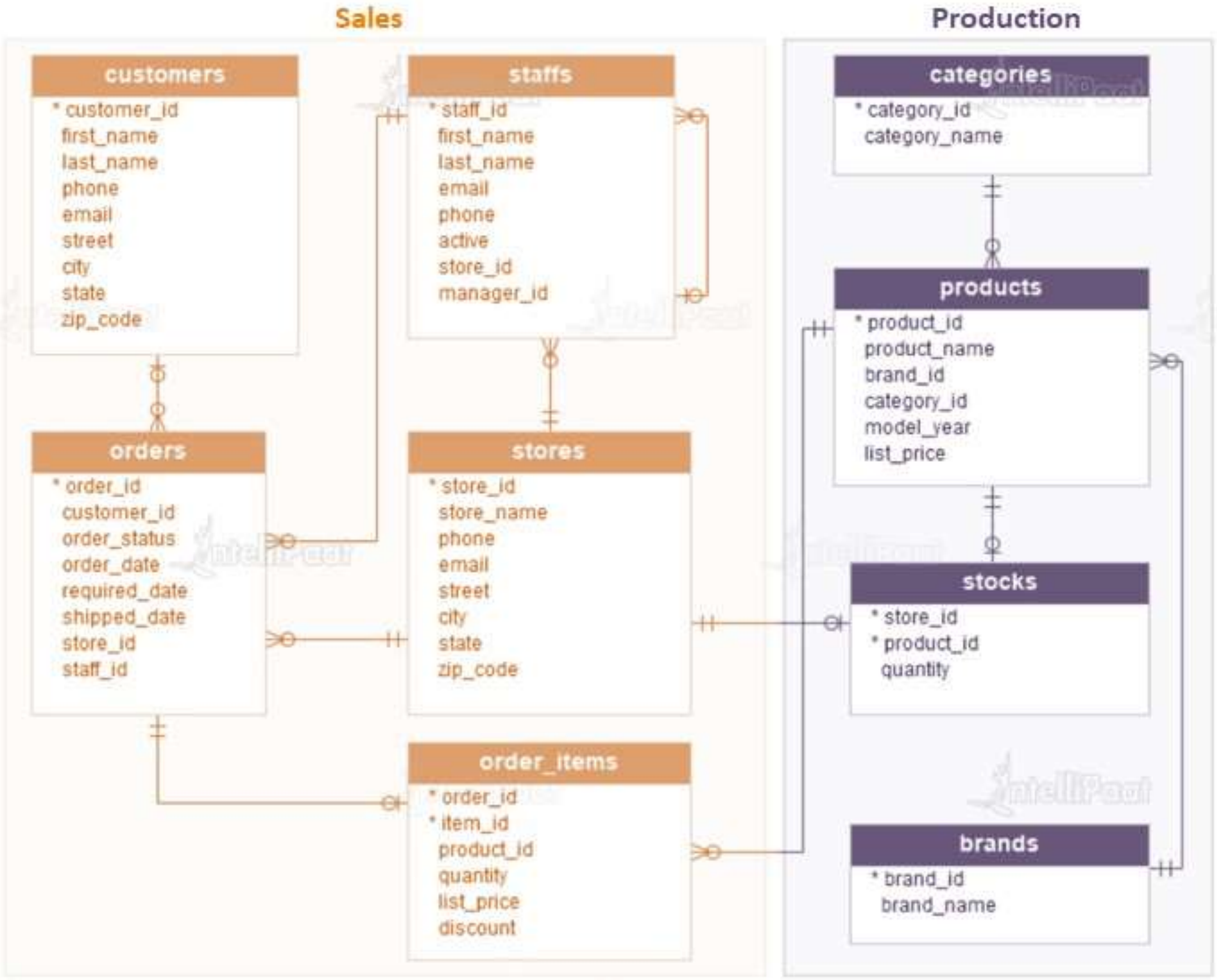


GET
Candidate row from outer query

EXECUTE
Import query using candidate row value

USE
Values from inner query to quality or disqualify candidate row

Corelated Subqueries: Example



Sales

customers
- * customer_id
- first_name
- last_name
- phone
- email
- street
- city
- state
- zip_code

staffs
- * staff_id
- first_name
- last_name
- email
- phone
- active
- store_id
- manager_id

orders
- * order_id
- customer_id
- order_status
- order_date
- required_date
- shipped_date
- store_id
- staff_id

stores
- * store_id
- store_name
- phone
- email
- street
- city
- state
- zip_code

order_items
- * order_id
- * item_id
- product_id
- quantity
- list_price
- discount

Production

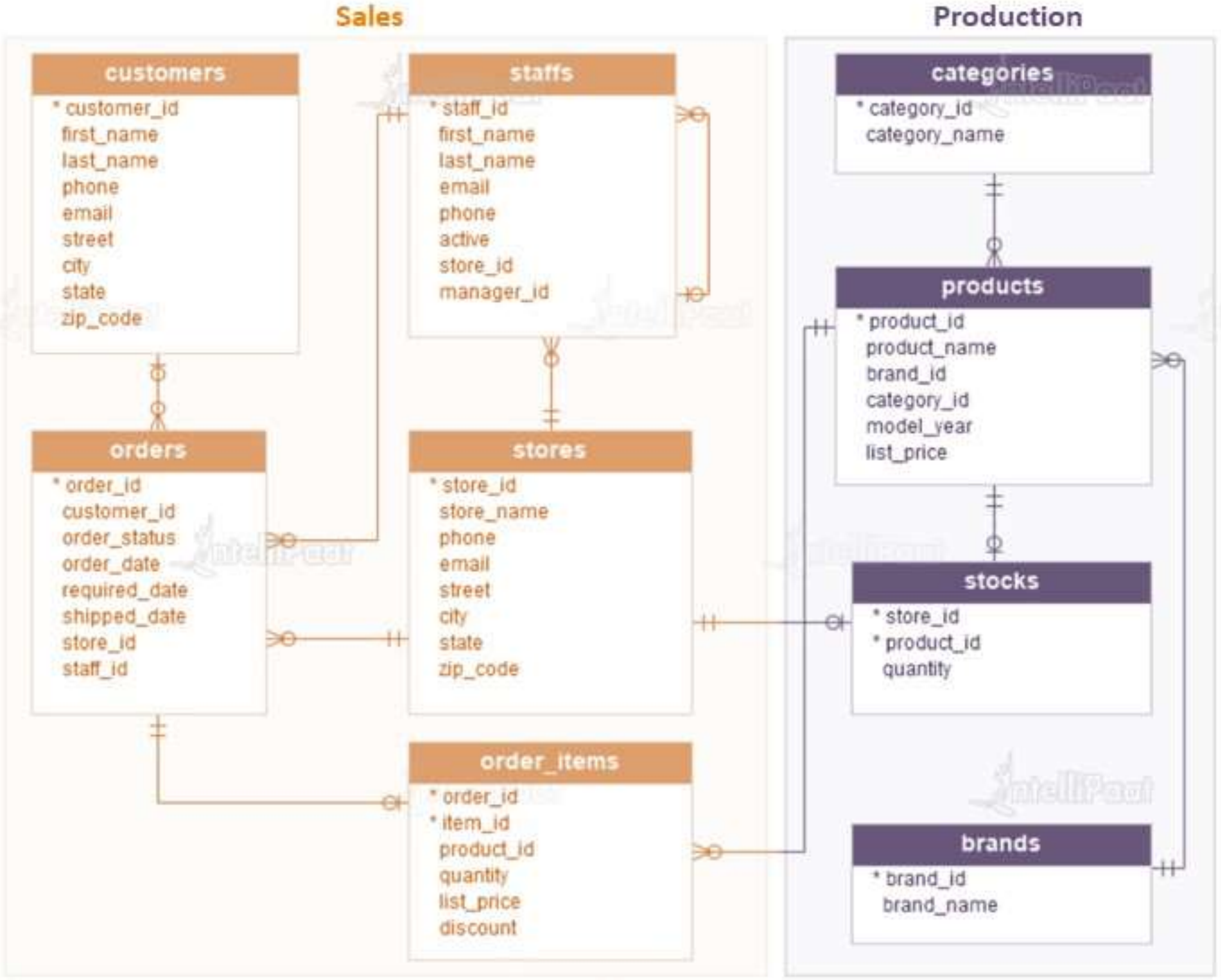
categories
- * category_id
- category_name

products
- * product_id
- product_name
- brand_id
- category_id
- model_year
- list_price

stocks
- * store_id
- * product_id
- quantity

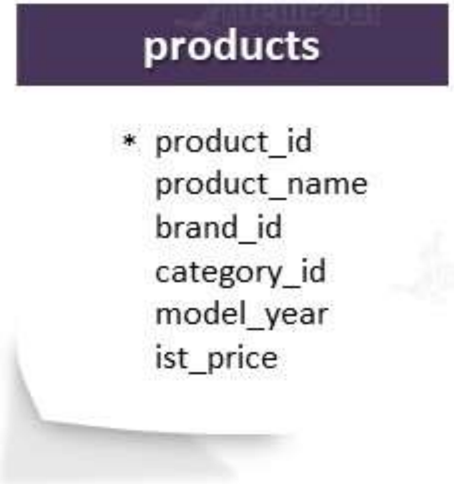
brands
- * brand_id
- brand_name

Corelated Subqueries: Example



Corelated Subqueries: Example

Consider the following `products` table from the sample database:

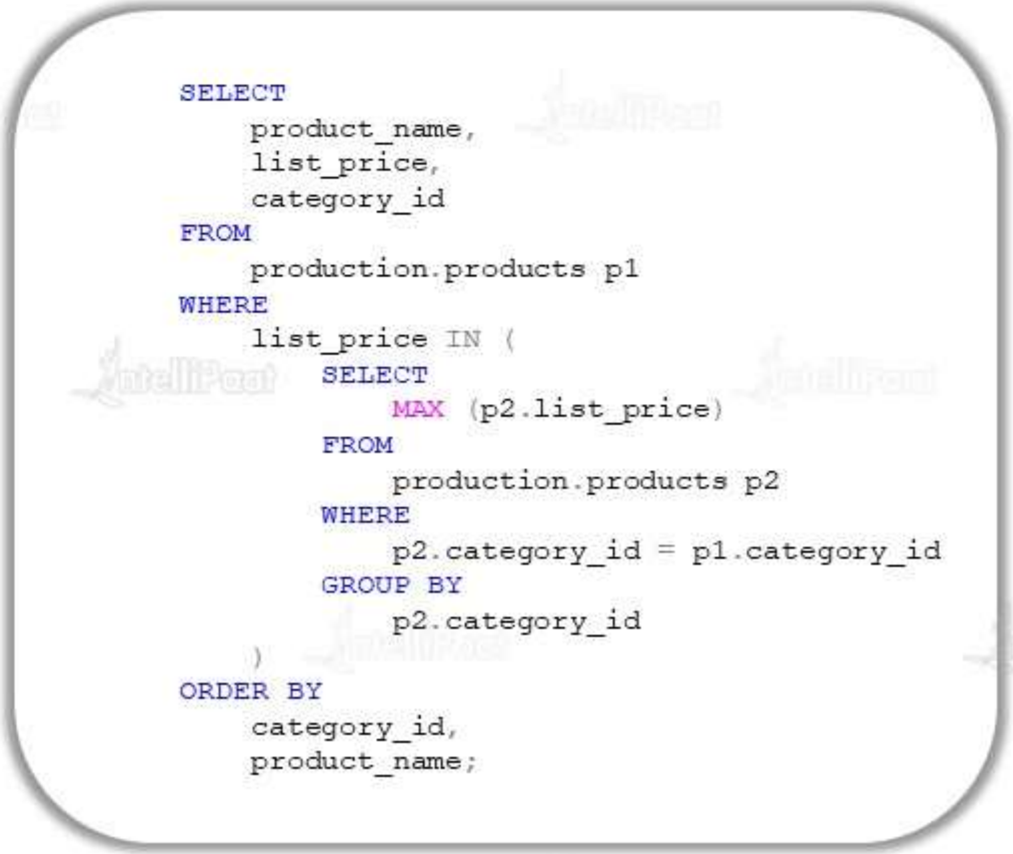


products

- * product_id
- product_name
- brand_id
- category_id
- model_year
- ist_price

Corelated Subqueries: Example

The following example finds the products whose list price is equal to the highest list price of the products within the same category:



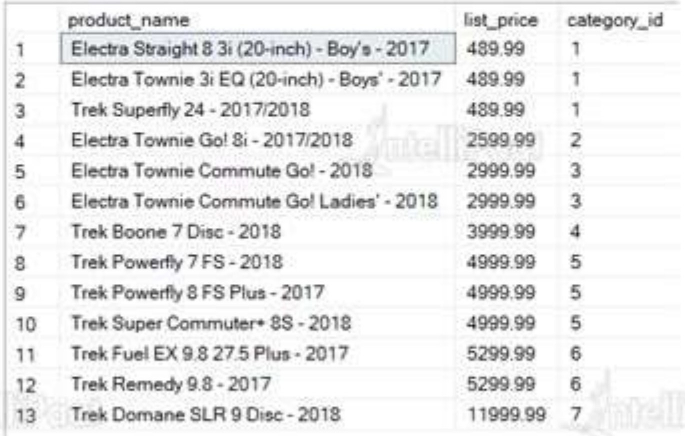
```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Corelated Subqueries: Example

The following example finds the products whose list price is equal to the highest list price of the products within the same category:

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Output



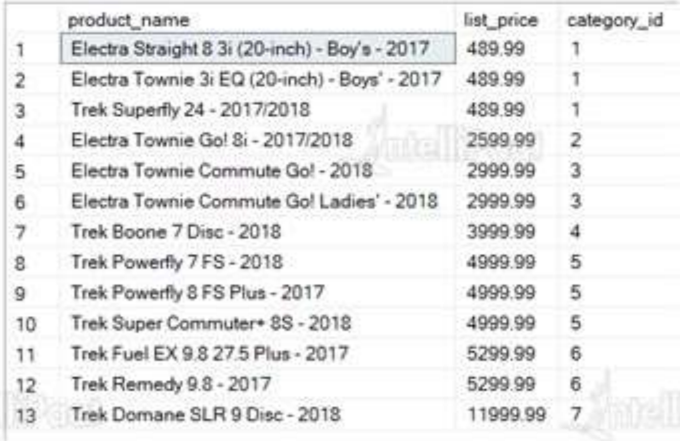
	product_name	list_price	category_id
1	Electra Straight 8 3i (20-inch) - Boy's - 2017	489.99	1
2	Electra Townie 3i EQ (20-inch) - Boys' - 2017	489.99	1
3	Trek Superfly 24 - 2017/2018	489.99	1
4	Electra Townie Go! 8i - 2017/2018	2599.99	2
5	Electra Townie Commute Go! - 2018	2999.99	3
6	Electra Townie Commute Go! Ladies' - 2018	2999.99	3
7	Trek Boone 7 Disc - 2018	3999.99	4
8	Trek Powerfly 7 FS - 2018	4999.99	5
9	Trek Powerfly 8 FS Plus - 2017	4999.99	5
10	Trek Super Commuter+ 8S - 2018	4999.99	5
11	Trek Fuel EX 9.8 27.5 Plus - 2017	5299.99	6
12	Trek Remedy 9.8 - 2017	5299.99	6
13	Trek Domane SLR 9 Disc - 2018	11999.99	7

Corelated Subqueries: Example

For each product evaluated by the outer query, the subquery finds the highest price of all products in its category.

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Output



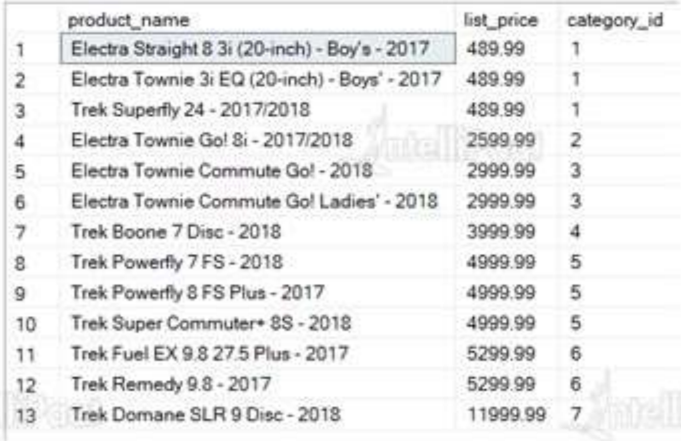
	product_name	list_price	category_id
1	Electra Straight 8 3i (20-inch) - Boy's - 2017	489.99	1
2	Electra Townie 3i EQ (20-inch) - Boys' - 2017	489.99	1
3	Trek Superfly 24 - 2017/2018	489.99	1
4	Electra Townie Go! 8i - 2017/2018	2599.99	2
5	Electra Townie Commute Go! - 2018	2999.99	3
6	Electra Townie Commute Go! Ladies' - 2018	2999.99	3
7	Trek Boone 7 Disc - 2018	3999.99	4
8	Trek Powerfly 7 FS - 2018	4999.99	5
9	Trek Powerfly 8 FS Plus - 2017	4999.99	5
10	Trek Super Commuter+ 8S - 2018	4999.99	5
11	Trek Fuel EX 9.8 27.5 Plus - 2017	5299.99	6
12	Trek Remedy 9.8 - 2017	5299.99	6
13	Trek Domane SLR 9 Disc - 2018	11999.99	7

Corelated Subqueries: Example

If the price of the current product is equal to the highest price of all products in its category, the product is included in the result set. This process continues for the next product and so on.

```
SELECT
    product_name,
    list_price,
    category_id
FROM
    production.products p1
WHERE
    list_price IN (
        SELECT
            MAX (p2.list_price)
        FROM
            production.products p2
        WHERE
            p2.category_id = p1.category_id
        GROUP BY
            p2.category_id
    )
ORDER BY
    category_id,
    product_name;
```

Output



	product_name	list_price	category_id
1	Electra Straight 8 3i (20-inch) - Boy's - 2017	489.99	1
2	Electra Townie 3i EQ (20-inch) - Boys' - 2017	489.99	1
3	Trek Superfly 24 - 2017/2018	489.99	1
4	Electra Townie Go! 8i - 2017/2018	2599.99	2
5	Electra Townie Commute Go! - 2018	2999.99	3
6	Electra Townie Commute Go! Ladies' - 2018	2999.99	3
7	Trek Boone 7 Disc - 2018	3999.99	4
8	Trek Powerfly 7 FS - 2018	4999.99	5
9	Trek Powerfly 8 FS Plus - 2017	4999.99	5
10	Trek Super Commuter+ 8S - 2018	4999.99	5
11	Trek Fuel EX 9.8 27.5 Plus - 2017	5299.99	6
12	Trek Remedy 9.8 - 2017	5299.99	6
13	Trek Domane SLR 9 Disc - 2018	11999.99	7

Exists

Exists

The EXISTS operator is a logical operator that allows you to check whether a subquery returns any row.

Exists

Exists

The EXISTS operator is a logical operator that allows you to check whether a subquery returns any row.

The EXISTS operator returns TRUE if the subquery returns one or more rows.

```
Syntax:   EXISTS(subquery)
```

Exists

Exists

The EXISTS operator is a logical operator that allows you to check whether a subquery returns any row.

The EXISTS operator returns TRUE if the subquery returns one or more rows.

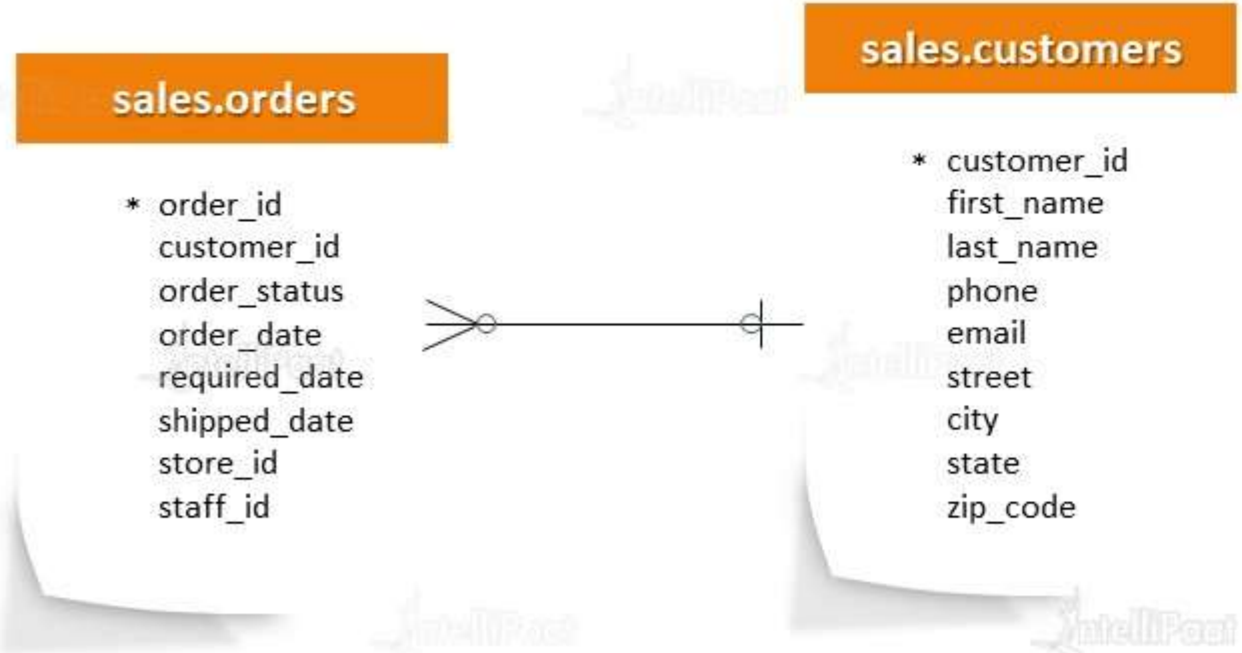
```
Syntax:   EXISTS(subquery)
```

NOTE

In this syntax, the subquery is a SELECT statement only. As soon as the subquery returns rows, the EXISTS operator returns TRUE and stop processing immediately. Though the subquery returns a NULL value, the EXISTS operator is still evaluated to TRUE.

Using EXISTS with a correlated subquery

Consider the following customers and orders tables



sales.orders

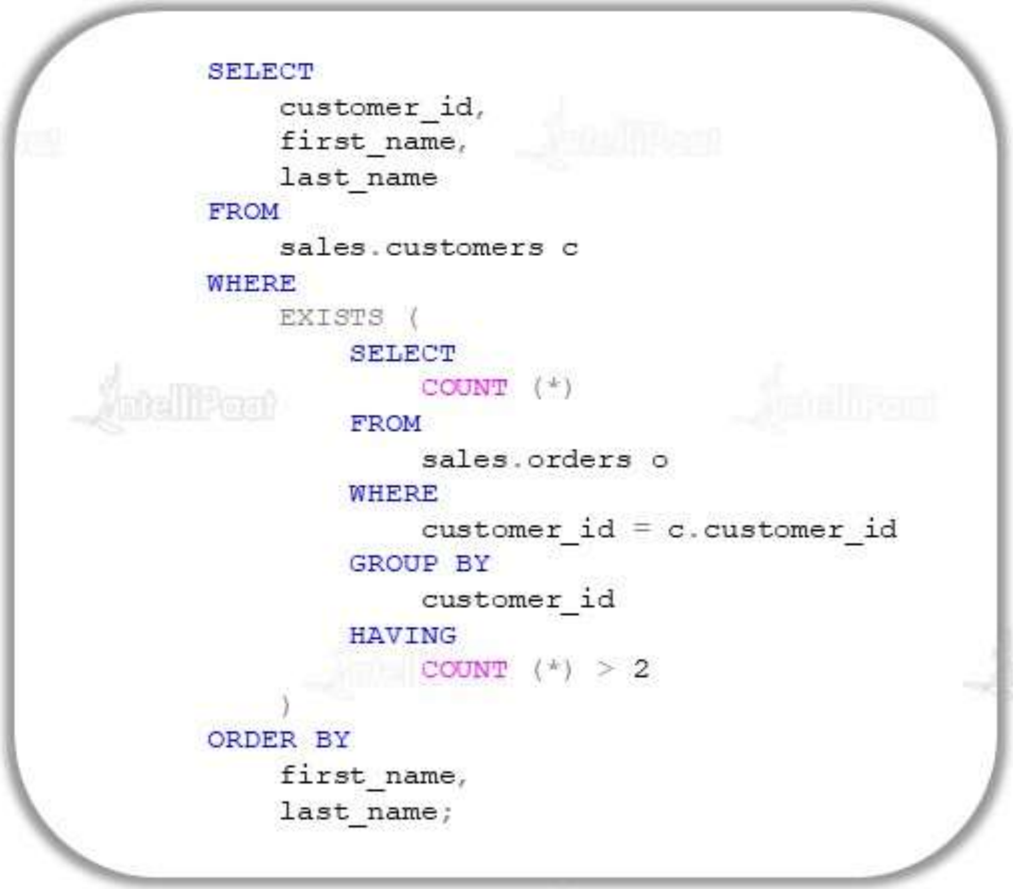
* order_id
customer_id
order_status
order_date
required_date
shipped_date
store_id
staff_id

sales.customers

* customer_id
first_name
last_name
phone
email
street
city
state
zip_code

Using EXISTS with a correlated subquery

This following example finds all customers who have placed more than two orders



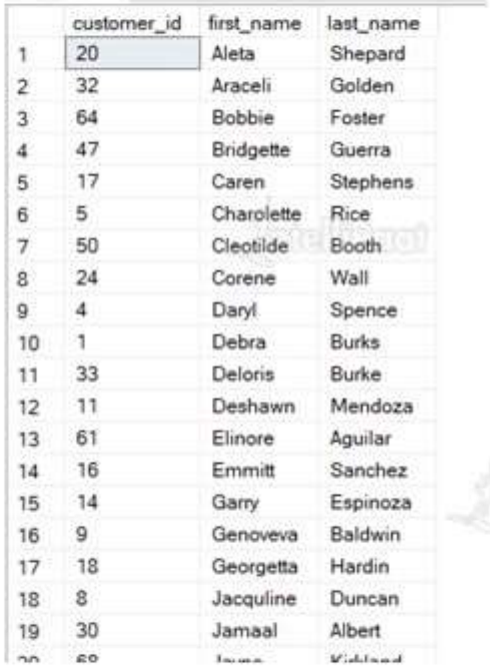
```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers c
WHERE
    EXISTS (
        SELECT
            COUNT (*)
        FROM
            sales.orders o
        WHERE
            customer_id = c.customer_id
        GROUP BY
            customer_id
        HAVING
            COUNT (*) > 2
    )
ORDER BY
    first_name,
    last_name;
```

Using EXISTS with a correlated subquery

This following example finds all customers who have placed more than two orders

```
SELECT
    customer_id,
    first_name,
    last_name
FROM
    sales.customers c
WHERE
    EXISTS (
        SELECT
            COUNT (*)
        FROM
            sales.orders o
        WHERE
            customer_id = c.customer_id
        GROUP BY
            customer_id
        HAVING
            COUNT (*) > 2
    )
ORDER BY
    first_name,
    last_name;
```

Output



	customer_id	first_name	last_name
1	20	Aleta	Shepard
2	32	Araceli	Golden
3	64	Bobbie	Foster
4	47	Bridgette	Guerra
5	17	Caren	Stephens
6	5	Charolette	Rice
7	50	Cleotilde	Booth
8	24	Corene	Wall
9	4	Daryl	Spence
10	1	Debra	Burks
11	33	Deloris	Burke
12	11	Deshawn	Mendoza
13	61	Elinore	Aguilar
14	16	Emmitt	Sanchez
15	14	Garry	Espinoza
16	9	Genoveva	Baldwin
17	18	Georgetta	Hardin
18	8	Jacquline	Duncan
19	30	Jamaal	Albert
20	68	Jayne	Kirkland

Agenda

2 GROUPING SETS

GROUPING SETS

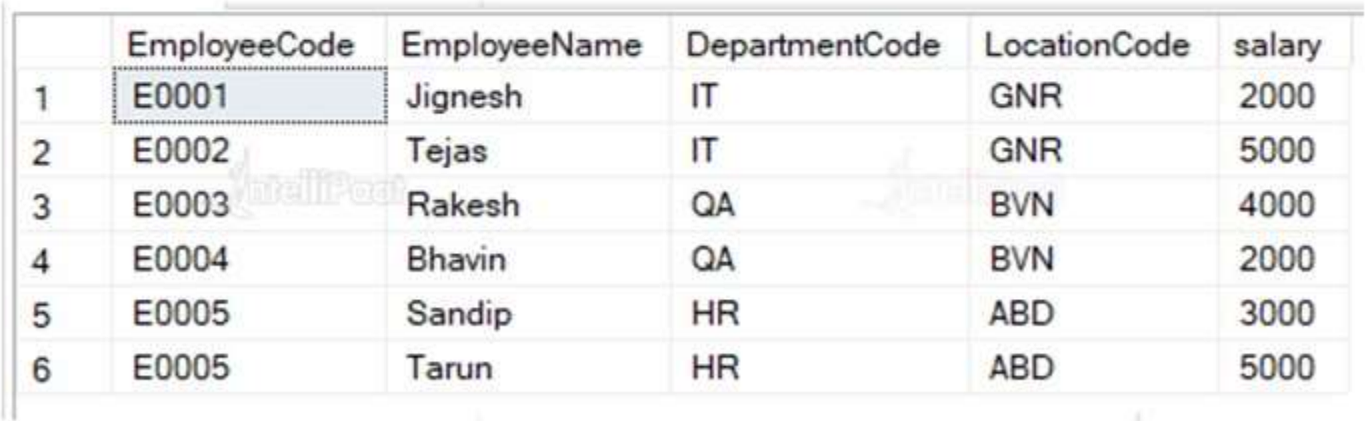
Grouping Sets

GROUPING SET is able to generate a result set that can be generated by a UNION ALL of multiple simple GROUP BY clauses. It is capable of generating a result set that is equivalent to the result set generated by ROLL UP or CUBE operations.

<table>
<tr><th colspan="2">GROUPING SETS Equivalent of UNION ALL</th></tr>
<tr><td>SELECT Col1, Col2, SUM(Col3) FROM Table1 GROUP BY GROUPING SETS((Col1), (Col2))</td><td>SELECT Col1, NULL as Col2, SUM(Col3) FROM Table1 GROUP BY Col1 UNION ALL SELECT NULL as Col1, Col2, SUM(Col3) From Table1 GROUP BY Col2</td></tr>
<tr><th colspan="2">GROUPING SETS ROLLUP Equivalents</th></tr>
<tr><td>GROUP BY ROLLUP(Col1, Col2, Col3)</td><td>GROUP BY GROUPING SETS((Col1, Col2, Col3),(Col1, Col2), (Col),())</td></tr>
<tr><th colspan="2">GROUPING SETS CUBE Equivalents</th></tr>
<tr><td>GROUP BY CUBE(Col1, Col2, Col3)</td><td>GROUP BY GROUPING SETS(Col1, Col2, Col3), (Col1, Col2), (Col1, Col3), (Col2, Col3), (Col1), (Col2), (Col3),())</td></tr>
</table>

GROUPING SETS

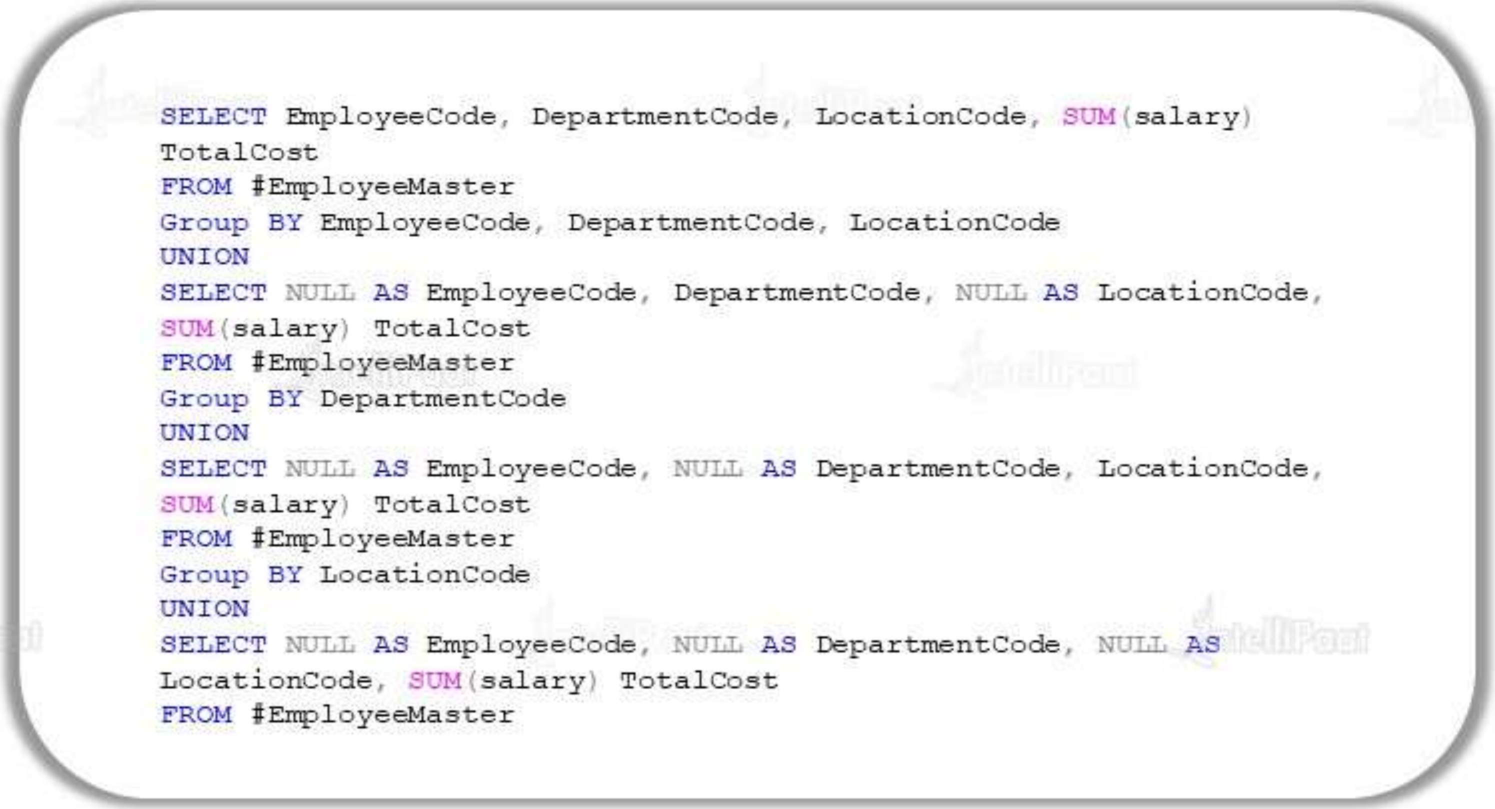
From the below table, I want some summarized data, like total cost by Employee, total cost by Department, total cost by location and total cost for all employees with all locations in a single result set.



	EmployeeCode	EmployeeName	DepartmentCode	LocationCode	salary
1	E0001	Jignesh	IT	GNR	2000
2	E0002	Tejas	IT	GNR	5000
3	E0003	Rakesh	QA	BVN	4000
4	E0004	Bhavin	QA	BVN	2000
5	E0005	Sandip	HR	ABD	3000
6	E0005	Tarun	HR	ABD	5000

GROUPING SETS

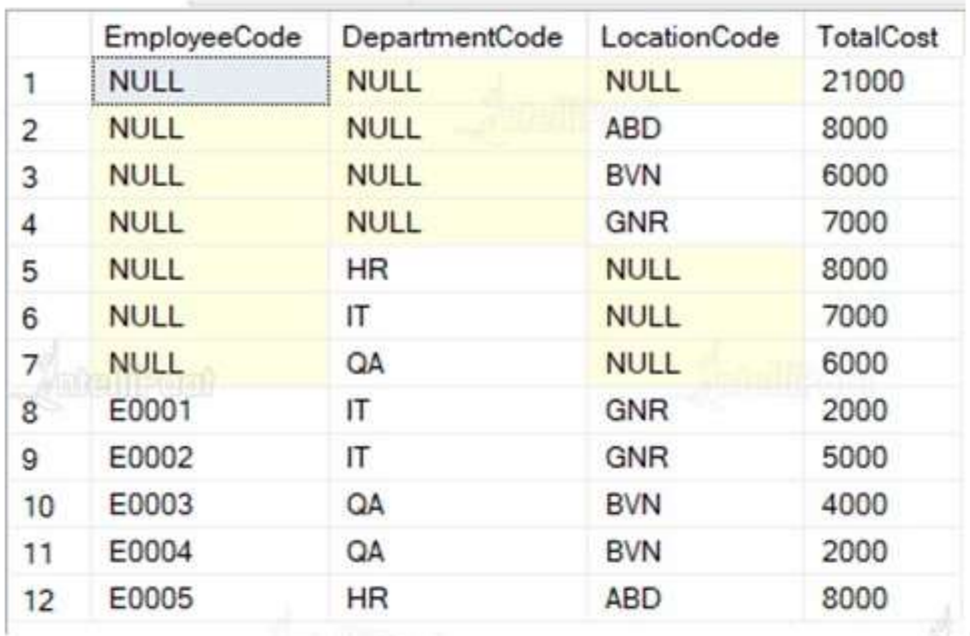
We must write a different query and UNION these queries



```
SELECT EmployeeCode, DepartmentCode, LocationCode, SUM(salary)
TotalCost
FROM #EmployeeMaster
Group BY EmployeeCode, DepartmentCode, LocationCode
UNION
SELECT NULL AS EmployeeCode, DepartmentCode, NULL AS LocationCode,
SUM(salary) TotalCost
FROM #EmployeeMaster
Group BY DepartmentCode
UNION
SELECT NULL AS EmployeeCode, NULL AS DepartmentCode, LocationCode,
SUM(salary) TotalCost
FROM #EmployeeMaster
Group BY LocationCode
UNION
SELECT NULL AS EmployeeCode, NULL AS DepartmentCode, NULL AS
LocationCode, SUM(salary) TotalCost
FROM #EmployeeMaster
```

GROUPING SETS

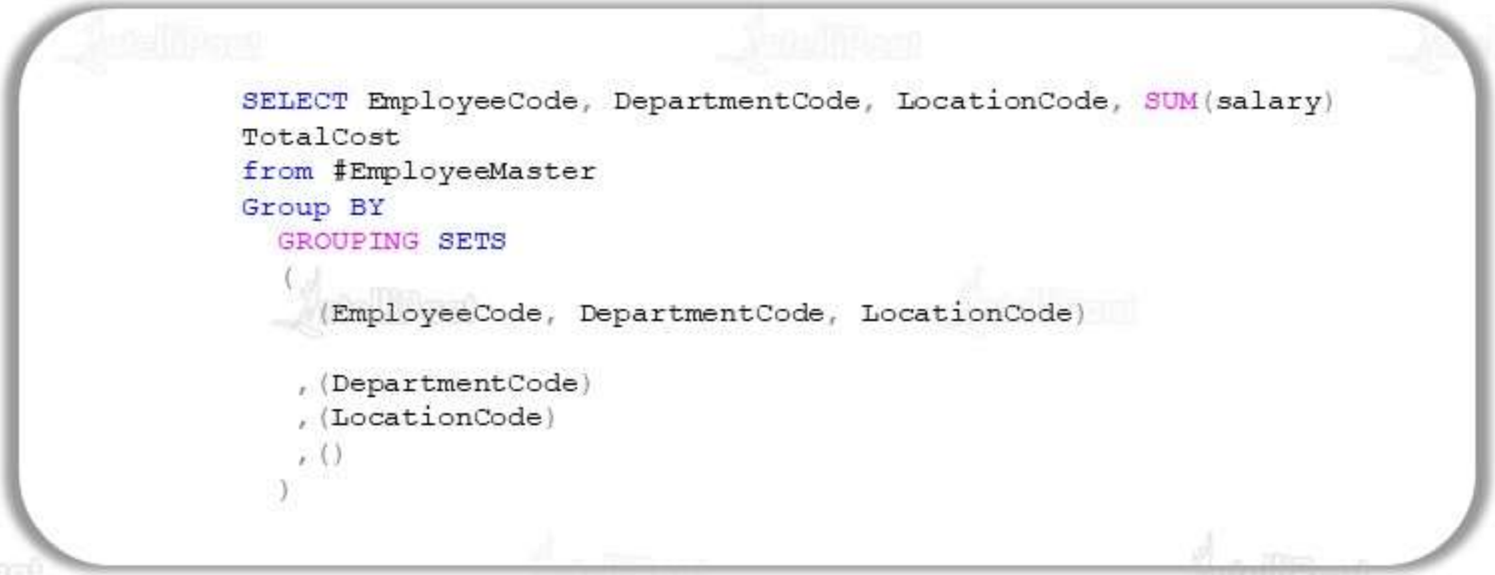
Output: Result Set from Union Query



	EmployeeCode	DepartmentCode	LocationCode	TotalCost
1	NULL	NULL	NULL	21000
2	NULL	NULL	ABD	8000
3	NULL	NULL	BVN	6000
4	NULL	NULL	GNR	7000
5	NULL	HR	NULL	8000
6	NULL	IT	NULL	7000
7	NULL	QA	NULL	6000
8	E0001	IT	GNR	2000
9	E0002	IT	GNR	5000
10	E0003	QA	BVN	4000
11	E0004	QA	BVN	2000
12	E0005	HR	ABD	8000

GROUPING SETS

Using GROUPING SET Query

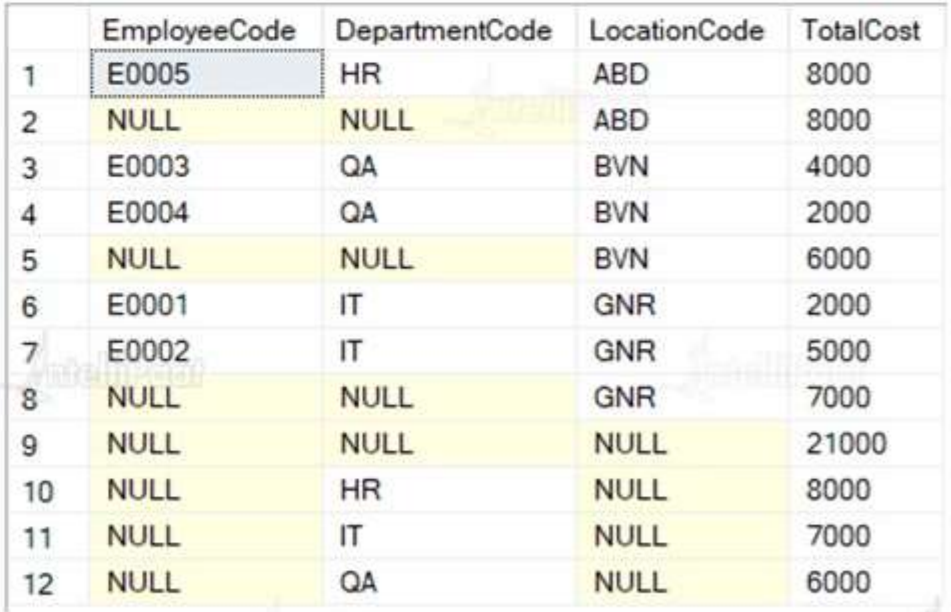


```
SELECT EmployeeCode, DepartmentCode, LocationCode, SUM(salary)
TotalCost
from #EmployeeMaster
Group BY
  GROUPING SETS
  (
    (EmployeeCode, DepartmentCode, LocationCode)

   ,(DepartmentCode)
   ,(LocationCode)
   ,()
  )
```

GROUPING SETS

Output: Result Set from Grouping Set Query

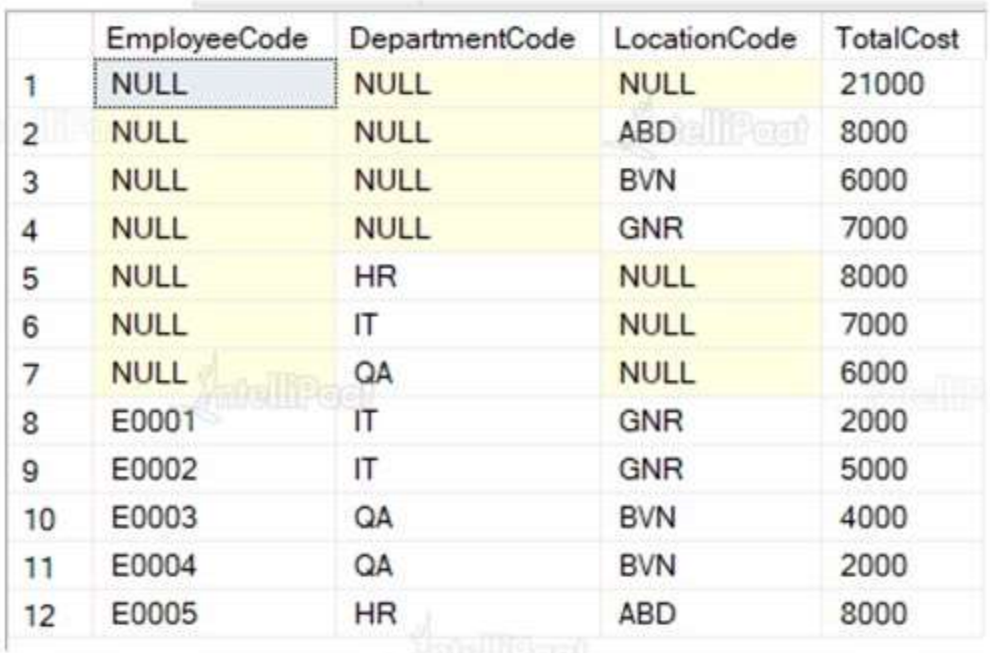


	EmployeeCode	DepartmentCode	LocationCode	TotalCost
1	E0005	HR	ABD	8000
2	NULL	NULL	ABD	8000
3	E0003	QA	BVN	4000
4	E0004	QA	BVN	2000
5	NULL	NULL	BVN	6000
6	E0001	IT	GNR	2000
7	E0002	IT	GNR	5000
8	NULL	NULL	GNR	7000
9	NULL	NULL	NULL	21000
10	NULL	HR	NULL	8000
11	NULL	IT	NULL	7000
12	NULL	QA	NULL	6000

GROUPING SETS

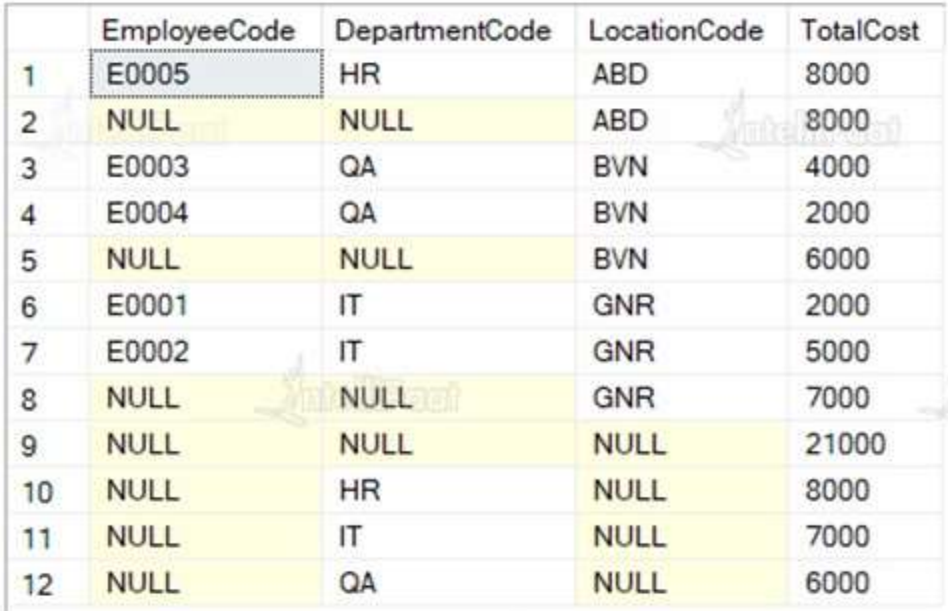
Using GROUPING SETS, we can write multiple "Group By" clauses within a single query and get a single result set. Also it can be used as equivalent to as well as with ROLLUP and CUBE.

Output: Result Set from Union Query



	EmployeeCode	DepartmentCode	LocationCode	TotalCost
1	NULL	NULL	NULL	21000
2	NULL	NULL	ABD	8000
3	NULL	NULL	BVN	6000
4	NULL	NULL	GNR	7000
5	NULL	HR	NULL	8000
6	NULL	IT	NULL	7000
7	NULL	QA	NULL	6000
8	E0001	IT	GNR	2000
9	E0002	IT	GNR	5000
10	E0003	QA	BVN	4000
11	E0004	QA	BVN	2000
12	E0005	HR	ABD	8000

Output: Result Set from Grouping Set Query



	EmployeeCode	DepartmentCode	LocationCode	TotalCost
1	E0005	HR	ABD	8000
2	NULL	NULL	ABD	8000
3	E0003	QA	BVN	4000
4	E0004	QA	BVN	2000
5	NULL	NULL	BVN	6000
6	E0001	IT	GNR	2000
7	E0002	IT	GNR	5000
8	NULL	NULL	GNR	7000
9	NULL	NULL	NULL	21000
10	NULL	HR	NULL	8000
11	NULL	IT	NULL	7000
12	NULL	QA	NULL	6000

Agenda

3 ROLLUP

ROLLUP

ROLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

Let's consider an example. The following `CUBE (d1,d2,d3)` defines eight possible grouping sets:

```
(d1, d2, d3)
(d1, d2)
(d2, d3)
(d1, d3)
(d1)
(d2)
(d3)
()
```

ROLLUP

ROLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

And the `ROLLUP(d1,d2,d3)` creates only four grouping sets, assuming the hierarchy d1 > d2 > d3, as follows:

```
(d1, d2, d3)
(d1, d2)
(d1)
()
```

ROLLUP

ROLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

The ROLLUP is commonly used to calculate the aggregates of hierarchical data such as sales by year > quarter > month.

ROLLUP

ROLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

Syntax

```
SELECT
    d1,
    d2,
    d3,
    aggregate_function(c4)
FROM
    table_name
GROUP BY
    ROLLUP (d1, d2, d3);
```

In this syntax, d1, d2, and d3 are the dimension columns. The statement will calculate the aggregation of values in the column c4 based on the hierarchy d1 > d2 > d3.

ROLLUP

ROLUP

ROLLUP is a subclause of the GROUP BY clause which provides a shorthand for defining multiple grouping sets. Unlike the CUBE subclause, ROLLUP does not create all possible grouping sets based on the dimension columns

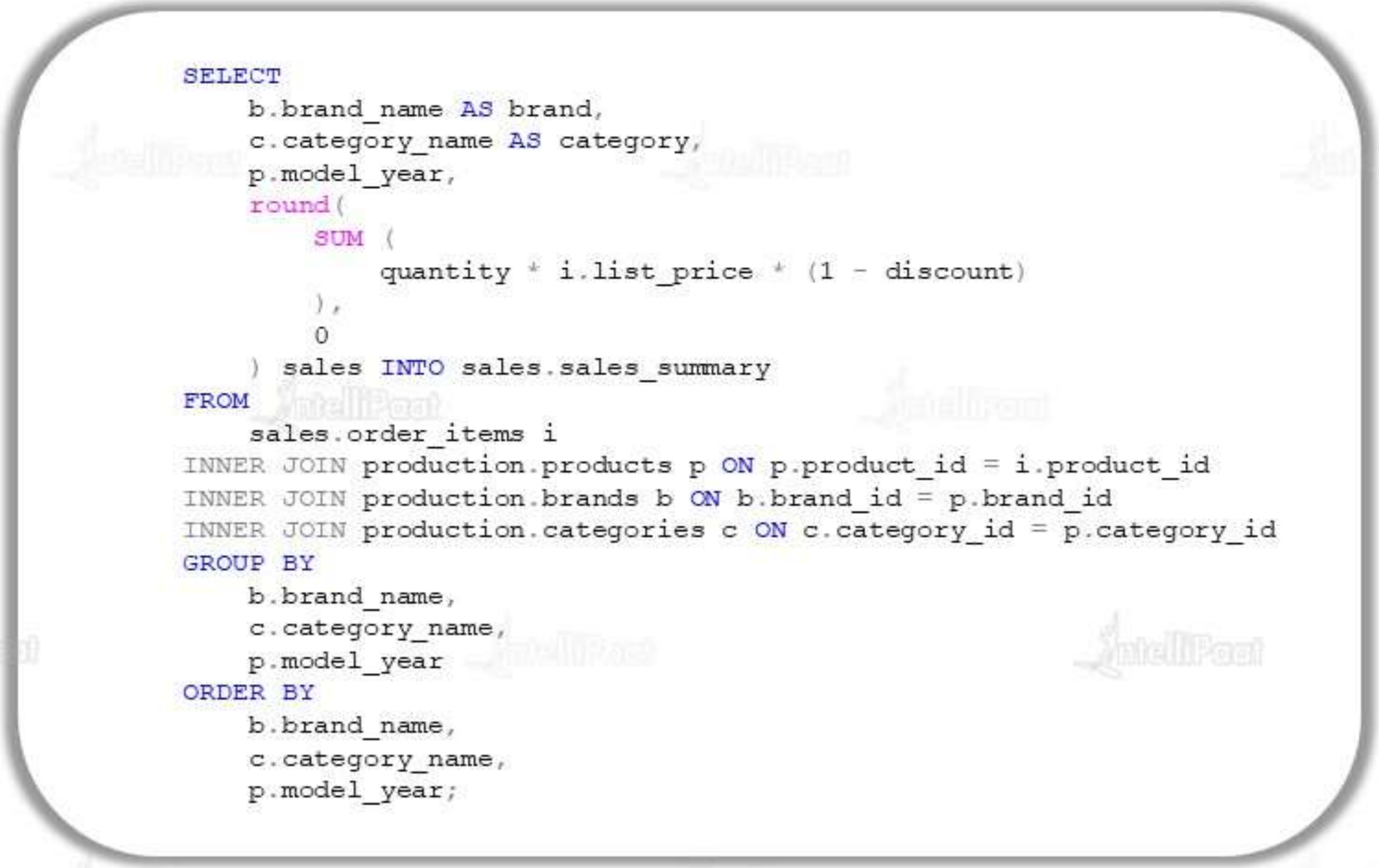
Syntax

```
SELECT
    d1,
    d2,
    d3,
    aggregate_function(c4)
FROM
    table_name
GROUP BY
    d1,
    ROLLUP (d2, d3);
```

You can also do a partial roll up to reduce the subtotals generated by using the following syntax

ROLLUP

Let’s create the sales summary table



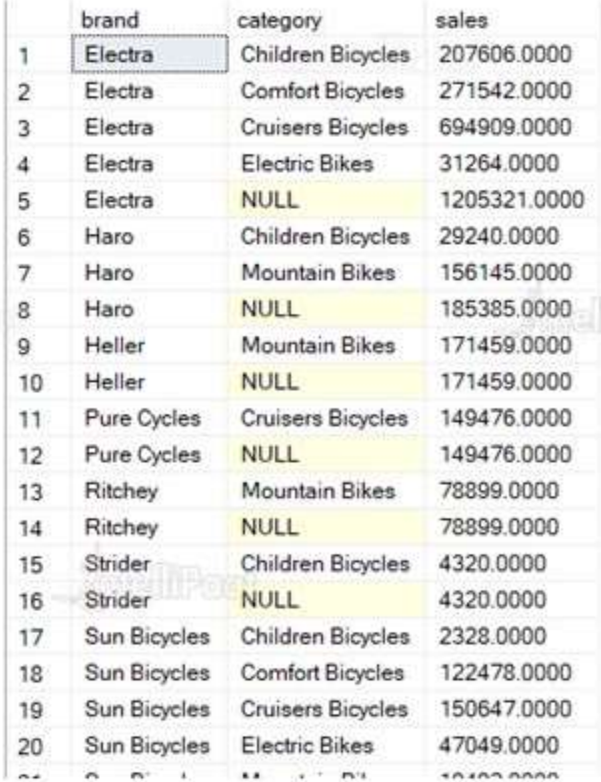
```
SELECT
    b.brand_name AS brand,
    c.category_name AS category,
    p.model_year,
    round(
        SUM (
            quantity * i.list_price * (1 - discount)
        ),
        0
    ) sales INTO sales.sales_summary
FROM
    sales.order_items i
INNER JOIN production.products p ON p.product_id = i.product_id
INNER JOIN production.brands b ON b.brand_id = p.brand_id
INNER JOIN production.categories c ON c.category_id = p.category_id
GROUP BY
    b.brand_name,
    c.category_name,
    p.model_year
ORDER BY
    b.brand_name,
    c.category_name,
    p.model_year;
```

ROLLUP

The following query uses the ROLLUP to calculate the sales amount by brand (subtotal) and both brand and category (total).

```
SELECT
    brand,
    category,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    ROLLUP(brand, category);
```

Output



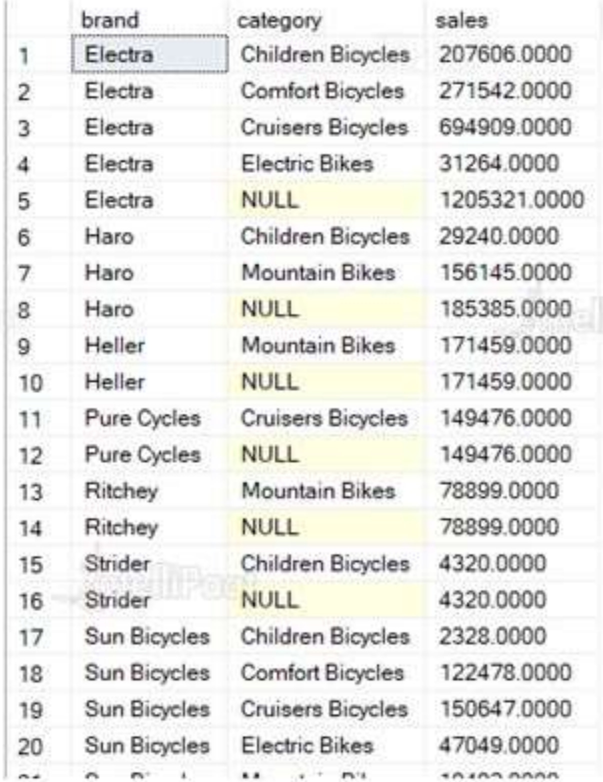
	brand	category	sales
1	Electra	Children Bicycles	207606.0000
2	Electra	Comfort Bicycles	271542.0000
3	Electra	Cruisers Bicycles	694909.0000
4	Electra	Electric Bikes	31264.0000
5	Electra	NULL	1205321.0000
6	Haro	Children Bicycles	29240.0000
7	Haro	Mountain Bikes	156145.0000
8	Haro	NULL	185385.0000
9	Heller	Mountain Bikes	171459.0000
10	Heller	NULL	171459.0000
11	Pure Cycles	Cruisers Bicycles	149476.0000
12	Pure Cycles	NULL	149476.0000
13	Ritchey	Mountain Bikes	78899.0000
14	Ritchey	NULL	78899.0000
15	Strider	Children Bicycles	4320.0000
16	Strider	NULL	4320.0000
17	Sun Bicycles	Children Bicycles	2328.0000
18	Sun Bicycles	Comfort Bicycles	122478.0000
19	Sun Bicycles	Cruisers Bicycles	150647.0000
20	Sun Bicycles	Electric Bikes	47049.0000

ROLLUP

In this example, the query assumes that there is a hierarchy between brand and category, which is the brand > category.

```
SELECT
    brand,
    category,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    ROLLUP(brand, category);
```

Output



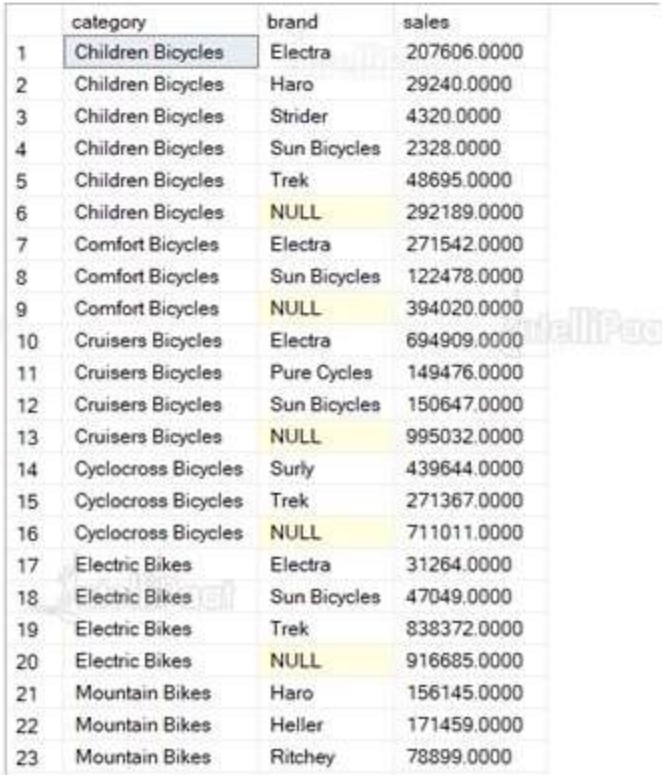
	brand	category	sales
1	Electra	Children Bicycles	207606.0000
2	Electra	Comfort Bicycles	271542.0000
3	Electra	Cruisers Bicycles	694909.0000
4	Electra	Electric Bikes	31264.0000
5	Electra	NULL	1205321.0000
6	Haro	Children Bicycles	29240.0000
7	Haro	Mountain Bikes	156145.0000
8	Haro	NULL	185385.0000
9	Heller	Mountain Bikes	171459.0000
10	Heller	NULL	171459.0000
11	Pure Cycles	Cruisers Bicycles	149476.0000
12	Pure Cycles	NULL	149476.0000
13	Ritchey	Mountain Bikes	78899.0000
14	Ritchey	NULL	78899.0000
15	Strider	Children Bicycles	4320.0000
16	Strider	NULL	4320.0000
17	Sun Bicycles	Children Bicycles	2328.0000
18	Sun Bicycles	Comfort Bicycles	122478.0000
19	Sun Bicycles	Cruisers Bicycles	150647.0000
20	Sun Bicycles	Electric Bikes	47049.0000

ROLLUP

If you change the order of brand and category, the result will be different as shown in the following query:

```
SELECT
    category,
    brand,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    ROLLUP (category, brand);
```

Output



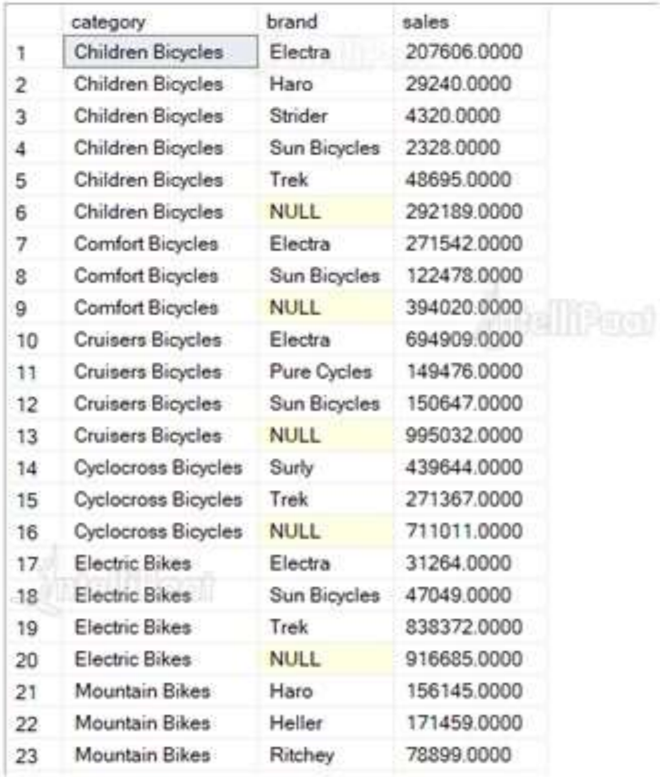
	category	brand	sales
1	Children Bicycles	Electra	207606.0000
2	Children Bicycles	Haro	29240.0000
3	Children Bicycles	Strider	4320.0000
4	Children Bicycles	Sun Bicycles	2328.0000
5	Children Bicycles	Trek	48695.0000
6	Children Bicycles	NULL	292189.0000
7	Comfort Bicycles	Electra	271542.0000
8	Comfort Bicycles	Sun Bicycles	122478.0000
9	Comfort Bicycles	NULL	394020.0000
10	Cruisers Bicycles	Electra	694909.0000
11	Cruisers Bicycles	Pure Cycles	149476.0000
12	Cruisers Bicycles	Sun Bicycles	150647.0000
13	Cruisers Bicycles	NULL	995032.0000
14	Cyclocross Bicycles	Surly	439644.0000
15	Cyclocross Bicycles	Trek	271367.0000
16	Cyclocross Bicycles	NULL	711011.0000
17	Electric Bikes	Electra	31264.0000
18	Electric Bikes	Sun Bicycles	47049.0000
19	Electric Bikes	Trek	838372.0000
20	Electric Bikes	NULL	916685.0000
21	Mountain Bikes	Haro	156145.0000
22	Mountain Bikes	Heller	171459.0000
23	Mountain Bikes	Ritchey	78899.0000

ROLLUP

In this example, the hierarchy is the brand > segment

```
SELECT
    category,
    brand,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    ROLLUP (category, brand);
```

Output



	category	brand	sales
1	Children Bicycles	Electra	207606.0000
2	Children Bicycles	Haro	29240.0000
3	Children Bicycles	Strider	4320.0000
4	Children Bicycles	Sun Bicycles	2328.0000
5	Children Bicycles	Trek	48695.0000
6	Children Bicycles	NULL	292189.0000
7	Comfort Bicycles	Electra	271542.0000
8	Comfort Bicycles	Sun Bicycles	122478.0000
9	Comfort Bicycles	NULL	394020.0000
10	Cruisers Bicycles	Electra	694909.0000
11	Cruisers Bicycles	Pure Cycles	149476.0000
12	Cruisers Bicycles	Sun Bicycles	150647.0000
13	Cruisers Bicycles	NULL	995032.0000
14	Cyclocross Bicycles	Surly	439644.0000
15	Cyclocross Bicycles	Trek	271367.0000
16	Cyclocross Bicycles	NULL	711011.0000
17	Electric Bikes	Electra	31264.0000
18	Electric Bikes	Sun Bicycles	47049.0000
19	Electric Bikes	Trek	838372.0000
20	Electric Bikes	NULL	916685.0000
21	Mountain Bikes	Haro	156145.0000
22	Mountain Bikes	Heller	171459.0000
23	Mountain Bikes	Ritchey	78899.0000

Agenda

4 CUBE

CUBE

CUBE

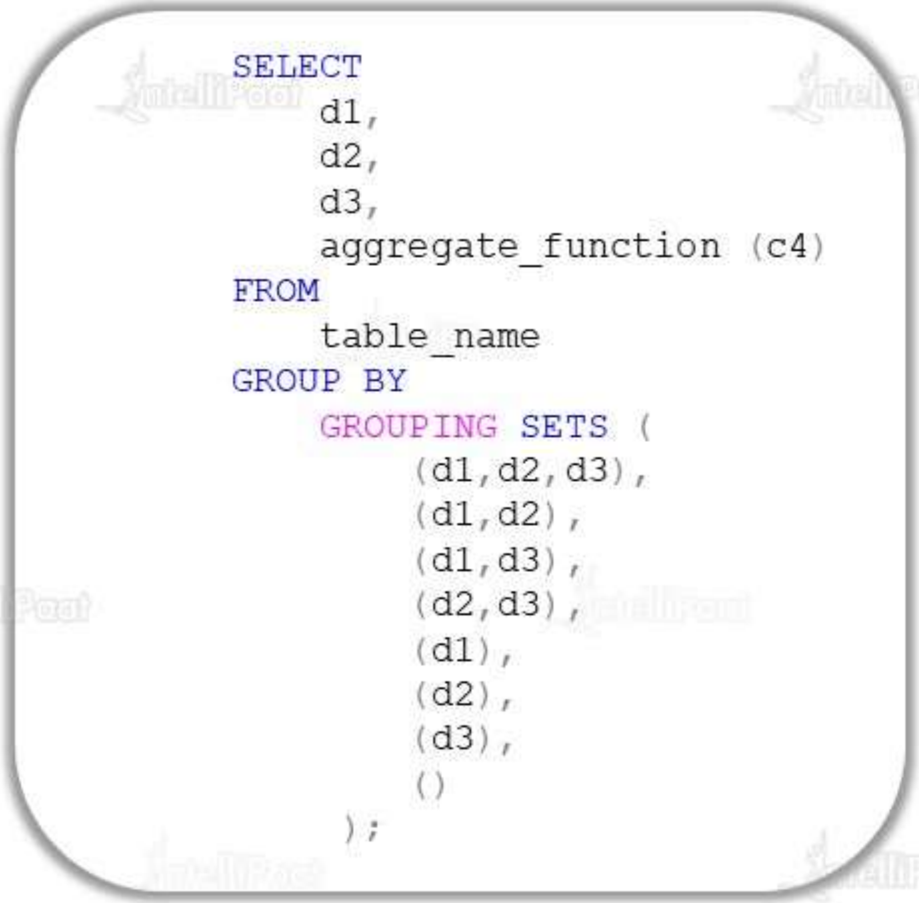
The CUBE is a subclause of the GROUP BY clause that allows you to generate multiple grouping sets

Syntax

```
SELECT
    d1,
    d2,
    d3,
    aggregate_function (c4)
FROM
    table_name
GROUP BY
    CUBE (d1, d2, d3);
```

In this syntax, the CUBE generates all possible grouping sets based on the dimension columns d1, d2, and d3 that you specify in the CUBE clause

CUBE

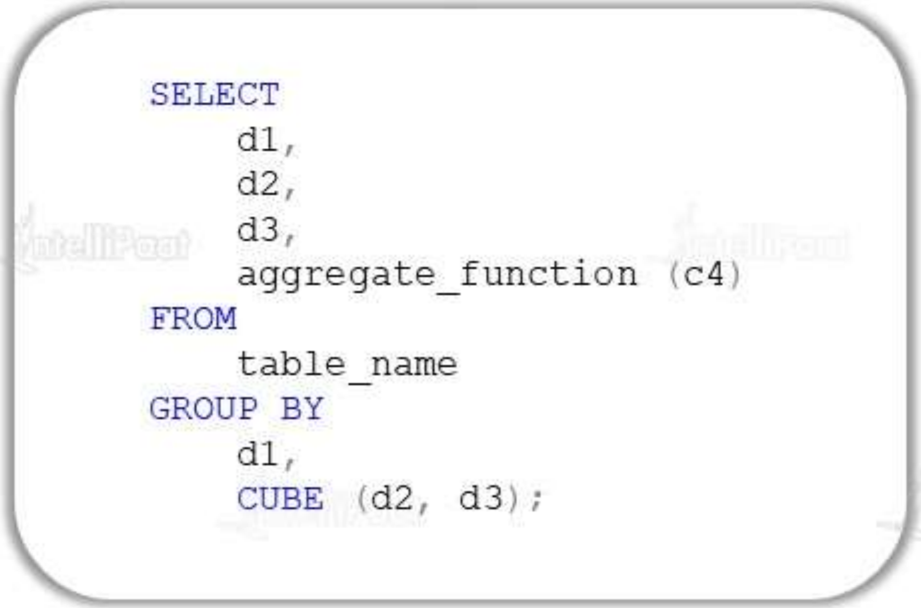


```
SELECT
    d1,
    d2,
    d3,
    aggregate_function (c4)
FROM
    table_name
GROUP BY
    GROUPING SETS (
        (d1,d2,d3),
        (d1,d2),
        (d1,d3),
        (d2,d3),
        (d1),
        (d2),
        (d3),
        ()
     );
```

In this syntax, the CUBE generates all possible grouping sets based on the dimension columns d1, d2, and d3 that you specify in the CUBE clause

CUBE

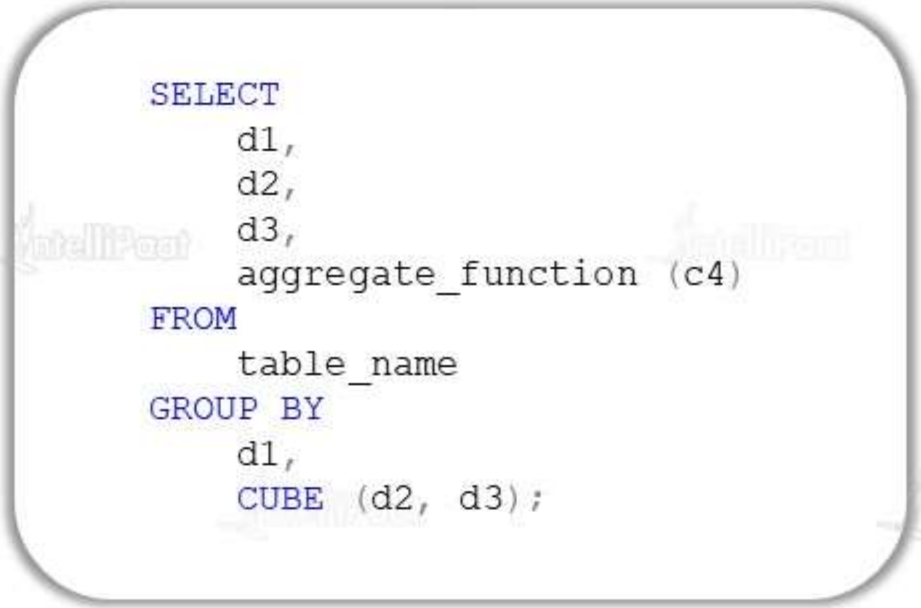
If you have N dimension columns specified in the CUBE, you will have 2^n grouping sets. It is possible to reduce the number of grouping sets by using the CUBE partially as shown in the following query:



```
SELECT
    d1,
    d2,
    d3,
    aggregate_function (c4)
FROM
    table_name
GROUP BY
    d1,
    CUBE (d2, d3);
```

CUBE

In this case, the query generates four grouping sets because there are only two dimension columns specified in the CUBE.



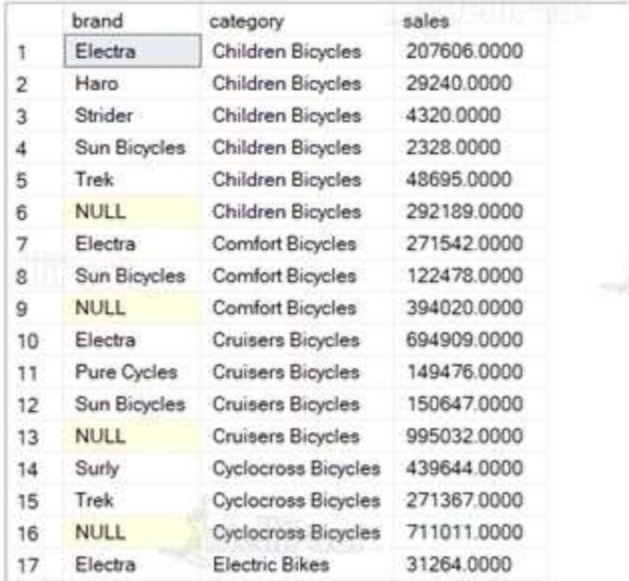
```
SELECT
    d1,
    d2,
    d3,
    aggregate_function (c4)
FROM
    table_name
GROUP BY
    d1,
    CUBE (d2, d3);
```

CUBE

Lets use the CUBE to generate four grouping sets: (brand, category), (brand), (category), ()

```
SELECT
    brand,
    category,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    CUBE(brand, category);
```

Output



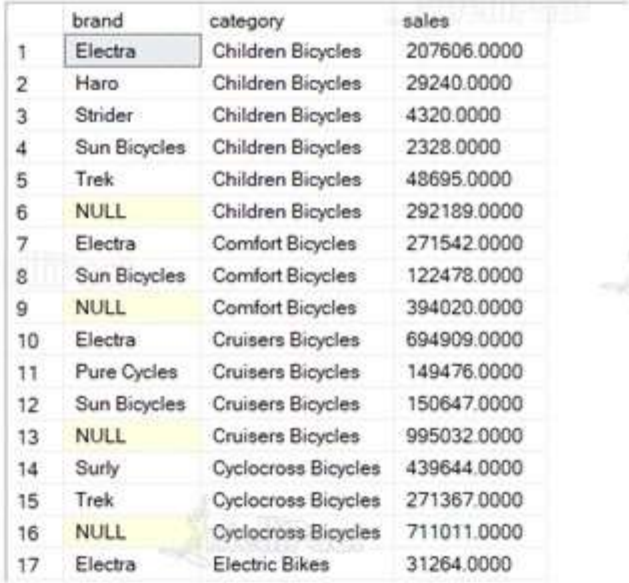
	brand	category	sales
1	Electra	Children Bicycles	207606.0000
2	Haro	Children Bicycles	29240.0000
3	Strider	Children Bicycles	4320.0000
4	Sun Bicycles	Children Bicycles	2328.0000
5	Trek	Children Bicycles	48695.0000
6	NULL	Children Bicycles	292189.0000
7	Electra	Comfort Bicycles	271542.0000
8	Sun Bicycles	Comfort Bicycles	122478.0000
9	NULL	Comfort Bicycles	394020.0000
10	Electra	Cruisers Bicycles	694909.0000
11	Pure Cycles	Cruisers Bicycles	149476.0000
12	Sun Bicycles	Cruisers Bicycles	150647.0000
13	NULL	Cruisers Bicycles	995032.0000
14	Surly	Cyclocross Bicycles	439644.0000
15	Trek	Cyclocross Bicycles	271367.0000
16	NULL	Cyclocross Bicycles	711011.0000
17	Electra	Electric Bikes	31264.0000

CUBE

In this example, we have two dimension columns specified in the CUBE clause, therefore, we have a total of four grouping sets.

```
SELECT
    brand,
    category,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    CUBE(brand, category);
```

Output



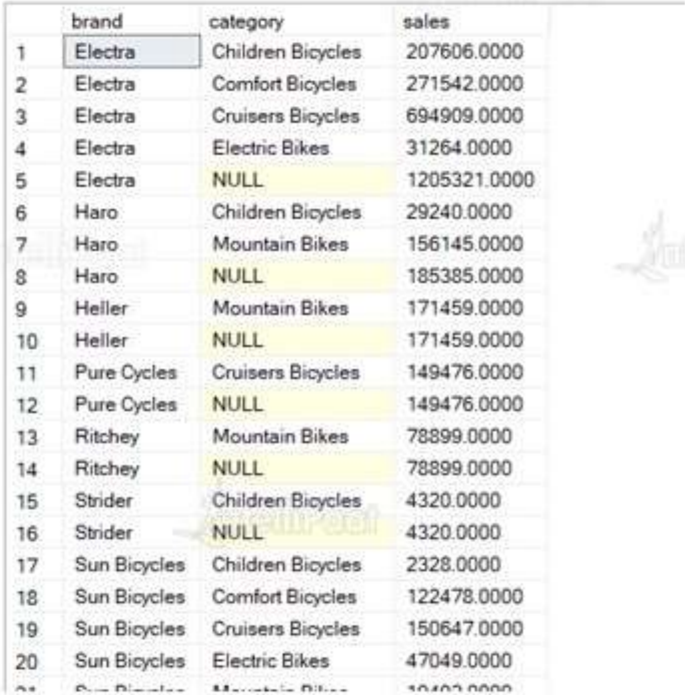
	brand	category	sales
1	Electra	Children Bicycles	207606.0000
2	Haro	Children Bicycles	29240.0000
3	Strider	Children Bicycles	4320.0000
4	Sun Bicycles	Children Bicycles	2328.0000
5	Trek	Children Bicycles	48695.0000
6	NULL	Children Bicycles	292189.0000
7	Electra	Comfort Bicycles	271542.0000
8	Sun Bicycles	Comfort Bicycles	122478.0000
9	NULL	Comfort Bicycles	394020.0000
10	Electra	Cruisers Bicycles	694909.0000
11	Pure Cycles	Cruisers Bicycles	149476.0000
12	Sun Bicycles	Cruisers Bicycles	150647.0000
13	NULL	Cruisers Bicycles	995032.0000
14	Surly	Cyclocross Bicycles	439644.0000
15	Trek	Cyclocross Bicycles	271367.0000
16	NULL	Cyclocross Bicycles	711011.0000
17	Electra	Electric Bikes	31264.0000

CUBE

Below example illustrates how to perform a partial CUBE to reduce the number of grouping sets generated by the query:

```sql
SELECT
    brand,
    category,
    SUM (sales) sales
FROM
    sales.sales_summary
GROUP BY
    brand,
    CUBE(category);
```

Output



	brand	category	sales
1	Electra	Children Bicycles	207606.0000
2	Electra	Comfort Bicycles	271542.0000
3	Electra	Cruisers Bicycles	694909.0000
4	Electra	Electric Bikes	31264.0000
5	Electra	NULL	1205321.0000
6	Haro	Children Bicycles	29240.0000
7	Haro	Mountain Bikes	156145.0000
8	Haro	NULL	185385.0000
9	Heller	Mountain Bikes	171459.0000
10	Heller	NULL	171459.0000
11	Pure Cycles	Cruisers Bicycles	149476.0000
12	Pure Cycles	NULL	149476.0000
13	Ritchey	Mountain Bikes	78899.0000
14	Ritchey	NULL	78899.0000
15	Strider	Children Bicycles	4320.0000
16	Strider	NULL	4320.0000
17	Sun Bicycles	Children Bicycles	2328.0000
18	Sun Bicycles	Comfort Bicycles	122478.0000
19	Sun Bicycles	Cruisers Bicycles	150647.0000
20	Sun Bicycles	Electric Bikes	47049.0000
21	Sun Bicycles	Mountain Bikes	[illegible]

IntelliPaat

India: +91-7847955955

US: 1-800-216-8930 (TOLL FREE)

sales@intellipaat.com

24/7 Chat with Our Course Advisor