


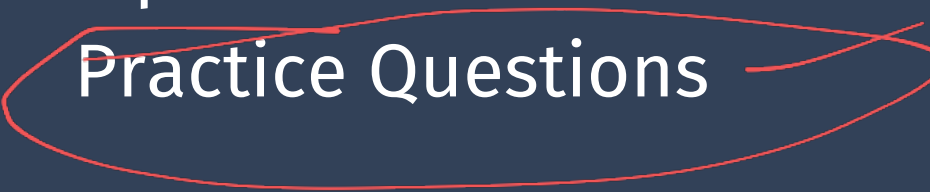


SQL Programming for Data Engineers

Overview for the class

- Topics covered in the prep material
- SQL statements - order of execution 
- JOINS —
- Windows/Analytical functions 
- Tips 
- Practice Questions 

Topics covered in the prep material

- SQL Query Structure
- Set Operations (UNION, INTERSECT, etc.)
- Aggregate functions (sum(), avg(), min(), max(), etc.)
- Datetime functions
- Group by/HAVING clauses
- CASE WHEN statements
- Subqueries

QUESTIONS???



**INTERVIEW
KICKSTART**

Constructing a SQL query

SQL query has the following format:

SELECT

col_A,, etc.

FROM ~~TABLE~~ t1

JOIN some_table t2 **ON** t1.some_column = t2.some_column

WHERE some_column = some_condition

GROUP BY (if aggregation function like **SUM()**, **COUNT()**, **MIN()**, etc. is used in the **SELECT**)

ORDER BY some_column

Dimension
Measure

SUM

ASC
DESC

col 2, col 3

Order of execution of SQL statements

Order	Command	Function
1	From	select the table and also the joins if exists
2	Where	filtering the base data
3	Group by	aggregating the data
4	Having	filtering based on aggregated data
5	Select	returns the final data
6	Order by	sorting the final data
7	Limit	limits the return count on rows

→ i/o

TOP
LIMIT

Order of execution (examples)

```
SELECT product_id FROM products p WHERE p.vendor = 'V1'
```

Order of execution

- the table (products) will be accessed first
- where clause will be used (i.e. filtering on 'V1' vendor)
- Finally the product_id will be selected

```
SELECT product_id, sum(sales) AS total_sales FROM orders WHERE customer_id = 'C1' GROUP BY product_id
```

- The table (orders) will be accessed first
- Filtering will be done on the customer_id (= 'C1')
- Group by (on product_id) will happen for different products which match the above criteria
- Finally, all those product_id will be returned

Order of execution (continued)

```
select product_id, sum(sales) as total_sales from orders where customer_id = 'C1' group by product_id having sum(sales) > 100
```

- The table (orders) will be accessed first
- Filtering will be done on the customer_id (= 'C1')
- Group by (on product_id) will happen for different products which match the above criteria
- Filtering will be done for products having total sales more than 100
- Finally, all those product_id will be returned

```
select product_name, sum(sales) as total_sales from orders o join products p on o.product_id = p.product_id where customer_id = 'C1' group by product_id having sum(sales) > 100
```

- Orders will be joined to products first on product_id
- Filtering will be done on the customer_id (= 'C1')
- Group by (on product_id) will happen for different products which match the above criteria
- Filtering will be done for products having total sales more than 100
- Finally, all those product_id will be returned

Examples continued

There are times when we have to use **distinct** statement in the SQL query.

Ex: if we have given a table with customer_id and some other dimension (as shown below)

Customer_id	Service	Start_date	end_date
C1	netflix	1/1/21	2/1/21
C1	prime	2/1/21	
C2	netflix	1/1/21	

- Find the number of customers who use one or other service?

Output: 2

```
SELECT Count(DISTINCT customer_id) FROM table
```

- Find the number of customers who use netflix service?

Output: 2

```
SELECT Count(DISTINCT customer_id)
FROM table
WHERE service = 'netflix'
```

Please note that the total number of customers is same as customers using netflix service. So, while counting customers, please take care of the dupes.-

Sample question

Date Customer_id

D1 C1

D2 C1

D3 C2

D4 C3

.,...

....

We need to find daily, weekly, monthly customers. In such cases, it is a good idea to have separate queries for each of the different time dimension and then a union all is performed

```
Select date as value, 'daily' as  
time_period, count(distinct  
customer_id) as num_customers from  
table
```

```
Group by 1,2
```

```
UNION ALL
```

```
Select week(date) as value, 'week' as  
time_period, count(distinct  
customer_id) as num_customers from  
table
```

```
Group by 1,2
```

```
UNION ALL
```

```
Select month(date) as value, 'month'  
as time_period, count(distinct  
customer_id) as num_customers from  
table
```

```
Group by 1,2
```



INTERVIEW
KICKSTART

JOINS

- Are used to link 2 or more tables to get desired result
- ~~6 types~~ - Left join, right join, inner join, full join, cartesian (cross) join (avoid this as it join each row of one table to all rows of other table and mostly, the query will run into performance issues) and self join
- LEFT join returns all the rows from the left table (the matching rows based on join condition doesn't matter)
- RIGHT join returns all the rows from the right table (the matching rows based on join condition doesn't matter)
- INNER join returns the matching rows (based on join condition) between 2 or more tables
- FULL join returns all rows from both the tables
- JOINS are heavily used in writing queries, data modeling, ETL. Please make sure that the examples are understood clearly

JOINS

- INNER JOIN
- OUTER JOIN

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

- CROSS JOIN

- SELF JOIN ← NOT a JOIN TYPE!

matching

left

all

Right

all

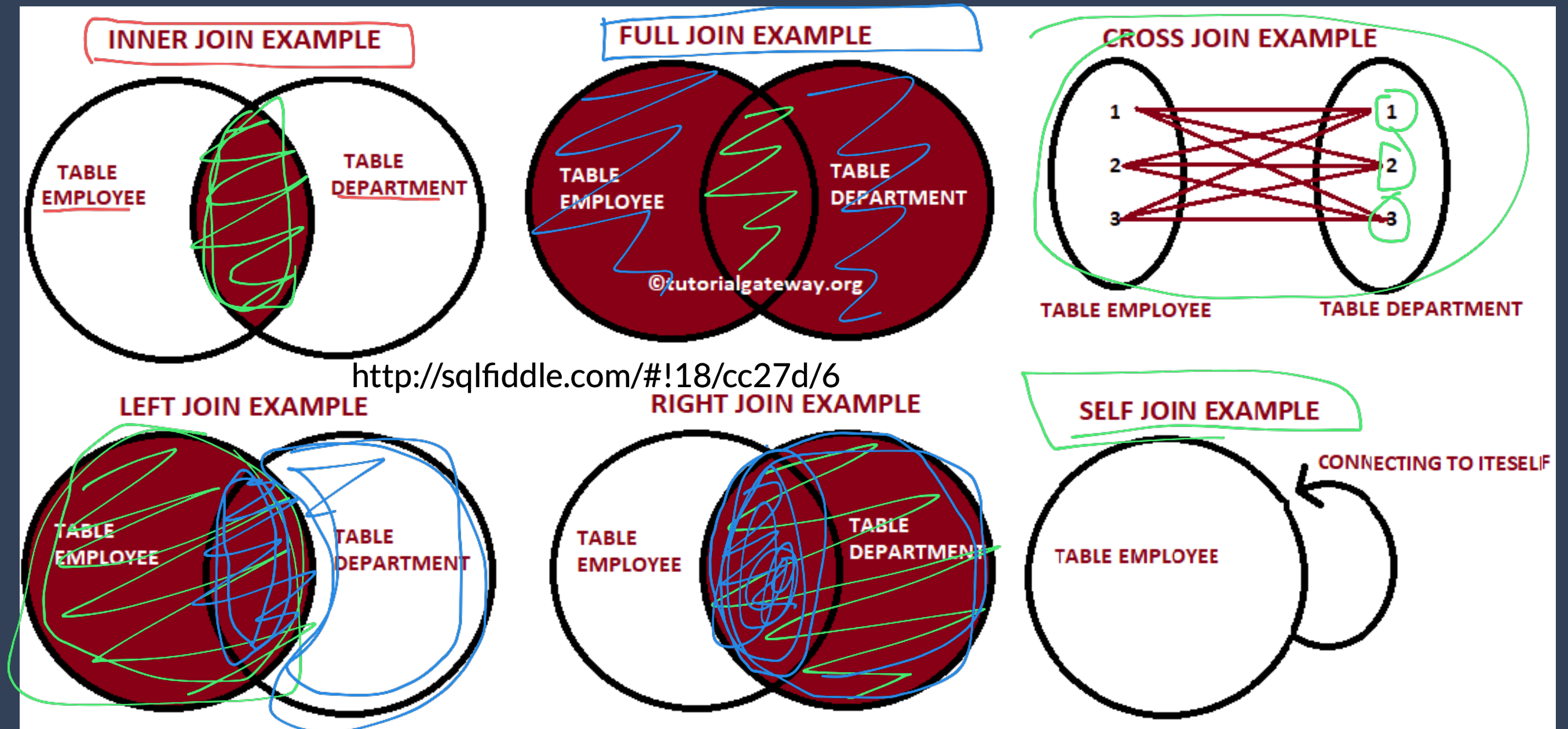
all

NULL

5

CONCEPT

Join interpretation



Source: theartofpostgresql.com

LEFT JOIN SYNTAX/EXAMPLE

```
select t1.col1, t2.col2
```

```
From t1
```

```
left join t2
```

```
on t1.col3 = t2.col3
```

first the join happens based on the join condition. After that, all the rows from t1 will be selected and only the matching rows from t2 will be selected.

RIGHT JOIN SYNTAX/EXAMPLE

```
select t1.col1, t2.col2
```

```
From t1
```

```
right join t2
```

```
on t1.col3 = t2.col3
```

first the join happens based on the join condition. After that, all the rows from t2 will be selected and only the matching rows from t1 will be selected.

INNER JOIN

```
select t1.col1, t2.col2
```

```
From t1
```

```
inner join t2
```

```
on t1.col3 = t2.col3
```

first the join happens based on the join condition. After the join, only the matching rows from both the tables will be selected.

FULL JOIN

```
select t1.col1, t2.col2
```

```
From t1
```

```
full join t2
```

```
on t1.col3 = t2.col3
```

first the join happens based on the join condition. After the join, the matching rows will be selected along with unmatched (uncommon) rows from both the tables

SELF JOIN

WHEN / WHY ?

Example

Let's say there are 2 tables:

Users:

Id (PK)
name

Id	Name
U1	UN1
U2	UN2

User follows:

User_id
User_follow_id

User_id	follower_id	
U1	U2	(U2 follows U1)
U1	U3	(U3 follows U1)
U1	U4	
U2	U1	

Write a query which outputs the number of users users having more than 10 followers and number of users having less than 10 followers

```
SELECT
    CASE
        WHEN num_user_follows > 10
            THEN 'Greater than 10
followers'
        ELSE 'Less than 10 followers'
    END AS followers_bucket,

    Count(user_id) AS num_users
FROM
    (
        SELECT uf.user_id, u.name,
count(uf.user_follow_id) AS
num_user_follows
        FROM user_follows uf
        JOIN Users u ON uf.user_id = u.id
        GROUP BY 1, 2
    )
GROUP BY 1
```


Window/Analytical functions

- These functions are similar to aggregated functions like `sum()`, `max()`, `min()`, etc.
- However, they provide the functionality of aggregated functions along with **window** to the dataset which is being queried
- Querying the dataset specific to different **windows** is a powerful way of slicing and dicing the data in various ways
- Ex: `sum() over (...)`, `max() over ()`, `lead() over ()`, etc.
- Below is an example:
- `sum(any column) over (partition by column1, etc. order by column, etc.)`

Window function (continued)

- Window Functions can be used to create running totals, moving averages and much more. The three main keywords to create a Window Function are:
 - **Over** - Indicates the beginning of a Window Function, this causes the results of the aggregation to be added as a column to the output table.
 - **Partition by** - Creates groups of data in the table, that the aggregation will be performed on
 - **Order by** - Sorts the data based on the given column
- **SELECT** '[aggregate function]' (The column to perform the aggregate function on) **OVER** (Optional: **PARTITION BY** and/or **ORDER BY**) **AS** '(name)' **FROM** '(table)';

Common examples of window functions

Some of the common examples of window functions include:

- Finding the overall top N products
 - use `rank()` over `()`
- Finding the top N products for every month or every city, etc.
 - use `rank()` over(`PARTITION BY` city, etc)
- Finding the previous/next value within a particular window
 - use `lead()/lag()` over()
- Find the rolling sum after every occurrence *use `sum()` over ()

Sample Question:

Let's say we have sales data of certain products for various cities in a month (assume Feb 2021)

We need to find for every city what are the top selling products

Date	Sale	Rank	Dense_Rank	Row_number	LAG	LEAD
1/1/2021	100	1	1	1	NULL	100
1/2/2021	100	1	1	2	100	50
1/3/2021	50	3	2	3	100	25
1/4/2021	25	4	3	4	50	NULL

Sample data in next slide

Ex:

Product_id	Product_name	city	Date	Sales amount
P1	PN1	Portland	2/3/21	100
P1	PN1	Seattle	2/5/21	900
P1	PN1	Seattle	2/8/21	300
P2	PN2	Chicago	2/4/21	200
P3	PN3	Austin	2/5/21	300
P4	PN4	Chicago	2/6/21	400
P5	PN5	Spokane	2/7/21	500
P6	PN6	Austin	2/8/21	600
P7	PN7	Seattle	2/9/21	700
P8	PN8	Portland	2/10/21	800
P9	PN9	Vancouver	2/11/21	900
P10	PN10	Spokane	2/12/21	1000



```
SELECT product_id,  
       product_name,  
       city  
FROM   (SELECT product_id,  
              product_name,  
              city,  
              Row_number()  
                OVER (  
                    partition BY city  
                    ORDER BY total_sales DESC) AS rnk  
FROM   (SELECT product_id,  
              product_name,  
              city,  
              Sum(sales) AS total_sales  
FROM   sales  
GROUP BY 1,  
         2,  
         3))  
WHERE  rnk = 1
```

General tips of approaching a SQL Problem

1. Understand the question and try to get a sense of what is the required output
2. Once the required output is clear, work backwards from there to understand what is to be done to arrive at that metric/dataset
3. Develop a thought process of which tables need to be used or joined to answer the question/parts of questions
4. While developing the thought process, it might require creation of multiple sub-query or temporary tables or CTEs and these will be used later in the query to arrive at final solution
5. Understand the schema and PKs

Some practice questions



INTERVIEW
KICKSTART

JOINS

Let's say there are 2 tables: customers and orders. What is the difference between these 2 queries?

Customers:

customer_id
Customer_name

Orders:

Order_id
Customer_id

```
SELECT c.customer_id
FROM   customers c
       LEFT JOIN orders o
           ON c.customer_id = o.customer_id
           AND o.customer_id = 'c1'
```

```
SELECT c.customer_id
FROM   customers c
       LEFT JOIN orders o
           ON c.customer_id = o.customer_id
WHERE  o.customer_id = 'c1'
```

Sales and exchange rate

Let's say there are 2 tables - sales and exchange rates. The sales table captures the amount of sales taking place along with the currency in which the sale has happened.

Schema for both the tables are:

Sales (sales_date, sales_amount, currency)

Exchange_rate (source_currency, target_currency, exchange_rate, effective_start_date)

Write a query which converts the local currency to USD and displays the rates which are applied (based on the sales_date and effective_start_date)

Sample data:

Sales

sales_date	sales_amount	currency
1/1/20	500	INR
1/1/20	100	GBP
1/2/20	1000	INR
1/2/20	500	GBP
1/3/20	500	INR
1/17/20	200	GBP

Exchange rates:

source_currency	target_currency	exchange_rate	effective_start_date
INR	USD	0.01	12/31/19
INR	USD	0.02	1/2/20
GBP	USD	1.32	12/20/19
GBP	USD	1.3	1/1/20
GBP	USD	1.35	1/16/20



INTERVIEW
KICKSTART

Thank you!!!