Prepare SQL Interview Using Its R Translation

OVERVIEW

Students are usually lack of **Database** experience in the school. However, it is a **must have skill** for **Data Analysts** and **Data Scientists** in the industry. Thus, you would expect it to appear very often in the **Interviews**. This Cheat Sheet aims to help you get fully prepared for those interviews. I will only cover **data query** syntax here. And I will also provide a **dplyr equivalent in R** for your reference, as you can directly **access database use dplyr functions** though with limited capability.

SAMPLE DATA

Table: posts_data

post_id	poster	post_text	post_keywords	post_date
0	2	The Lakers game from last night was great.	[basketball,lakers,nba]	2019/1/1
1	1	Lebron James is top class.	[basketball,lebron_james,nba]	2019/1/2
2	2	Asparagus tastes OK.	[asparagus,food]	2019/1/1
3	1	Spaghetti is an Italian food.	[spaghetti,food]	2019/1/2
4	3	User 3 is not sharing interests	[#spam#]	2019/1/1

Table: posts_view

post_id	viewer_id
4	0
4	1
4	2
5	0
5	1

dplyr setup:

```
library(dplyr)
#I highly recommand to use odbc to connect the database
con=DBI::dbConnect(odbc::odbc, dsn='odbc name')
#access the table using table name and the connection
posts_data = tbl(con,"posts_data")
```

SELECT

SELECT post_id, poster, post_text

FROM posts_data

--select columns from the table

SELECT * --select all columns

R equivalent:

```
posts_data %>% select(post_id, poster, post_text)
posts_data %>% select(everything())
# select() provides more options like below
#  for selecting columns from a big table
posts_data %>% select(post_id:post_text)
posts_data %>% select(!post_id:post_text)
posts_data %>% select(starts_with('post'))
```

WHERE & HAVING clauses

Where and Having are both for filtering rows, while having is only used on data being grouped. And for most of the time, we use having with an aggregate function.

```
SELECT *
FROM posts_data
WHERE post_id = 1
-- post_id in (1,2,3)
-- post_id % 2 =0
SELECT poster, count(post_id) as post_num
FROM posts_data
GROUP BY poster
HAVING count(post_id) > 5
```

R equivalent:

```
posts_data %>% select(everything()) %>%
  filter(post_id == 1)
posts_data %>%
  select(poster, post_id) %>%
  group_by(poster) %>%
  summarise(post_num=n()) %>%
  filter(post_num>5)
```

I assume you are familiar with R. The dplyr package offers similar structure as SQL. You can apply more advanced functions to manipulate your data after you collect data from database. And remember, we are using dplyr commands on a table in database here. The results should be the same.

ORDER BY & LIMIT

ORDER BY helps you sort the resulted data and LIMIT helps you select a limit number of rows.

```
SELECT poster, count(post_id) as post_num
FROM posts_data
GROUP BY poster
HAVING count(post_id) > 5
ORDER BY count(post_id) DESC --ASC
LIMIT 10
```

R equivalent:

```
posts_data %>%
  select(poster, post_id) %>%
  group_by(poster) %>%
  summarise(post_num=n()) %>%
  filter(post_num>5) %>%
  arrange(-n) %>%  #arrange(n) for ASC
  head(10)
```

GROUP BY & Aggregation

GROUP BY is just the same as group_by() in dplyr. The reason we use it is that we want to aggregate the data and return information like mean(), sum(), count(), median(), count(distinct). And I will introduce window function later.

```
SELECT poster,
      count(post_id) as post_num,
      count(distinct post_id),
      --avg(expr)
      min(posts_date),
      max(posts_date),
FROM posts_data
GROUP BY poster
```

R equivalent:

```
posts_data %>%
  select(poster, post_id) %>%
  group_by(poster) %>%
  summarise(post_distinct_num=n_distinct(post_id),
            post_num=n(),
            max(post_date), min(post_date))
```

CASE WHEN

CASE WHEN is a logical operator that helps you create values base on column values.

```
SELECT
   CASE WHEN post_date>='2021-10-01' then 'new post'
   WHEN post_date>='2021-06-01' and post_date<'2021-10-01' then 'not that new'
   ELSE 'old post'
   END as recency
FROM posts_data
```

R equivalent:

posts_data %>% mutate(recency=ifelse(*expr*))

WINDOW FUNCTION

aggregate_fun over (partition by col1 order by col2)

Aggregate function: count, sum, avg, median, max, min, and row_number(), rank(), dense_rank(). e.g., we can achieve cumulative sum by window function.

RANK() and DENSE_RANK() both gives rank to each row, while DENSE_RANK() will use the next chronological ranking value when there are ties and RANK() will skip and create gap in ranking numbers.

ROW_NUMBER() is also a very useful tool for labeling the rows and find the rows you need.

PARTITION BY col1 means the function will start the aggregating when the value of col1 changed.

ORDER BY col2 DESC/ASC refers to the order to apply the aggregation function.

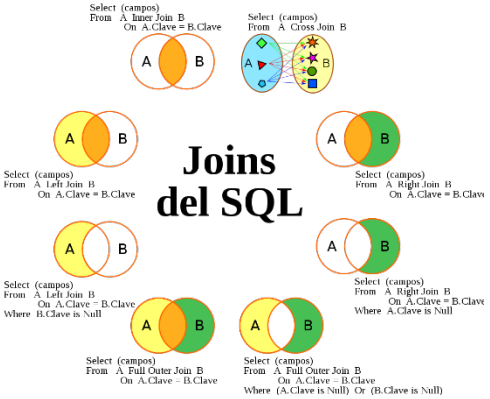
R equivalent:

'Vectorized function' is the term you want to search for. They are fully listed on dplyr cheat sheet.

```
posts_data %>%
  group_by(poster) %>%
  mutate(rn=row_number(post_id)) #row_number(-post_id) for DESC
# OR
posts_data %>%
  group_by(poster) %>%
  mutate(rn=order_by(post_id, row_number(post_id)))
# row_number() over (partition by poster order by post_id ASC)
```

JOIN & UNION

I found the following graph from Wikipedia very helpful to learn SQL joins. It is a little bit more complicated than the joins in dplyr.



R equivalent:

```
left_join(posts_data, posts_view, by=c("post_id"="post_id"))
# dim: 7, 6
inner_join(posts_data, posts_view, by=c("post_id"="post_id"))
# dim: 3, 6
right_join(posts_data, posts_view, by=c("post_id"="post_id"))
# dim: 5, 6
full_join(posts_data, posts_view, by=c("post_id"="post_id"))
# dim: 9, 6
anti_join(posts_data, posts_view, by=c("post_id"="post_id"))
# dim: 4, 5
# anti_join returns all rows in posts_data w/o a match in posts_view
semi_join(posts_data, posts_view, by=c("post_id"="post_id"))
# dim: 1, 5
# semi_join returns all rows in posts_data with a match in posts_view

posts_data %>%
  left_join(posts_view, by=c("post_id"="post_id")) %>%
  filter(!is.na(viewer_id))
# apply filter to achieve some more joins like the picture above
# this is equivalent to the inner_join
```

SAMPLE QUESTION to TRY\

Calculate the percentage of spam posts in all viewed posts by day. Note that the **posts_data** table stores all posts posted by users. The **posts_view** table is an action table denoting if a user has viewed a post.