SQL INTERVIEW CHEAT SHEET

EMMA ding

EMMA ding

emmading.com

info@datainterviewpro.com

SQL Interview Cheat Sheet

There are three main question types in a SQL interview:

1. **Basic Functions**
2. **Complex queries for calculating metrics**
3. **Conceptual questions**

This summary walks you through all three types of questions when preparing for a SQL Interview.

> ✔️ This quick reference lists common pitfalls and important details to keep in mind when working with SQL, making it easier for you to avoid mistakes and write bug-free queries.

> 💯 Practice makes perfect! Speed matters!

Quick Checks

▼ Keywords

Check if the query contains all the necessary keywords

1. `SELECT`
2. `FROM`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

▼ Order of Statements

The order of the statement is the following. Mixing up the statement order will result in a syntax error.

1. `SELECT`
2. `FROM`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

The most common mistake is to have `WHERE` after `GROUP BY`.

```
// ✅ Correct Syntax

SELECT
  COUNT(*)
FROM table1
WHERE date = '2022-01-01'
```

```
// ❌ Wrong Syntax

SELECT
  COUNT(*)
FROM table1
GROUP BY col2
GROUP BY col2
HAVING COUNT(*) > 100;
```

```
WHERE date = '2022-01-01'
HAVING COUNT(*) > 100;
```

Table and Column Names

Make sure the table and columns are EXACTLY the same as in the problem statement, including CAPITALIZATION.

> 💡 Pro tip: copy and paste the table and column names to avoid typos

Table: dropoff_records

id	dropoff_date
1	2022-01-01
2	2022-01-02
…	…

```
// ❌ Wrong table and column names

SELECT dropoff_data
FROM dropoff_record;
```

Parentheses

Check if ALL parentheses are paired.

🔎 Can you spot the mistake in the query?

```
SELECT p.name, d.department_count, s.total_sales
FROM persons AS p
JOIN (
   SELECT
      department_id,
      COUNT(people) AS department_count
   FROM department
   GROUP BY department_id
) AS d
ON d.department_id = p.department_id
JOIN (
   SELECT
      person_id,
      SUM(sales) AS total_sales
   FROM orders
   GROUP BY person_id
AS s
ON s.person_id = p.person_id;
```

Aliases

Don’t forget to create aliases when tables have the same names, i.e. when self-joining tables.

```
// ✅ Correct Syntax

SELECT
  *
FROM
  table1 AS today,
  table1 AS yesterday;
```

```
// ❌ Wrong Syntax

SELECT
  *
FROM
  table1,
  table1;
```

Single Quotes

Use single quotes in queries for strings.

- [**S**]ingle quotes are for [**S**]tring Literals (date literals are also strings).
- [**D**]ouble quotes are for [**D**]atabase Identifiers.

```
// ✅ Correct Syntax

SELECT *
FROM table1
WHERE date = '2022-01-01'
AND status = 'ready';
```

```
// ❌ Wrong Syntax

SELECT *
FROM table1
WHERE date = "2022-01-01"
AND status = "ready";
```

Carefully use Indentation & White spaces

Ident after a keyword, and when you use a subquery or a derived table.

```
// ✅ Correct Syntax

SELECT c.id,
       c.name,
       p.date
FROM customers c
LEFT JOIN (
              SELECT customer_id,
                     MIN(date) as date
              FROM purchases
              GROUP BY customer_id
          ) AS p
            ON p.customer_id = c.id
WHERE c.age <= 30;
```

```
// ❌ Correct Syntax but low readability

SELECT c.id, c.name, p.date
FROM customers c
LEFT JOIN ( SELECT customer_id, MIN(date) as date
 FROM purchases GROUP BY customer_id ) AS p
ON p.customer_id = c.id
WHERE c.age<=30;
```

Go for the ANSI-92 JOIN Syntax (explicit join)

```
// ✅ Correct Syntax

SELECT c.id,
       c.name,
       COUNT(t.id) as count_transactions
FROM customers c
JOIN transactions t ON c.id = t.customer_id
```

```
// ❌ Wrong Syntax

SELECT c.id,
       c.name,
       COUNT(t.id) as count_transactions
FROM customers c, transactions t
WHERE c.id = t.customer_id
WHERE c.age <= 30
GROUP BY 1,2;
```

```
      AND c.age <= 30
GROUP BY 1,2;
```

Basic Functions

▼ SELECT

Add commas between columns. No comma after the last column.

```
// ✅ Correct Syntax

SELECT
  col1,
  col2,
  col3
FROM table1;
```

```
// ❌ Wrong Syntax

SELECT
  col1
  col2,
  col3,
FROM table1;
```

▼ CASE WHEN

Don’t forget the `END` keyword at the end.

```
// ✅ Correct Syntax

SELECT
  CASE
    WHEN score > 95 THEN 'Excellent'
    WHEN score > 80 THEN 'Good'
    WHEN score > 60 THEN 'Fair'
    ELSE 'Poor'
  END
FROM table1;
```

```
// ❌ Wrong Syntax

SELECT
  CASE
    WHEN score > 95 THEN 'Excellent'
    WHEN score > 80 THEN 'Good'
    WHEN score > 60 THEN 'Fair'
    ELSE 'Poor'
FROM table1;
```

👉 **Syntax Details**

- If there is no `ELSE` part and no conditions are true, it returns NULL.
- The `CASE` statement evaluates its conditions **sequentially** and stops with the first condition whose condition is satisfied.

▼ ROUND

Don’t forget to use round for a ratio result to increase the readability.

```
// ✅ Correct Syntax
```

```
// ❌ Correct Syntax but not clean result
```

```
SELECT
  seller,
  ROUND(COUNT(shipped_flag)::FLOAT/COUNT(order_i
d),2) as ship_rate
FROM sales
GROUP BY seller;
```

```
SELECT
  seller,
  COUNT(shipped)::FLOAT/COUNT(shipped_flag) as shi
p_rate
FROM sales
GROUP BY seller;
```

JOIN

- `(INNER) JOIN`: Returns records that have matching values in both tables
- `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table

Don’t forget the `ON` statement unless you are using implicit join.

```
// ✅ Correct Syntax

SELECT
  *
FROM table1 AS t2
JOIN table2 AS t1
ON t2.col1 = t1.col1;
```

```
// ❌ Wrong Syntax

SELECT
  *
FROM table1 AS t2
JOIN table2 AS t1;
```

Multiple Filters

Be careful when combining `OR` with `AND` statements. Use parentheses `()` to combine filters when necessary.

Example:

Select records with the ***status ‘ready’ or ‘shipped’*** on ***2022-01-01***.

```
// ✅ Correct Logic

SELECT
  *
FROM table1
WHERE date = '2022-01-01'
AND (status = 'ready'
OR status = 'shipped');
```

```
// ❌ Wrong Logic

SELECT
  *
FROM table1
WHERE date = '2022-01-01'
AND status = 'ready'
OR status = 'shipped';
```

LIKE

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character

Example:

Find any values that have "94401" in any position

```
// ✅ Correct Syntax

SELECT
  *
FROM table1 AS t1
WHERE address LIKE '%94401%';
```

```
// ❌ Correct Syntax but wrong result

SELECT
  *
FROM table1 AS t1
WHERE address LIKE '_94401_';
```

▼ <>

Find the best combo sell items, use `<>` to filter out the same item in an order.

```
// ✅ Correct Syntax

SELECT t1.item_id,
        t2.item_id,
        COUNT(*)
FROM table1 AS t1
JOIN table1 AS t2
ON t1.order_id = t2.order_id
WHERE t1.item_id <> t2.item_id
GROUP BY 1,2
ORDER BY 3 DESC
LIMIT 1;
```

```
// ❌ Correct Syntax but wrong result

SELECT t1.item_id,
        t2.item_id,
        COUNT(*)
FROM table1 AS t1
JOIN table1 AS t2
ON t1.order_id = t2.order_id
GROUP BY 1,2
ORDER BY 3 DESC
LIMIT 1;
```

> 🎡 You can use either != or <> both in your queries as both are technically the same but <> is preferred as that is SQL-92 standard.

▼ GROUP BY

When mixing aggregate functions (`SUM`, `AVG`, `COUNT`, etc) with an unaggregated column, ensure it's in the `GROUP BY` statement.

```
// ✅ Correct Syntax

SELECT
  COUNT(*),
  city
FROM table1
GROUP BY city;
```

```
// ❌ Correct Syntax but wrong
// result

SELECT
  COUNT(*),
  city
FROM table1;
```

When there are multiple unaggregated columns, ensure ALL are in the `GROUP BY` statement.

```
// ✅ Correct Syntax

SELECT
  COUNT(*),
  city,
  country
FROM table1
GROUP BY city, country;
```

```
// ❌ Wrong Syntax

SELECT
  COUNT(*),
  city,
  country
FROM table1
GROUP BY city;
```

▼ ORDER BY

If you use ORDER BY without any keywords, it is ascending order by default. To sort the records in descending order, use the `DESC` keyword.

Example:

Return score from highest to lowest:

```
// ✅ Correct Syntax

SELECT
  name,
  score
FROM table1
ORDER BY score DESC;
```

```
// ❌ Correct Syntax but wrong result, missing key
word DESC

SELECT
  name,
  score
FROM table1 t1
ORDER BY score;
```

▼ LIMIT N

Example:

Return top 5 scores

```
// ✅ Correct Syntax

SELECT
  name,
  score
FROM table1
ORDER BY score DESC
LIMIT 5;
```

```
// ❌ Correct Syntax in MS SQL Server

SELECT TOP 5
  name,
  score
FROM table1 t1
ORDER BY score DESC;
```

▼ UNION and UNION ALL

`UNION` returns unique rows and `UNION ALL` returns all rows.

Example:

Combine ALL data from table a and table b

Table a **Table b**

id
1
2

id
2
3

```
// ✅ Correct Logic

SELECT *
FROM a
UNION ALL
SELECT *
FROM b;
```

```
// ❌ Wrong Logic

SELECT *
FROM a
UNION
SELECT *
FROM b;
```

Complex Queries

▼ DATE Format

▼ DATE()

> 📆 Check the date column format before using it. If it is `TIMESTAMP` or `VARCHAR`, you need to convert it to `DATE` first. Otherwise, the result will not be aggregated to the date level.

Example:

Aggregate the sales by date:

```
// ✅ Correct Syntax

SELECT
  DATE(timestamp),
  SUM(sales)
FROM table1
GROUP BY 1;
```

```
// ❌ Correct Syntax but wrong result

SELECT
  timestamp,
  SUM(sales)
FROM table1
GROUP BY 1;
```

▼ CURDATE()

Example:

Find out last 7 days page visitor count

```
// ✅ Correct Syntax

SELECT
  pagename,
  COUNT(DISTINCT user_id)
FROM table1
```

```
// ❌ Correct Syntax but hard coding is not preferred

SELECT
  pagename,
  COUNT(DISTINCT user_id)
FROM table1
```

```
WHERE visit_timestamp > CURDATE -7
GROUP BY 1;
```

```
WHERE visit_timestamp BETWEEN 'YYYY-MM-DD' AND
 'YYYY-MM-DD'
GROUP BY 1;
```

▼ DATEDIFF()

🪒 If we want positive results, the first value must be less than the 2nd value.

```
// ✅ Correct Syntax

SELECT
  COUMT(seller_id)
FROM table1
WHERE DATEDIFF(signup_date, order_date) >=7;
```

```
// ❌ Correct Syntax but wrong result, confused
 with start date and end date

SELECT
  COUMT(seller_id)
FROM table1
WHERE DATEDIFF(order_date, signup_date) >=7;
```

🗓️ Go to this link for more detailed info:

https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_datediff

▼ Window function

▼ ROW_NUMBER

`ROW_NUMBER` always returns **unique** rankings.

⚠️ Window function column cannot be called by where clause when it is generated

```
// ✅ Correct Syntax

SELECT dep_name,
        emp_no,
        salary,
        enroll_date
FROM
  (SELECT *,
   rank() OVER (PARTITION BY depname ORDER BY sa
lary DESC, empno) AS pos
   FROM empsalary
  ) AS salary_pos
WHERE pos < 3;
```

```
// ❌ Wrong Syntax

SELECT dep_name,
        emp_no,
        salary,
        enroll_date,
         rank() OVER (PARTITION BY depname ORDER
BY salary DESC, empno) AS pos
FROM empsalary
WHERE pos < 3;
```

▼ RANK/DENSE RANK

`RANK` and `DENSE_RANK` return the rank of values in a column with and without gaps, respectively.

```
// ✅ Correct Syntax

SELECT name, score
FROM
(SELECT name, score,
      RANK() OVER(ORDER BY grades DESC) AS ranking
FROM t1) scores
WHERE ranking <=10;
```

```
// ❌ Correct Syntax but wrong result, missing keyword DESC

SELECT name, score
FROM
(SELECT name, score,
      RANK() OVER(ORDER BY grades) AS ranking
FROM t1) scores
WHERE ranking <=10;
```

LEAD/LAG

`LAG` pulls from previous rows and `LEAD` pulls from following rows.

Example:

Compute MoM growth Rate from a monthly sales table.

```
// ✅ Correct Syntax

SELECT seller,
        years,
        months,
        sales,
        sales/LAG(sales) OVER(PARTITION BY seller ORDER BY years, months)-1 AS growth_rate
FROM table1;
```

```
// ❌ Correct Syntax but wrong result, wrong keyword is used

SELECT seller,
        years,
        months,
        sales,
        sales/LEAD(sales) OVER(PARTITION BY seller ORDER BY years, months)-1 AS growth_rate
FROM table1;
```

COUNT(), SUM(), MIN(), MAX(), MAX()

`COUNT`, `SUM`, `MIN`, `MAX`, and `AVG` can all be used in functions to compute aggregated results of each partition and return the result to every row.

Example:

Return the cumulative balance by account. Use `ORDER BY` for cumulative, otherwise, it returns a total for each row.

```
// ✅ Correct Syntax

SELECT
    dates,
    SUM(Amount) OVER (PARTITION BY Account ORDER BY Months)
FROM
    AccountBalances;
```

```
// ❌ Correct Syntax but wrong result, missing ORDER BY clause

SELECT
    dates,
    SUM(Amount) OVER (PARTITION BY Account)
FROM
    AccountBalances;
```

Common Table Expression (CTE)

- Using CTE improves the readability of your query
- A CTE is defined once then can be referred multiple times

You declare a CTE with the instruction **WITH … AS**:

```
WITH my_cte AS
(
  SELECT col1, col2 FROM table
)
SELECT * FROM my_cte;
```

> 👉 Use "," to separate multiple CTEs

```
// ✅ Correct Syntax

WITH first_purchase AS
(
   SELECT customer_id,
          MIN(date) as date
   FROM purchases
   GROUP BY customer_id
),
persona AS
(
   SELECT age,
          gender,
          AVG(salary) as avg_salary
   FROM customers
   GROUP BY age, gender
)
SELECT c.name,
       p.avg_salary as avg_salary,
       fp.date
FROM customers c
JOIN first_purchase fp ON fp.customer_id = c.id
JOIN persona p ON p.age = p.age
                       AND p.gender = p.gender
WHERE c.age <= 30;
```

```
// ❌ Correct Syntax but low readability

SELECT c.names,
       p.avg_salary as avg_salary,
       fp.date
FROM customers c
JOIN (
          SELECT customer_id,
                 MIN(date) as date
          FROM purchases
          GROUP BY customer_id
      ) AS fp
        ON fp.customer_id = c.id
JOIN (
          SELECT age,
             gender,
             AVG(salary) as avg_salary
         FROM customers
         GROUP BY age, gender
      ) AS p
        ON p.age = c.age
           AND p.gender = c.gender
WHERE c.age <= 30;
```

Conceptual Questions

▼ How to write efficient SQL query

- **Use indexes**
    - Indexes are used to quickly locate rows in a table based on the values in a specific column. Make sure that you have proper indexes on the columns used in your WHERE clause and JOIN conditions, as this can greatly improve the performance of your queries.
- **Avoid using** `SELECT *`

    - Instead of "`SELECT *`", list only the columns you need in your result set. This will reduce the amount of data that needs to be transferred from the database to your application, which can improve performance.

- **Avoid using subqueries**
    - Subqueries can be slow, especially when they are used in the `WHERE` clause. Instead, try to use `JOIN`s or derived tables to achieve the same result.
- **Avoid using functions in the WHERE clause**
    - Functions like `UPPER()`, `LOWER()`, and `CONVERT()` can prevent the use of indexes, as they change the values of the columns. Instead, try to use these functions in the `SELECT` statement, after the data has been filtered.
- **Use proper JOIN conditions**
    - When using JOINs, make sure to specify the correct `JOIN` conditions. Using a wrong join type or missing join conditions can result in a large number of unnecessary rows being returned, which can negatively impact performance.
- **Use `LIMIT` and `OFFSET` clauses**
    - If you only need a limited number of rows from your query, use the `LIMIT` and `OFFSET` clauses to specify the number of rows to return. This can help reduce the amount of data returned, improving performance.
- **Use the appropriate data type**
    - Choose the appropriate data type for each column in your table. For example, using an `INT` data type for a column that only contains numbers with two decimal places may not be the best choice, as it will result in more storage space being used than necessary.

▼ Primary key vs foreign key

PRIMARY KEY	FOREIGN KEY
To ensure data in the specific column is unique.	A column or group of columns in a relational database table provides a link between data in two tables.
Uniquely identifies a record in the relational database table.	It refers to the field in a table which is the primary key of another table.
Only one primary key is allowed in a table.	More than one foreign key is allowed in a table.

▼ [Advanced] SQL Rewrite in Pandas Equivalently

SQL	Pandas
select * from df	**df**
select * from df **limit** 3	df.**head**(3)
select **id** from df **where** name = 'LAX'	df[df.name **==** 'LAX'].**id**

SQL	Pandas
select **distinct** type from df	df.type.**unique**()
select * from df **where** region = 'US-CA' **and** type = 'sea'	df[(df.region == 'US-CA') **&** (df.type == 'sea')]
select **id, name, length** from df **where** region = 'US-CA' **and** type = 'large'	df[(df.region == 'US-CA') **&** (df.type == 'large')][[**'id', 'name', 'length'**]]
select * from df where name = 'LAX' **order by** type	df[df.name == 'LAX'].**sort_values**('type')
select * from df where name = 'LAX' **order by** type **desc**	df[df.name == 'LAX'].**sort_values**('type', **ascending=False**)
select * from df where type **in** ('heliport', 'balloonport')	df[df.type.**isin**(['heliport', 'balloonport'])]
select * from df where type **not in** ('heliport', 'balloonport')	df[**~**df.type.**isin**(['heliport', 'balloonport'])]
select country, type, **count**(*) from df **group by** country, type order by country, type	df.**groupby**(['country', 'type']).**size**()
select country, type, count(*) from df group by country, type **order by** country, count(*) **desc**	df.groupby(['country','type']).size().**to_frame('size').reset_index().sort_values**(['country', 'size'], **ascending=[True, False]**)
select type, count(*) from df where country = 'US' group by type **having** count(*) > 1000 order by count(*) desc	df[df.country == 'US'].groupby('type').filter(**lambda g: len(g) > 1000)**.groupby('type').size().sort_values(ascending=False)
select country from df **order by** size **desc limit** 10	df.**nlargest**(10, 'country')
select country from df **order by** size **desc limit** 10 **offset** 10	df.**nlargest**(20, 'country').**tail**(10)
select **max**(length_ft), **min**(length_ft), **avg**(length_ft), **median**(length_ft) from df	df.agg({'length_ft': [**'min', 'max', 'mean', 'median'**]})
select name, type, description, frequency from df_freq **join** df on df_freq.ref_id = df.id where df.name = 'LAX'	df_freq.**merge**(df[df.name == 'LAX'][['id']], left_on='df_ref', right_on='id', **how='inner'**)[['name', 'type', 'description', 'frequency']]

SQL	Pandas
row_number() over(**partition by** seller_id **order by** order_date)	df.**groupby**([‘seller_id’])[‘order_date’].**rank**(**method=’**first’, ascending = False)
rank() over(partition by seller_id order by order_date)	df.groupby(['seller_id'])['order_date'].**rank**(**method='min'**)
dense_rank() over(partition by seller_id order by order_date)	df.groupby(['seller_id'])['order_date'].**rank(method='dense'**)
sum(amount) over(partition by seller_id, order_month order by order_date rows unbounded preceding)	df.groupby(['seller_id', 'order_month'])['amount'].**cumsum**()
avg(amount) over(partition by seller_id, order_month)	df.groupby(['seller_id', 'order_month'])['amount'].**transform('mean')**.round(1)
lag(sales, 1) over(partition by seller order by date)	df.groupby('seller')['sales'].**shift(-1)**
avg(sales) over(order by date **rows between 6 preceding and current row**)	df['sales'].**rolling(7).mean()**.round(1)