

SQL Basic Interview Questions

1. What is SQL?

SQL (Structured Query Language) is a standard programming language used to communicate with **relational databases**. It allows users to create, read, update, and delete data, and provides commands to define **database schema** and manage database security.

2. What is a database?

A **database** is an **organized collection of data** stored electronically, typically structured in tables with rows and columns. It is managed by a **database management system** (DBMS), which allows for efficient **storage**, **retrieval**, and **manipulation** of data.

3. What are the main types of SQL commands?

SQL commands are broadly classified into:

- **DDL (Data Definition Language)*:* CREATE, ALTER, DROP, TRUNCATE.
- **DML (Data Manipulation Language)*:* SELECT, INSERT, UPDATE, DELETE.
- **DCL (Data Control Language)*:* GRANT, REVOKE.
- **TCL (Transaction Control Language)*:* COMMIT, ROLLBACK, SAVEPOINT.

4. What is the difference between CHAR and VARCHAR2 data types?

- **CHAR:** Fixed-length storage. If the defined length is not fully used, it is padded with spaces.
- **VARCHAR2:** Variable-length storage. Only the actual data is stored, saving space when the full length is not needed.

5. What is a primary key?

A **primary key** is a unique identifier for each record in a table. It ensures that no two rows have the same value in the primary key column(s), and it does not allow NULL values.

6. What is a foreign key?

A **foreign key** is a column (or set of columns) in one table that refers to the primary key in another table. It establishes and enforces a relationship between the two tables, ensuring data integrity.

7. What is the purpose of the DEFAULT constraint?

The **DEFAULT constraint** assigns a default value to a column when no value is provided during an **INSERT operation**. This helps maintain consistent data and simplifies data entry.

8. What is normalization in databases?

Normalization is the process of organizing data in a database to ***reduce redundancy*** and ***improve data integrity***. This involves dividing large tables into smaller, related tables and defining relationships between them to ensure consistency and avoid anomalies.

1NF (第一范式)

1NF要求数据库表中的每个列（字段）都保持“**原子性**”，即每个列中都只能存储单一的数据值，**不能包含多个值（例如数组、列表等）**。所有非主键列必须直接，或者至少间接依赖于，整个主键或主键的一部分。

如果某些非主键列完全不依赖于主键，这将违背关系数据库的基本规范，因为这些列与当前表中的主要实体没有逻辑联系。（0 NF）

2NF (第二范式)

2NF要求在满足1NF的基础上，表中 **所有** 非主键列都必须 **（直接或者间接）完全依赖于主键**。如果主键是由多个列组成的“复合主键”，则所有非主键列必须依赖于复合主键的所有部分，而不能只依赖其中一部分。

注意：传递依赖在第二范式（2NF）* 中是允许的。第二范式的规则只要求* 消除部分依赖，即非主键列必须**完全依赖于主键**，但对**传递依赖**没有限制。因此，满足2NF的表可能依然存在传递依赖。

3NF (第三范式)

3NF要求在**满足2NF的基础上**，所有非主键列都**直接依赖于主键**，而**不能通过其他非主键列间接依赖**。这意味着非主键列之间不应存在传递依赖。

什么是传递依赖？

传递依赖指的是，非主键列通过其他非主键列“间接”依赖于主键，而不是直接依赖。例如：

- 如果表中的一个非主键列A依赖于主键，而另一个非主键列B依赖于A，那么B就通过A“间接”依赖于主键，这就是传递依赖。

在3NF中，**非主键列必须直接依赖于主键**，而不能依赖其他非主键列。因此，如果有传递依赖，就需要进行分解，以消除这种依赖关系。

9. What is denormalization, and when is it used?

Denormalization is the process of combining ***normalized tables*** into larger tables for performance reasons. It is used when ***complex queries*** and joins slow down data retrieval, and the performance benefits outweigh the ***drawbacks of redundancy***.

10. What is a query in SQL?

A query is a SQL statement used to retrieve, update, or manipulate data in a ***database***. The most common type of query is a **SELECT statement**, which fetches data from one or more tables based on specified conditions.

11. What are the different operators available in SQL?

- ***Arithmetic Operators:*** +, -, *, /, %
- ***Comparison Operators:*** =, !=, <>, >, <, >=, <=
- ***Logical Operators:*** AND, OR, NOT

- **Set Operators** UNION, INTERSECT, EXCEPT
- **Special Operators** BETWEEN, IN, LIKE, IS NULL

12. What is a view in SQL?

A view in MySQL is essentially a **stored SQL query** that you can refer to by name. It doesn't store data itself but **acts as a shortcut to run a predefined query**, which means the underlying query is executed whenever you reference the view in a **SELECT** statement or another operation.

People also call it: A *virtual table* based on the result of a SQL query.

Cannot accept parameters.

Does not store data itself — just saves the query definition.

13. What is the purpose of the UNIQUE constraint?

The **UNIQUE constraint** ensures that all values in a column are **different**.

14. What are the different types of joins in SQL?

- **INNER JOIN**
只返回 **两张表中 join condition 能匹配上的行**。
换句话说，**两边都有的数据才会出现**。
- **LEFT JOIN (LEFT OUTER JOIN)**
返回 **left table 的所有行**，
right table 只有在匹配成功时才返回数据，
如果没有匹配，right table 的字段会是 NULL。
- **RIGHT JOIN (RIGHT OUTER JOIN)**
和 **LEFT JOIN 正好相反**，
返回 **right table 的所有行**，
left table 只有匹配成功时才有值，否则为 NULL。
- **FULL OUTER JOIN**
返回 **两张表的所有行**，
只要任意一边能匹配就会出现，
匹配不到的一侧用 NULL 填充。
- **CROSS JOIN**
会生成 **Cartesian product**，
也就是 **第一张表的每一行都会和第二张表的每一行组合一次**。

15. What is the difference between INNER JOIN and OUTER JOIN?

- ***INNER JOIN*** Returns only rows where there is a match in both tables.
- ***OUTER JOIN*** Returns all rows from one table (LEFT, RIGHT, or FULL), and the matching rows from the other table. If there is no match, NULL values are returned for the non-matching side.

16. What is the purpose of the GROUP BY clause?

- **GROUP BY** 不是去重操作，而是将数据按指定的列进行分组。在分组的过程中，所有属性依然存在，只是数据会被按照分组列进行组织。
- **GROUP BY** 常常与聚合函数（如 **COUNT()**, **SUM()**, **AVG()** 等）结合使用，来对每个分组进行计算和汇总。

17. What are aggregate functions in SQL?

Aggregate functions perform calculations on a set of values and return a single value. Common aggregate functions include:

- **COUNT()**: Returns the number of rows.
- **SUM()**: Returns the total sum of values.
- **AVG()**: Returns the average of values.
- **MIN()**: Returns the smallest value.
- **MAX()**: Returns the largest value.

18. What is a subquery?

A **subquery** is a query nested within another query. It is often used in the ***WHERE clause*** to filter data based on the results of another query, making it easier to handle complex conditions.

19. What is the difference between the WHERE and HAVING clauses?

- **WHERE**
用来 在 **GROUP BY** 之前过滤 rows,
它作用于 individual rows,
并且 不能直接使用 aggregate functions，比如 **COUNT**、**SUM**。
- **HAVING**
用来 在 **GROUP BY** 之后过滤 grouped data,
它作用于 groups,
并且 通常和 aggregate functions 一起使用。

```
SELECT department, COUNT(*)
FROM employees
WHERE salary > 50000
GROUP BY department
HAVING COUNT(*) > 5;
```

20. What are indexes, and why are they used?

Index 是一种 database object，
主要作用是 提高 query performance，
让数据库可以 更快地定位和检索 rows，
而不需要对整个 table 做 **full table scan**。

它的工作原理类似 book index :

通过提前建立有序的数据结构,

数据库可以直接跳到目标位置, 而不是逐行查找。

不过, Index 也有代价:

它会占用 additional storage ,

并且在执行 INSERT 、 UPDATE 、 DELETE 等 data modification operations 时,

需要同步维护索引, 因此会带来 额外开销。

一句话总结:

Index 用空间换时间, 提升读性能, 牺牲部分写性能。

21. drop、 delete 与 truncate 的区别?

DROP 是物理删除, 用来删除整张表, 包括表结构, 且不能回滚。

DELETE 支持行级删除, 可以带 WHERE 条件, 可以回滚。

TRUNCATE 用于清空表中的所有数据, 但会保留表结构, 不能回滚。

22. What is the purpose of the SQL ORDER BY clause?

The ORDER BY clause sorts the result set of a query in either ***ascending*** (default) or ***descending order*** , based on one or more columns.

```
SELECT * FROM table_name ORDER BY column_name ASC | DESC;
```

23. What are the differences between SQL and NoSQL databases?

- ***SQL Databases:***
 - Use structured tables with rows and columns.
 - Rely on a fixed schema.
 - Offer ***ACID*** properties.
- ***NoSQL Databases:***
 - Use flexible, schema-less structures (e.g., key-value pairs, document stores).
 - Are designed for horizontal scaling.
 - Often focus on performance and scalability over strict consistency.

24. What is a table in SQL?

A table is a **structured collection** of related data organized into rows and columns. Columns define the type of data stored, while rows contain individual records.

25. What are the types of constraints in SQL?

Common constraints include:

- ***NOT NULL:*** Ensures a column cannot have NULL values.
- ***UNIQUE:*** Ensures all values in a column are distinct.

- ***PRIMARY KEY:*** Uniquely identifies each row in a table.
- ***FOREIGN KEY:*** Ensures referential integrity by linking to a primary key in another table.
- ***CHECK:*** Ensures that all values in a column satisfy a specific condition.
- ***DEFAULT:*** Sets a default value for a column when no value is specified.

26. What is a cursor (游标) in SQL?

`result set` 就是：一条 `SELECT` query 的输出结果。

`Cursor` 是一种 database object，
用于 **逐行地** traverse result set，
也就是 one row at a time 地 `retrieve` 和 `manipulate` 数据。

和普通 SQL 的 set-based operations 不同，
`Cursor` 允许我们 **按顺序 sequentially 处理 rows**，
当业务逻辑 **必须依赖上一行结果** 时，`Cursor` 会比较有用。

不过在实际开发中，
`Cursor` 通常 performance 较差，
因为逐行处理不如 set-based 操作高效，
所以一般 **只在无法用普通 SQL 表达逻辑时才使用**。

🧠 超好记的一句话口诀

`Cursor` = row by row + sequential processing + slower performance

27. What is a trigger in SQL?

A `trigger` is a set of SQL statements that automatically execute in response to certain events on a table, such as `*INSERT*` , `*UPDATE*` , or `*DELETE*` . Triggers help maintain ***data consistency*** , enforce business rules, and implement complex integrity constraints.

28. What is the purpose of the SQL SELECT statement?

The `SELECT` statement retrieves data from one or more tables. It is the most commonly used command in SQL, allowing users to filter, sort, and display data based on specific criteria.

29. What are NULL values in SQL?

`*NULL*` represents a missing or unknown value. It is different from zero or an empty string. NULL values indicate that the data is not available or applicable.

30. What is a stored procedure?

`Stored Procedure` 是一种 database object，
用于在数据库中 **保存并执行一段 SQL logic**，
它本身 **不存储数据**，只存储 programmatic logic。

当你 `EXECUTE` 一个 `Stored Procedure` 时，
里面的代码可以对 table 执行 `SELECT`、`INSERT`、`UPDATE`、`DELETE`，
但数据始终是存放在 table 中，而不是存放在 procedure 里。

可以把 `Stored Procedure` 理解为：
运行在 database 内部的 function，用来封装业务操作逻辑。

🧠 和 `View` 的对比 (这是面试官最爱问的)

- `View`
只保存 query definition，
用于 data representation，
看起来像 table，但本身不存数据 (virtual table)。
- `Stored Procedure`
只保存 logic，
用于 data operations，
更像 function，而不是 table。
- Stored procedure = programmatic logic for operations.

SQL Intermediate Interview Questions

31. What is the difference between DDL and DML commands?

DDL (Data Definition Language)

DDL 用来 定义和修改 database objects 的结构，
关注的是 schema，而不是具体数据。

常见的 DDL commands 包括：

- `CREATE`：创建 table、index、view
- `ALTER`：修改 table 结构
- `DROP`：删除 database object

一句话总结：

DDL 决定数据库“长什么样”。

DML (Data Manipulation Language)

DML 用来 操作 table 中的实际数据，
但 不改变表的结构。

常见的 DML commands 包括：

- `INSERT`：插入 rows
- `UPDATE`：修改 rows
- `DELETE`：删除 rows

一句话总结：

DML 决定数据库“存了什么数据”。

💡 超好记的一句话口诀

DDL = structure / schema

DML = data / rows

💡 你不用背但要看懂的例子

```
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

👉 **DDL** : 定义 table structure

```
INSERT INTO Employees (ID, Name)
VALUES (1, 'Alice');
```

👉 **DML** : 操作 table data

32. What is the purpose of the ALTER command in SQL?

- **ALTER** 是一种 **DDL command**，
用于 **修改已有 database object 的结构**，
而不是创建或删除对象本身。
- 它的主要作用是：
在 **不删除原有数据** 的前提下，
对 **database schema** 进行调整，
以适应业务需求的变化。
- 常见的 **ALTER** 用途包括：
 - 给 table **add 或 drop column**
 - 修改 column 的 **data type**
 - **add 或 remove constraints**
 - **rename table 或 column**
 - 调整 **Index** 等结构性设置
- 一句话总结：
ALTER **用来改结构，不改数据。**

33. What is a composite primary key?

A composite primary key is a primary key made up of two or more columns.

34. How is data integrity maintained in SQL databases?

- ***Constraints:*** Ensuring that certain conditions are always met. For example, **NOT NULL** ensures a column cannot have missing values, **FOREIGN KEY** ensures a valid relationship between tables, and **UNIQUE** ensures no duplicate values.
- ***Transactions:*** Ensuring that a series of operations either all succeed or all fail, preserving data consistency.

- ***Triggers:*** Automatically enforcing rules or validations before or after changes to data.
- ***Normalization:*** Organizing data into multiple related tables to minimize redundancy and prevent anomalies. These measures collectively ensure that the data remains reliable and meaningful over time.

35. What are the advantages of using stored procedures?

- ***Improved Performance:*** Stored procedures are precompiled and cached in the database, making their execution faster than sending multiple individual queries.
- ***Reduced Network Traffic:*** By executing complex logic on the server, fewer round trips between the application and database are needed.
- ***Enhanced Security:*** Stored procedures can restrict direct access to underlying tables, allowing users to execute only authorized operations.
- ***Reusability and Maintenance:*** Once a procedure is written, it can be reused across multiple applications. If business logic changes, you only need to update the stored procedure, not every application that uses it.

36. What is a UNION operation, and how is it used?

UNION 是一种 set operation ,
用于把 多个 **SELECT** queries 的 result sets 合并成一个 result set。

UNION 的特点是：

- 自动去除 duplicate rows
- 每个 **SELECT** 必须 有相同数量的 columns
- 对应 columns 的 data types 必须兼容

可以把 **UNION** 理解为：

把多次查询的结果纵向拼接在一起，并做去重。

Example:

```
SELECT Name FROM Customers
UNION
SELECT Name FROM Employees;
```

37. What is the difference between UNION and UNION ALL?

UNION 和 **UNION ALL** 都是 set operations ,
用于把多个 **SELECT** 的 result sets 合并成一个 result set ,
但它们在 **duplicate handling** 和 **performance** 上有本质区别。

- **UNION**
在合并 result sets 时,
会自动 remove duplicate rows ,
只返回 unique records。
- **UNION ALL**
只负责合并 result sets ,

不会去除 duplicates,
所有 rows 都会被保留下。

在 performance 方面:

UNION ALL 通常更快,

因为它 **不需要额外做 duplicate elimination**。

一句话总结:

要去重用 UNION, 不要去重、追求性能用 UNION ALL。

Example:

```
SELECT Name FROM Customers
UNION ALL
SELECT Name FROM Employees;
```

38. How does the CASE statement work in SQL?

- **CASE** 是 SQL 中用于实现 conditional logic 的语句,
可以在 query 里 **根据不同条件返回不同的值**。
- **CASE** 的执行方式是:
从上到下依次判断 WHEN 条件,
一旦某个条件为 true,
就 **返回对应的结果并停止继续判断**。
- 如果所有 **WHEN** 条件都不满足,
就会执行 **ELSE clause**,
如果没有 **ELSE**, 结果会返回 **NULL**。
- 一句话总结:
CASE 就像 SQL 里的 if-else。

Example:

```
SELECT ID,
CASE
    WHEN Salary > 100000 THEN 'High'
    WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'
    ELSE 'Low'
END AS SalaryLevel
FROM Employees;
```

39. What are scalar functions in SQL?

- **Scalar functions** 是 SQL 中的一类 functions,
它们 **作用在单个值上**,
并且 **返回一个单一的值作为结果**。
- **Scalar functions** 通常用于:

- **formatting data**
 - **data type conversion**
 - 或对单个值做简单计算
- 常见的 **Scalar functions** 包括:
 - **LEN()** : 返回 string 的长度
 - **ROUND()** : 对 numeric value 进行四舍五入
 - **CONVERT()** : 在不同 data types 之间进行转换
- 一句话总结:

Scalar functions = input 一个值, output 一个值。

Example:

```
SELECT LEN('Example') AS StringLength;
```

40. What is the purpose of the COALESCE function?

Purpose: Returns the **first non-NULL value** in a list of expressions.

Why: Useful for handling **NULL** values and providing default fallbacks.

Syntax:

```
COALESCE(expr1, expr2, expr3, ...)
```

→ SQL evaluates expressions in order and returns the first one that is not **NULL**.

```
SELECT COALESCE(MiddleName, FirstName, 'N/A') AS DisplayName
FROM Employees;
```

41. What are the differences between SQL's COUNT() and SUM() functions?

1. COUNT(): Counts the number of rows or non-NULL values in a column.

Example:

```
SELECT COUNT(*) FROM Orders;
```

2. SUM(): Adds up all numeric values in a column.

Example:

```
SELECT SUM(TotalAmount) FROM Orders;
```

42. What is the difference between the NVL (*null value logic*) and NVL2 functions?

NVL(expr1, expr2)

- NVL 的作用是 用指定值替换 NULL。
 - 如果 expr1 是 NULL , 返回 expr2
 - 如果 expr1 不是 NULL , 返回 expr1

一句话理解:

NVL = 给 NULL 一个默认值。

- Example:

```
SELECT NVL(commission, 0) FROM employees;
```

→ If commission is NULL , returns 0 .

NVL2(expr1, expr2, expr3)

- NVL2 在 NVL 的基础上 增加了一层 conditional logic。
 - 如果 expr1 是 NOT NULL, 返回 expr2
 - 如果 expr1 是 NULL, 返回 expr3

一句话理解:

NVL2 = 根据是否为 NULL, 在两个结果中二选一。

- Example:

```
SELECT NVL2(commission, 'Has Commission', 'No Commission') FROM employees;
```

→ If commission is not null → 'Has Commission'

→ If commission is null → 'No Commission'

43. How does the RANK() function differ from DENSE_RANK()?

◆ RANK()

- RANK() 在遇到 tied rows 时,
会给它们 相同的 rank ,
但在并列之后 会跳过排名数字。

可以理解为:

排名值等于“前面有多少个比它大的 rows + 1”。

Example:

```
SELECT name, score,
       RANK() OVER (ORDER BY score DESC) AS rank_num
    FROM students;
```

name	score	RANK()
Ann	100	1
Bob	95	2
Cara	95	2
Dave	90	4

◆ DENSE_RANK()

- DENSE_RANK() 同样会给 tied rows 相同的 rank ,
但在并列之后 不会跳号 ,
排名是 连续递增的。

可以理解为:

排名只关心“有多少个不同的值在它前面”。

Example:

```
SELECT name, score,
       DENSE_RANK() OVER (ORDER BY score DESC) AS dense_rank_num
FROM students;
```

name	score	DENSE_RANK()
Ann	100	1
Bob	95	2
Cara	95	2
Dave	90	3

44. What is the difference between ROW_NUMBER() and RANK()?

ROW_NUMBER()

- Assigns a unique sequential number to each row.
- Does not care about ties: even if two rows have the same value, they get different numbers.
- Commonly used for tasks like pagination or picking the “first” row in a group.

Example:

```
SELECT Name, Salary,
       ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNum
FROM Employees;
```

Name	Salary	RowNum
Alice	1000	1
Bob	1000	2
Carol	900	3
Dave	800	4

Notice how Alice and Bob had the same salary, but they still got **different row numbers**.

RANK()

- Assigns a **rank number based on ordering**, but **ties get the same rank**.
- After a tie, it **skips** numbers (leaves gaps).
- Often used when you want to show competition rankings (e.g., “1st place, 2nd place, 2nd place, 4th place”).

Example:

```
SELECT Name, Salary,
       RANK() OVER (ORDER BY Salary DESC) AS Rank
FROM Employees;
```

Name	Salary	Rank
Alice	1000	1
Bob	1000	1
Carol	900	3
Dave	800	4

Notice how Carol jumped to **rank 3** because Alice and Bob both occupied **rank 1**.

45. What are common table expressions (CTEs) in SQL?

CTE (Common Table Expression) 是一种 **临时的、命名的 query definition**，
用于在 **单条 SQL query 内** 提高 **readability 和可维护性**。

它本身 **不存数据**，

而是由 SQL engine 在执行 query 时

把 CTE 展开并内联到主查询中，

本质上就像一个 **named subquery**。

CTE = temporary named subquery (one query only)

Example:

```

WITH TopSalaries AS (
    SELECT Name, Salary
    FROM Employees
    WHERE Salary > 50000
)
SELECT * FROM TopSalaries WHERE Name LIKE 'A%';

```

⚠ 和 **View** 的关键区别 (面试必考)

- **CTE**
 - lifetime: 只在一条 query 内
 - storage: query definition in memory
 - reuse: 不能跨 query 使用
 - 常用于: 复杂逻辑拆分、recursion
- **View**
 - lifetime: persistent
 - storage: query definition saved in schema
 - reuse: 可以被多个 queries 使用
 - 常用于: 复用逻辑、security abstraction

👉 一句话对比:

CTE 解决“这一条 query 太复杂”，

View 解决“这段逻辑要反复用”。

46. What are window functions, and how are they used?

- **Window functions** 是 SQL 中一类 analytical functions，
它们可以 在一组相关 rows 上做计算，
但仍然为每一行返回一个结果。
- 和 **GROUP BY** 不同的是：
Window functions 不会 collapse rows，
而是 在保留所有 rows 的同时，计算“和当前 row 相关的统计值”。
- 什么是“window”
当前 row 在计算时，允许“看到”的 rows 范围。
- window 是通过 **OVER(...)** 定义的：
- - **PARTITION BY**
→ 把数据分成多个 groups (类似 mini **GROUP BY**)
 - **ORDER BY**
→ 定义 group 内 rows 的顺序，
常用于 **RANK()**、**DENSE_RANK()**、running total
- 一句话总结：

OVER(...) 决定了 window 的范围和顺序。

📝 Example Table (before)

Imagine we have this **Employees** table:

Name	Department	Salary
Alice	Sales	50,000
Bob	Sales	60,000
Charlie	Sales	55,000
Diana	HR	70,000
Eve	HR	65,000

◆ What **GROUP BY** would do

If we try to find the total salary by department:

```
SELECT Department, SUM(Salary)
FROM Employees
GROUP BY Department;
```

Result (rows collapsed):

Department	SUM(Salary)
HR	135,000
Sales	165,000

👉 Notice we **lost the individual employees** — only one row per group remains.

◆ What a Window Function does

Now, if we use a window function:

```
SELECT
  Name,
  Department,
  Salary,
  SUM(Salary) OVER (PARTITION BY Department) AS DeptTotal
FROM Employees;
```

Result (rows preserved, new column added):

Name	Department	Salary	DeptTotal
Alice	Sales	50,000	165,000
Bob	Sales	60,000	165,000
Charlie	Sales	55,000	165,000
Diana	HR	70,000	135,000
Eve	HR	65,000	135,000

👉 Here:

- We still see every employee's row.
- A new column (**DeptTotal**) shows the department's total salary for each row.
- The "window" is defined by **PARTITION BY Department**.

◆ With **ORDER BY** in **OVER**

```
SELECT
    Name,
    Salary,
    SUM(Salary) OVER (ORDER BY Salary) AS RunningTotal
FROM Employees;
```

Result:

Name	Salary	RunningTotal
Alice	50,000	50,000
Charlie	55,000	105,000
Bob	60,000	165,000
Eve	65,000	230,000
Diana	70,000	300,000

含义是：

👉 按照 **Salary** 排序，
对当前 row 及其之前的 rows 计算 cumulative sum (running total)

47. What is the difference between an index and a key in SQL?

1. Index

- **Index** 是一种 data structure，
用于 加快数据查找和访问速度。

- 数据库会在 table 数据之外，
维护一份 **有序结构** (例如 B-Tree) ,
通过指针快速定位 rows,
从而避免 `full table scan`。
- **Index** 的作用是：
 - 加快 `SELECT`
 - 加快 `WHERE / JOIN / ORDER BY`
 - 但会增加 `storage` 和 `write cost`
- 👍 一句话总结：
Index = 用空间换时间，提高查询性能。

Common types:

- **Primary Key index** (automatically created for `PRIMARY KEY`)
 - Always indexed **automatically** → very fast lookups.
The DB guarantees uniqueness and builds an index to enforce it.
- **Unique index** (enforces uniqueness on a column)
 - Always indexed **automatically** → very fast lookups.
The DB guarantees uniqueness and builds an index to enforce it.
- **Non-unique index** (just for faster lookups)
 - Also provide fast lookups (not for uniqueness, but for speeding up searches, filters, joins, ORDER BY, etc.).
- **Composite index** (on multiple columns)
 - Also provide fast lookups (not for uniqueness, but for speeding up searches, filters, joins, ORDER BY, etc.).
- **Foreign Key** →
 - The foreign key constraint itself does **not** automatically create an index in most databases.
- ◆ **How Does It Make Queries Faster?**

1) . 无索引情况

The diagram shows a table with three columns: id, name, and age. The rows contain 10 entries. A large red circle with a diagonal slash is drawn over the first 9 rows, with the Chinese characters '全表扫描' (Full-table Scan) written in red inside it. A red arrow points to the last row (id 10, name 赵敏, age 20), which is highlighted with a red border.

	id	name	age
0x07	1	金庸	36
0x56	2	张无忌	22
0x6A	3	杨逍	33
0xF3	4	韦一笑	48
0x90	5	常遇春	63
0x77	6	小昭	19
0xD1	7	灭绝	45
0x32	8	周芷若	17
0xE5	9	丁敏君	23
0xF2	10	赵敏	20

无索引

在无索引情况下，就需要从第一行开始扫描，一直扫描到最后一行，我们称之为 全表扫描，性能很低。

如果我们针对于这张表建立了索引，假设索引结构就是二叉树，那么也就意味着，会对age这个字段建立一个二叉树的索引结构。



此时我们在进行查询时，只需要扫描三次就可以找到数据了，极大的提高了查询的效率。

备注： 这里我们只是假设索引的结构是二叉树，介绍一下索引的大概原理，只是一个示意图，并不是索引的真实结构，索引的真实结构，后面会详细介绍。

2.3 特点

优势	劣势
提高数据检索的效率，降低数据库的IO成本	索引列也是要占用空间的。
通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗。	索引大大提高了查询效率，同时却也降低更新表的速度，如对表进行INSERT、UPDATE、DELETE时，效率降低。

2. Key

- **Key** 是一种 logical constraint，

用于 **约束数据规则和表之间的关系**,

而不是为了性能。

Key 关注的是:

- rows 如何被 **唯一标识**
- tables 之间如何 **建立关系**
- 如何保证 **data integrity**

常见的 **Key** 包括:

- **Primary Key** : 唯一标识一行 (no duplicates, no NULLs)
- **Unique Key** : 保证唯一性 (通常允许 NULL)
- **Foreign Key** : 保证 referential integrity

👉 一句话总结:

Key = 定义数据规则, 不是优化性能。

- Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,      -- uniquely identifies each employee
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

48. How does indexing improve query performance?

- ◆ How Does It Make Queries Faster?

1) . 无索引情况

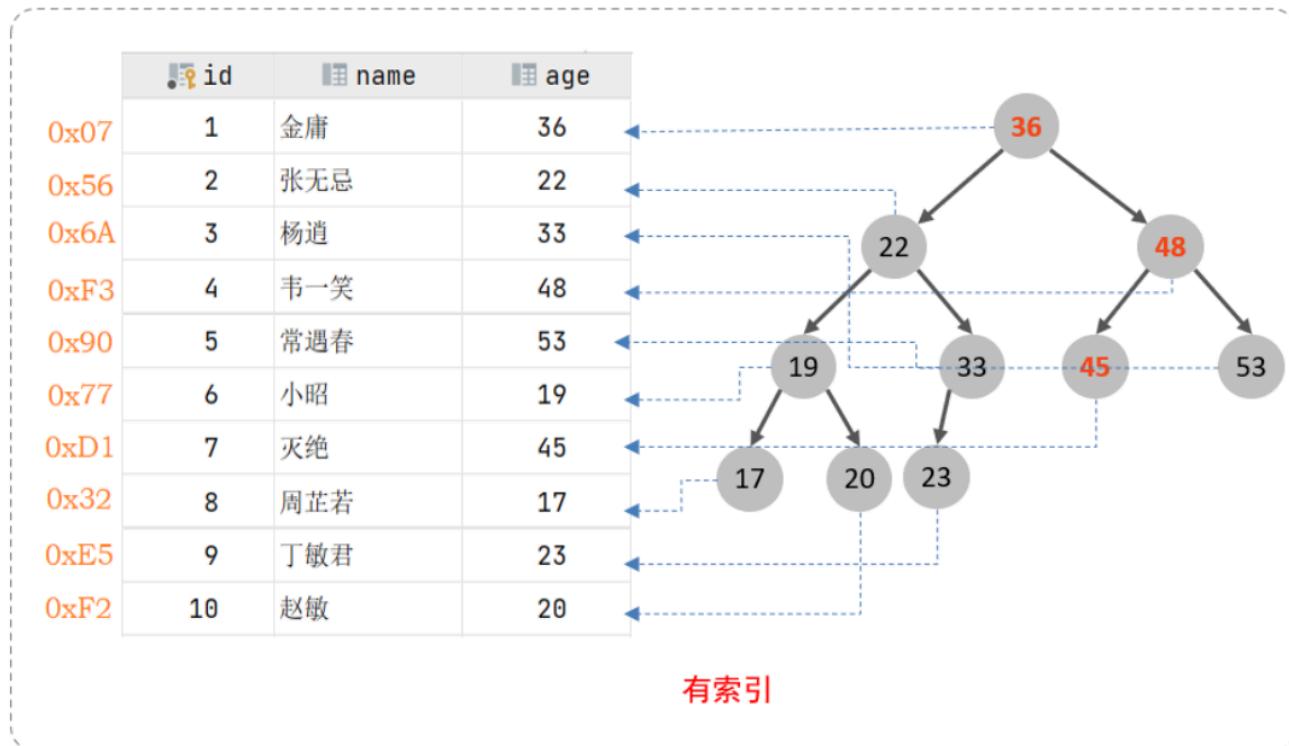
The diagram shows a table with three columns: id, name, and age. The rows contain 10 entries. A large red circle with a diagonal slash is drawn over the first 9 rows, with the Chinese characters '全表扫描' (Full-table Scan) written in red inside it. A red arrow points to the last row (index 10). Below the table, the word '无索引' (No Index) is written in red.

	id	name	age
0x07	1	金庸	36
0x56	2	张无忌	22
0x6A	3	杨逍	33
0xF3	4	韦一笑	48
0x90	5	常遇春	63
0x77	6	小昭	19
0xD1	7	灭绝	45
0x32	8	周芷若	17
0xE5	9	丁敏君	23
0xF2	10	赵敏	20

无索引

在无索引情况下，就需要从第一行开始扫描，一直扫描到最后一行，我们称之为 全表扫描，性能很低。

如果我们针对于这张表建立了索引，假设索引结构就是二叉树，那么也就意味着，会对age这个字段建立一个二叉树的索引结构。



此时我们在进行查询时，只需要扫描三次就可以找到数据了，极大的提高了查询的效率。

备注： 这里我们只是假设索引的结构是二叉树，介绍一下索引的大概原理，只是一个示意图，并不是索引的真实结构，索引的真实结构，后面会详细介绍。

2.3 特点

优势	劣势
提高数据检索的效率，降低数据库的IO成本	索引列也是要占用空间的。
通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗。	索引大大提高了查询效率，同时却也降低更新表的速度，如对表进行INSERT、UPDATE、DELETE时，效率降低。

Example:

```

CREATE INDEX idx_lastname ON Employees(LastName);
SELECT * FROM Employees WHERE LastName = 'Smith';

```

The index on `Lastname` lets the database quickly find all rows matching ‘Smith’ without scanning every record.

49. What are the trade-offs of using indexes in SQL databases?

2.3 特点

优势	劣势
提高数据检索的效率，降低数据库的IO成本	索引列也是要占用空间的。
通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗。	索引大大提高了查询效率，同时却也降低更新表的速度，如对表进行INSERT、UPDATE、DELETE时，效率降低。

50. What is the difference between clustered and non-clustered (secondary) indexes?

2.3.2 聚集索引&二级索引

而在InnoDB存储引擎中，根据索引的存储形式，又可以分为以下两种：

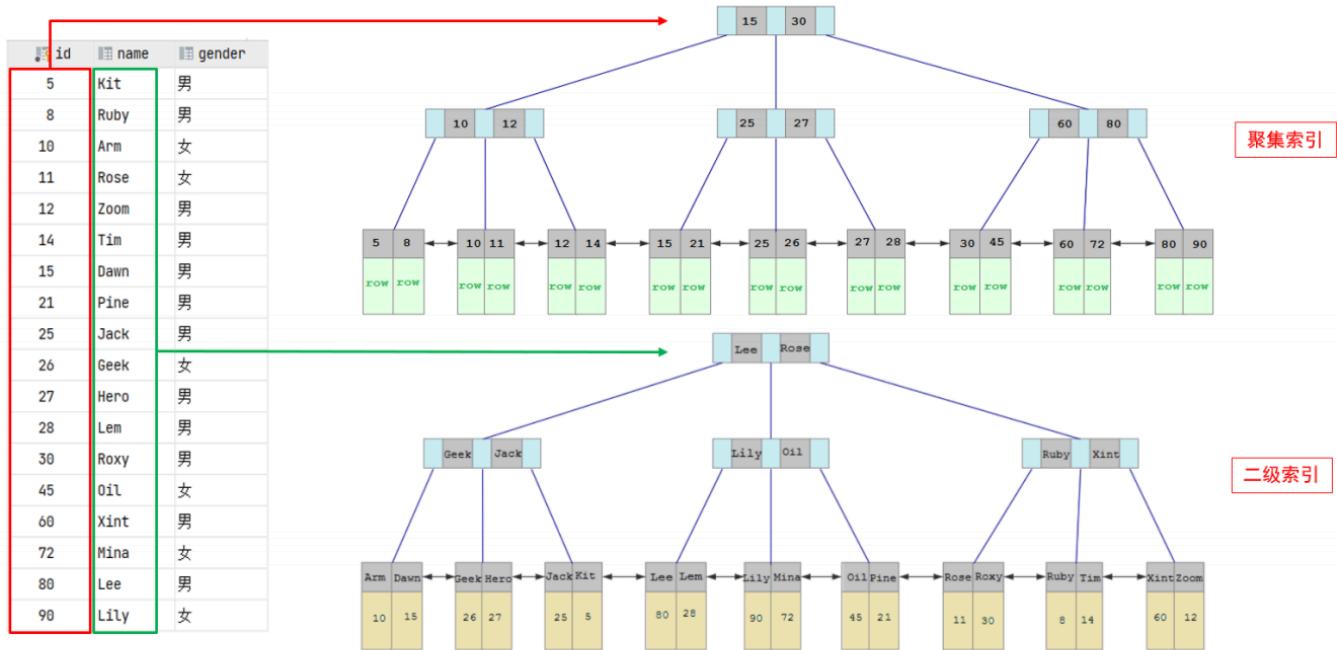
分类	含义	特点
聚集索引(Clustered Index)	将数据存储与索引放到了一块，索引结构的叶子节点保存了行数据	必须有，而且只有一个
二级索引(Secondary Index)	将数据与索引分开存储，索引结构的叶子节点关联的是对应的主键	可能存在多个

聚集索引选取规则：

- 如果存在主键，主键索引就是聚集索引。

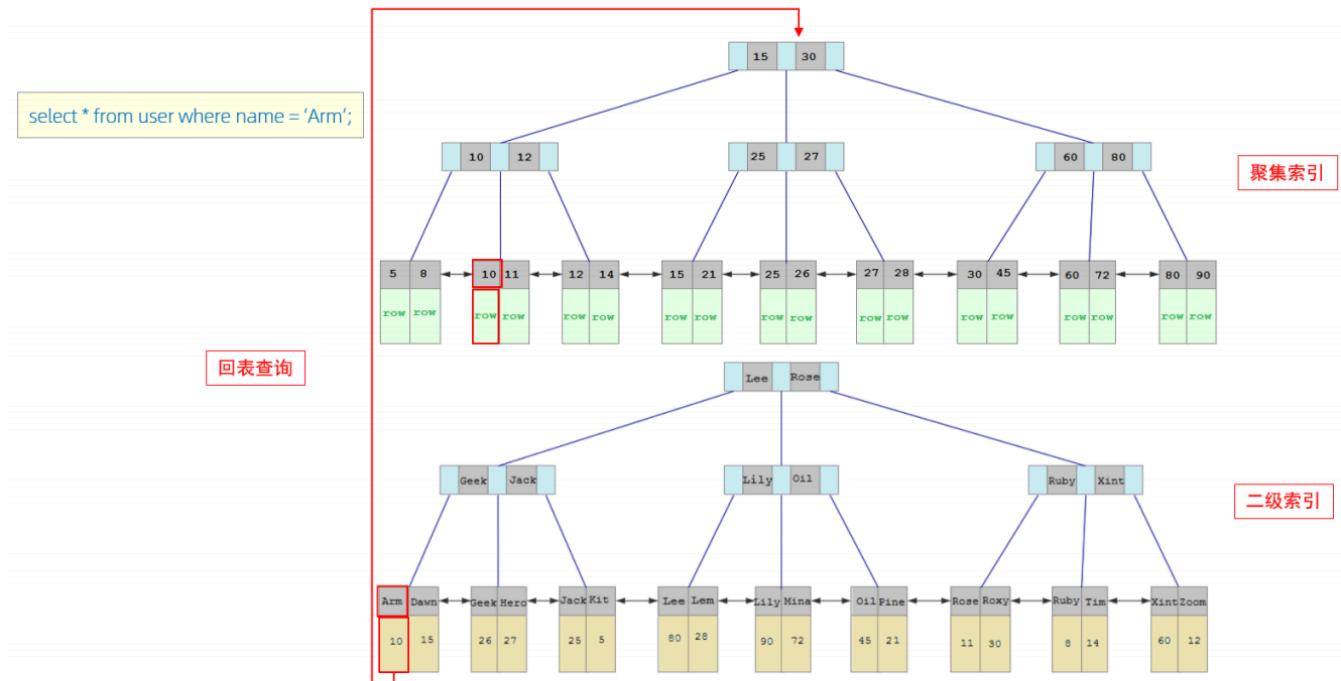
- 如果不存在主键，将使用第一个唯一（UNIQUE）索引作为聚集索引。
- 如果表没有主键，或没有合适的唯一索引，则InnoDB会自动生成一个rowid作为隐藏的聚集索引。

聚集索引和二级索引的具体结构如下：



- 聚集索引的叶子节点下挂的是这一行的数据。
- 二级索引的叶子节点下挂的是该字段值对应的主键值。

接下来，我们来分析一下，当我们执行如下的SQL语句时，具体的查找过程是什么样子的。



具体过程如下：

- ①. 由于是根据name字段进行查询，所以先根据`name='Arm'`到name字段的二级索引中进行匹配查找。但是在二级索引中只能查找到 Arm 对应的主键值 10。
- ②. 由于查询返回的数据是*，所以此时，还需要根据主键值10，到聚集索引中查找10对应的记录，最终找到10对应的行row。
- ③. 最终拿到这一行的数据，直接返回即可。

回表查询：这种先到二级索引中查找数据，找到主键值，然后再到聚集索引中根据主键值，获取数据的方式，就称之为回表查询。

51. What are temporary tables, and how are they used?

Temporary tables 是一种临时存在的 tables，只在 session 或 transaction 的生命周期内有效，常用于存储中间结果，而不会影响正式的 production tables。

它们的主要用途包括：

- 保存 intermediate results

- 简化复杂 queries
- 对部分数据做临时计算或转换

一句话总结：

Temporary tables 用来临时存数据，查询结束后自动消失。

一、 Local Temporary Tables

- 以 # 开头 (例如 `#TempTable`)
- 只对创建它的 session 可见
- session 结束时 自动 drop

👉 适合 单个 session 内的中间处理逻辑。

二、 Global Temporary Tables

- 以 ## 开头 (例如 `##GlobalTempTable`)
- 对所有 sessions 可见
- 当 所有引用它的 sessions 结束后才会 drop

👉 适合 多个 sessions 共享临时数据 的场景。

🧠 超好记的一句话口诀

```
#TempTable → one session  
##TempTable → all sessions
```

Example:

```
CREATE TABLE #TempResults (ID INT, Value VARCHAR(50));  
INSERT INTO #TempResults VALUES (1, 'Test');  
SELECT * FROM #TempResults;
```

52. What is a materialized view, and how does it differ from a standard view?

一、 Standard View

Standard View 是一个 virtual table，
本身 不存储数据，
只保存一条 query definition。

每次查询 **Standard View** 时：

- 底层的 `SELECT` query 都会重新执行
- 返回的是 real-time data
- 读取速度取决于原始 tables 的复杂度

一句话总结:

Standard View = saved query, 不存 data。

二、Materialized View

Materialized View 是一个 physical table,
会实际存储 query 的执行结果。

它的特点是:

- 数据是 precomputed 并存储的
- 查询时 不需要重新跑复杂 query
- 因此 read performance 非常快
- 但数据 可能不是最新的

为了保持数据更新,

Materialized View 需要定期执行 refresh,
可以是手动或 scheduled。

一句话总结:

Materialized View = saved data, 用性能换 freshness。

🧠 超好记的一句话口诀 (必背)

View : query 每次跑

Materialized View : 结果先存好

53. What is a sequence in SQL?

Sequence 是一种 database object,

用于 生成一系列唯一的 numeric values ,

常用于给 Primary Key 或其他需要唯一值的 column 提供 ID。

Sequence 的特点是:

- 每次调用都会返回一个新的值
- 生成的值 不依赖 table
- 即使 transaction rollback, 已生成的值 通常也不会回退

一句话总结:

Sequence 是一个独立的“数字生成器”，专门用来产生唯一 ID。

💡 你不用背但要看懂的例子

```
CREATE SEQUENCE seq_emp_id
START WITH 1
INCREMENT BY 1;
SELECT NEXT VALUE FOR seq_emp_id; -- 1
SELECT NEXT VALUE FOR seq_emp_id; -- 2
```

解释：

每次调用 `NEXT VALUE FOR` ,
`Sequence` 都会返回一个新的唯一值。

54. What are the advantages of using sequences over identity columns?

1. Greater Flexibility:

- `Sequence` 可以自由配置：
 - `START WITH`
 - `INCREMENT BY`
 - `MAXVALUE / MINVALUE`

而 `Identity Column` 的生成规则
通常固定在 table 上，灵活性较低。

2. Dynamic Adjustment: Can alter the sequence without modifying the table structure.

- `Sequence` 可以通过 `ALTER SEQUENCE`
- 直接修改生成规则，
- 而 不需要改 table schema。

3. Cross-Table Consistency:

- `Sequence` 是一个 独立的 database object ,
- 可以被 多个 tables 共享使用 ,
- 从而保证 跨表的唯一性。

55. How do constraints improve database integrity?

`Constraints` 用于 强制数据必须遵守一组规则，
从而防止 invalid 或 inconsistent data 被写入数据库，
是保证 database integrity 的核心机制。

• 不同类型的 `Constraints` 从不同层面约束数据：

- `NOT NULL`
确保 column 不能存 NULL ,
防止缺失关键数据。
- `UNIQUE`
确保 column 中的值 不重复 ,
防止出现重复记录。
- `PRIMARY KEY`
同时具备 `NOT NULL + UNIQUE` ,
保证每一行 都能被唯一标识。
- `FOREIGN KEY`

通过要求引用值必须存在于另一张 table 中，
来保证 referential integrity，
防止出现“孤儿数据”。

- **CHECK**

用于验证 column 的值 **是否满足业务规则**，
比如 `CHECK (Salary > 0)`。

56. What is the difference between a local and a global temporary table?

- Local Temporary Table:

- Prefixed with `#` (e.g., `#TempTable`).
- Exists only within the session that created it.
- Automatically dropped when the session ends.

- Global Temporary Table:

- Prefixed with `##` (e.g., `##GlobalTempTable`).
- Visible to all sessions.
- Dropped only when all sessions referencing it are closed.

Example:

```
CREATE TABLE #LocalTemp (ID INT);
CREATE TABLE ##GlobalTemp (ID INT);
```

57. What is the purpose of the SQL MERGE statement?

MERGE 是 SQL 中的一个 **single statement**，
用于 **根据 source 和 target 之间的匹配关系**，
在 target table 中 **执行 INSERT、UPDATE，甚至 DELETE 操作**。

MERGE 的核心思想是：

把“判断是否存在 + 对应操作”合并成一条语句，
因此也常被称为 **upsert** (**update + insert**)。

在使用 **MERGE** 时：

- 如果 source 和 target **匹配** (`WHEN MATCHED`)，可以执行 `UPDATE` 或 `DELETE`
- 如果 **不匹配** (`WHEN NOT MATCHED`)，可以执行 `INSERT`

一句话总结：

MERGE 用一条语句完成“存在就更新，不存在就插入”的逻辑。

💡 超好记的一句话口诀（必背）

```
MERGE = match → update, no match → insert
```

💡 你不用背但要看懂的例子

```

MERGE INTO TargetTable T
USING SourceTable S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.Value = S.Value
WHEN NOT MATCHED THEN
    INSERT (ID, Value) VALUES (S.ID, S.Value);

```

解释:

- **ON** : 定义 source 和 target 的匹配条件
- **WHEN MATCHED** : 已有 row → **UPDATE**
- **WHEN NOT MATCHED** : 新 row → **INSERT**

58. How can you handle duplicates in a query without using DISTINCT?

*1. **GROUP BY:*** Aggregate rows to eliminate duplicates

```

SELECT Column1, MAX(Column2)
FROM TableName
GROUP BY Column1;

```

2. ***ROW_NUMBER():*** Assign a unique number to each row and filter by that

```

WITH CTE AS (
    SELECT Column1, Column2, ROW_NUMBER() OVER (PARTITION BY Column1 ORDER BY Column2) AS RowNum
    FROM TableName
)
SELECT * FROM CTE WHERE RowNum = 1;

```

59. What is a correlated subquery?

A **normal subquery** runs once, produces a result set, and the outer query uses it.

A **correlated subquery** is different:

- It depends on values from the outer query.
- It is executed once per row of the outer query.

Example:

```

SELECT Name,
    (SELECT COUNT(*)
     FROM Orders
     WHERE Orders.CustomerID = Customers.CustomerID) AS OrderCount
FROM Customers;

```

◆ Example Tables

Customers

CustomerID	Name
1	Alice
2	Bob
3	Charlie

Orders

OrderID	CustomerID
101	1
102	1
103	2

◆ Result of Query

Name	OrderCount
Alice	2
Bob	1
Charlie	0

60. What are partitioned tables, and when should we use them?

- **Partitioned tables** 是指把一张 **大 table** 按照某个 **partition key** 的取值，拆分成多个 **独立的 partitions** 来存储。
- 每个 partition 只包含 **一部分 rows**，但在逻辑上它们 **仍然属于同一张 table**。
- 这样做的核心目的，是：
减少扫描的数据量，提高 query performance，并降低大表的维护成本。

💡 常见的 partition 类型 (知道名字就够)

- **RANGE** : 按数值或时间范围分区 (最常见)
- **LIST** : 按枚举值分区
- **HASH / KEY** : 按 hash 规则均匀分布数据

☰ 你不用背但要看懂的例子 (Range partition)

```
PARTITION BY RANGE (YEAR(SaleDate)) (
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025)
);
```

解释：

- 每一年数据进一个 partition
- 查某一年，只扫描对应 partition
- 删除旧年份，只需 drop partition

SQL Advanced Interview Questions

61. What are the ACID properties of a transaction? 事务四大特性

Atomicity (原子性)

Atomicity 表示一个 transaction 是 **不可分割的最小单元**，
要么 **所有操作全部成功并 commit**，
要么 **出现错误就全部 rollback**，
不会只成功一部分。

👉 一句话：

transaction 要么全做，要么全不做。

Consistency (一致性)

Consistency 表示 transaction **开始前和结束后**，
数据库都必须处于 **合法、一致的状态**，
所有 **constraints、rules** 都必须被满足。

👉 一句话：

transaction **不能破坏数据库规则**。

Isolation (隔离性)

Isolation 表示多个 transactions **并发执行时互不干扰**，
每个 transaction 都像是在 **独立环境中运行**。

数据库通过不同的 **isolation levels**
来控制 transaction 之间的可见性。

👉 一句话：

transaction **之间“互相看不见”**。

Durability (持久性)

Durability 表示 transaction 一旦 commit,

它对数据的修改就是 **永久的**,

即使发生 crash 或重启, 数据也不会丢失。

👉 一句话:

commit 之后, 数据一定在。

62. What are the differences between isolation levels in SQL?

Isolation levels 用来定义 一个 transaction 在并发环境中, 能看到其他 transactions 多少影响,

一、并发事务的三类经典问题 (先记这个)

- Dirty Read
 - 👉 读到了 未 commit 的数据
- Non-Repeatable Read
 - 👉 同一行数据 在同一个 transaction 中, 两次读取结果不同
- Phantom Read
 - 👉 同一个查询条件, 后一次读多出或少了 rows

二、四种 **Isolation levels** (从弱到强)

1 Read Uncommitted (最低)

- 允许 Dirty Read
- 一个 transaction 可以读到另一个未 commit 的修改
- 几乎没有隔离, concurrency 最高

👉 一句话:

什么都挡不住, 性能最好, 几乎不用。

2 Read Committed

- ✗ Dirty Read
- ✓ 只能读到 已 commit 的数据
- 但同一行数据 两次读可能不同

👉 会发生:

Non-Repeatable Read

👉 一句话:

看不到脏数据, 但结果可能变。

3 Repeatable Read

- ✗ Dirty Read
- ✗ Non-Repeatable Read
- 保证 读过的 rows 不会变

但:

- 可能出现 Phantom Read

- 新插入、满足条件的 rows 可能“突然出现”

👉 一句话：

旧数据不变，新数据可能冒出来。

4 Serializable (最高)

- ✗ Dirty Read
- ✗ Non-Repeatable Read
- ✗ Phantom Read

实现方式是：

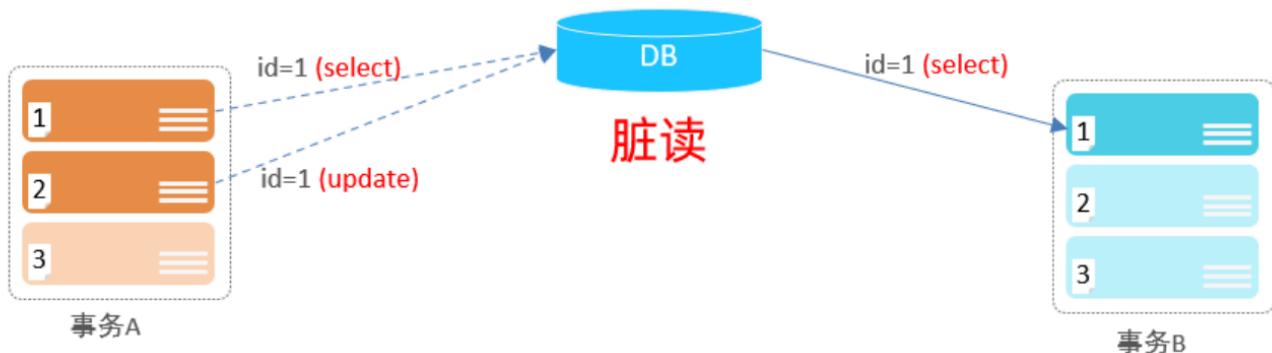
- 像是 transactions 串行执行
- 通过 range lock 阻止其他 transaction 插入或修改相关数据

👉 一句话：

最安全，但 concurrency 最低。

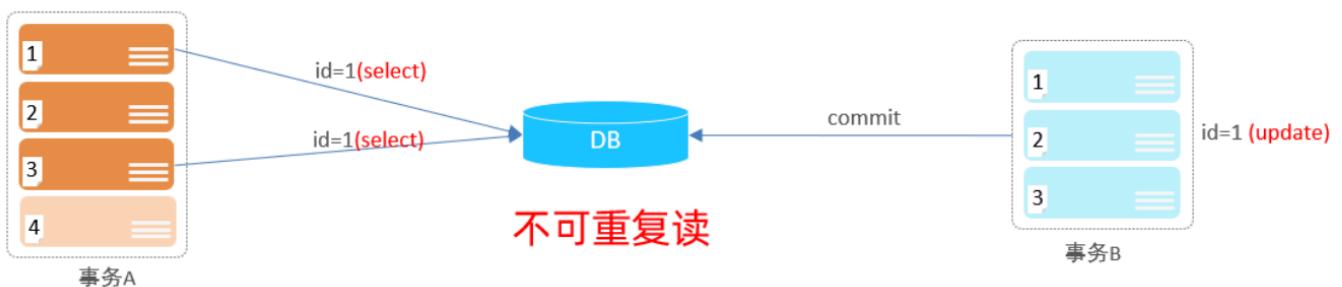
并发事务问题

- 1) 脏读：一个事务读到另外一个事务还没有提交的数据。



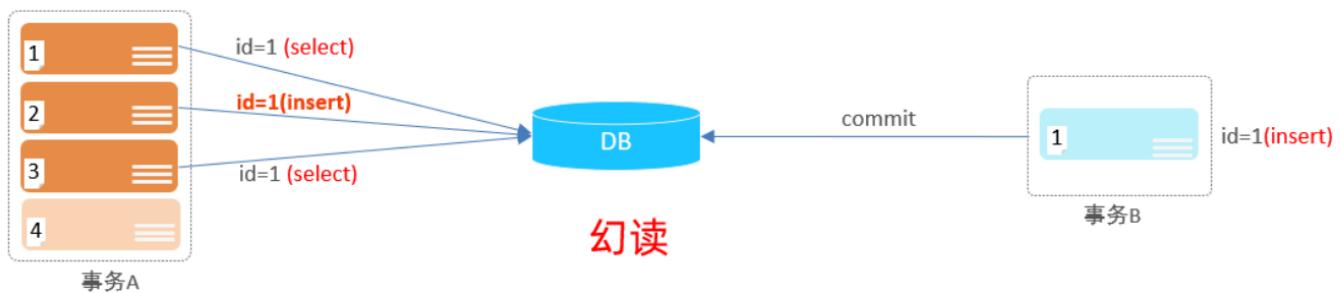
比如 B 读取到了 A 未提交的数据。

- 2) 不可重复读：一个事务先后读取同一条记录，但两次读取的数据不同，称之为不可重复读。



事务 A 两次读取同一条记录，但是读取到的数据却是不一样的。

3). 幻读：一个事务按照条件查询数据时，没有对应的数据行，但是在插入数据时，又发现这行数据已经存在，好像出现了“幻影”。



为了解决并发事务所引发的问题，在数据库中引入了事务隔离级别。主要有以下几种：

隔离级别	脏读	不可重复读	幻读
<code>Read uncommitted</code>	√	√	√
<code>Read committed</code>	✗	√	√
<code>Repeatable Read (默认)</code>	✗	✗	√
<code>Serializable</code>	✗	✗	✗

Isolation levels define the extent to which the operations in one `transaction` are isolated from those in other transactions. They are critical for *managing concurrency* and ensuring data integrity. Common isolation levels include:

- ◆ Setup

We have a table:

Orders

OrderID	Amount
1	120
2	80
3	150

Transaction A: Wants to read all orders where `Amount > 100`.

Transaction B: Will insert or update orders during A's work.

1. Read Uncommitted

Transaction A

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Reads rows 1 and 3.

Transaction B (not committed yet)

```
UPDATE Orders SET Amount = 200 WHERE OrderID = 2; -- from 80 → 200
```

Transaction A (still running)

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Now sees row 2 too, **even though B hasn't committed yet**.

If B rolls back, A saw data that never really existed = **Dirty Read**.

✓ Both can finish independently (A doesn't wait for B).

2. Read Committed

Transaction A (first query)

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Reads rows 1 and 3.

Transaction B

```
UPDATE Orders SET Amount = 200 WHERE OrderID = 2;  
COMMIT;
```

Transaction A (second query)

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Now sees row 2 also, but only **after B committed**.

⚠ Same row gave different results between A's first and second read → **Non-Repeatable Read**.

✓ A never sees uncommitted data, but results change mid-transaction.

3. Repeatable Read

Transaction A (first query)

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Reads rows 1 and 3.

Transaction B

```
UPDATE Orders SET Amount = 200 WHERE OrderID = 2;  
COMMIT;
```

Transaction A (second query)

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Still sees **only rows 1 and 3** (row 2 is “invisible” until A finishes).

No dirty reads, no non-repeatable reads.

But...

If B does:

```
INSERT INTO Orders VALUES (4, 180);  
COMMIT;
```

Then when A runs the query again, it now sees row 4 as well → **Phantom Row**.

Rows A touched don't change, but *new rows matching the condition* can sneak in.

4. Serializable

Transaction A (first query)

```
SELECT * FROM Orders WHERE Amount > 100;
```

👉 Reads rows 1 and 3. Database also places a **range lock** on “all rows where Amount > 100.”

Transaction B tries:

```
INSERT INTO Orders VALUES (4, 180);
```

👉 Blocked until A finishes because that would change A's result set.

Transaction A (second query)

👉 Still sees rows 1 and 3. No new rows can appear.

Prevents dirty reads, non-repeatable reads, and phantom reads — but at cost of concurrency.

63. What is the purpose of the WITH (NOLOCK) hint in SQL Server?

- **WITH (NOLOCK)** 是 SQL Server 中的一种 table hint,
用于告诉数据库:
在读取数据时不获取 shared locks。

使用 **WITH (NOLOCK)** 时,
query 的行为等同于 **READ UNCOMMITTED** isolation level,
也就是说 **可以读取未 commit 的数据**。

它的核心目的, 是:
减少锁竞争, 提高并发查询的 performance。
- **查询 不会被其他 transactions 阻塞**

在 **large table + high concurrency** 场景下
可以明显降低等待时间

Downsides (the trade-off)

- You might read **dirty data** (data that another transaction hasn't committed yet).

Example:

```
SELECT *
FROM Orders WITH (NOLOCK);
```

This query fetches data from the `Orders` table without waiting for other transactions to release their locks.

64. How do you handle deadlocks in SQL databases?

Deadlock 指的是 **两个或多个 transactions 互相等待对方持有的 locks**，
形成一个 **循环依赖**，
导致所有相关 transactions **都无法继续执行**。

◆ **What is a deadlock?**

Transaction A **持有 lock 1 等 lock 2**，
Transaction B **持有 lock 2 等 lock 1**，
双方都卡住 → deadlock。

◆ **How databases handle deadlocks (built-in)**

- 大多数数据库（如 SQL Server、MySQL InnoDB、PostgreSQL、Oracle）
都内置 **deadlock detector**。

当数据库检测到 **lock dependency cycle** 时，会：

1. 选择一个 transaction 作为 **victim**
2. **rollback** 这个 transaction
3. 释放 locks，让其他 transaction 继续执行

👉 一句话总结：

数据库自动“牺牲一个”，系统不会真的卡死。

◆ **What developers can do to reduce deadlocks**

1. **Keep transactions short**
 - The longer a transaction holds locks, the higher the chance of conflict.
2. **Use indexes / efficient queries**
 - If the query touches fewer rows, fewer locks are needed.
3. **Access resources in a consistent order**
 - For example, always update `Customer` before `Orders`.
 - This avoids cycles (everyone grabs locks in the same order).
4. **Break work into smaller steps**
 - Instead of one giant update, break into smaller transactions.
5. **Tune isolation level if appropriate**
 - Sometimes a lower isolation level (e.g., `READ COMMITTED`) reduces locking.

- In other cases, higher levels (e.g., **SERIALIZABLE**) enforce a strict order.
- ⚠ Trade-off: lower isolation = more anomalies, higher = less concurrency.

65. What is a database snapshot, and how is it used?

- A **database snapshot** is a **read-only, static copy** of a database at a specific point in time.

👉 Snapshot 是如何工作的 (这是核心)

当你创建 **Database Snapshot** 时:

- 不会复制整库
- 只创建一个 **empty sparse file**
- 所有数据 **最开始仍然从 source database 读取**

之后只要 source database 中的某个 **data page** 被修改:

- SQL Server 会先把 **修改前的 page** 写入 **snapshot 的 sparse file**
- 再允许 source database 修改该 page

👉 因此:

- **unchanged data** → 直接从 live database 读
- **changed data** → 从 **snapshot 的 sparse file** 读

一句话总结:

Snapshot 只保存“被改之前的数据”。

Example:

```
CREATE DATABASE SalesDB_Snapshot_2025
ON
(
    NAME  = SalesDB_Data,
    FILENAME = 'C:\Snapshots\SalesDB_Snapshot_2025.ss'
)
AS SNAPSHOT OF SalesDB;
```

- 含义是:
 - **MySnapshot** : snapshot database 名称
 - **NAME** : source database 的 logical data file name (必须真实存在)
 - 👉 **NAME** 用的是 LogicalName, 不是文件路径, 也不是数据库名。
 - LogicalName是原数据库的一个属性, 可通过查询获得!
 - **FILENAME** : snapshot 的 sparse file 存储路径
 - **AS SNAPSHOT OF** : 指定 snapshot 的源数据库

① **CREATE DATABASE SalesDB_Snapshot_2025**

👉 创建一个新的 database

👉 名字叫 **SalesDB_Snapshot_2025**

② **NAME = SalesDB_Data**

👉 告诉 SQL Server:

“我要对 SalesDB 里那个 logical name 叫 **SalesDB_Data** 的 data file 建 snapshot。”

③ **FILENAME = 'C:\Snapshots\SalesDB_Snapshot_2025.ss'**

👉 这是 snapshot 的 sparse file :

- 初始时几乎是空的
- 只会存 **被修改前的数据页**
- 不是完整数据库

④ **AS SNAPSHOT OF SalesDB**

👉 指定:

“这个 snapshot 的源数据库是 **SalesDB** ”

- Snapshot **依赖这个 source database**
- Source database 被 drop → snapshot 就失效

四、为什么 SQL Server 非要你写 logical file name?

因为一个 database:

- 可能有 **多个 data files**
- 每个 data file 都有 **不同的 logical name**
- SQL Server 需要你**明确告诉它**:

“你要 snapshot 哪一个 data file。 ”

66. What are the differences between OLTP and OLAP systems?

- **Transactional queries (OLTP)**

- Small, fast operations (INSERT, UPDATE, DELETE).
 - Optimized for speed and concurrency.
 - Example: updating a customer's order status.

- **Analytical queries (OLAP)**

- Complex aggregations over large datasets.
 - Read-heavy, often slower but more data-intensive.
 - Example: finding total sales by region over the past year.

👉 Key difference: transactional = day-to-day operations, analytical = insights and reporting.

67. What is a live lock, and how does it differ from a deadlock?

1) Deadlock (死锁) 你要怎么说才满分

定义：两个或多个 transaction/thread 各自持有部分 lock，并且互相等待对方释放需要的 lock，导致永久等待。

状态特征： blocked (卡住不动)。

经典例子：

- Txn A: lock(row1) → wants lock(row2)
- Txn B: lock(row2) → wants lock(row1)
- 两边都在等 → 永远等不到

面试加分点： deadlock 常见触发条件 (背 4 个词)

- mutual exclusion
- hold and wait
- no preemption
- circular wait

2) Livelock (活锁) 你要怎么说才满分

定义：多个 transaction/thread 没有被 blocked，它们持续执行一些“避免冲突”的动作 (比如 abort/retry, release/reacquire, backoff)，但因为节奏一致或策略问题，彼此一直干扰，导致始终无法完成。

状态特征： running / spinning (在跑，在做事)，但 no progress (没有前进)。

你给的“两个礼貌的人走廊让路”比喻非常标准。

3) 最关键区别 (面试官就想听这一句)

- Deadlock: 大家都在 waiting / blocked (不干活)
- Livelock: 大家都在 working / retrying (干活但不产出)

68. What is the purpose of the SQL EXCEPT operator?

SQL **EXCEPT** 用做 set difference：返回“在第一个 result set 里出现、但不在第二个 result set 里出现”的行 (A - B)，并且默认是 DISTINCT (去重)。

1. 两边 **SELECT** 必须返回 same number of columns
2. 对应列的数据类型要 compatible data types
3. 默认会去重：相当于 EXCEPT = EXCEPT DISTINCT
 - 有些数据库支持 **EXCEPT ALL**：保留重复次数 (multiset difference)

Example:

```
SELECT ProductID FROM ProductsSold  
EXCEPT  
SELECT ProductID FROM ProductsReturned;
```

含义：找“卖出去但没退货”的 `ProductID` (Sold 里有, Returned 里没有)。

69. How do you implement dynamic SQL, and what are its advantages and risks?

Dynamic SQL = SQL statements built as strings and executed at runtime, not pre-written in the code.

It's like constructing a query on the fly depending on inputs or conditions.

In SQL Server: Use `sp_executesql` or `EXEC`.

Syntax:

```
DECLARE @sql NVARCHAR(MAX)  
SET @sql = 'SELECT * FROM ' + @TableName  
EXEC sp_executesql @sql;
```

Downside:

- SQL injection + performance issues

70. What is the difference between horizontal and vertical partitioning?

Horizontal partitioning = 按 rows 切 (同样的 columns, 不同的行)。

Vertical partitioning = 按 columns 切 (同样的行, 不同的列)。

记忆: Horizontal = row-wise; Vertical = column-wise。

Horizontal Partitioning

- **What it is:** Breaks the table by rows.
- Each partition has the **same columns**, but only a **subset of rows**.
- Often based on a condition, like **region**, **date**, or **ID range**.

Example:

`Customers` table (all columns stay the same):

- Partition 1: Customers in `North America`
- Partition 2: Customers in `Europe`
- Partition 3: Customers in `Asia`

Vertical Partitioning

- **What it is:** Breaks the table by columns.
- Each partition has a **subset of columns**, but all rows.

- Used when some columns are large, rarely used, or need to be stored separately.

Example:

`Customers` table:

- Partition 1: `(CustomerID, Name, Email)`
- Partition 2: `(CustomerID, ProfilePicture, Notes)`

71. What are the considerations for indexing very large tables?

- Only index columns that are **heavily used** in `WHERE`, `JOIN`, or `ORDER BY` clauses.
- Avoid indexing every column → increases storage and slows down inserts/updates.
- Remove unused or rarely accessed indexes to reduce maintenance costs.

72. What is the difference between database sharding and partitioning?

Sharding

把数据拆到多个 **independent databases/servers** (多个 shards) → 主要为 **horizontal scaling** (容量/吞吐扩展)。

- Example: A global user DB split into `North America`, `Europe`, and `Asia` shards, each hosted on a separate server.
- Key point: Shards live on **different servers/databases** → increases capacity.

Partitioning

Partitioning: 在同一个 DB 里把一张表切成多个 **partitions** (逻辑还是一张表) → 主要为 **performance + maintenance**。

- Example: A `Sales` table partitioned by year (`2019`, `2020`, `2021` partitions). Queries on `2021` sales only scan that partition.

73. What are the best practices for writing optimized SQL queries?

- ◆ 1. Keep Queries Simple and Clear
- ◆ 2. Filter Data Early
 - Use `WHERE` conditions as soon as possible to reduce the dataset size.
- ◆ 3. Avoid `SELECT`
- ◆ 4. Use Indexes Wisely

- Periodically check for **unused or duplicate indexes** and remove them.
-

- ◆ 5. Leverage Execution Plans

- Use the database's query execution plan to spot slow operations.
- Look for missing indexes, table scans, or expensive joins.

74. How can you monitor query performance in a production database?

看 Top queries + 看 execution plan + 看 runtime metrics/waits + 看 plan regression，用 slow query log / query stats / APM tracing 把“谁慢、为啥慢、什么时候开始慢”定位出来。

- ◆ 1. Execution Plans

- 目的：看 DB 到底怎么跑你的 SQL（走不走 index、有没有 full table scan、join order 对不对、有没有 sort/hash 爆内存）。
 - MySQL/Postgres: `EXPLAIN` (需要更真实可用 `EXPLAIN ANALYZE`，但生产慎用)

- ◆ 2. Query Profiling / EXPLAIN

- Use `EXPLAIN` (MySQL/Postgres) or Query Store (SQL Server) to see query cost.
- Tells you which part of the query is expensive.

75. What are the trade-offs of using indexing versus denormalization?

Indexing

- Pros:
 - Makes reads faster (lookups, joins, filters).
- Cons:
 - Slows down writes (INSERT/UPDATE/DELETE must update indexes too).
 - Uses extra storage.

Denormalization

- Pros:
 - Reduces complex joins by storing redundant/pre-joined data.
 - Great for read-heavy workloads where performance is more important than strict normalization.
- Cons:
 - Data redundancy → risk of inconsistency.
 - Harder updates (must keep duplicates in sync).

76. How does SQL handle recursive queries?(不会遇见的，算了)

- SQL 处理递归查询主要靠 recursive CTE (Common Table Expression)：先用 anchor member 找起点，再用 recursive member 不断自连接，直到没有新行（或到 depth limit）为止。
- 递归 CTE 的 3 个组成（必背）
 1. Anchor member：起始集合 (root)
 2. Recursive member：把 CTE 自己再拿来 JOIN，一层层扩展
 3. Termination：当递归那一层产不出新行就停；很多 DB 也可设置 max recursion depth

🔍 Example (Org Chart)

Find all employees under a given manager:

```
WITH EmployeeHierarchy AS (
    -- Anchor: start with the manager
    SELECT EmployeeID, ManagerID, 1 AS Level
    FROM Employees
    WHERE ManagerID IS NULL    -- top-level boss

    UNION ALL

    -- Recursive: find direct reports
    SELECT e.EmployeeID, e.ManagerID, h.Level + 1
    FROM Employees e
    INNER JOIN EmployeeHierarchy h
    ON e.ManagerID = h.EmployeeID
)
SELECT * FROM EmployeeHierarchy;
```

- Starts at the boss (`ManagerID IS NULL`).
- Keeps joining employees to their manager until no more levels remain.

77. What are the differences between transactional and analytical queries?

Transactional queries (OLTP) are short, frequent operations like `INSERT/UPDATE/DELETE` or point `SELECT`，optimized for **low latency** and **high concurrency**, often requiring ACID.

Analytical queries (OLAP) are long, read-heavy queries that scan large datasets and do complex **aggregations** and **joins** for reporting and insights, optimized for **throughput** rather than per-request latency.

两个超直观例子 (面试官最爱)

- OLTP：`UPDATE Orders SET status='Shipped' WHERE order_id=123;`
- OLAP：`SELECT region, SUM(amount) FROM Sales WHERE year=2025 GROUP BY region;`

78. How can you ensure data consistency across distributed databases?

- ◆ 1. Distributed Transactions (Strong Consistency)
 - Use protocols like **two-phase commit** (2PC).
 - All databases either **commit together or roll back together**.
 - Guarantees consistency, but slower and harder to scale.
- ◆ How it works

Phase 1 — Prepare (Voting phase)

1. A **coordinator** asks all databases (participants) if they can commit.
2. Each participant checks if it can commit (locks resources, validates constraints).
3. Each replies **Yes** (ready to commit) or **No** (abort).

Phase 2 — Commit (Decision phase)

1. If all said **Yes** → coordinator tells everyone to **commit**.
2. If anyone said **No** → coordinator tells everyone to **roll back**.

-
- ◆ 2. Replication & Synchronization

79. What is the purpose of the SQL PIVOT operator?

SQL **PIVOT** 的目的：把“行里的分类值”旋转成“列”，做 **reporting / summary table**（报表汇总）更直观。

它到底做了 3 件事（背这三个词就够）

PIVOT = group by + aggregate + rotate

1. **GROUP BY**: 按某个 key (比如 **ProductID**) 分组
 2. **AGGREGATE**: 对每组做聚合 (通常 **SUM**, **COUNT**, **AVG**)
 3. **ROTATE**: 把某列里的分类值 (比如 **Year**) 变成多列 (**2021**, **2022** ...)
-

超标准解释（面试用）

PIVOT transforms rows into columns by taking:

- 一个 **pivot column** (要变成列的那个分类, 比如 **Year**)
 - 一个 **value column** (要填进新列的数值, 比如 **Amount**)
 - 一个 **aggregate function** (因为同一个 **ProductID + Year** 可能有多行, 需要聚合)
-

你给的例子（补全“为什么必须 aggregate”）

原表：

ProductID	Year	Amount
1	2021	500
1	2022	700

PIVOT 后:

ProductID	2021	2022
1	500	700

👉 重点: 如果同一个 `ProductID=1` 在 `2021` 有多条销售记录, 那 `2021` 这一列必须用 `SUM(Amount)` 或 `MAX(Amount)` 来合成一个值, 所以 `PIVOT` 本质上一定带 aggregation。

80. What is a bitmap index, and how does it differ from a B-tree index?

1. Bitmap index 到底是什么? (你要能讲清“它怎么存、怎么查”)

假设表有 N 行, 某列是 `Gender`, 只有两个值: `M/F` (low-cardinality)。

Bitmap index 会为每个 distinct value 存一个 `bitmap(bitset)`, 长度 = N:

row#	Gender
1	M
2	F
3	F
4	M

- $\text{bitmap}(M) = 1001$
- $\text{bitmap}(F) = 0110$

查询:

```
WHERE Gender='F'
```

直接取 `bitmap(F)` 就知道哪些行命中。

更强的是多条件:

```
WHERE Gender='F' AND Status='Active'
```

如果 `Status` 也有 bitmap:

- $\text{bitmap}(Active) = 1101$

那么结果就是:

`bitmap(F) AND bitmap(Active) = (0 1 1 0) AND (1 1 0 1) = 0 1 0 0`

一条 bitwise AND 就把两个过滤条件合并了，非常快（CPU 位运算 + 常常还能压缩）。

2) B-tree index 是什么？（面试常见“正常索引”）

B-tree index 把 key 按排序放在 balanced tree 里，叶子节点（leaf node）通常存 `(key -> rowid/tuple pointer)`，并且叶子常有链表方便扫描。

它擅长：

- range query: `WHERE OrderDate BETWEEN ...`
- high-cardinality: 比如 `UserID` 这种几乎都不同的值
- OLTP 场景下频繁 `INSERT/UPDATE/DELETE` 也更友好

3) 核心区别（你背这 6 点基本满分）

1. 数据结构

- Bitmap index: value → bit vector
- B-tree index: value → tree nodes (有序)

1. 最适合的 cardinality

- Bitmap: low-cardinality (少量取值，比如 `Gender`, `IsActive`, `Status`)
- B-tree: high-cardinality (很多唯一值，比如 `UserID`, `Email`, `Timestamp`)

1. 多条件过滤

- Bitmap: 多个条件用 bitwise AND/OR/NOT，合并超快（特别是 star schema / OLAP）
- B-tree: 通常是“用一个索引定位 + 回表”，或多个索引用 `index intersection`（看优化器能力），但没 bitmap 那么天然

1. range query

- Bitmap: 不擅长（本质不是“按排序连续扫描”）
- B-tree: 强项（叶子有序，范围扫描很自然）