数据库MySQL面试题

> 基于简历：熟练使用MySQL，了解索引结构（B+树）、事务ACID特性、InnoDB存储引擎、锁机制

一、索引

Q1: 为什么MySQL使用B+树而不是B树或哈希表?

B+树 vs B树：

B树：

```
      [10, 20]
     /    |    \
[5]  [15]  [25, 30]
```

- 每个节点都存数据
- 非叶子节点也存数据

B+树：

```
      [10, 20]
     /    |    \
[5]  [15]  [25, 30]  ← 叶子节点链表
```

- 只有叶子节点存数据
- 非叶子节点只存索引
- 叶子节点形成链表

B+树的优势：

1. **非叶子节点不存数据，可以存更多索引**
   - 一个节点4KB，能存更多key
   - 树的高度更低，I/O次数更少
2. **叶子节点链表，范围查询效率高**

   ```sql
   SELECT * FROM users WHERE age BETWEEN 20 AND 30;
   ```

   - B+树：找到20后，顺着链表扫描到30
   - B树：需要多次回到根节点
3. **所有查询路径长度相同**

- 性能稳定，都是O(log n)

B+树 vs 哈希表：

特性	B+树	哈希表
范围查询	✅ 高效	❌ 不支持
排序	✅ 有序	❌ 无序
精确查询	O(log n)	O(1)

适用场景：

- B+树：范围查询、排序（InnoDB默认）
- 哈希：精确查询（Memory引擎）

Q2: 聚簇索引和非聚簇索引的区别?

聚簇索引（Clustered Index）：

- 叶子节点存储整行数据
- InnoDB主键索引就是聚簇索引
- 一个表只能有一个聚簇索引

```
主键索引（聚簇索引）：
      [10, 20]
     /    |    \
[完整行1] [完整行2] [完整行3]
```

非聚簇索引（Secondary Index）：

- 叶子节点存储主键值
- 需要回表查询

```
age索引（非聚簇索引）：
      [25, 30]
     /    |    \
[pk:1]  [pk:2]  [pk:3]  → 再通过主键索引查找完整数据
```

回表：

```
SELECT * FROM users WHERE age = 25;

步骤：
1. 通过age索引找到主键id=5
2. 通过主键索引（聚簇索引）找到完整行
```

索引覆盖（Covering Index）：

```
SELECT id, age FROM users WHERE age = 25;
```

- 只需要age索引的叶子节点（包含id和age）
- 不需要回表，效率更高

与你项目的联系： "在设计KV存储时，如果用MySQL存储，可以将key作为主键（聚簇索引），查询效率高。"

Q3: 联合索引和最左前缀原则?

联合索引：

```
CREATE INDEX idx_name_age ON users(name, age);
```

B+树结构：

```
     [(Alice,20), (Bob,25)]
    /                      \
[(Alice,18)]           [(Bob,23), (Bob,27)]
```

- 先按name排序，name相同再按age排序

最左前缀原则：

✅ **会使用索引：**

```
WHERE name = 'Alice'                  -- 用到name
WHERE name = 'Alice' AND age = 20     -- 用到name和age
WHERE name > 'Alice'                  -- 用到name
```

❌ **不会使用索引：**

```
WHERE age = 20                        -- 跳过name，无法使用
WHERE name LIKE '%Alice%'             -- 前缀模糊，无法使用
```

原因： B+树按(name, age)排序，必须先确定name才能利用age的排序。

索引设计建议：

```
-- 如果有查询：
-- 1. WHERE name = ?
-- 2. WHERE name = ? AND age = ?
-- 3. WHERE name = ? AND age = ? AND city = ?

-- 建立索引：
CREATE INDEX idx ON users(name, age, city);
-- 一个索引满足三种查询
```

Q4: 什么情况下索引会失效?

1. 使用函数或表达式

```
-- ❌ 索引失效
WHERE YEAR(birthday) = 1990
WHERE age + 1 = 20

-- ✅ 正确
WHERE birthday BETWEEN '1990-01-01' AND '1990-12-31'
WHERE age = 19
```

2. 隐式类型转换

```
-- phone字段是VARCHAR类型
-- ❌ 索引失效（数字转字符串）
WHERE phone = 12345678

-- ✅ 正确
WHERE phone = '12345678'
```

3. 前导模糊查询

```
-- ❌ 索引失效
WHERE name LIKE '%Alice%'
WHERE name LIKE '%Alice'

-- ✅ 正确
WHERE name LIKE 'Alice%'
```

4. OR条件有非索引列

```
-- ❌ 索引失效（name有索引，email没有）
WHERE name = 'Alice' OR email = 'alice@example.com'
```

```
-- ✅ 两个都有索引，会使用索引
WHERE name = 'Alice' OR age = 20
```

5. 联合索引不满足最左前缀

```
-- 索引: (name, age)
-- ❌ 索引失效
WHERE age = 20
```

6. 优化器选择全表扫描

```
-- 如果查询结果占表的大部分，优化器可能选择全表扫描
WHERE age > 10  -- 如果90%的数据都满足
```

二、事务

Q5: ACID特性详细解释?

A - 原子性（Atomicity）：

- 事务是不可分割的最小单位
- 要么全部成功，要么全部失败

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;  -- 两条SQL要么都成功，要么都失败
```

实现： undo log（回滚日志）

C - 一致性（Consistency）：

- 事务前后，数据的完整性约束不被破坏
- 例如：转账前后总金额不变

实现： 应用层保证 + 数据库约束

I - 隔离性（Isolation）：

- 并发事务之间互相隔离，互不干扰

实现： MVCC（多版本并发控制）+ 锁

D - 持久性（Durability）：

- 事务提交后，数据永久保存

实现： redo log（重做日志）

Q6: 事务隔离级别及其问题?

四个隔离级别：

1. 读未提交（Read Uncommitted）

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

- 可以读到其他事务未提交的数据
- **问题：脏读**

脏读示例：

```
时间  事务A                    事务B
1     BEGIN
2                              BEGIN
3                              UPDATE SET balance = 500
4     SELECT (读到500)         -- 未提交
5                              ROLLBACK
6     SELECT (读到1000)        -- 数据不一致！
```

2. 读已提交（Read Committed）

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

- 只能读到已提交的数据
- **问题：不可重复读**

不可重复读示例：

```
时间  事务A                    事务B
1     BEGIN
2     SELECT (读到1000)
3                              UPDATE SET balance = 500
4                              COMMIT
5     SELECT (读到500)         -- 两次读结果不同！
```

3. 可重复读（Repeatable Read）

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

- 同一事务多次读取结果相同
- **问题：幻读**
- **MySQL默认级别**

幻读示例：

```
时间  事务A                          事务B
1     BEGIN
2     SELECT COUNT(*) (结果10)
3                                     INSERT新行
4                                     COMMIT
5     SELECT COUNT(*) (结果仍然10)   -- MVCC解决
6     UPDATE所有行                    -- 更新了11行！(幻读)
```

4. 串行化（Serializable）

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- 完全串行执行，无并发问题
- 性能最差

总结：

隔离级别	脏读	不可重复读	幻读
读未提交	✓	✓	✓
读已提交	✗	✓	✓
可重复读	✗	✗	✓（InnoDB解决了）
串行化	✗	✗	✗

Q7: MVCC如何实现？

MVCC（Multi-Version Concurrency Control）：

- 通过数据的多个版本实现并发控制
- 读不加锁，写不阻塞读

隐藏列： 每行数据包含：

- `DB_TRX_ID`：最后修改该行的事务ID
- `DB_ROLL_PTR`：指向undo log的指针
- `DB_ROW_ID`：隐藏主键（如果没有主键）

undo log版本链：

```
当前版本: name='Bob', TRX_ID=100
   ↓
旧版本: name='Alice', TRX_ID=50
   ↓
更旧版本: name='Tom', TRX_ID=20
```

Read View（读视图）： 事务开始时，记录：

- `m_ids`：当前活跃事务列表
- `min_trx_id`：最小活跃事务ID
- `max_trx_id`：下一个将分配的事务ID
- `creator_trx_id`：当前事务ID

可见性判断：

```
if (DB_TRX_ID < min_trx_id):
    可见（在所有活跃事务之前提交）
elif (DB_TRX_ID >= max_trx_id):
    不可见（在Read View创建之后才开始）
elif (DB_TRX_ID in m_ids):
    不可见（活跃事务，未提交）
else:
    可见（已提交事务）
```

示例：

```
事务100：SELECT * FROM users WHERE id = 1;
Read View: m_ids=[98, 99], min=98, max=101

数据: name='Bob', TRX_ID=99  → 在活跃列表中 → 不可见
      ↓ 沿着undo log查找
旧数据: name='Alice', TRX_ID=50  → <min → 可见！
返回: name='Alice'
```

RC和RR的区别：

- RC：每次SELECT生成新的Read View
- RR：事务开始时生成Read View，一直用这个

三、锁机制

Q8: 表锁、行锁、间隙锁的区别?

表锁（Table Lock）：

```
LOCK TABLES users WRITE;
-- 其他事务无法读写整个表
UNLOCK TABLES;
```

- 锁粒度大，并发性差
- MyISAM使用表锁

行锁（Row Lock）：

```
SELECT * FROM users WHERE id = 1 FOR UPDATE;
-- 只锁id=1这一行
```

- 锁粒度小，并发性好
- InnoDB使用行锁
- 基于索引实现（无索引会升级为表锁）

间隙锁（Gap Lock）：

```
数据: id = 5, 10, 15
间隙: (-∞,5), (5,10), (10,15), (15,+∞)

锁定id>5 AND id<10:
锁定间隙(5,10)，防止插入7、8、9
```

- 防止幻读
- 只在RR隔离级别生效

Next-Key Lock：

- 行锁 + 间隙锁
- 锁定记录及其前面的间隙
- InnoDB默认使用

示例：

```
-- 数据: id = 5, 10, 15
SELECT * FROM users WHERE id = 10 FOR UPDATE;

加锁范围: (5, 10]
- 锁定id=10的行（行锁）
- 锁定间隙(5,10)（间隙锁）
- 防止插入6,7,8,9
```

Q9: 乐观锁和悲观锁?

悲观锁（Pessimistic Lock）：

- 认为冲突一定会发生
- 先加锁再操作

```
START TRANSACTION;
SELECT * FROM products WHERE id = 1 FOR UPDATE;  -- 加锁
-- 处理业务逻辑
UPDATE products SET stock = stock - 1 WHERE id = 1;
COMMIT;
```

优点： 数据一致性强 **缺点：** 并发性能差

乐观锁（Optimistic Lock）：

- 认为冲突很少发生
- 不加锁，通过版本号检测冲突

```
-- 1. 读取数据和版本号
SELECT id, stock, version FROM products WHERE id = 1;
-- stock=10, version=5

-- 2. 业务处理

-- 3. 更新时检查版本号
UPDATE products
SET stock = 9, version = version + 1
WHERE id = 1 AND version = 5;

-- 4. 检查影响行数
if (affected_rows == 0):
    -- 版本号变了，说明有冲突，重试
```

优点： 并发性能好 **缺点：** 冲突时需要重试

选择：

- 读多写少 → 乐观锁
- 写冲突频繁 → 悲观锁

与你项目的联系： "在KV存储系统中，如果实现乐观锁，可以为每个key增加version字段，更新时检查版本号。"

Q10: 如何避免死锁?

死锁示例：

```
-- 事务A
START TRANSACTION;
UPDATE users SET name='A' WHERE id=1;  -- 锁住id=1
UPDATE users SET name='A' WHERE id=2;  -- 等待id=2（被B锁定）

-- 事务B
START TRANSACTION;
UPDATE users SET name='B' WHERE id=2;  -- 锁住id=2
UPDATE users SET name='B' WHERE id=1;  -- 等待id=1（被A锁定）
-- 死锁！
```

避免死锁：

1. 固定加锁顺序

```
-- 总是按id顺序加锁
UPDATE users SET name='X' WHERE id=1;
UPDATE users SET name='X' WHERE id=2;
```

2. 一次性获取所有锁

```
SELECT * FROM users WHERE id IN (1, 2) FOR UPDATE;
-- 一次性锁住多行
```

3. 减小事务粒度

```
-- ❌ 长事务
START TRANSACTION;
SELECT ...  -- 耗时操作
UPDATE ...
COMMIT;

-- ✅ 短事务
SELECT ...  -- 不在事务中
START TRANSACTION;
UPDATE ...  -- 只有更新在事务中
COMMIT;
```

4. 使用超时

```
SET innodb_lock_wait_timeout = 5;  -- 5秒超时
```

5. 死锁检测

- InnoDB自动检测死锁
- 回滚其中一个事务

查看死锁日志：

```
SHOW ENGINE INNODB STATUS;
```

四、SQL优化

Q11: EXPLAIN的关键字段?

```
EXPLAIN SELECT * FROM users WHERE age = 20;
```

关键字段：

1. type（访问类型）：

```
性能从好到坏：
system > const > eq_ref > ref > range > index > ALL
```

- `const`：主键或唯一索引查询
- `ref`：普通索引查询
- `range`：范围查询
- `index`：索引全扫描
- `ALL`：全表扫描（最差）

2. possible_keys：

- 可能使用的索引

3. key：

- 实际使用的索引

4. rows：

- 预计扫描的行数

5. Extra：

- `Using index`：索引覆盖，不需要回表
- `Using where`：使用WHERE过滤
- `Using filesort`：需要额外排序（性能差）
- `Using temporary`：使用临时表（性能差）

优化目标：

- type尽量达到ref以上
- 避免`Using filesort`和`Using temporary`

Q12: 慢查询如何优化?

1. 分析慢查询日志

```
-- 开启慢查询日志
SET GLOBAL slow_query_log = ON;
SET GLOBAL long_query_time = 2;  -- 超过2秒的查询

-- 查看慢查询
mysqldumpslow /var/log/mysql/slow.log
```

2. 使用EXPLAIN分析

```
EXPLAIN SELECT * FROM orders WHERE user_id = 1 AND status = 'paid';
```

3. 添加索引

```
-- 没有索引 → type=ALL（全表扫描）
CREATE INDEX idx_user_status ON orders(user_id, status);
-- 有索引 → type=ref
```

4. 避免SELECT *

```
-- ❌ 查询所有字段
SELECT * FROM users WHERE id = 1;

-- ✅ 只查询需要的字段
SELECT id, name, age FROM users WHERE id = 1;
```

5. 分页优化

```
-- ❌ 深分页效率低
SELECT * FROM users ORDER BY id LIMIT 100000, 10;

-- ✅ 使用子查询
SELECT * FROM users
WHERE id > (SELECT id FROM users ORDER BY id LIMIT 100000, 1)
LIMIT 10;
```

```
-- ✅ 记录上次的ID
SELECT * FROM users WHERE id > 上次最大ID LIMIT 10;
```

6. JOIN优化

```
-- 小表驱动大表
SELECT * FROM small_table
JOIN large_table ON small_table.id = large_table.fk;

-- 确保JOIN字段有索引
```

五、存储引擎

Q13: InnoDB和MyISAM的区别?

特性	InnoDB	MyISAM
事务	✅ 支持	❌ 不支持
行锁	✅ 支持	❌ 只有表锁
外键	✅ 支持	❌ 不支持
崩溃恢复	✅ 支持（redo log）	❌ 不支持
MVCC	✅ 支持	❌ 不支持
全文索引	✅ 支持（5.6+）	✅ 支持
表空间	表空间文件（.ibd）	数据文件+索引文件
性能	写性能好	读性能好

选择:

- 需要事务、高并发写 → InnoDB（推荐）
- 只读、查询为主 → MyISAM

MySQL 5.5+默认InnoDB

六、快速复习清单

索引

- ☐ B+树 vs B树 vs 哈希表
- ☐ 聚簇索引 vs 非聚簇索引
- ☐ 联合索引最左前缀原则
- ☐ 索引失效场景

事务

- [ ] ACID特性（实现原理）
- [ ] 四个隔离级别
- [ ] MVCC原理

锁

- [ ] 表锁、行锁、间隙锁
- [ ] 乐观锁 vs 悲观锁
- [ ] 如何避免死锁

SQL优化

- [ ] EXPLAIN关键字段
- [ ] 慢查询优化方法
- [ ] 分页优化

存储引擎

- [ ] InnoDB vs MyISAM

面试技巧

画图说明：

- B+树结构
- MVCC版本链
- 间隙锁范围

结合场景： "在电商系统中，库存扣减需要用悲观锁..." "在KV存储中，key作为主键，利用聚簇索引..."

深入追问准备：

- "B+树为什么高度通常是3-4层？" → 答：InnoDB页大小16KB，一个节点能存约1000个key，4层可以存10亿条记录

记住：原理 + 场景 + 优化 = 完美回答