计算机考研复试面试常问问题 数据库篇

>

章节导读:

1.事务

> **概念**：事务指的是满足 ACID 特性的一组操作，可以通过 Commit 提交一个事务，也可以使用 Rollback 进行回滚。

> **ACID特性**：
> （1）原子性 (Atomicity)：事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。
>
> （2）一致性 (Consistency)：数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对一个数据的读取结果都是相同的。
>
> （3）隔离性 (Isolation)：一个事务所做的修改在最终提交以前，对其它事务是不可见的。
>
> （4）持久性 (Durability)：一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

> 使用重做日志来保证持久性。

> 事务的 ACID 特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：
> 只有满足一致性，事务的执行结果才是正确的。
> 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时只要能满足原子性，就一定能满足一致性。
> 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性。事务满足持久性是为了能应对数据库崩溃的情况。

2.并发一致性问题

> **丢失数据**

> 丢失数据：T_1 和 T_2 两个事务都对一个数据进行修改，T_1 先修改，T_2 随后修改，T_2 的修改覆盖了 T_1 的修改。简记为 `同时修改`。

> **读脏数据**

> 读脏数据：T_1 对一个数据做了修改，T_2 读取这一个数据。若 T_1 执行 ROLLBACK 操作，则 T_2 读取的结果和第一次的结果不一样。简记为 `读取失败的修改`。最简单的场景是修改完成后，紧接着查询检验结果。

> **不可重复读**

> 不可重复读：T_2 读取一个数据，T_1 对该数据做了修改。如果 T_2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。简记为 `读时修改`，重复读取的结果不一样。

> **幻影读**

> 幻影读：T_1 读取某个范围的数据，T_2 在这个范围内插入新的数据，T_1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。简记为 `读时插入`，重复读取的结果不一样。

> **解决方案**

> 在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。产生并发不一致性问题的主要原因是破坏了事务的隔离性。解决方法是通过 `并发控制` 来保证隔离性。并发控制可以通过 `封锁` 来实现，但是封锁操作需要用户自己控制，相当复杂。数据库管理系统提供了事务的 `隔离级别`，让用户以一种更轻松的方式处理并发一致性问题。

3.封锁

> **封锁粒度**
> MySQL 中提供了两种封锁粒度：`行级锁` 以及 `表级锁`。
>
> 应尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。但是加锁需要消耗资源，锁的各种操作 (包括获取锁、释放锁、以及检查锁状态) 都会增加系统开销。因此封锁粒度越小，系统开销就越大。为此，我们在选择封锁粒度时，需在 `锁开销` 和 `并发程度` 之间做一个 `权衡`。

> **封锁类型**

> **（1）读写锁**
> 排它锁 (Exclusive)，简写为 `X 锁`，又称 `写锁`。
>
> 共享锁 (Shared)，简写为 `S 锁`，又称 `读锁`。
>
> 有以下两个规定：
> 一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。

> 一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

> **（2）意向锁**
> 使用意向锁 (Intention Locks)，可以更容易地支持多粒度封锁，使得行锁和表锁能够共存。
> 在存在行级锁和表级锁的情况下，事务 T 想要对表 A 加 X 锁，就需要先检测是否有其它事务对表 A 或者表 A 中的任意一行加了锁，那么就需要对表 A 的每一行都检测一次，这是非常耗时的。
>
> 意向锁在原来的 X/S 锁之上引入了 IX / IS，IX / IS 都是 `表级别的锁`，用来表示一个事务稍后会对表中的某个数据行上加 X 锁或 S 锁。整理可得以下两个规定：
>
> 一个事务在获得某个数据行对象的 S 锁之前，必须先获得表的 IS 锁或者更强的锁；
> 一个事务在获得某个数据行对象的 X 锁之前，必须先获得表的 IX 锁。

> **封锁协议**

> **三级封锁协议**
>
> `一级封锁协议`：事务 T 要修改数据 A 时必须加 X 锁，直到 T 结束才释放锁。`防止同时修改`,可解决 `丢失修改` 问题，因不能同时有两个事务对同一个数据进行修改，那么事务的修改就不会被覆盖。
>
> `二级封锁协议`：在一级的基础上，要求读取数据 A 时必须加 S 锁，读取完马上释放 S 锁。`防止修改时读取`,可解决 `丢失修改` 和 `读脏数据` 问题，因为一个事务在对数据 A 进行修改，根据 1 级封锁协议，会加 X 锁，那么就不能再加 S 锁了，也就是不会读入数据。
>
> `三级封锁协议`：在二级的基础上，要求读取数据 A 时必须加 S 锁，直到事务结束了才能释放 S 锁。`防止读取时修改`,可解决 `丢失修改` 和 `读脏数据` 问题，还进一步防止了 `不可重复读` 的问题，因为读 A 时，其它事务不能对 A 加 X 锁，从而避免了在读的期间数据发生改变。

> **两段锁协议**
> 两段锁协议是指每个事务的执行可以分为两个阶段：生长阶段 (加锁阶段) 和衰退阶段 (解锁阶段)。
> 两段封锁法可以这样来实现：
>
> 事务开始后就处于加锁阶段，一直到执行 ROLLBACK 和 COMMIT 之前都是加锁阶段。
> ROLLBACK 和 COMMIT 使事务进入解锁阶段，即在 ROLLBACK 和 COMMIT 模块中 DBMS 释放所有封锁。

4.关系数据库设计理论

> **函数依赖**

> - 记 $A \to B$ 表示 A 函数决定 B，也可以说 B 函数依赖于 A。
> - 若 $\{A_1，A_2，\ldots，A_n\}$ 是关系的一个或多个属性的集合，该集合函数决定了关系的其它所>有属性并且是 `最小的`，那么该集合就称为 `键码`。
> - 对于 $A \to B$，如果能找到 A 的真子集 A'，使得 $A' \to B$，那么 $A \to B$ 就是 `部分函数>依赖`，否则就是 `完全函数依赖`。
> - 对于 $A \to B$，$B \to C$，则 $A \to C$ 是一个 `传递函数依赖`。

> **异常**

> - 如表所示，展示了学生课程关系的函数依赖为 {Sno, Cname} -> {Sname, Sdept, Mname, Grade}，键码为 {Sno, Cname}。也就是说，确定学生和课程后就能确定其它信息。

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80

Sno	Sname	Sdept	Mname	Cname	Grade
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

> 不符合范式的关系，会产生很多异常，主要有以下四种异常：
>
> > `冗余数据`：例如 `学生-2` 出现了两次。
> >
> > `修改异常`：修改了一个记录中的信息，但是另一个记录中相同的信息却没有被修改。
> >
> > `删除异常`：删除一个信息，那么也会丢失其它信息。例如删除了 `课程-1` 需要删除第一>行和第三行，那么 学生-1 的信息就会丢失。
> >
> > `插入异常`：例如想要插入一个学生的信息，如果这个学生还没选课，那么就无法插入。

> **范式**
>
> 范式理论是为了解决以上提到四种异常。高级别范式的依赖于低级别的范式，1NF 是最低级别的范式。

> **第一范式 (1NF)**
>
> 属性不可分。即数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

> **第二范式 (2NF)**
>
> 每个非主属性完全函数依赖于键码。可以通过分解来满足 2NF。

分解前:

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

> 以下学生课程关系中，{Sno, Cname} 为键码，有如下函数依赖：
>
> - Sno -> Sname, Sdept
> - Sdept -> Mname
> - Sno, Cname-> Grade

> 函数依赖状况分析：
> Grade 完全函数依赖于键码，它没有任何冗余数据，每个学生的每门课都有特定的成绩。
>
> Sname, Sdept 和 Mname 都部分依赖于键码，当一个学生选修了多门课时，这些数据就会出现多次，造成大量冗余数据。

分解后:

> 关系-1：

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

- 有以下函数依赖：
- Sno -> Sname, Sdept
- Sdept -> Mname

- 关系-2：

Sno	Cname	Grade
1	课程-1	90
2	课程-2	80
2	课程-1	100
3	课程-2	95

- 有以下函数依赖： Sno, Cname → Grade

第三范式 (3NF)

- 非主属性不传递函数依赖于键码。简而言之，第三范式就是属性不依赖于其它非主属性。
- 上面的 关系-1 中存在以下传递函数依赖：Sno → Sdept → Mname。

分解后

- 关系-1.1：

Sno	Sname	Sdept
1	学生-1	学院-1
2	学生-2	学院-2
3	学生-3	学院-2

- 关系-1.2：

Sdept	Mname
学院-1	院长-1
学院-2	院长-2

巴斯-科德范式（BCNF）

Boyce-Codd Normal Form（巴斯-科德范式）
在3NF基础上，任何非主属性不能对主键子集依赖（在3NF基础上消除对主码子集的依赖）
巴斯-科德范式（BCNF）是第三范式（3NF）的一个子集，即满足巴斯-科德范式（BCNF）必须满>足第三范式（3NF）。通常情况下，巴斯-科德范式被认为没有新的设计规范加入，只是对第二

> 范式与第三范式中设计规范要求更强，因而被认为是修正第三范式，也就是说，它事实上是对第三范式的修正，使数据库冗余度更小。这也是BCNF不被称为第四范式的原因。某些书上，根据范式要求的递增性将其称之为第四范式是不规范，也是更让人不容易理解的地方。而真正的第四范式，则是在设计规范中添加了对多值及依赖的要求。

5.ER 图

> 实体关系图 (Entity-Relationship，E-R)，有三个组成部分：实体、属性、联系。用来进行关系型数据库系统的概念设计。

> - `实体`：用矩形表示，矩形框内写明实体名.
> - `属性`：用椭圆形表示，并用无向边将其与相应的实体连接起来。
> - `联系`：用菱形表示，菱形框内写明联系名，并用无向边分别与有关实体连接起来，同时在无向>边旁标上联系的类型（1...1，1...* 或 *...*）就是指存在的三种关系 (一对一、一对多或多对多)。

> - ER 模型转换为关系模式的原则：
> - `一对一`：遇到一对一关系的话，在两个实体任选一个添加另一个实体的主键即可。
> - `一对多`：遇到一对多关系的话，在多端添加另一端的主键。
> - `多对多`：遇到多对多关系的话，我们需要将联系转换为实体，然后在该实体上加上另外两个实体的主键，作为联系实体的主键，然后再加上该联系自身带的属性即可。

6.索引

> **索引的数据结构**

> B-Tree (平衡树, Balance Tree)：也称为 `多路平衡查找树`，并且所有叶子节点位于同一层。

> B+Tree：它不仅具有 B-Tree 的平衡性，并且可通过 `顺序访问指针` 来提高 `区间查询` 的性能。
>
> 在 B+Tree 中，一个节点中的 key 从左到右非递减排列，若某个指针的 key_i 左右相邻分别是 key_{i-1} 和 key_{i+1}，且不为 null，则该指针指向节点的所有 key 满足 $key_{i-1} \leq key_i \leq key_{i+1}$。
>
> B+Tree 与 B-Tree 最大区别是，B+Tree 的非叶子结点不保存数据，只用于索引，所有数据都保存在叶子结点中。而且叶子结点间按照从小到大顺序链接起来。

> B-Tree/B+Tree 的增删改查：

> `查找操作`：首先在`根节点`进行 `二分查找`，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在 `叶子节点` 上进行 `二分查找`，找出 key 所对应的 data。
>
> 二分查找要求表有序，正好 B-Tree 和 B+Tree 结点中的 key 从左到右非递减有序排列。
>
> `增删操作`：会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

> **MySQL 索引**

> 索引，在 MySQL 也称为键 (Key)，是 `存储引擎` 快速找到记录的一种 `数据结构`。相当于图书的目录，可根据目录中的页码快速找到所需的内容。

> **索引结构类型**

> **（1）B+Tree 索引**

- B+Tree 索引是大多数 MySQL 存储引擎的默认索引类型。
- 因为 B+ Tree 的 `有序性`，因此可用于 `部分查找`、`范围查找`、`排序` 和 `分组`。

- 适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。若不是按照索引列的顺序进行查找，则无法使用索引。

（2）Hash 索引

- Hash 索引能以 O(1) 时间进行查找，但是失去了有序性。因此无法用于排序与分组，无法用于部分查找和范围查找，只支持 `精确查找`。

> Hash 索引仅满足 `=`，`IN` 和 `<=>` 查询，不能使用范围查询。因为 Hash 索引比较的是 Hash 运算后的 Hash 值，所以它只能用于等值的过滤。

（3）全文索引

- 全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。

（4）空间数据索引

- 空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。
- 必须使用 GIS 相关的函数来维护数据。

索引的优点缺点

优点

- 大大减少了服务器需要扫描的数据行数。
- 避免服务器进行排序和分组操作，以避免创建 `临时表`。

> B+Tree 索引是有序的，可以用于 ORDER BY 和 GROUP BY 操作。临时表主要是在排序和分组过程中创建，不需要排序和分组，也就不需要创建临时表。

- 将 `随机 I/O` 变为 `顺序 I/O`。

> B+Tree 索引是有序的，会将相邻的数据都存储在一起。

缺点

- 索引并不是越多越好，索引固然可以提高相应的 SELECT 的效率，但同时也降低了 INSERT 及 UPDATE 的效率，因为 INSERT 或 UPDATE 时有可能会 `重建索引`。

索引的设计原则

> 从索引的优、缺点考虑索引的设计原则。

- `忌过度索引`：索引需要额外的磁盘空间，而且会降低写操作的性能。
  - 在修改表内容时，索引会进行更新甚至重构，索引列越多花销时间越长。为此优化检索性能，只保持需要的索引即可。
  - 经常用在 `排列`、`分组` 和 `范围搜索` 的列适合创建索引，因为索引是有序的。
  - 经常出现在 `WHERE` 子句的列，或是 `JOIN` 连接子句中指定的列适合创建索引。
- `使用短索引`：若对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间。

索引的优化策略

- `独立的列`：在进行查询时，索引列不能是 `表达式` 的一部分，也不能是 `函数参数`，否则无法使用索引。
- `多列索引`：在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。
- `索引列的顺序`：让选择性最强的索引列放在前面。
- `前缀索引`：对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。前缀长度的选取需要根据索引选择性来确定。

- `覆盖索引`：索引包含所有需要查询的字段的值。具有以下优点：
  - 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
  - 一些存储引擎（例如 MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。

> **索引的使用场景**

- 对于 `非常小的表`：大部分情况下简单的 `全表扫描` 比建立索引更高效；
- 对于 `中大型的表`：`建立索引` 非常有效；
- 对于 `特大型的表`：建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配。例如可以使用 `分区技术`。