MySQL

基础

数据库三大范式 (中)

- 第一范式: 要求数据库表的每一列都是不可分割的原子数据项
  - 如详细地址可以分割为 省市区等.
- 第二范式: 非主键属性必须完全依赖于主键, 不能部分依赖
  - 第二范式要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关
- 第三范式: 任何非主键属性不依赖于其它非主键属性
  - 第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关, 避免传递依赖
  - 一张表, 有学生id, 学号, 姓名, 年龄, 班主任姓名, 班主任年龄
  - 此时班主任年龄依赖于班主任姓名或者班主任id, 不应该依赖于学生id, 所以这就是间接相关.

数据库三范式也并不是必须遵守的, 适当添加冗余信息, 可以减少多表查询, 提高效率.

CHAR和VARCHAR有什么区别？ (中)

- CHAR是固定长度的字符串类型，定义时需要指定固定长度，存储时会在末尾补足空格。CHAR适合存储长度固定的数据，如固定长度的代码、状态等，存储空间固定，对于短字符串效率较高。

- VARCHAR是可变长度的字符串类型，定义时需要指定最大长度，实际存储时根据实际长度占用存储空间。VARCHAR适合存储长度可变的数据，如用户输入的文本、备注等，节约存储空间。

SQL语句的执行顺序 (中)

当一个查询语句同时出现了where,group by,having,order by的时候, 编写顺序

```
select 字段
from 表名
where 条件列表
group by 分组条件
having 分组后筛选
order by 排序条件
limit 条数
```

实际执行顺序:

1. 执行from查看表
2. 执行where xx对全表数据做筛选，返回第1个结果集。
3. 针对第1个结果集使用group by分组，返回第2个结果集。
4. 针对第2个结果集执行having xx进行筛选，返回第3个结果集。
5. 针对第3个结果集执行select xx，返回第4个结果集。
6. 针对第4个结果集order by排序, 返回第5个结果集
7. 针对第5个结果集使用limit进行条数限制, 返回第6个结果集

架构/引擎

SQL语句的执行过程/MySQL架构是什么 (中)

- **取得链接**，使用使用到 MySQL 中的连接器
  - 连接的过程需要先经过 TCP 三次握手，因为 MySQL 是基于 TCP 协议进行传输的
  - 校验客户端的用户名和密码
  - 校验用户权限
- **查询缓存**，key 为 SQL 语句，value 为查询结果，如果查到就直接返回。不建议使用此缓存， 在 MySQL 8.0 版本已经将查询缓存删除，也就是说 MySQL 8.0 版本后不存在此功能
  - 更新比较频繁的表，查询缓存的命中率很低的，因为只要一个表有更新操作，那么这个表的查询缓存就会被清空
- **分析器**，分为词法分析和语法分析。词法分析就是提取sql语句关键字, 语法分析就是语法校验, 构建 SQL 语法树, 方便后续模块读取表名、字段、语句类型
- **执行阶段**
  - **预处理**:
    - 检查 SQL 查询语句中的表或者字段是否存在；
    - 将 `select *` 中的 `*` 符号，扩展为表上的所有列；
  - **优化阶段(优化器)**:
    - 在表里有多个索引的时候，决定使用哪个索引；
    - 或者一个语句中存在多表关联的时 候（join），决定各个表的连接顺序
  - **执行阶段(执行器):** 据表的引擎定义(Innodb或者MyISAM), 去使用这个引擎提供的接口

数据库存储引擎有哪些 (高)

innodb

- 事务, 外键, 行级锁
- 适合事务要求高, 数据完整性高的场景

MyISAM

- 全表锁, 不支持事务, 不支持外键, 并发性低
- 适合对事务要求不高, 数据完整性要求不高, 并发性要求不高的场景

memory

- 全表锁
- 数据存储在内存中, 默认使用hash索引, 检索速度非常高.
- 适合做缓存 (被redis代替)

innodb和MyISAM的区别 (高)

- Innodb有事务 外键 行级锁
- InnoDB 支持数据库异常崩溃后的安全恢复, 依赖于`redo log`, 而MyISAM不支持
- innodb支持MVCC, MyISAM不支持

索引 (常考)

MySQL中的索引类型 (高)

逻辑维度

- 主键索引: 针对表主键的索引, 默认创建, 只能有一个
- 唯一索引: 避免一个表中的索引重复, 可以多个
- 常规索引: 快速定位数据, 可以多个
- 前缀索引: 在文本类型如CHAR,VARCHAR,TEXT类列上创建索引时，可以指定索引列的长度. 但是数值类型不 能指定长度
- 联合索引: 多个列组合的索引 (最左前缀匹配原则, 索引下推, 避免回表, select *)
- 全文索引: 查找文本中的关键词. 像es一样. 可以多个

innodb中根据索引的物理存储形式, 又可以分为两种

- 聚集索引: 一般主键索引就是聚集索引, 且只有一个. 索引的叶子节点是id, id下挂了行数据.
- 二级索引: 索引的叶子节点是该列的值, 下面挂了id
    - 如果走二级索引, 那么就先从二级索引中拿到id, 再根据id从聚集索引中查行数据. 这个过程叫回表, 一般要避免回表(不要使用select *).

为什么InnoDB存储引擎选择使用B+树索引结构? (高)

- 相对于二叉树，层级更少，搜索效率高；
- B树无论是叶子节点还是非叶子节点，都会保存数据，这样导致一页中存储的键值减少，指针跟着减少，要同样保存大量数据，只能增加树的高度，导致性能降低；

- 相对Hash索引，B+树支持范围匹配及排序操作；

什么是覆盖索引/什么是回表 (高)

如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为 **覆盖索引（Covering Index)**

- 在 InnoDB 存储引擎中，非主键索引的叶子节点包含的是主键的值
- 这意味着，当使用非主键索引进行查询时，数据库会先找到对应的主键值，然后再通过主键索引来定位和检索完整的行数据。这个过程被称为“回表”

覆盖索引即需要查询的字段正好是索引的字段，那么直接根据该索引，就可以查到数据了，而无需回表查询。

为了尽可能避免回表, 所以往往会使用联合索引.

索引的使用原则/索引失效场景 (高)

(有非常多的场景和原则, 这里给出最常见的)

- 最左前缀匹配原则
    - 如果索引了多列（联合索引）,要遵守最左前缀法则。最左前缀法则指的是查询从索引的最左列开始，并且不跳过索引中的列。如果跳跃某一列，索引将部分失效（后面的字段索引失效）

    - ```
      联合索引(a, b, c)

      where a=1 and b=2  # 走联合索引
      where b=2 and a=1  # 走联合索引, 不会因为where后的字段顺序就失效
      where b=2 and c=1  # 不会走联合索引, 因为a字段在索引最左侧, where中没有a
      where c=1 and a=2  # 会走联合索引关于a的部分
      ```
- 索引列运算/函数会使索引失效;
    - `where substring(phone, 10, 2) = '12'`
- 模糊查询
    - `where name like '王%'` 尾部模糊, 索引生效
    - `where name like '王%王'` 尾部模糊, 索引生效
    - `where name like '%三'` 头部模糊, 索引失效
- 覆盖索引
    - 尽量使用覆盖索引（查询使用了索引，并且需要返回的列，在该索引中已经全部能够找到）,避免回表.
- 在WHERE子句中，如果在OR前的条件列是索引列，而在OR后的条件列不是索引列，那么索引会失效。

创建联合索引时需要注意什么？（高）

- 最左前缀匹配法则
- 把区分度大的字段排在前面性能会更高, 把性别这种区分度小的字段应该放在后面.
      ```

什么情况不走联合索引? (高)

设置联合索引(a,b,c), 查询条件如下

- where a=1 and b=2
  - 会走联合索引, 符合最左前缀匹配法则
- where b=2 and a=1
  - 会走联合索引, 符合最左前缀匹配法则. 注意, 最左前缀匹配法则与sql书写顺序无关
- where b=2 and c=1
  - 不会走联合索引, 不符合最左前缀匹配法则, 最左边的a不在
- where a=1 and c=2
  - 会走联合索引, 符合最左前缀匹配法则. 但是只会走a这部分的索引, 无法走c部分的索引, 因为没有b.

索引的优缺点？（高）

索引最大的好处是提高查询速度，但是索引也是有缺点的

- 需要占用物理空间，数量越大，占用空间越大；
- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增大；
- 会降低表的增删改的效率，因为每次增删改索引，B+树为了维护索引有序性，都需要进行动态维护。

索引不是万能的，它也是根据场景来使用的

索引设计原则 (高)

- 数据量大的, 查询频繁的列建立索引
- 对于经常where, order by, group by的列建立索引
- 选择区分度高的列做索引. 身份证号适合索引, 性别和状态不适合索引
- 字符串类型且比较长的, 可以使用前缀索引
- 尽量使用联合索引, 而不是单列索引, 联合索引很多时候可以覆盖索引, 避免回表
- 控制索引数量, 索引不是多多益善, 太多了占空间, 维护索引需要的代价也越多, 增删改反而会比较慢.

常见sql优化手段 (高)

- 查询语句中不要使用select *, 避免回表查询
- 数据库主键要保证自增(UUID不适合做主键), 且插入的数据主键也要交保证自增插入, 否则会引起页分裂
- 尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描
- 尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表 扫描
- 使用update时, where条件尽量用带索引的字段, 上行锁. InnoDB的行锁是**针对索引加的锁, 不是针对记录加的锁**.
- count(*)效率最高, 因为innodb做了优化.
- 表关联查询的效率高于子查询, 所以尽量少用子查询, 用关联查询替代.
- 关联查询时, on的条件列最好加上索引, 否则非常慢

SQL优化详解 (中)

插入优化

- insert优化

➢ 批量插入

```
Insert into tb_test values(1,'Tom'),(2,'Cat'),(3,'Jerry');
```

批量插入每次插入500~1000比较好
大于1000就分几次批量插入

➢ 手动提交事务

```
start transaction;
insert into tb_test values(1,'Tom'),(2,'Cat'),(3,'Jerry');
insert into tb_test values(4,'Tom'),(5,'Cat'),(6,'Jerry');
insert into tb_test values(7,'Tom'),(8,'Cat'),(9,'Jerry');
commit;
```

避免每添加一条数据, 就提交一次事务
避免多次提交事务

➢ 主键顺序插入

```
主键乱序插入 : 8  1  9  21  88  2  4  15  89  5  7  3
主键顺序插入 : 1  2  3  4  5  7  8  9  15  21  88  89
```

主键顺序插入的效率高于乱序插入

- 大批量插入数据

如果一次性需要插入大批量数据，使用insert语句插入性能较低，此时可以使用MySQL数据库提供的load指令进行插入。操作如下：

```
1,jdTmmKQlwu1,jdTmmKQlwu,jdTmmKQlwu,2020-10-13,1
2,BTJOeWjRiw2,BTJOeWjRiw,BTJOeWjRiw,2020-6-12,2
3,waQTJIIlHI3,waQTJIIlHI,waQTJIIlHI,2020-6-2,0
4,XmeFHwozIo4,XmeFHwozIo,XmeFHwozIo,2020-1-11,1
5,xRrvQSHcZn5,xRrvQSHcZn,xRrvQSHcZn,2020-10-18,2
6,gTDfGFNLEj6,gTDfGFNLEj,gTDfGFNLEj,2020-1-13,0
7,nBETIlVCle7,nBETIlVCle,nBETIlVCle,2020-9-27,1
8,vmePKKZjJU8,vmePKKZjJU,vmePKKZjJU,2020-10-20,2
9,pWjaLhJVaB9,pWjaLhJVaB,pWjaLhJVaB,2020-5-7,0
10,zimgGFPEQe10,zimgGFPEQe,zimgGFPEQe,2020-8-1,1
```

load

id	username	password	name	birthday	sex
1	jdTmmKQlwu1	jdTmmKQlwu	jdTmmKQlwu	2020-10-13	1
2	BTJOeWjRiw2	BTJOeWjRiw	BTJOeWjRiw	2020-06-12	2
3	waQTJIIlHI3	waQTJIIlHI	waQTJIIlHI	2020-06-02	0
4	XmeFHwozIo4	XmeFHwozIo	XmeFHwozIo	2020-01-11	1
5	xRrvQSHcZn5	xRrvQSHcZn	xRrvQSHcZn	2020-10-18	2
6	gTDfGFNLEj6	gTDfGFNLEj	gTDfGFNLEj	2020-01-13	0
7	nBETIlVCle7	nBETIlVCle	nBETIlVCle	2020-09-27	1
8	vmePKKZjJU8	vmePKKZjJU	vmePKKZjJU	2020-10-20	2
9	pWjaLhJVaB9	pWjaLhJVaB	pWjaLhJVaB	2020-05-07	0
10	zimgGFPEQe10	zimgGFPEQe	zimgGFPEQe	2020-08-01	1

```
#客户端连接服务端时，加上参数 --local-infile
mysql --local-infile -u root -p
#设置全局参数local_infile为1，开启从本地加载文件导入数据的开关
set global local_infile = 1;
#执行load指令将准备好的数据，加载到表结构中
load data local infile '/root/sql1.log' into table `tb_user` fields terminated by ',' lines terminated by '\n' ;
```

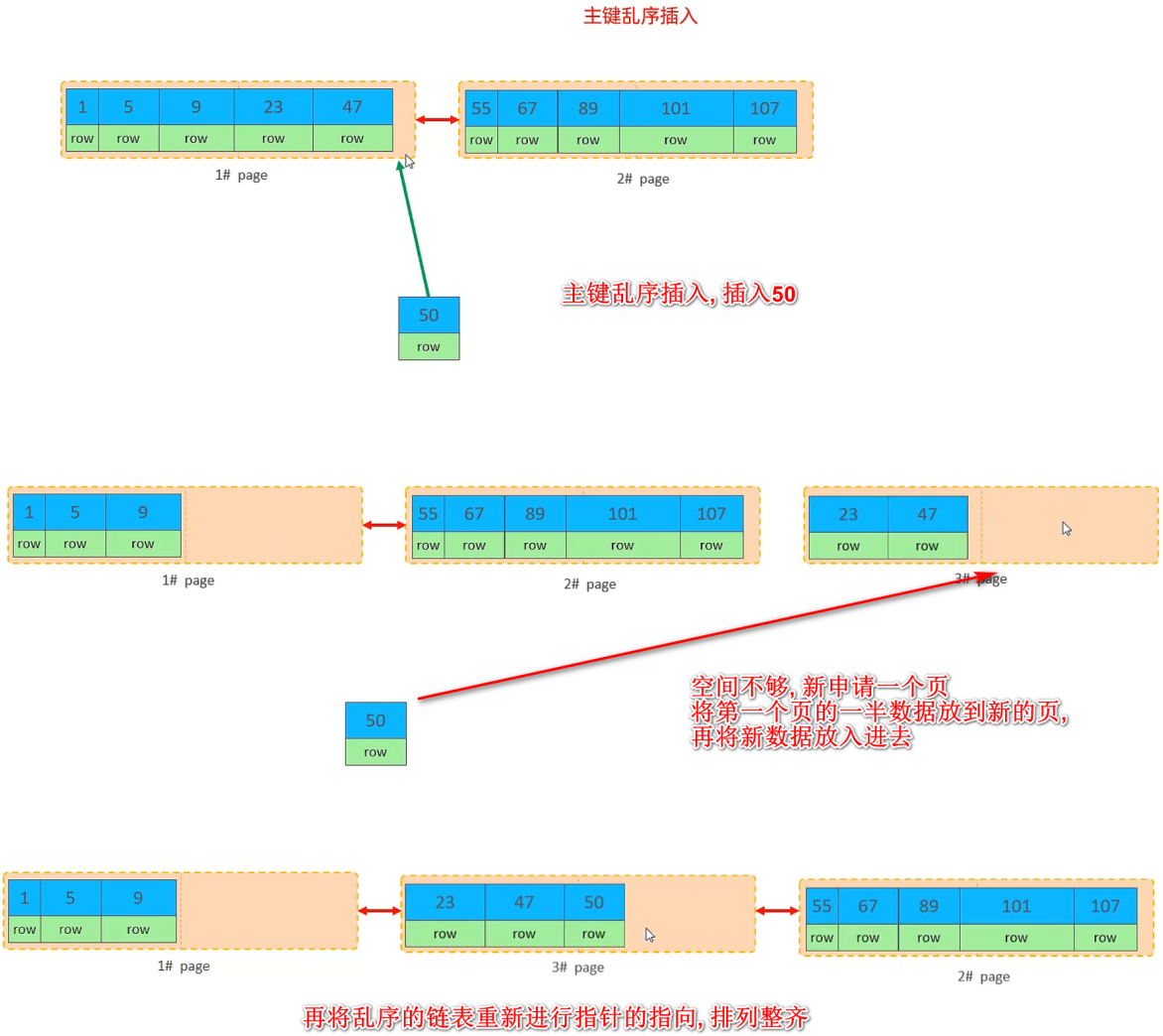
主键顺序插入性能高于乱序插入

主键优化

- 主键乱序会导致页分裂问题, 性能较差

- 页分裂

页可以为空，也可以填充一半，也可以填充100%。每个页包含了2-N行数据(如果一行数据多大，会行溢出)，根据主键排列。



- 主键设计原则
  - 满足业务需求的情况下，尽量降低主键的长度。 二级索引有很多个，二级索引的叶子节点存储了主键
当主键太长时，很浪费存储空间
  - 插入数据时，尽量选择顺序插入，选择使用AUTO_INCREMENT自增主键。主键顺序插入效率高
  - 尽量不要使用UUID做主键或者是其他自然主键，如身份证号。UUID和身份证都不是自增的，都是无序的，所以插入效率低
  - 业务操作时，避免对主键的修改。

order by优化

①. Using filesort : 通过表的索引或全表扫描，读取满足条件的数据行，然后在排序缓冲区sort buffer中完成排序操作，所有不是通过索引直接返回排序结果的排序都叫 FileSort 排序。

②. Using index : 通过有序索引顺序扫描直接返回有序数据，这种情况即为 using index，不需要额外排序，操作效率高。

```
#没有创建索引时，根据age, phone进行排序
explain select id,age,phone from tb_user order by age , phone;      file sort
#创建索引
create index idx_user_age_phone_aa on tb_user(age,phone);
#创建索引后，根据age, phone进行升序排序
explain select id,age,phone from tb_user order by age , phone;              index sort
#创建索引后，根据age, phone进行降序排序
explain select id,age,phone from tb_user order by age desc , phone desc ;    倒序 index sort
#根据age, phone进行降序一个升序，一个降序
explain select id,age,phone from tb_user order by age asc , phone desc;    age 用index sort, phone用file sort
#创建索引
create index idx_user_age_phone_ad on tb_user(age asc ,phone desc);
#根据age, phone进行降序一个升序，一个降序
explain select id,age,phone from tb_user order by age asc , phone desc;    创建索age升序, phone降序的索引后,
                                                                           再查找就是 index sort
```

- 根据排序字段建立合适的索引，多字段排序时，也遵循最左前缀法则。
- 尽量使用覆盖索引。
- 多字段排序, 一个升序一个降序，此时需要注意联合索引在创建时的规则（ASC/DESC）。
- 如果不可避免的出现filesort，大数据量排序时，可以适当增大排序缓冲区大小 sort_buffer_size(默认256k)。

group by 优化

```
#删除掉目前的联合索引 idx_user_pro_age_sta
drop index idx_user_pro_age_sta on tb_user;
#执行分组操作，根据profession字段分组
explain select profession , count(*) from tb_user group by profession ;      using temporary 使用临时表
#创建索引
Create index idx_user_pro_age_sta on tb_user(profession , age , status);
#执行分组操作，根据profession字段分组
explain select profession , count(*) from tb_user group by profession;      走索引
#执行分组操作，根据profession字段分组
explain select profession , count(*) from tb_user group by profession, age;     走索引
```

- 在分组操作时，可以通过索引来提高效率。
- 分组操作时，索引的使用也是满足最左前缀法则的。

limit优化

有张blog表, 字段 `id, content, create_time, ...` , 需要分页按创建时间倒序显示blog

常规分页查询sql: `select * from blog order by create_time desc limit 2000000, 10`

这里mysql需要排序前面的2000010条记录, 然后丢弃其他记录, 只返回2000000~2000010条记录. 所以存在严重的性能问题.

优化: 建立排序列索引 + id子查询避免回表

- 建立create_time的索引
- 使用子查询分页 `select * from blog inner join (select id from blog order by create_time desc limit 2000000, 10)`

一个常见又非常头疼的问题就是 limit 2000000,10 ，此时需要MySQL排序前2000010 记录，仅仅返回2000000 - 2000010 的记录，其他记录丢弃，查询排序的代价非常大 。

优化思路: 一般分页查询时，通过创建 覆盖索引 能够比较好地提高性能，可以通过覆盖索引加子查询形式进行优化。

原sql: `select * from user limit 200000, 10`

优化后的sql: `select * from user inner join (select id from user limit 2000000, 10) uid on user.id = uid.id`

覆盖索引(不用回表)

通过子查询查id组成的表, 起别名为 uid表, 和user表进行内连接
就能获取到最终分页结果

几种count对比

- count的几种用法
  - count（主键）

    InnoDB 引擎会遍历整张表，把每一行的 主键id 值都取出来，返回给服务层。服务层拿到主键后，直接按行进行累加(主键不可能为null)。
  - count（字段）

    没有not null 约束 : InnoDB 引擎会遍历整张表把每一行的字段值都取出来，返回给服务层，服务层判断是否为null，不为null，计数累加。
    有not null 约束：InnoDB 引擎会遍历整张表把每一行的字段值都取出来，返回给服务层，直接按行进行累加。
  - count（1）

    InnoDB 引擎遍历整张表，但不取值。服务层对于返回的每一行，放一个数字“1”进去，直接按行进行累加。
  - count（*）

    InnoDB引擎并不会把全部字段取出来，而是专门做了优化，不取值，服务层直接按行进行累加。

按照效率排序的话，count(字段) < count(主键 id) < count(1) ≈ count(*)，所以尽量使用 count(*)。

update优化

InnoDB的行锁是**针对索引加的锁, 不是针对记录加的锁**. 如果where条件是不带索引的字段, 那么就会是表锁. 如果where条件是带索引的字段, 那么是行锁. 并且该索引不能失效, 否则会从行锁升级为表锁. 表锁的并发性能低

在有事务的情况下, update进行更新的时候,

1. 如果where条件是id等带索引的字段, 则update会对该上行锁. 那么其他事务不能对该行进行操作, 但是可以对该表的其他行进行操作.
2. 如果where条件是不带索引的字段, 则update会上表锁, 其他事务对整张表都不能进行操作 (会阻塞).

因此, **使用update时, where条件尽量用带索引的字段, 上行锁.**

where优化

应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫 描，如： select id from t where num is null 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询： select id from t where num=0

什么时候不要使用索引 (高)

1. 经常增删改的列不要建立索引
2. 有大量重复的列不建立索引

3. 表记录太少不要建立索引

索引下推 (中)

可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，然后再去做回表, 从而减少了回表次数, 提升了性能.

组合索引满足最左匹配，但是遇到非等值判断时匹配停止。name like '陈%' 不是等值匹配，所以 age = 20 这里就用不上 (name,age) 组合索引了。如果没有索引下推，组合索引只能用到 name，age 的判定就需要回表才能做了。5.6之后有了索引下推，age = 20 可以直接在组合索引里判定

举例说明

假设存在表 `user`，其索引为 `(name, age)`，查询语句如下：

```sql
SELECT *
FROM user
WHERE name LIKE '王%'
  AND age = 30;
```

联合索引先按name排序，name一样再按age排序，如果是 name="张三" and age > 18，这个就能使用联合索引的所有列. 不需要索引下推.

- **无索引下推**：
  - 这里走联合索引先筛选出姓名以王开头的用户
  - 由于这里是模糊匹配, 不是等值匹配, 故获取所有以王开头的用户后, 他们的age不一定有序的.

        - 如 [张一, 20], [张二, 18], [张三, 30]
    - 所以无法继续使用联合索引的特性来筛选age, 只能拿到以王开头的用户的id, 去回表, 然后再筛选出age=30的人.
- **有索引下推**:
    - 走联合索引先筛选出姓名以王开头且同时age=30的用户
        - 结果为[张三, 30]
    - 回表只需要根据张三的id查即可

可以看到使用了索引下推后, 大大减少了回表操作.

新手可能不是很理解, 建议配合视频或者网上博客理解.

怎么找到慢sql? 可以从哪些角度优化? (高)

寻找慢sql:

- 打开慢查询日志
- 使用explain执行计划来对慢 SQL 进行分析, 查询是否使用了索引 (sql语句前加上 explain即可)

sql优化:

- 避免使用select *, 避免查询不需要的列
- 尝试给where, order by, limit后面的列添加索引
- 如果有索引, 避免给重复值很多的列添加索引
- 添加索引尽量使用联合索引, 尽量覆盖索引, 避免回表
- 对于update和delete慢, 应该where后面跟索引列, 使用行锁, 避免使用表锁
- 对于insert慢, 应该使用递增的主键, 避免页分裂
- 如果有limit, 应该先查id, 再根据id查询 (覆盖索引+子查询优化)
- 尽量使用count(*)

- 避免索引失效
- 数据量大使用分库分表
- 可以加缓存, 加es

什么是慢查询日志(slow query log) (高)

慢查询日志记录了执行时间超过 long_query_time（默认是 10s，通常设置为 1s）的所有查询语句，在解决 SQL 慢查询（SQL 执行时间过长）问题的时候经常会用到

找到慢 SQL 是优化 SQL 语句性能的第一步，然后再用EXPLAIN 命令可以对慢 SQL 进行分析，获取执行计划的相关信息

explain执行计划 (高)

```
#先执行一条sql
select * from user;

#在该sql前加上explain关键字
explain select * from user;
```

- explain执行计划

EXPLAIN 执行计划各字段含义：

- Id

  select查询的序列号，表示查询中执行select子句或者是操作表的顺序(id相同，执行顺序从上到下；id不同，值越大，越先执行)。

- select_type

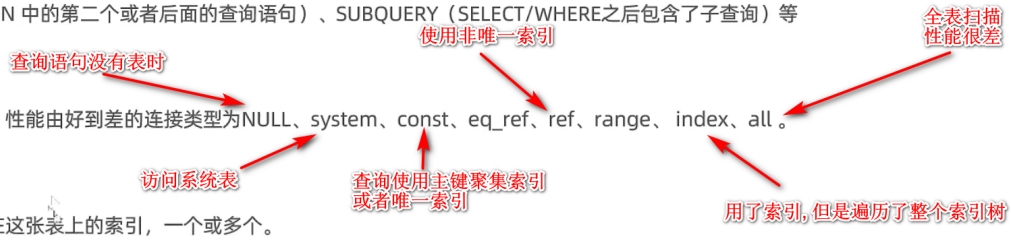
  表示 SELECT 的类型，常见的取值有 SIMPLE（简单表，即不使用表连接或者子查询）、PRIMARY（主查询，即外层的查询）、UNION（UNION 中的第二个或者后面的查询语句）、SUBQUERY（SELECT/WHERE之后包含了子查询）等

- type

  表示连接类型，性能由好到差的连接类型为NULL、system、const、eq_ref、ref、range、 index、all 。

- possible_key

  显示可能应用在这张表上的索引，一个或多个。



- Key

  实际使用的索引，如果为NULL，则没有使用索引。

- Key_len

  表示索引中使用的字节数， 该值为索引字段最大可能长度，并非实际使用长度，在不损失精确性的前提下， 长度越短越好 。

- rows

  MySQL认为必须要执行查询的行数，在innodb引擎的表中，是一个估计值，可能并不总是准确的。

- filtered

  表示返回结果的行数占需读取行数的百分比， filtered 的值越大越好。

  最后一个字段Extra, 代表额外信息, 额外信息会展示没有查到的信息

重要点关注: type, prossible_key, key, key_len, extra.

发现查询速度很慢，怎么解决 (高)

- 分析查询语句：使用EXPLAIN命令分析SQL执行计划，找出慢查询的原因，比如是否使用了全表扫描，是否存在索引未被利用的情况等，并根据相应情况对索引进行适当修改。
- 创建或优化索引：根据查询条件创建合适的索引，特别是经常用于WHERE子句的字段、Orderby排序的字段、Join连表查询的字典、groupby的字段，并且如果查询中经常涉及多个字段，考虑创建联合索引
- 避免索引失效：比如不要用左模糊匹配、函数计算、表达式计算等等。
- 查询优化：避免使用 `SELECT *`,只查询真正需要的列；使用覆盖索引，即索引包含所有查询的字段；联
  表查询最好要以小表驱动大表，并且被驱动表的字段要有索引，当然最好通过穴余字段的设计，避免联
  表查询。
- 分页优化：针对深分页的查询优化

- 优化数据库表：如果单表的数据超过了千万级别，考虑是否需要将大表拆分为小表，减轻单个表的查询压力。也可以将字段多的表分解成多个表，有些字段使用频率高，有些低，数据量大时，会由于使用频率低的存在而变慢，可以考虑分开
- 使用缓存技术：引入缓存, 存储热点数据和频繁查询的结果

Explain发现执行的索引不正确的话，怎么办？（高）

可以使用force index, 强制走索引

事务 (常考)

什么是数据库事务/事务四大特性（高）

事务: 一系列sql语句, 要么全成功, 要么全失败.

原子性 (Atomicity): 事务是不可分割的最小单元, n个连续操作失败了一个, 前面的操作回滚 (要么都成功, 要么都失败)

- 原子性通过undolog回滚来实现

一致性(Consistency): 执行事务前后，数据总量保持一致. 例如转账业务中，无论事务是否成功，转账者和收款人的总额应该是不变的;

- 保证了其他三个特性, 一致性就自然实现了.

持久性 (Durability): 持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的, 无法撤销

- redolog来实现

隔离性 (Isolation): 多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离, 保证每个事务不受并发影响, 独立执行.

- mvcc+锁 配合undolog来实现

隔离性产生的问题 (高)

脏读: 一个事务读取到另一个事务未提交的数据

1. 在事务A执行过程中，事务A对数据资源进行了修改，事务B读取了事务A修改后的数据。
2. 由于某些原因，事务A并没有完成提交，发生了RollBack操作，则事务B读取的数据就是脏数据。

这种读取到另一个事务未提交的数据的现象就是脏读(Dirty Read)。

不可重复读:

- 事务B读取了两次数据资源，在这两次读取的过程中事务A修改了数据，导致事务B在这两次读取出来的数据不一致。
- 这种在同一个事务中，前后两次读取的数据不一致的现象就是不可重复读(Nonrepeatable Read)。

幻读:

- 事务A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，好像出现了幻觉。(由于解决了不可重复读, 所以该事务读取不到别的事务已提交的数据)
- 幻读和不可重复读有些类似，但是幻读强调的是集合的增减，而不是单条数据的更新。(比如第一次读是有0条数据, 但是第二次读却有了1条数据).

不可重复读和幻读区别: **不可重复读**的重点是修改比如多次读取一条记录发现其中**某些列的值被修改**，**幻读**的重点在于新增或者删除比如多次读取一条记录发现**记录增多或减少**了

事务的隔离级别 (高)

为了解决以上的问题，主流的关系型数据库都会提供四种事务的隔离级别。事务隔离级别从低到高分别是：读未提交、读已提交、可重复读、串行化。

事务隔离级别等级越高，越能保证数据的一致性和完整性，但是执行效率也越低。

所以在设置数据库的事务隔离级别时需要做一下权衡，**MySQL默认是可重复读的级别。**

读未提交

- 读未提交(Read Uncommitted)，是最低的隔离级别，所有的事务都可以看到其他未提交的事务的执行结果。不能解决脏读，可重复读，幻读，所以很少应用于实际项目。

读已提交

- 读已提交(Read Committed)， 在该隔离级别下，一个事务的更新操作结果只有在**该事务提交之后，另一个事务才可能读取到同一笔数据更新后的结果。**
- 可以防止脏读，但是不能解决可重复读和幻读的问题。

可重复读 (mysql默认隔离级别)

- 可重复读(Repeatable Read)，MySQL默认的隔离级别。

- **可重复读是快照读**, 在该隔离级别下，一个事务多次读同一个数据, 实际上读的是数据快照, 其他事务修改数据在当前事务是不可见的, 这样就可以保证在同一个事务内两次读到的数据是一样的。
- 可以防止脏读、不可重复读、第一类更新丢失、第二类更新丢失的问题，不过还是会出现幻读。

串行化

- 串行化(Serializable)，这是最高的隔离级别。
- 它要求事务序列化执行，事务只能一个接着一个地执行，不能并发执行(会阻塞)。
- 在这个级别，可以解决上面提到的所有并发问题，但可能导致大量的超时现象和锁竞争，通常不会用这个隔离级别

注意: 事务的隔离级别越高, 数据安全性就越高, 但是执行效率越低. 事务的隔离级别越低, 执行效率就越高, 但是数据安全性就越低.

MySQL 的隔离级别怎么实现的 (中)

MySQL 的隔离级别基于锁和 MVCC 机制共同实现的。

- SERIALIZABLE 隔离级别，是通过锁来实现的
- 除了 SERIALIZABLE 隔离级别，其他的隔离级别都是基于 MVCC 实现
- 不过， SERIALIZABLE 之外的其他隔离级别可能也需要用到锁机制，就比如 REPEATABLE-READ 在当前读情况下需要使用加锁读来保证不会出现幻读

单条update语句是原子性的吗？ (中)

- 是原子性的
- 主要通过锁+undolog日志保证原子性的
  - 执行update的时候，会加行级别锁，保证了一个事务更新一条记录的时候，不会被其他事务干扰。
  - 事务执行过程中，会生成undolog,如果事务执行失败，就可以通过undolog日志进行回滚。

MVCC

什么是 MVCC (中)

MVCC, 多版本并发控制

指维护一个数据的多个版本，使得读写操作没有冲突, 具体实现就是快照读, 快照读为MySQL实现MVCC提供了一个非阻塞读功能

MVCC的具体实现，还需要依赖于数据库记录中的隐式字段、undo log日志、readView。

MVCC 可以为数据库解决什么问题（中)

在并发读写数据库时, 可以做到在 读 (select) 操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能.

同时还可以解决脏读、幻读、不可重复读等事务隔离问题

MVCC 的实现原理 (中)

MVCC的具体实现，依赖于数据库记录中的隐式字段(最近更新的事务id和回滚指针)、undo log日志、readView。

在内部实现中，InnoDB 通过数据行的 DB_TRX_ID(最近更新的事务 id) 和 Read View 来判断数据的可见性，如不可见，则通过数据行的 DB_ROLL_PTR(回滚指针) 找到 undo log 版本链中的历史版本。这就是快照读

每个事务读到的数据版本可能是不一样的，在同一个事务中，用户只能看到该事务创建 Read View 之前已经提交的修改和该事务本身做的修改

ReadView是什么 (中)

Read View是MVCC中用来判断数据的可见性的, 里面记录了活跃事务 id 列表, 全局事务中最大的事务 id 值, 创建该 Read View 的事务的事务 id等.

通过比较当前事务id和ReadView中记录的事务id, 就能知道该版本的记录对当前事务是否可见. 如不可见，则通过数据行的 DB_ROLL_PTR(回滚指针) 找到 undo log 版本链中的历史版本。这就是快照读

如果想详细了解readView中具体有什么, 怎么对比的, 建议网上查对应视频. 文字很难讲解清楚, 一般面试也不会问这么详细.

当前读与快照读 (中)

当前读: 读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁

- 对于我们日常的操作，如：select...lock in share mode(共享锁）,select...for update、update、insert、delete(排他锁）都是一种当前读
- **当前读**： 使用临键锁进行加锁来保证不出现幻读

快照读: 不加锁的select就是快照读，快照读读取的是记录数据的可见版本有可能是历史数据，不加锁

- 读已提交: 每次select都会生成一个快照读
- 可重复读: 事务开始后的第一个select才是快照读的地方
- 串行化: 快照度会退化为当前读
- **快照读**： 由 MVCC 机制来保证不出现幻读

MVCC是怎么实现不可重复读的 (中)

在读已提交下, 在事务中每一次执行快照读时生成ReadView, 这也就造成了每次读取就有不同ReadView, 那么就会读到已提交的事务修改的内容, 造成不可重复读的问题.

解决不可重复读主要靠readview, 在隔离级别为可重复读时, 仅在事务中第一次执行快照读时生成ReadView, 后续复用该ReadView.

由于后续复用了ReadView, 所以数据对当前事务的可见性和第一次是一样的, 所以从undolog中读到的数据快照和第一次是一样的, 即便过程中有其他事务修改也读不到.

MVCC是怎么防止幻读的 (中)

`InnoDB`存储引擎在 RR 级别下通过 `MVCC`和 `Next-key Lock`(临键锁) 来解决幻读问题

1、执行普通 `select`，此时会以 `MVCC` 快照读的方式读取数据

快照读: 避免加锁, 通过MVCC来进行控制, 使其他事务所做的更新对当前事务不可见, 从而防止幻读.

在快照读的情况下，RR 隔离级别只会在事务开启后的第一次查询生成 `Read View` 。所以在生成 `Read View` 之后其它事务所做的更新、插入记录版本对当前事务并不可见，实现了可重复读和防止快照读下的 “幻读”

2、执行 select...for update/lock in share mode、insert、update、delete 等为当前读

这些语句执行前都会查询最新版本的数据, 所以是当前读

当前读: 通过临键锁next-key-lock锁住空隙, 防止其他事务在查询的范围内插入数据, 从而防止幻读.

在当前读下，读取的都是最新的数据，如果其它事务有插入新的记录，并且刚好在当前事务查询范围内，就会产生幻读！`InnoDB` 使用 `Next-key Lock` 临键锁来防止这种情况。当执行当前读时，会锁定读取到的记录的同时，锁定它们的间隙，防止其它事务在查询范围内插入数据。只要我不让你插入，就不会发生幻读

但是MVCC并没有彻底防止幻读问题, 只是解决了大多数幻读问题, 在一些极端场景还是会有幻读问题.

锁

(锁这一节非常复杂, 锁非常多, 各种情况也非常多, 往往不是面试重点, 所以了解即可, 这里只给出常见考点)

锁的分类 (中)

全局锁：锁定数据库中的所有表。

- 全局锁就是**对整个数据库实例加锁**，加锁后整个实例就处于**只读状态**，后续的DML的写语句，DDL语句，已经更新操作的事务提交**语句都将被阻塞**
- 表级锁
  - 表锁
    - 每次操作锁住整张表
    - 开销小，加锁快
    - 并发度最低
  - 元数据锁（meta data lock,MDL)
    - MDL不需要显式使用，在访问一个表的时候会被自动加上。
    - MDL的作用：保证读写的正确性。
    - 如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，删了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。
    - 当对一个表做增删改查操作的时候，加MDL读锁；
    - 当要对表做结构变更操作的时候，加MDL写锁。
    - 读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性。
    - 如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。
- 行级锁

- 顾名思义，行锁就是针对数据表中行记录的锁（也有人称为记录锁）。
- 事务A更新了一行，而这时候事务B也要更新同一行，则必须等事务A的操作完成后才能进行更新。
- 特点：
  - 每次操作锁住一行数据
  - 开销大，加锁慢
  - 发生锁冲突的概率是最低的，并发度是最高的

共享锁和排他锁 (中)

- 共享锁（共享锁也叫读锁或S锁）
  - 共享锁锁定的资源可以被其他用户读取，但不能修改。
  - 在进行SELECT的时候，会将对象进行共享锁锁定，当数据读取完毕之后，就会释放共享锁，这样就可以保证数据在读取时不被修改。
  - 加锁语句：`SELECT xxx from xxx LOCK IN SHARE MODE;`
- 排他锁（排它锁也叫独占锁、写锁或X锁）
  - 排它锁锁定的数据只允许进行锁定操作的事务使用，其他事务无法对已锁定的数据进行查询或修改
  - 加锁语句: `SELECT xxx from xxx FOR UPDATE;`
  - 对数据进行更新的时候，也就是INSERT、DELETE或者UPDATE的时候，数据库也会自动使用排它锁，防止其他事务对该数据行进行操作。

当然, 共享锁和排他锁不仅可以锁住一行，也可以锁住一张表.

表级锁和行级锁的区别 (中)

表级锁（table-level locking)一锁就锁整张表, 是针对非索引字段加的锁

行级锁的粒度更小，只针对当前操作的行记录进行加锁

- InnoDB的行锁是针对索引字段加的锁，表级锁是针对非索引字段加的锁
- 当我们执行UPDATE、DELETE语句时，如果操作不走索引，就会升级为表锁

InnoDB 有哪几类行锁？ (中)

MySQL InnoDB 支持三种行锁定方式：

- **记录锁（Record Lock）** ：也被称为记录锁，属于单个行记录上的锁。
- **间隙锁（Gap Lock）** ：锁定一个范围，不包括记录本身。
  - `SELECT * FROM table WHERE id BETWEN 1 AND 10 FOR UPDATE;`
  - 即所有在`（1，10）`区间内的记录行都会被锁住，所有id 为 2、3、4、5、6、7、8、9 的数据行的插入会被阻塞，但是 1 和 10 两条记录行并不会被锁住
- **临键锁(Next-key Lock)**: 行锁和间隙锁组合，同时锁住数据，并锁住数据前面的间隙Gap. 相当于锁定一个范围，包含记录本身, 左闭右开
  - 有一个列age, 已有的记录中age分别为2, 8. 则潜在的临键锁为`(-∞, 2]` `(2,8]` `(8,+∞]`

数据库的表锁和行锁有什么作用？（中）

表锁

- 整体控制：表锁可以用来控制整个表的并发访问，当一个事务获取了表锁时，其他事务无法对该表进行任何读写操作，从而确保数据的完整性和一致性。
- 粒度大：表锁的粒度比较大，在锁定表的情况下，可能会影响到整个表的其他操作，可能会引起锁竞争和性能问题。
- 适用于大批量操作：表锁适合于需要大批量操作表中数据的场景，例如表的重建、大量数据的加载等。

行锁

- 细粒度控制：行锁可以精确控制对表中某行数据的访问，使得其他事务可以同时访问表中的其他行数据，在并发量大的系统中能够提高并发性能
- 减少锁冲突：行锁不会像表锁那样造成整个表的锁冲突，减少了锁竞争的可能性，提高了并发访问的效率
- 适用于频繁单行操作：行锁适合于需要频繁对表中单独行进行操作的场景

MySQL两个事务的update语句同时更新同一条数据，会发生什么情况？（中）

- 两个事务同时使用update, 首先要明确是当前读
- 当事务A对id=1这行记录进行更新时，会对主键id为1的记录加行锁
- 事务B对id=1进行更新时，发现已经有行锁了，就会陷入阻塞状态

两条update语句修改处理同一张表的不同范围的数据, 一个<5,一个>10, 会阻塞吗? (中)

得分情况.

如果update的where条件是索引列, 那么会加行锁.

- 第一条 update xxx where 索引列 < 5, 锁住的范围是（-无穷,5)
- 第二条 update xxx where 索引列 > 10, 锁住的范围是（10, +无穷)

如果两个update的where条件不是索引列, 那么由于没有用到索引, 所以会触发全表扫描, 会加表锁. 此时第二条update执行的时候，就会被阻塞.

日志

三大日志（高）

- **undo log（回滚日志）**： 主要**用于事务回滚和 MVCC**, 实现了事务中的**原子性**
- **redo log（重做日志）**： 主要**用于掉电重启等故障恢复**, 实现了事务中的**持久性**
- **binlog （归档日志/二进制日志）**： 主要**用于数据备份和主从复制**;

binlog主要记录了什么？有什么用？ （高）

binlog, 即二进制日志, 主要记录了对 MySQL 数据库执行了更改的所有操作(数据库执行的所有 DDL 和 DML 语句)

- 包括表结构变更（CREATE、ALTER、DROP TABLE...）

- 表数据修改（INSERT、UPDATE、DELETE...）
- 但不包括 SELECT、SHOW 这类不会对数据库造成更改的操作。

数据库的**数据备份、主备、主从**需要依靠`binlog`来同步数据，保证数据一致性。

redo log主要记录了什么？有什么用？（高）

redo log重做日志，记录的是事务提交时数据页的物理修改，是用来实现事务的持久性. 它让`MySQL`拥有了崩溃恢复能力.

redo log基本过程（高）

`MySQL` 中数据是以页为单位，你查询一条记录，会从硬盘把一页的数据加载出来，加载出来的数据叫数据页，会放入到 `Buffer Pool` 中。

后续的查询都是先从 `Buffer Pool` 中找，没有命中再去硬盘加载，减少硬盘 `IO` 开销，提升性能。

更新表数据的时候，也是如此，发现 `Buffer Pool` 里存在要更新的数据，就直接在 `Buffer Pool` 里更新.

一个事务提交之后，我们对 Buffer Pool 中对应的页的修改可能还未持久化到磁盘。这个时候，如果 MySQL 突然宕机的话，这个事务的更改是不是直接就消失了呢?

MySQL InnoDB 引擎使用 redo log 来保证事务的持久性，redo log 主要做的事情就是记录页的修改，比如某个页面某个偏移量处修改了几个字节的值以及具体被修改的内容是什么.

在事务提交时，我们会将 redo log 按照刷盘策略刷到磁盘上去. 即使 MySQL 宕机了，重启之后也能恢复未能写入磁盘的数据，从而保证事务的持久性。也就是说，redo log 让 MySQL 具备了崩溃回复能力.

为什么事务提交后不直接将Buffer Pool的数据同步到磁盘（中)

实际上，数据页大小是`16KB`，刷盘比较耗时，可能就修改了数据页里的几 `Byte` 数据，有必要把完整的数据页刷盘吗?

而且数据页刷盘是**随机写**，因为一个数据页对应的位置可能在硬盘文件的随机位置，所以**性能是很差**。

如果是**写** `redo log`，一行记录可能就占几十 `Byte`，只包含表空间号、数据页号、磁盘文件偏移 量、更新值，**内容少**, 再加上是**顺序写**，所以**刷盘速度很快**。

所以用 `redo log` 形式记录修改内容，性能会远远超过刷数据页的方式，这也让数据库的并发能力更强。

binlog 和 redolog 有什么区别？ (中)

1. binlog 主要用于数据库还原，属于数据级别的数据恢复，主从复制是 binlog 最常见的一个应用场景。redolog 主要用于保证事务的持久性，属于事务级别的数据恢复。
2. redolog 属于 InnoDB 引擎特有的，binlog 属于所有存储引擎共有的，因为 binlog 是 MySQL 的 Server 层实现的。
3. redolog 属于物理日志，主要记录的是某个页的修改。binlog 属于逻辑日志，主要记录的是数据库执行的所有 DDL 和 DML 语句。

4. binlog 通过追加的方式进行写入，大小没有限制。redo log 采用循环写的方式进行写入，大小固定，当写到结尾时，会回到开头循环写日志。
    - 循环写日志是否会覆盖: CheckPoint 机制可以帮助解决这个问题。一旦不够用需要覆盖之前的日志内容时，为保证被覆盖的日志内容是不再需要的、无用的，则需要将 Buffer Pool 中的脏页同步到硬盘中，并进行 Checkpoint 操作.

为什么需要 redo log（高)

- **实现事务的持久性，让 MySQL 有 崩溃恢复 的能力**，能够保证 MySQL 在任何时间段突然崩溃，重启后之前已提交的记录都不会丢失;
- **将写操作从「随机写」变成了「顺序写」**，提升 MySQL 写入磁盘的性能。

两阶段提交是什么？(中)

在执行更新语句过程，会记录redo log与bin log两块日志，以基本的事务为单位，redo log在事务执行过程中可以不断写入，而bin log只有在提交事务时才写入，所以redo log与bin log的写入时机不一样。

假设`id=2`的记录，字段c值是0，把字段c值更新成1, sql为 `update T set c=1 where id=2`

假设执行过程中写完`redo log`日志后，`binlog`日志写期间发生了异常，会出现什么情况呢?

由于binlog没写完就异常，这时候binlog里面没有对应的修改记录。但是redo log中有数据. 两份日志数据不一致.

- 在主从架构中, 主库通过redo log恢复数据后, 主库的c为0
- 从库同步主库的binlog, 从库的c为1
- 此时就会出现主从数据不一致的场景

为了解决两份日志之间的逻辑一致问题，`InnoDB`存储引擎使用**两阶段提交**方案。将`redo log`的写入拆成了两个步骤`prepare`和`commit`，这就是**两阶段提交**。

- 开始事务 -> 更新数据 -> 写入redo log(redo log prepare阶段) -> 提交事务(写入binlog, redolog 设置为commit)

此时, 如过发生写入binlog异常. 那么mysql根据redo log进行日志数据恢复时, 会发现redo log处于prepare阶段，并且没有对应`binlog`日志, 那么就会回滚事务.

`redo log`设置`commit`阶段发生异常，那会不会回滚事务呢？ 并不会回滚事务 , 虽然`redo log`是处于`prepare`阶段，但是能通过事务`id`找到对应的`binlog`日志，所以`MySQL`认为是完整的，就会提交事务恢复数据

undo log（高）

Undo Log（回滚日志）记录了事务操作前的数据状态，确保事务回滚时能恢复原始数据，并为并发事务提供数据的历史版本。

核心作用

- **事务回滚（Rollback）**：当事务执行失败或显式调用 `ROLLBACK` 时，通过 Undo Log 将数据恢复到修改前的状态。
- **MVCC（多版本并发控制）**：提供数据的历史版本，使其他事务能读取到一致的快照（Read View），避免读写冲突。