

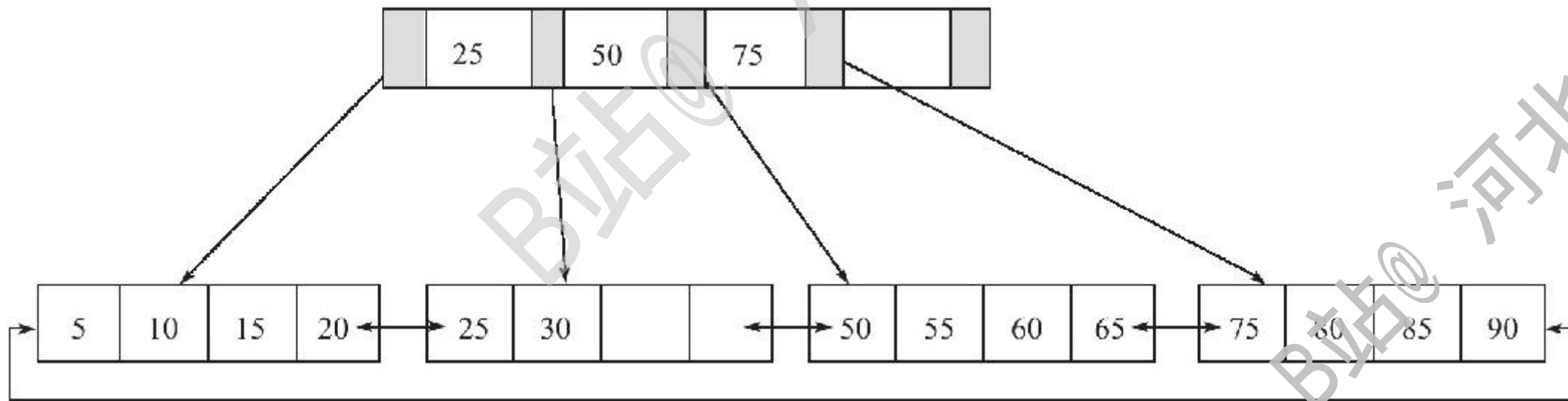
# 1. MySQL 数据库使用的索引数据结构是什么？

B+树。

B+树是为磁盘或其他直接存取辅助设备设计的一种**平衡查找树**。在B+树中，所有记录节点都是按键值的大小顺序存放在同一层的叶子节点上，由各叶子节点指针进行连接。

B+索引在数据库中有一个特点是**高扇出性**，因此在数据库中，B+树的高度一般都在2~4层，这也就是说查找某一键值的行记录时最多只需要2到4次IO，这倒不错。因为当前一般的机械磁盘每秒至少可以做100次IO，2~4次的IO意味着查询时间只需0.02~0.04秒。

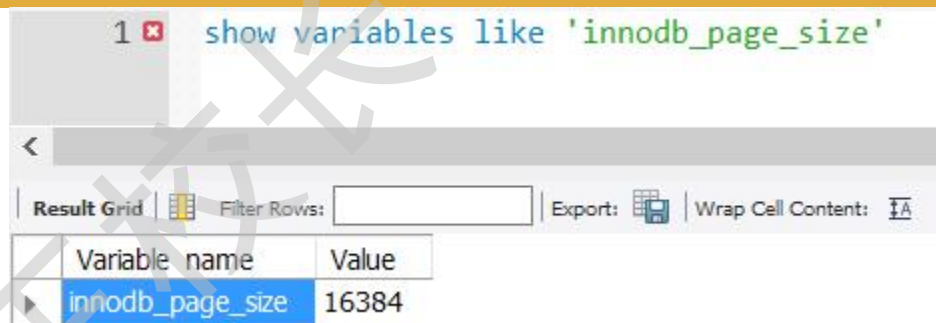
数据库中的B+树索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。



## 2. 假设每条sql信息为1kb，主键ID为bigint型，一颗高度为4的b+树能存储多少数据

在 innodb 存储引擎里面，最小的存储单元是页（page），一个页的大小是 16KB。如果我们在数据库的命令行输入如下命令，那么可以返回如右图所示。

这就说明了一个页的大小为 16384B，也就是 16kb。



```
1 show variables like 'innodb_page_size'
```

Variable name	Value
innodb_page_size	16384

假设一行数据的大小是 1k，那么一个页可以存放 16 行这样的数据。那如果想查找某个页里面的一个数据的话，得首先找到他所在的页。innodb 存储引擎我们都知道使用 B + 树的结构来组织数据。如果是在主键上建立的索引就是聚簇索引，即**只有在叶子节点才存储行数据，而非叶子节点里面的内容其实是键值和指向数据页的指针。**

因此，我们首先解决一个简单一点的问题：那么如果是 2 层的 B + 树，最多可以存储多少行数据？

如果是 2 层的 B + 树，即存在一个根节点和若干个叶子节点，那么这棵 B + 树的存放总记录数为：根节点指针数 \* 单个叶子节点记录行数。因为单个页的大小为 16kb，而一行数据的大小为 1kb，也就是说一页可以存放 16 行数据。然后因为非叶子节点的结构是：“页指针 + 键值”，我们假设主键 ID 为 bigint 类型，长度为 8 字节（byte），而指针大小在 InnoDB 源码中设置为 6 字节（byte），这样一共 14 字节（byte），因为一个页可以存放 16k 个 byte，所以一个页可以存放的指针个数为  $16384/14=1170$  个。因此一个两层的 B + 树可以存放的数据行的个数为： $1170*16=18720$ （行）。

那么对于高度为3的B+树呢？也就是说第一层的页，即根页可以存放 1170 个指针，然后第二层的每个页也可以存放 1170 个指针。这样一共可以存放  $1170 \times 1170$  个指针，所以一共可以存放  $1170 \times 1170 \times 16 = 21902400$ （2千万左右）行记录。也就是说一个三层的 B + 树就可以存放千万级别的数据了。而每经过一个节点都需要 IO 一次，把这个页数据从磁盘读取到缓存，也就是说读取一个数据只需要三次 IO。

继续来说，高度为4的B+树呢？  $1170 \times 1170 \times 1170 \times 16$  约等于 2000万\*1000。5个 2000 万是 1个亿。1000个 2000 万就是 200亿。

### 3. 为什么选用B+树做索引而不选用二叉树或者B树

b 树 (balance tree) 和 b + 树应用在数据库索引, 可以认为是 m 叉的多路平衡查找树, 但是从理论上讲, 二叉树查找速度和比较次数都是最小的, 为什么不用二叉树呢?

因为我们要考虑磁盘 IO 的影响, 它相对于内存来说是很慢的。数据库索引是存储在磁盘上的, 当数据量大时, 就不能把整个索引全部加载到内存了, 只能逐一加载每一个磁盘页 (对应索引树的节点)。所以我们要减少 IO 次数, 对于树来说, IO 次数就是树的高度, 而 “矮胖” 就是 b 树的特征之一, 它的每个节点最多包含 m 个孩子, m 称为 b 树的阶。

为什么不用B树呢?

b + 树, 是 b 树的一种变体, 查询性能更好。

b + 树相比于 b 树的查询优势:

- 1.b + 树的中间节点不保存数据, 所以磁盘页能容纳更多节点元素, 更 “矮胖”。B 树不管叶子节点还是非叶子节点, 都会保存数据, 这样导致在非叶子节点中能保存的指针数量变少 (有些资料也称为扇出), 指针少的情况下要保存大量数据, 只能增加树的高度, 导致 IO 操作变多, 查询性能变低;
- 2.b + 树查询必须查找到叶子节点, b 树只要匹配到即可直接返回。因此 b + 树查找更稳定 (并不慢), 必须查找到叶子节点; 而B树, 如果数据在根节点, 最快, 在叶子节点最慢, 查询效率不稳定。
- 3.对于范围查找来说, b + 树只需遍历叶子节点链表即可, 并且不需要排序操作, 因为叶子节点已经对索引进行了排序操作。b 树却需要重复地中序遍历, 找到所有的范围内的节点。

## 4. 为什么用 B+ 树做索引而不用哈希表做索引?

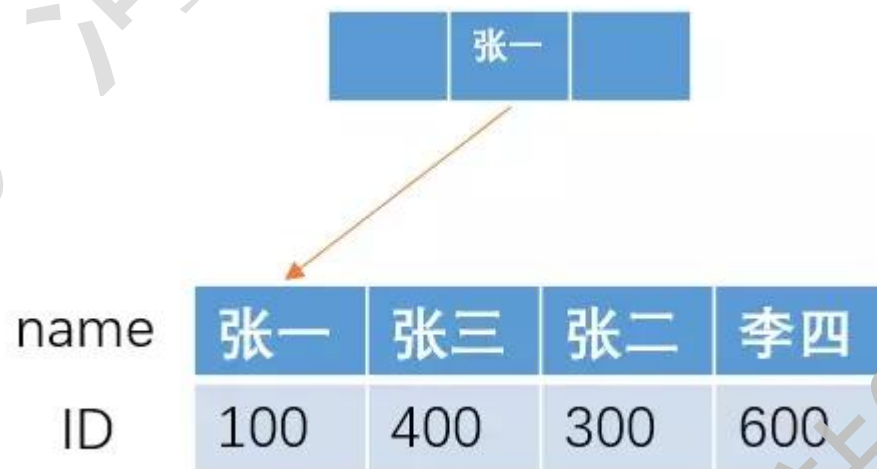
- 1、**模糊查找不支持**：哈希表是把索引字段映射成对应的哈希码然后再存放在对应的位置，这样的话，如果我们要进行模糊查找的话，显然哈希表这种结构是不支持的，只能遍历这个表。而 B + 树则可以通过最左前缀原则快速找到对应的数据。
- 2、**范围查找不支持**：如果我们要进行范围查找，例如查找 ID 为 100 ~ 400 的人，哈希表同样不支持，只能遍历全表。
- 3、**哈希冲突问题**：索引字段通过哈希映射成哈希码，如果很多字段都刚好映射到相同值的哈希码的话，那么形成的索引结构将会是一条很长的链表，这样的话，查找的时间就会大大增加。

## 5. 什么是最左前缀原则？（基于 MySQL 的 InnoDB 引擎）

例如右侧的表：

ID	name	age
100	张一	10
300	张二	20
400	张三	40
600	李四	10

如果我们按照 name 字段来建立索引的话，采用 B + 树的结构，大概的索引结构如右图：



如果我们要进行**模糊查找**，查找 name 以 “张 ” 开头的所有人的 ID，即 sql 语句为

```
select ID from table where name like '张%'
```

由于在 B + 树结构的索引中，索引项是按照索引定义里面出现的字段顺序排序的，索引在查找的时候，可以快速定位到 ID 为 100 的张一，然后直接向右遍历所有张开头的人，直到条件不满足为止。也就是说，我们找到第一个满足条件的人之后，直接向右遍历就可以了，由于索引是有序的，所有满足条件的人都会聚集在一起。

而这种定位到最左边，然后向右遍历寻找，就是我们所说的最左前缀原则。



## 6. 什么是聚集索引

InnoDB存储引擎表是索引组织表，即表中数据按照主键顺序存放。而**聚集索引**（clustered index）就是**按照每张表的主键构造一棵B+树**，同时叶子节点中存放的即为整张表的行记录数据，也将聚集索引的叶子节点称为数据页。每个数据页都通过一个双向链表来进行链接。

由于实际的数据页只能按照一棵B+树进行排序，因此**每张表只能拥有一个聚集索引**。在多数情况下，查询优化器倾向于采用聚集索引。因为聚集索引能够在B+树索引的叶子节点上直接找到数据。此外，由于定义了数据的逻辑顺序，它对于**主键**的排序查找和范围查找速度非常快。。叶子节点的数据就是用户所要查询的数据

如:用户需要查询一张注册用户的表，查询最后注册的10位用户，由于B+树索引是双向链表的，用户可以快速找到最后一个数据页，并取出10条记录

```
SELECT * FROM Profile ORDER BY id LIMIT 10;
```

虽然使用ORDER BY对**主键id**记录进行排序，但是在实际过程中并没有进行所谓的filesort操作，而这就是因为聚集索引的特点。另一个是范围查询（range query），即如果要查找主键某一范围内的数据，通过叶子节点的上层中间节点就可以得到页的范围，之后直接读取数据页即可。如：SELECT \* FROM Profile where id > 1 and id <100;



## 7. 什么是辅助索引

对于辅助索引（Secondary Index，也称非聚集索引），叶子节点并不包含行记录的全部数据。叶子节点除了**包含键值**以外，每个叶子节点中的索引行中还包含了一个**书签（bookmark）**。该书签用来告诉InnoDB存储引擎哪里可以找到与索引相对应的行数据。由于InnoDB存储引擎表是索引组织表，因此InnoDB存储引擎的辅助索引的**书签就是相应行数据的聚集索引键**。

辅助索引的存在并不影响数据在聚集索引中的组织，因此**每张表上可以有多个辅助索引**。当通过辅助索引来寻找数据时，InnoDB存储引擎会遍历**辅助索引并通过叶级别的指针获得指向主键索引的主键**，然后再通过主键索引来找到一个完整的行记录。举例来说，如果在一棵高度为3的辅助索引树中查找数据，那需要对这棵辅助索引树遍历3次找到指定主键，如果聚集索引树的高度同样为3，那么还需要对聚集索引树进行3次查找，最终找到一个完整的行数据所在的页，因此一共需要6次逻辑IO访问以得到最终的一个数据页。

## 8. 说说FIC (Fast Index Creation) 原理，与普通Index操作有什么不同

MySQL 5.5版本之前（不包括5.5）存在的一个普遍被人诟病的问题是：MySQL数据库对于索引的添加或者删除的这类DDL操作，MySQL数据库的操作过程为：

- ❑ 首先**创建**一张新的**临时表**，表结构为通过命令ALTER TABLE新定义的结构。
- ❑ 然后把原表中数据**导入到临时表**。
- ❑ 接着**删除原表**。
- ❑ 最后把临时表**重命名**为原来的表名。

可以发现，若用户对于一张大表进行索引的添加和删除操作，那么这会需要很长的时间。更关键的是，若有大量事务需要访问正在被修改的表，这意味着数据库服务不可用。MySQL数据库的索引维护始终让使用者感觉非常痛苦。

InnoDB存储引擎从InnoDB 1.0.x版本开始支持一种称为Fast Index Creation（快速索引创建）的索引创建方式——简称FIC。

对于辅助索引的创建，InnoDB存储引擎会对创建索引的表加上一个**S锁**。在创建的过程中，不需要重建表，因此速度较之前提高很多，并且数据库的可用性也得到了提高。删除辅助索引操作就更简单了，InnoDB存储引擎只需更新内部视图，并将辅助索引的空间标记为可用（不影响附注索引的使用，因为可读，后边的同时删除四个字非常传神），**同时删除**MySQL数据库内部视图上对该表的索引定义即可。

由于FIC在索引的创建的过程中对表加上了S锁，因此在创建的过程中只能对该表进行读操作，若有大量的事务需要对目标表进行写操作，那么数据库的服务同样不可用。此外，FIC方式只限定于辅助索引，对于主键的创建和删除同样需要重建一张表。

## 9. 有没有比FIC更好的方式?

虽然FIC可以让InnoDB存储引擎避免创建临时表，从而提高索引创建的效率。但正如前面面试题中所说的，索引创建时会阻塞表上的DML操作（除读操作）。OSC（一个FaceBook的PHP脚本）虽然解决了上述的部分问题，但是还是有很大的局限性。**MySQL 5.6版本开始支持Online DDL（在线数据定义）操作，其允许辅助索引创建的同时，还允许其他诸如INSERT、UPDATE、DELETE这类DML操作，这极大地提高了MySQL数据库在生产环境中的可用性。**

不仅是辅助索引，以下这几类DDL操作都可以通过“在线”的方式进行操作：

- ❑ 辅助索引的创建与删除
- ❑ 改变自增长值
- ❑ 添加或删除外键约束
- ❑ 列的重命名

使用语法：

```
ALTER TABLE tbl_name  
|ADD{INDEX|KEY}[index_name]  
[index_type](index_col_name,...)[index_option]...  
ALGORITHM[=]{DEFAULT|INPLACE|COPY}  
LOCK[=]{DEFAULT|NONE|SHARED|EXCLUSIVE}
```

**ALGORITHM**指定了创建或删除索引的算法，**COPY**表示按照MySQL 5.1版本之前的工作模式，即创建临时表的方式。**INPLACE**表示索引创建或删除操作不需要创建临时表。**DEFAULT**表示根据参数old\_alter\_table来判断是通过INPLACE还是COPY的算法，该参数的默认值为OFF，表示采用INPLACE的方式。

**LOCK**部分为索引创建或删除时对表添加锁的情况：

(1) **NONE**

执行索引创建或者删除操作时，对目标表不添加任何的锁，即事务仍然可以进行读写操作，不会收到阻塞。因此这种模式可以获得最大的并发度。

(2) **SHARE**

这和之前的FIC类似，执行索引创建或删除操作时，对目标表加上一个**S锁**。对于并发地读事务，依然可以执行，但是遇到写事务，就会发生等待操作。如果存储引擎不支持SHARE模式，会返回一个错误信息。

(3) **EXCLUSIVE**

在EXCLUSIVE模式下，执行索引创建或删除操作时，对目标表加上一个**X锁**。读写事务都不能进行，因此会阻塞所有的线程，这和COPY方式运行得到的状态类似，但是不需要像COPY方式那样创建一张临时表。

(4) **DEFAULT**

DEFAULT模式首先会判断当前操作是否可以使用NONE模式，若不能，则判断是否可以使用SHARE模式，最后判断是否可以使用EXCLUSIVE模式。也就是说**DEFAULT**会通过判断事务的最大并发性来判断执行DDL的模式。

InnoDB存储引擎实现Online DDL的原理是在执行创建或者删除操作的同时，将INSERT、UPDATE、DELETE这类DML操作日志写入到一个缓存中。待完成索引创建后再将重做应用到表上，以此达到数据的一致性。这个缓存的大小由参数innodb\_online\_alter\_log\_max\_size控制，默认的大小为128MB。

需要特别注意的是，由于Online DDL在创建索引完成后再通过重做日志达到数据库的最终一致性，这意味着在索引创建过程中，SQL优化器不会选择正在创建中的索引。

## 10. 如何选择表的列作为索引更加高效

并不是在所有的查询条件中出现的列都需要添加索引。对于什么时候添加B+树索引，一般的经验是，在访问表中很少一部分时使用B+树索引才有意义。对于性别字段、地区字段、类型字段，它们可取值的范围很小，称为低选择性不建议添加索引，当然也要根据自身项目和场景的需求。如：

```
SELECT * FROM student WHERE sex='M'
```

按性别进行查询时，可取值的范围一般只有'M'、'F'。因此上述SQL语句得到的结果可能是该表50%的数据（假设男女比例1：1），这时添加B+树索引是完全没有必要的。相反，如果某个字段的取值范围很广，几乎没有重复，即属于高选择性，则此时使用B+树索引是最适合的。

怎样查看索引是否是高选择性的呢？可以通过SHOW INDEX结果中的列Cardinality来观察。**Cardinality值非常关键，表示索引中不重复记录数量的预估值**。同时需要注意的是，Cardinality是一个预估值，而不是一个准确值，基本上用户也不可能得到一个准确的值。在实际应用中，**Cardinality/n\_rows\_in\_table应尽可能地接近1**。如果非常小，那么用户需要考虑是否还有必要创建这个索引。



举例：SELECT\*FROM **member** WHERE usernick='David'。表**member**大约有500万行数据。usernick字段上有一个唯一的索引。这时如果查找用户名为David的用户，将会得到如下的执行计划：

```
mysql> EXPLAIN SELECT*FROM member
-> WHERE usernick='David';
id:1
select_type:SIMPLE
table:member
type:const
possible_keys:usernick
key:usernick
key_len:62
ref:const
rows:1
Extra:
1 row in set(0.00 sec)
```

```
mysql> SHOW INDEX FROM member;

Table:member
... ..
Column_name:usernick
... ..
Cardinality:1
```

可以看到使用了usernick这个索引，这也符合之前提到的高选择性，即SQL语句选取表中较少行的原则。

## 11. Cardinality统计是预估值，如何尽量确保其准确性

因为MySQL数据库中有各种不同的存储引擎，而每种存储引擎对于B+树索引的实现又各不相同，所以对**Cardinality**的统计是放在存储引擎层进行的。

，在生产环境中，索引的更新操作可能是非常频繁的。如果每次索引在发生操作时就对其进行Cardinality的统计，那么将会给数据库带来很大的负担。另外需要考虑的是，如果一张表的数据非常大，如一张表有50G的数据，那么统计一次Cardinality信息所需要的时间可能非常长。这在生产环境下，也是不能接受的。因此，**数据库对于Cardinality的统计都是通过采样（Sample）的方法**来完成的。默认InnoDB存储引擎对**8个叶子节点**（Leaf Page）进行采用。采样的过程如下：

- 取得B+树索引中叶子节点的数量，记为A。
- 随机取得B+树索引中的8个叶子节点。统计每个页不同记录的个数，即为P1, P2, ..., P8。
- 根据采样信息给出Cardinality的预估值： $\text{Cardinality} = (P1 + P2 + \dots + P8) * A / 8$ 。

在InnoDB存储引擎中，Cardinality值是通过**对8个叶子节点预估而得的**，不是一个实际精确的值。

在InnoDB存储引擎中，Cardinality统计信息的更新发生在两个操作中：INSERT和UPDATE。根据前面的叙述，不可能在每次发生INSERT和UPDATE时就去更新Cardinality信息，这样会增加数据库系统的负荷，同时对于大表的统计，时间上也不允许数据库这样去操作。因此，**InnoDB存储引擎内部对更新Cardinality信息的策略为：**

- 表中**1/16的数据已发生过变化**。自从上次统计Cardinality信息后，表中1/16的数据已经发生过变化，这时需要更新Cardinality信息。

- stat\_modified\_counter > 2 000 000 000（20 亿）**。InnoDB存储引擎内部有一个计数器stat\_modified\_counter，用来表示发生变化的次数，当stat\_modified\_counter大于2 000 000 000时，则同样需要更新Cardinality信息。



当执行SQL语句ANALYZE TABLE、SHOW TABLE STATUS、SHOW INDEX以及访问INFORMATION\_SCHEMA架构下的表TABLES和STATISTICS时会导致InnoDB存储引擎去重新计算索引的Cardinality值。若表中的数据量非常大，并且表中存在多个辅助索引时，执行上述这些操作可能会非常慢。虽然用户可能并不希望去更新Cardinality值。

**Cardinality值非常关键，优化器会根据这个值来判断是否使用这个索引。**但是这个值并不是实时更新的，即并非每次索引的更新都会更新该值，因为这样代价太大了。如果需要更新索引Cardinality的信息，除了上边所说的，也可以使用**ANALYZE TABLE**命令。

在某些情况下可能会发生索引建立了却没有用到的情况，可能会出现Cardinality为NULL，致使优化器不选择使用索引。这时最好的解决办法就是做一次ANALYZE TABLE的操作。**因此在场景允许的情况下，建议在一个非高峰时间，对应用程序下的几张核心表做ANALYZE TABLE操作，这能使优化器和索引更好地为你工作。**

## 12. 什么是联合索引，联合索引的使用经验有吗？

联合索引是指对表上的多个列进行索引。

```
CREATE TABLE buy_log(  
  userid INT UNSIGNED NOT NULL,  
  buy_date DATE  
)ENGINE=InnoDB;
```

```
ALTER TABLE buy_log ADD KEY(userid);  
ALTER TABLE buy_log ADD KEY(userid,buy_date);
```

以上代码建立了两个索引来进行比较。两个索引都包含了userid字段。

**情况1：** 如果只对于userid进行查询，如：**SELECT\*FROM buy\_log WHERE userid=2;**

索引选择：优化器最终的选择是索引userid，因为该索引的叶子节点包含单个键值，所以理论上一个页能存放的记录应该更多。

**情况2：** **SELECT\*FROM buy\_log WHERE userid=1 ORDER BY buy\_date DESC LIMIT 3;**

索引选择：优化器使用了 (userid, buy\_date) 的联合索引userid\_2，因为在这个联合索引中buy\_date已经排序好了。根据该联合索引取出数据，无须再对buy\_date做一次额外的排序操作。

**情况 3：**假如三个字段的联合索引。如：对于联合索引（a，b，c）来说，下列语句同样可以直接通过联合索引得到结果，不需要filesort的排序操作：

```
SELECT...FROM TABLE WHERE a=xxx ORDER BY b
```

```
SELECT...FROM TABLE WHERE a=xxx AND b=xxx ORDER BY c
```

但是对于下面的语句，联合索引不能直接得到结果，其还需要执行一次filesort排序操作，因为索引（a，c）并未排序：

```
SELECT...FROM TABLE WHERE a=xxx ORDER BY c
```

## 13. 什么是覆盖索引，什么情况下优化器会选择使用覆盖索引

InnoDB存储引擎支持覆盖索引（covering index，或称索引覆盖），即从辅助索引中就可以得到查询的记录（此时不能够使用select \* 操作，只能对特定的索引字段进行select），而不需要查询聚集索引中的记录。使用覆盖索引的一个好处是辅助索引不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以减少大量的IO操作。

对于InnoDB存储引擎的辅助索引而言，由于其包含了主键信息，因此其叶子节点存放的数据为（primary key1, primary key2, ..., key1, key2, ...）。例如，下列语句都可仅使用一次辅助联合索引来完成查询：

```
SELECT key2 FROM table WHERE key1=xxx;
```

```
SELECT primary key2,key2 FROM table WHERE key1=xxx;
```

```
SELECT primary key1,key2 FROM table WHERE key1=xxx;
```

```
SELECT primary key1,primary key2, key2 FROM table WHERE key1=xxx;
```

覆盖索引的另一个好处是对某些统计问题而言的。  
还是对于上题创建的表buy\_log，要进行举例说明。

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

InnoDB存储引擎并不会选择通过查询聚集索引来进行统计。由于buy\_log表上还有辅助索引，而辅助索引远小于聚集索引，选择辅助索引可以减少IO操作。

```
CREATE TABLE buy_log(  
  userid INT UNSIGNED NOT NULL,  
  buy_date DATE  
)ENGINE=InnoDB;
```

```
ALTER TABLE buy_log ADD KEY(userid);  
ALTER TABLE buy_log ADD KEY(userid,buy_date);
```

在通常情况下，诸如 (a, b) 的联合索引，一般是不可以选择列b中所谓的查询条件。但是如果是统计操作，并且是覆盖索引的，则优化器会选择，如下述语句：

```
SELECT COUNT(*)FROM buy_log WHERE buy_date >='2011-01-01'AND buy_date < '2011-02-01'
```

表buy\_log有 (userid, buy\_date) 的联合索引，这里只根据列b进行条件查询，一般情况下是不能进行该联合索引的，但是这句SQL查询是统计操作，并且可以利用到覆盖索引的信息，因此优化器会选择该联合索引：

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	1	SIMPLE	buy_log	index	NULL	userid_2	8	NULL	7	Using where; Using index

从图5-27中可以发现列possible\_keys依然为NULL，但是列key为userid\_2，即表示 (userid, buy\_date) 的联合索引。在列Extra同样可以发现Using index提示，表示为覆盖索引。

## 14. 什么是离散读？

在某些情况下，当执行EXPLAIN命令进行SQL语句的分析时，会发现优化器并没有选择索引去查找数据，而是通过扫描聚集索引，也就是直接进行全表的扫描来得到数据。这种情况多发生于范围查找、JOIN链接操作等情况下。

假设表：t\_index。其中id为主键；c1与c2组成了联合索引（c1，c2）；此外，c1还是一个单独索引。进行如下查询操作：

```
SELECT * FROM t_index WHERE c1 > 1 and c1 < 100000;
```

可以看到表t\_index有（c1，c2）的联合主键，此外还有对于列c1的单个索引。上述这句SQL显然是可以通过扫描OrderID上的索引进行数据的查找。然而通过EXPLAIN命令，用户会发现优化器并没有按照OrderID上的索引来查找数据。

在最后的索引使用中，优化器选择了PRIMARY id 聚集索引，也就是表扫描（table scan），而非c1辅助索引扫描（index scan）。

这是为什么呢？因为如果强制使用c1索引，就会造成离散读。具体原因在于用户要选取的数据是整行信息，而c1作为辅助索引不能覆盖到我们要查询的信息，因此在对c1索引查询到指定数据后，还需要一次书签访问来查找整行数据的信息。虽然c1索引中数据是顺序存放的，但是再一次进行书签查找的数据则是无序的，因此变为了磁盘上的离散读操作。如果要求访问的数据量很小，则优化器还是会选择辅助索引，但是当访问的数据占整个表中数据的蛮大一部分时（一般是20%左右），优化器会选择通过聚集索引来查找数据。



## 15. 优化器如何优化离散读？你是如何避免离散读的

MySQL 5.6之前，优化器在进行离散读决策的时候，如果数据量比较大，会选择使用聚集索引，全表扫描。

MySQL 5.6版本开始支持Multi-Range Read (MRR) 优化。Multi-Range Read优化的目的是为了减少磁盘的随机访问，并且将随机访问转化为较为顺序的数据访问，这对于IO-bound类型的SQL查询语句可带来性能极大的提升。Multi-Range Read优化可适用于range, ref, eq\_ref类型的查询。

MRR优化有以下几个好处：

- ❑ MRR使数据访问变得较为顺序。在查询辅助索引时，首先根据得到的查询结果，按照主键进行排序，并按照主键排序的顺序进行书签查找。
- ❑ 减少缓冲池中页被替换的次数。（顺序查找可以对一个页进行顺序查找，无需离散加载数据页）
- ❑ 批量处理对键值的查询操作。

对于InnoDB和MyISAM存储引擎的范围查询和JOIN查询操作，MRR的工作方式如下：

- ❑ 将查询得到的辅助索引键值存放于一个缓存中，这时缓存中的数据是根据辅助索引键值排序的。
- ❑ 将缓存中的键值根据RowID进行排序。
- ❑ 根据RowID的排序顺序来访问实际的数据文件。



举例说明：SELECT \* FROM salaries WHERE salary > 10000 AND salary < 40000;

salary上有一个辅助索引idx\_s，因此除了通过辅助索引查找键值外，还需要通过书签查找来进行对整行数据的查询。当不启用Multi-Range Read特性时，看到的执行计划如图：

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	salaries	range	idx_s	idx_s	4	NULL	23378	Using index condition

若启用Mulit-Range Read特性，则除了会在列Extra看到Using index condition外，还会看见Using MRR选项

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
▶	1	SIMPLE	salaries	range	idx_s	idx_s	4	NULL	23378	Using index condition; Using MRR

是否启用 Multi-Range Read 的执行时间对比

	执行时间 ( 秒 )
不使用 Multi-Range Read	43.213
使用 Multi-Range Read	4.212

Multi-Range Read还可以将某些范围查询，拆分为键值对，以此来进行批量的数据查询。这样做的好处是可以在拆分过程中，直接过滤一些不符合查询条件的数据，例如：

```
SELECT*FROM t  
WHERE key_part1 >=1000 AND key_part1 < 2000  
AND key_part2=10000;
```

表t有 (key\_part1, key\_part2) 的联合索引，因此索引根据key\_part1, key\_part2的位置关系进行排序。若没有Multi-Read Range，此时查询类型为Range，SQL优化器会先将key\_part1大于1000且小于2000的数据都取出，即使key\_part2不等于1000。待取出行数据后再根据key\_part2的条件进行过滤。这会导致无用数据被取出。如果有大量的数据且其key\_part2不等于1000，则启用Multi-Range Read优化会使性能有巨大的提升。

倘若启用了Multi-Range Read优化，优化器会先将查询条件进行拆分，然后再进行数据查询。就上述查询语句而言，优化器会将查询条件拆分为 (1000, 10000) , (1001, 10000) , (1002, 10000) , ..., (1999, 10000) , 最后再根据这些拆分出的条件进行数据的查询。

我是如何优化的：在非必要的情况下，拒绝使用 select \* ; 在必须 select \* 的情况下，尽量使用MySQL 5.6+ 的版本开启MRR；在必须 select \* 的情况下且MySQL 小于 5.6 版本下，可以根据数据量进行离散读和聚集索引两种情况下的性能进行对比，必要时采用force index语句强制指定索引。

## 16. 什么是 ICP 优化

和Multi-Range Read一样，Index Condition Pushdown同样是MySQL 5.6开始支持的一种根据索引进行查询的优化方式。之前的MySQL数据库版本不支持Index Condition Pushdown，当进行索引查询时，首先根据索引来查找记录，然后再根据WHERE条件来过滤记录。在支持Index Condition Pushdown后，MySQL数据库会在取出索引的同时，判断是否可以进行WHERE条件的过滤，也就是将**WHERE的部分过滤操作放在了存储引擎层**。在某些查询下，可以大大减少上层SQL层对记录的索取（fetch），从而提高数据库的整体性能。

Index Condition Pushdown优化支持range、ref、eq\_ref、ref\_or\_null类型的查询，当前支持MyISAM和InnoDB存储引擎。当优化器选择Index Condition Pushdown优化时，可在执行计划的列Extra看到Using index condition提示。

MySQL 5.5 和 MySQL 5.6 中是否启用 Index Condition Pushdown 的执行时间对比

	执行时间（秒）
MySQL 5.5	46.738
MySQL 5.6 with ICP	37.924
MySQL 5.6 with ICP & MRR	7.816

## 17. MySQL是如何获取索引所在内存中的数据页的

哈希表，利用哈希算法。

哈希算法是一种常见算法，时间复杂度为 $O(1)$ ，且不只存在于索引中，每个数据库应用中都存在该数据库结构。

数据库中一般采用除法散列的方法，发生碰撞时采用链地址法。

在哈希函数的除法散列法中，通过取 $k$ 除以 $m$ 的余数，将关键字 $k$ 映射到 $m$ 个槽的某一个去，即哈希函数为：

$$h(k) = k \bmod m$$

InnoDB存储引擎使用**哈希算法**来对字典进行查找，其冲突机制采用**链表方式**，哈希函数采用**除法散列方式**。对于缓冲池页的哈希表来说，在缓冲池中的Page页都有一个chain指针，它指向相同哈希函数值的页。而对于除法散列， $m$ 的取值为略大于2倍的缓冲池页数量的质数。例如：当前参数innodb\_buffer\_pool\_size的大小为10M，则共有640个16KB的页。对于缓冲池页内存的哈希表来说，需要分配 $640 \times 2 = 1280$ 个槽，但是由于1280不是质数，需要取比1280略大的一个质数，应该是1399，所以在启动时会分配1399个槽的哈希表，用来哈希查询所在缓冲池中的页。

InnoDB存储引擎的表空间都有一个space\_id，用户所要查询的应该是某个表空间的某个连续16KB的页，即偏移量offset。InnoDB存储引擎将space\_id左移20位，然后加上这个space\_id和offset，即关键字 $K = \text{space\_id} \ll 20 + \text{space\_id} + \text{offset}$ ，然后通过除法散列到各个槽中去。

## 18. 什么是自适应哈希索引

自适应哈希索引采用之前讨论的哈希表的方式实现。不同的是，这仅是数据库自身创建并使用的，DBA本身并不能对其进行干预。自适应哈希索引经哈希函数映射到一个哈希表中，因此对于字典类型的查找非常快，如`SELECT * FROM TABLE WHERE index_col='xxx'`。但是对于范围查找就无能为力了。通过命令`SHOW ENGINE INNODB STATUS`可以看到当前自适应哈希索引的使用状况。

## 19. 什么是全文检索

例：SELECT \* FROM blog WHERE content like '%xxx%'

根据B+树索引的特性，上述SQL语句即便添加了B+树索引也是需要进行索引的扫描来得到结果。类似这样的需求在互联网应用中还有很多。例如，搜索引擎需要根据用户输入的关键字进行全文查找，电子商务网站需要根据用户的查询条件，在可能需要在商品的详细介绍中进行查找，这些都不是B+树索引所能很好地完成的工作。

**全文检索 (Full-Text Search)** 是将存储于数据库中的整本书或整篇文章中的任意内容信息查找出来的技术。它可以根据需要获得全文中有关章、节、段、句、词等信息，也可以进行各种统计和分析。

在之前的MySQL数据库中，InnoDB存储引擎并不支持全文检索技术。大多数的用户转向MyISAM存储引擎，这可能需要对表的拆分，并将需要进行全文检索的数据存储为MyISAM表。这样的确能够解决逻辑业务的需求，但是却丧失了InnoDB存储引擎的事务性，而这在生产环境应用中同样是非常关键的。

从InnoDB 1.2.x版本开始，InnoDB存储引擎开始支持全文检索，其支持MyISAM存储引擎的全部功能，并且还支持其他的一些特性。

InnoDB存储引擎从1.2.x版本开始支持全文检索的技术，其采用full inverted index的方式。在InnoDB存储引擎中，将(DocumentId, Position)视为一个“ilist”。因此在全文检索的表中，有两个列，一个是word字段，另一个是ilist字段，并且在word字段上有索引。



全文检索通常使用**倒排索引**（inverted index）来实现。倒排索引同B+树索引一样，也是一种索引结构。它在辅助表（auxiliary table）中存储了单词与单词自身在一个或多个文档中所在位置之间的映射。这通常**利用关联数组实现**，其拥有两种表现形式：

- ❑ **inverted file index**，其表现形式为{单词， 单词所在文档的ID}
- ❑ **full inverted index**，其表现形式为{单词， (单词所在文档的ID， 在具体文档中的位置)}

inverted file index 的关联数组

Number	Text	Documents	Number	Text	Documents
1	code	1, 4	8	old	3, 6
2	days	3, 6	9	pease	1, 2
3	hot	1, 4	10	porridge	1, 2
4	in	2, 5	11	pot	2, 5
5	it	4, 5	12	some	4, 5
6	like	4, 5	13	the	2, 5
7	nine	3, 6			

可以看到单词code存在于文档1和4中，单词days存在与文档3和6中。

full inverted index 的关联数组

Number	Text	Documents	Number	Text	Documents
1	code	(1:6), (4:8)	8	old	(3:3), (6:3)
2	days	(3:2), (6:2)	9	pease	(1:1,4), (2:1)
3	hot	(1:3), (4:4)	10	porridge	(1:2,5), (2:2)
4	in	(2:3), (5:4)	11	pot	(2:5), (5:6)
5	it	(4: 3,7), (5:3)	12	some	(4:1,5), (5:1)
6	like	(4:2,6), (5:2)	13	the	(2:4), (5:5)
7	nine	(3:1), (6:1)			

full inverted index还存储了单词所在的位置信息，如code这个单词出现在（1：6），即文档1的第6个单词为code。相比之下，full inverted index占用更多的空间，但是能更好地定位数据



## 20. 全文检索的语法有了解吗?

MySQL数据库支持全文检索 (Full-Text Search) 的查询, 其语法为:  
**MATCH(col1,col2,...)AGAINST(expr[search\_modifier])**

**search\_modifier:**

```
{  
  IN NATURAL LANGUAGE MODE  
  | IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION  
  | IN BOOLEAN MODE  
  | WITH QUERY EXPANSION  
}
```

MySQL数据库通过MATCH()...AGAINST()语法支持全文检索的查询, MATCH指定了需要被查询的列, AGAINST指定了使用何种方法进行查询。

### 1. **NATURAL LANGUAGE MODE**

全文检索通过MATCH函数进行查询, 默认采用Natural Language模式, 其表示查询带有指定word的文档

```
mysql > SELECT * FROM fts_a  
- > WHERE MATCH(body)  
- > AGAINST('Porridge' IN NATURAL LANGUAGE MODE);
```

## 2. BOOLEAN MODE

MySQL数据库允许使用IN BOOLEAN MODE修饰符来进行全文检索。当使用该修饰符时，查询字符串的前后字符会有特殊的含义，例如下面的语句要求查询有字符串Pease但没有hot的文档，其中+和-分别表示这个单词必须出现，或者一定不存在。

```
mysql > SELECT * FROM fts_a  
- > WHERE MATCH(body) AGAINST('+Pease-hot' IN BOOLEAN MODE) ;
```

## 3. WITH QUERY EXPANSION 或 IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION

MySQL数据库还支持全文检索的扩展查询。这种查询通常在查询的关键词太短，用户需要implied knowledge（隐含知识）时进行。例如，对于单词database的查询，用户可能希望查询的不仅仅是包含database的文档，可能还指那些包含MySQL、Oracle、DB2、RDBMS的单词。而这时可以使用Query Expansion模式来开启全文检索的implied knowledge。

```
mysql > SELECT * FROM articles  
- > WHERE MATCH(title,body)  
- > AGAINST('database' WITH QUERY EXPANSION);
```

## 21.INNODB 中锁是什么级别的，有几种

InnoDB存储引擎实现了如下**两种标准的行级锁**：

- ❑ **共享锁 (S Lock)**，允许事务读一行数据。
- ❑ **排他锁 (X Lock)**，允许事务删除或更新一行数据。

如果一个事务T1已经获得了行r的共享锁，那么另外的事务T2可以立即获得行r的共享锁，因为读取并没有改变行r的数据，称这种情况为锁兼容 (Lock Compatible)。但若有其他的事务T3想获得行r的排他锁，则其必须等待事务T1、T2释放行r上的共享锁——这种情况称为锁不兼容。

排他锁和共享锁的兼容性

	X	S
X	不兼容	不兼容
S	不兼容	兼容

## 22. INNODB除了s和x行级锁，还有其他锁类型吗

此外，InnoDB存储引擎支持多粒度（granular）锁定，这种锁定允许事务在行级上的锁和表级上的锁同时存在。为了支持在不同粒度上进行加锁操作，InnoDB存储引擎支持一种额外的锁方式，称之为意向锁（Intention Lock）。意向锁是将锁定的对象分为多个层次，意向锁意味着事务希望在更细粒度（fine granularity）上进行加锁

InnoDB存储引擎支持意向锁设计比较简练，其意向锁即为表级别的锁。设计目的主要是为了在一个事务中揭示下一行将被请求的锁类型。其支持两种意向锁：

- 1) 意向共享锁（IS Lock），事务想要获得一张表中某几行的共享锁
- 2) 意向排他锁（IX Lock），事务想要获得一张表中某几行的排他锁

由于InnoDB存储引擎支持的是行级别的锁，因此意向锁其实不会阻塞除全表扫以外的任何请求。

InnoDB 存储引擎中锁的兼容性

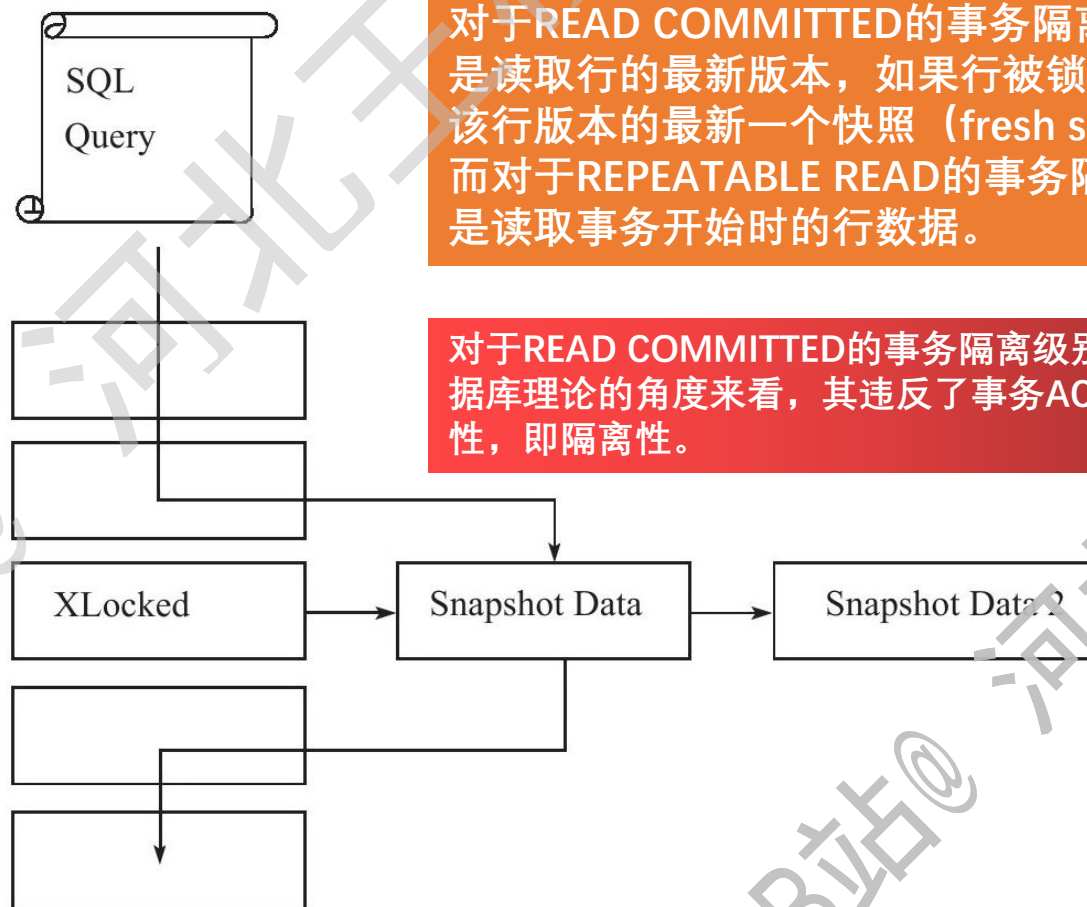
	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

## 23. 说说InnoDB是如何进行一致性非锁定读的（MVCC的原理）

**一致性的非锁定读**（consistent nonlocking read）是指InnoDB存储引擎通过**行多版本控制（multi versioning）**的方式来读取当前执行时间数据库中行的数据。如果读取的行正在执行DELETE或UPDATE操作，这时读取操作不会因此去等待行上锁的释放。相反地，InnoDB存储引擎会去**读取行的一个快照数据**。

之所以称其为非锁定读，因为**不需要等待访问的行上X锁的释放**。**快照数据**是指该行的**之前版本的数据**，该实现是通过**undo段来完成**。而**undo用来在事务中回滚数据**，因此快照数据本身是没有额外的开销。此外，**读取快照数据是不需要上锁的**，因为没有事务需要对历史的数据进行修改操作。

快照数据其实就是当前行数据之前的历史版本，每行记录**可能有多个版本**。如右图显示的，一个行记录可能有不止一个快照数据，一般称这种技术为行多版本技术。由此带来的并发控制，称之为**多版本并发控制（Multi Version Concurrency Control, MVCC）**。



对于READ COMMITTED的事务隔离级别，它总是读取行的最新版本，如果行被锁定了，则读取该行版本的最新一个快照（fresh snapshot）。而对于REPEATABLE READ的事务隔离级别，总是读取事务开始时的行数据。

对于READ COMMITTED的事务隔离级别而言，从数据库理论的角度来看，其违反了事务ACID中的I的特性，即隔离性。



## 24. InnoDB的主键自增长是如何保证的

自增长在数据库中是非常常见的一种属性，也是很多DBA或开发人员首选的主键方式。在InnoDB存储引擎的内存结构中，对每个含有自增长值的表都有一个**自增长计数器（auto-increment counter）**。当对含有自增长的计数器的表进行插入操作时，这个计数器会被初始化，执行如下的语句来得到计数器的值：

```
SELECT MAX(auto_inc_col)FROM t FOR UPDATE;
```

插入操作会依据这个自增长的计数器值加1赋予自增长列。这个实现方式称做**AUTO-INC Locking**。**这种锁其实是采用一种特殊的表锁机制，为了提高插入的性能，锁不是在一个事务完成后才释放，而是在完成对自增长值插入的SQL语句后立即释放。**

虽然**AUTO-INC Locking**从一定程度上提高了并发插入的效率，但还是**存在一些性能上的问题**。首先，对于有自增长值的列的并发插入性能较差，**事务必须等待前一个插入的完成**（虽然不用等待事务的完成）。其次，对于INSERT...SELECT的大数据量的插入会影响插入的性能，因为**另一个事务中的插入会被阻塞**。

**从MySQL 5.1.22版本开始**，InnoDB存储引擎中提供了一种轻量级互斥量的自增长实现机制，这种机制大大提高了自增长值插入的性能。并且从该版本开始，InnoDB存储引擎提供了一个参数**innodb\_autoinc\_lock\_mode**来控制自增长的模式，该参数的默认值为1。

## 参数 innodb\_autoinc\_lock\_mode 的说明

innodb_autoinc_lock_mode	说明
0	这是 MySQL 5.1.22 版本之前自增长的实现方式，即通过表锁的 <b>AUTO-INC Locking</b> 方式。因为有了新的自增长实现方式，0 这个选项不应该是新版用户的首选选项
1	<b>这是该参数的默认值。</b> 对于 <b>“simple inserts”</b> ，该值会用互斥量（mutex）去对内存中的计数器进行累加的操作。对于 <b>“bulk inserts”</b> ，还是使用传统表锁的 AUTO-INC Locking 方式。在这种配置下，如果不考虑回滚操作，对于自增值列的增长还是连续的。并且在这种方式下，statement-based 方式的 replication 还是能很好地工作。需要注意的是，如果已经使用 AUTO-INC Locking 方式去产生自增长的值，而这时需要再进行 <b>“simple inserts”</b> 的操作时，还是需要等待 AUTO-INC Locking 的释放
2	在这个模式下，对于所有 <b>“INSERT-like”</b> 自增长值的产生都是通过互斥量，而不是 AUTO-INC Locking 的方式。显然，这是性能最高的方式。然而，这会带来一定的问题。因为并发插入的存在，在每次插入时，自增长的值可能不是连续的。此外，最重要的是，基于 Statement-Base Replication 会出现问题。因此，使用这个模式，任何时候都应该使用 row-base replication。这样才能保证最大的并发性能及 replication 主从数据的一致

statement-based replication (SBR)，row-based replication (RBR). 主从复制的方式。后续会有专门的相关面试题。



## 插入类型

插入类型	说明
insert-like	insert-like 指 <u>所有的插入语句</u> ，如 INSERT、REPLACE、INSERT...SELECT，REPLACE...SELECT、LOAD DATA 等
simple inserts	simple inserts 指能在插入前就 <u>确定插入行数的语句</u> 。这些语句包括 INSERT、REPLACE 等。需要注意的是：simple inserts 不包含 INSERT ...ON DUPLICATE KEY UPDATE 这类 SQL 语句
bulk inserts	bulk inserts 指在插入前不能确定得到插入行数的语句，如 INSERT...SELECT，REPLACE...SELECT，LOAD DATA
mixed-mode inserts	mixed-mode inserts 指插入中有一部分的值是自增长的， <u>有一部分是确定的</u> 。如 INSERT INTO t1 (c1,c2) VALUES (1,'a'), (NULL,'b'), (5,'c'), (NULL,'d'); 也可以是指 INSERT ...ON DUPLICATE KEY UPDATE 这类 SQL 语句

## 25. InnoDB 行锁的 3 种算法

InnoDB存储引擎有3种行锁的算法，其分别是：

- ❑ **Record Lock**：单个行记录上的锁
- ❑ **Gap Lock**：间隙锁，锁定一个范围，但不包含记录本身
- ❑ **Next-Key Lock**：Gap Lock+Record Lock，锁定一个范围，并且锁定记录本身

Record Lock总是会去锁住索引记录，如果InnoDB存储引擎表在建立的时候没有设置任何一个索引，那么这时InnoDB存储引擎会使用隐式的主键来进行锁定。

Next-Key Lock是结合了Gap Lock和Record Lock的一种锁定算法，在Next-Key Lock算法下，InnoDB对于行的查询都是采用这种锁定算法。

在**InnoDB** 默认的事务隔离级别下，即REPEATABLE READ下，InnoDB存储引擎采用Next-Key Locking这种锁定算法。例如一个索引有10，11，13和20这四个值，那么该索引可能被Next-Key Locking的区间为：

- $(-\infty, 10]$
- $(10, 11]$
- $(11, 13]$
- $(13, 20]$
- $(20, +\infty)$

```
DROP TABLE IF EXISTS t;

CREATE TABLE t(a INT PRIMARY KEY);

INSERT INTO t SELECT 1;

INSERT INTO t SELECT 2;

INSERT INTO t SELECT 5;
```

当查询的索引含有**唯一属性**时，InnoDB存储引擎会对Next-Key Lock进行优化，将其降级为**Record Lock**，即仅锁住索引本身，而不是范围。

什么是唯一属性，其实就是我们所说的能够标识该行数据唯一的标识。  
unique 字段。比如：主键就是唯一的，不重复的。我们也可以自己设计多个字段组合不重复，唯一的。

表t共有1、2、5三个值。在上面的例子中，在会话A中首先对a=5进行X锁定。而由于a是主键且唯一，因此锁定的仅是5这个值，而不是(2, 5)这个范围，这样在会话B中插入值4而不会阻塞，可以立即插入并返回。即锁定由Next-Key Lock算法降级为了Record Lock，从而提高应用的并发性。

唯一索引的锁定示例

时 间	会话 A	会话 B
1	BEGIN;	
2	SELECT * FROM t WHERE a =5 FOR UPDATE;	
3		BEGIN;
4		INSERT INTO t SELECT 4;
5		COMMIT; # 成功，不需要等待
6	COMMIT	

```
CREATE TABLE z(a INT,b INT PRIMARY KEY(a),KEY(b));
```

```
INSERT INTO z SELECT 1,1;
```

```
INSERT INTO z SELECT 3,1;
```

```
INSERT INTO z SELECT 5,3;
```

```
INSERT INTO z SELECT 7,6;
```

```
INSERT INTO z SELECT 10,8;
```

表z的列b是辅助索引，若在会话A中执行下面的SQL语句：

```
SELECT*FROM z WHERE b=3 FOR UPDATE
```

很明显，这时SQL语句通过索引列b进行查询，该列不是唯一属性，因此其使用传统的Next-Key Locking技术加锁，并且由于有两个索引，其需要**分别进行锁定**。对于聚集索引（primary-key a），其仅对列a等于5的索引加上Record Lock。而对于辅助索引b，其加上的是Next-Key Lock，**锁定的范围是(1，3)**。特别需要注意的是，InnoDB存储引擎还会对辅助索引下一个键值加上gap lock，即还有一个辅助索引范围为**(3，6)**的锁。

**因此，若在新会话B中运行下面的SQL语句，都会被阻塞：**

```
SELECT*FROM z WHERE a=5 LOCK IN SHARE MODE;
```

```
INSERT INTO z SELECT 4,2;
```

```
INSERT INTO z SELECT 6,5;
```

第一个SQL语句不能执行，因为在会话A中执行的SQL语句已经对聚集索引中列a=5的值加上X锁，因此执行会被阻塞。第二个SQL语句，主键插入4，没有问题，但是插入的辅助索引值2在锁定的范围(1, 3)中，因此执行同样会被阻塞。第三个SQL语句，插入的主键6没有被锁定，5也不在范围(1, 3)之间。但插入的值5在另一个锁定的范围(3, 6)中，故同样需要等待。



## 26. InnoDB 行锁 next-key lock 有什么作用，避免了什么问题

在默认的事务隔离级别下，即REPEATABLE READ下，InnoDB存储引擎采用Next-Key Locking机制来避免Phantom Problem（幻读问题，也称不可重复读）。Phantom Problem是指在同一事务下，连续执行两次同样的SQL语句可能导致不同的结果。（在 READ COMMITTED 事务隔离级别下会出现）

```
CREATE TABLE z(a INT,b INT,PRIMARY KEY(a),KEY(b));
```

```
INSERT INTO z SELECT 1,1;
```

```
INSERT INTO z SELECT 3,1;
```

```
INSERT INTO z SELECT 5,3;
```

```
INSERT INTO z SELECT 7,6;
```

```
INSERT INTO z SELECT 10,8;
```

在同一事务中，若此时执行语句：**SELECT\*FROM z WHERE b=3 FOR UPDATE** 两次，中间间隔10秒时间执行。可以肯定的说，我们会得到第三行数据的结果，即（5,3）。此时我们知道，会有一个 Record Lock锁定主键 5，还会有一个 gap lock锁定（1,3）和（3,6）。

假设：我们分析下，若此时没有gap lock（1,3）和（3,6），如果只有Record Lock锁定主键 5 会不会造成幻读。

分析：我们在第一次 select 完成之后，第二次select 之前，插入一条数据: **INSERT INTO z SELECT 20,3;** 这条数据是可以插入成功的，因为我们只有一个record lock 锁定了 主键5，对于新插入的数据主键为 20，可以插入，且无重复。插入完成后，第二次 select 得到了两个值，（5,3）（20,3）。这就造成了同一事物中，第一次读取和第二次读取的结果不一样，出现幻读。如果有gap lock，插入就会被阻塞，不会出现幻读。

# 27. 什么是脏读，数据库如何避免的脏读

脏读指的就是在不同的事务下，当前事务可以读到另外事务未提交的数据，简单来说就是可以读到脏数据。

表 6-15 脏读的示例

Time	会话 A	会话 B
1	SET @@tx_isolation='read-ncommitted';	
2		SET @@tx_isolation='read-ncommitted';
3		BEGIN;
4		mysql> SELECT * FROM t\G; ***** 1. row *****  a: 1  1 row in set (0.00 sec)
5	INSERT INTO t SELECT 2;	
6		mysql> SELECT * FROM t\G; ***** 1. row *****  a: 1 ***** 2. row *****  a: 2  2 row in set (0.00 sec)

截图(Alt + A)

事务的隔离级别进行了更换，由默认的REPEATABLE READ换成了**READ UNCOMMITTED**。因此在会话A中，在事务并没有提交的前提下，会话B中的两次SELECT操作取得了不同的结果，并且2这条记录是在会话A中并未提交的数据，即产生了脏读，违反了事务的隔离性。

脏读现象在生产环境中并不常发生，从上面的例子中就可以发现，**脏读发生的条件是需要事务的隔离级别为READ UNCOMMITTED**，而目前绝大部分的数据库都至少设置成READ COMMITTED。



## 28. 什么是丢失更新，如何避免

丢失更新是另一个锁导致的问题，简单来说其就是一个事务的更新操作会被另一个事务的更新操作所覆盖，从而导致数据的不一致。例如：

- 1) 事务T1将行记录r更新为v1，但是事务T1并未提交。
- 2) 与此同时，事务T2将行记录r更新为v2，事务T2未提交。
- 3) 事务T1提交。
- 4) 事务T2提交。

**但是**，在当前数据库的任何隔离级别下，都不会导致数据库理论意义上的丢失更新问题。这是因为，即使是READ UNCOMMITTED的事务隔离级别，对于行的DML操作，需要对行或其他粗粒度级别的对象加锁。因此在上述步骤2) 中，事务T2并不能对行记录r进行更新操作，其会被阻塞，直到事务T1提交。

虽然数据库能阻止丢失更新问题的产生，但是在生产应用中还有另一个逻辑意义的丢失更新问题，而导致该问题的并不是因为数据库本身的问题。实际上，在所有多用户计算机系统环境下都有可能产生这个问题。简单地说来，出现下面的情况时，就会发生丢失更新：

- 1) 事务T1查询一行数据，放入本地内存，并显示给一个终端用户User1。
- 2) 事务T2也查询该行数据，并将取得的数据显示给终端用户User2。
- 3) User1修改这行记录，更新数据库并提交。
- 4) User2修改这行记录，更新数据库并提交。

显然，这个过程中用户User1的修改更新操作“丢失”了，而这可能会导致一个“恐怖”的结果。

要避免丢失更新发生，需要让事务在这种情况下的操作变成串行化，而不是并行的操作。即在上述四个步骤的1) 中，对用户读取的记录加上一个**排他X锁**。同样，在步骤2) 的操作过程中，用户同样也需要加一个排他X锁。通过这种方式，步骤2) 就必须等待步骤1) 和步骤3) 完成，最后完成步骤4)

## 29. 如何预防数据库死锁，生产环境如何避免死锁

死锁是指两个或两个以上的事务在执行过程中，因争夺锁资源而造成的一种互相等待的现象。若无外力作用，事务都将无法推进下去。解决死锁问题最简单的方式是不要有等待，将任何的等待都转化为回滚，并且事务重新开始。毫无疑问，这的确可以避免死锁问题的产生。然而在线上环境中，这可能导致并发性能的下降，甚至任何一个事务都不能进行。而这所带来的问题远比死锁问题更为严重，因为这很难被发现并且浪费资源。

解决死锁问题最简单的一种方法是超时，即当两个事务互相等待时，当一个等待时间超过设置的某一阈值时，其中一个事务进行回滚，另一个等待的事务就能继续进行。在InnoDB存储引擎中，参数innodb\_lock\_wait\_timeout用来设置超时的时间。

超时机制虽然简单，但是其仅通过超时后对事务进行回滚的方式来处理，或者说其是根据FIFO的顺序选择回滚对象。但若超时的事务所占权重比较大，如事务操作更新了很多行，占用了较多的undo log，这时采用FIFO的方式，就显得不合适了，因为回滚这个事务的时间相对另一个事务所占用的时间可能会很多。

因此，除了超时机制，当前数据库还都普遍采用**wait-for graph（等待图）**的方式来进行死锁检测。较之超时的解决方案，这是一种更为主动的死锁检测方式。InnoDB存储引擎也采用的这种方式。wait-for graph要求数据库保存以下两种信息：

- 锁的信息链表
- 事务等待链表

Transaction  
Wait Lists

t1
t2
t3
t4

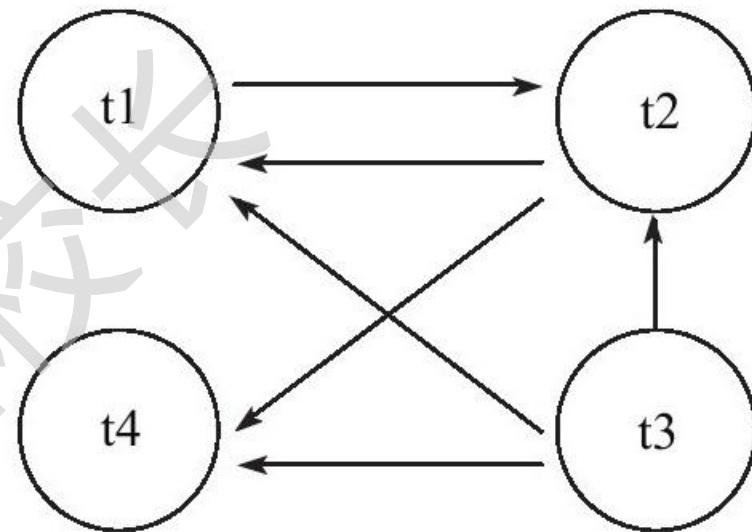
Lock Lists

row1

t2:x
t1:s

row2

t1:s
t4:s
t2:x
t3:x



事务T1指向T2边的定义为：

□ 事务T1等待事务T2所占用的资源

□ 事务T1最终等待T2所占用的资源，也就是事务之间在等待相同的资源，而事务T1发生在事务T2的后面

通过等待图可以发现存在回路（t1，t2），因此存在死锁。通过上述的介绍，可以发现wait-for graph是一种较为主动的死锁检测机制，在每个事务请求锁并发生等待时都会判断是否存在回路，若存在则有死锁，通常来说InnoDB存储引擎选择回滚undo量最小的事务。

## 30. 数据库靠什么保证事务的持久性的。详细说说

事务隔离性由之前讲述的锁来实现。**redo log称为重做日志，用来保证事务的持久性。**redo通常是物理日志，记录的是页的物理修改操作。

重做日志用来实现事务的持久性，即事务ACID中的D。其由两部分组成：

- 一是内存中的重做日志缓冲（redo logbuffer），其是易失的；
- 二是重做日志文件（redo log file），其是持久的。

InnoDB是事务的存储引擎，其通过Force Log at Commit机制实现事务的持久性，即当事务提交（COMMIT）时，必须先将该事务的所有日志写入到重做日志文件进行持久化，待事务的COMMIT操作完成才算完成。为了确保每次日志都写入重做日志文件，在每次将重做日志缓冲写入重做日志文件后，InnoDB存储引擎都需要调用一次fsync操作。由于重做日志文件打开并没有使用O\_DIRECT选项，因此重做日志缓冲先写入文件系统缓存。为了确保重做日志写入磁盘，必须进行一次fsync操作。由于fsync的效率取决于磁盘的性能，因此磁盘的性能决定了事务提交的性能，也就是数据库的性能。





## 31.重做日志刷新到磁盘的策略有几种，有什么优劣

通过参数`innodb_flush_log_at_trx_commit`用来控制重做日志刷新到磁盘的策略。

- 该参数的默认值为1，表示事务提交时必须调用一次`fsync`操作。还可以设置该参数的值为0和2。
- 0表示事务提交时不进行写入重做日志操作，这个操作仅在master thread中完成，而在master thread中每1秒会进行一次重做日志文件的`fsync`操作。
- 2表示事务提交时将重做日志写入重做日志文件，但仅写入文件系统的缓存中，不进行`fsync`操作。在这个设置下，当MySQL数据库发生宕机而操作系统不发生宕机时，并不会导致事务的丢失。而当操作系统宕机时，重启数据库后会丢失未从文件系统缓存刷新到重做日志文件那部分事务。

举例：逐条插入50万条数据。

`innodb_flush_log_at_trx_commit = 1`时：用时 2 分 13 秒。50 万次写入重做日志；`fsync` 操作 50 万次。

`innodb_flush_log_at_trx_commit = 0`时：用时 23 秒。约23次写如重做日志；`fsync` 操作约 23 次。

`innodb_flush_log_at_trx_commit = 2`时：用时 35 秒。50万次写入重做日志（仅缓存）；`fsync` 操作 0 次。

虽然用户可以通过设置参数`innodb_flush_log_at_trx_commit`为0或2来提高事务提交的性能，但是需要牢记的是，这种设置方法丧失了事务的ACID特性。而针对上述存储过程，为了提高事务的提交性能，应该在将50万行记录插入表后进行一次的COMMIT操作，而不是在每插入一条记录后进行一次COMMIT操作。这样做的好处是还可以使事务方法在回滚时回滚到事务最开始的确定状态。

正确方法：`innodb_flush_log_at_trx_commit = 1`，将50万条数据在一个事务或者多个事务中分派提交，减少`fsync`次数

## 32. redo log 和 bin log有什么区别？ bin log做什么用的

在MySQL数据库中还有一种二进制日志（binlog），其用来进行POINT-IN-TIME（PIT）的恢复及主从复制（Replication）环境的建立。从表面上看其和重做日志非常相似，都是记录了对于数据库操作的日志。然而，从本质上来看，两者有着非常大的不同。

首先，**重做日志是在InnoDB存储引擎层产生**，而**二进制日志是在MySQL数据库的上层产生的**，并且二进制日志不仅仅针对于InnoDB存储引擎，MySQL数据库中的任何存储引擎对于数据库的更改都会产生二进制日志。

其次，两种日志记录的内容形式不同。MySQL数据库上层的二进制日志 **bin log**是一种逻辑日志，其记录的是对应的SQL语句。而InnoDB存储引擎层面的**重做日志是物理格式日志**，其记录的是对于每个页的修改。



### 33. 什么是undo log，有什么用？

重做日志记录了事务的行为，可以很好地通过其对页进行“重做”操作。但是事务有时还需要进行回滚操作，这时就需要undo。因此在对数据库进行修改时，InnoDB存储引擎不但会产生redo，还会产生一定量的undo。这样如果用户执行的事务或语句由于某种原因失败了，又或者用户用一条ROLLBACK语句请求回滚，就可以利用这些undo信息将数据回滚到修改之前的样子。

redo存放在重做日志文件中，与redo不同，undo存放在数据库内部的一个特殊段（segment）中，这个段称为undo段（undo segment）。undo段位于共享表空间内。可以通过py\_innodb\_page\_info.py工具来查看当前共享空间中undo的数量。

除了回滚操作，undo的另一个作用是MVCC，即在InnoDB存储引擎中MVCC的实现是通过undo来完成。当用户读取一行记录时，若该记录已经被其他事务占用，当前事务可以通过undo读取之前的行版本信息，以此实现非锁定读取。最后也是最为重要的一点是，undo log会产生redo log，也就是undo log的产生会伴随着redo log的产生，这是因为undo log也需要持久性的保护。

## 34. purge 操作有什么作用

delete和update操作可能并不直接删除原有的数据。例如，

```
DELETE FROM t WHERE a=1;
```

表t上列a有聚集索引，列b上有辅助索引。对于上述的delete操作，仅是将主键列等于1的记录delete flag设置为1，记录并没有被删除，即记录还是存在于B+树中。其次，对辅助索引上a等于1，b等于1的记录同样没有做任何处理。而真正删除这行记录的操作其实被“延时”了，最终在purge操作中完成。

purge用于最终完成delete和update操作。这样设计是因为InnoDB存储引擎支持MVCC，所以记录不能在事务提交时立即进行处理。这时其他事物可能正在引用这行，故InnoDB存储引擎需要保存记录之前的版本。而是否可以删除该条记录通过purge来进行判断。若该行记录已不被任何其他事务引用，那么就可以进行真正的delete操作。可见，purge操作是清理之前的delete和update操作，将上述操作“最终”完成。而实际执行的操作作为delete操作，清理之前行记录的版本。

## 35. group commit有什么好处，使用时需要注意什么？

若事务为非只读事务，则每次事务提交时需要进行一次fsync操作，以此保证重做日志都已经写入磁盘。当数据库发生宕机时，可以通过重做日志进行恢复。虽然固态硬盘的出现提高了磁盘的性能，然而**磁盘的fsync性能是有限的**。为了提高磁盘fsync的效率，当前数据库都提供了group commit的功能，即一次fsync可以刷新确保多个事务日志被写入文件。对于InnoDB存储引擎来说，事务提交时会进行两个阶段的操作：

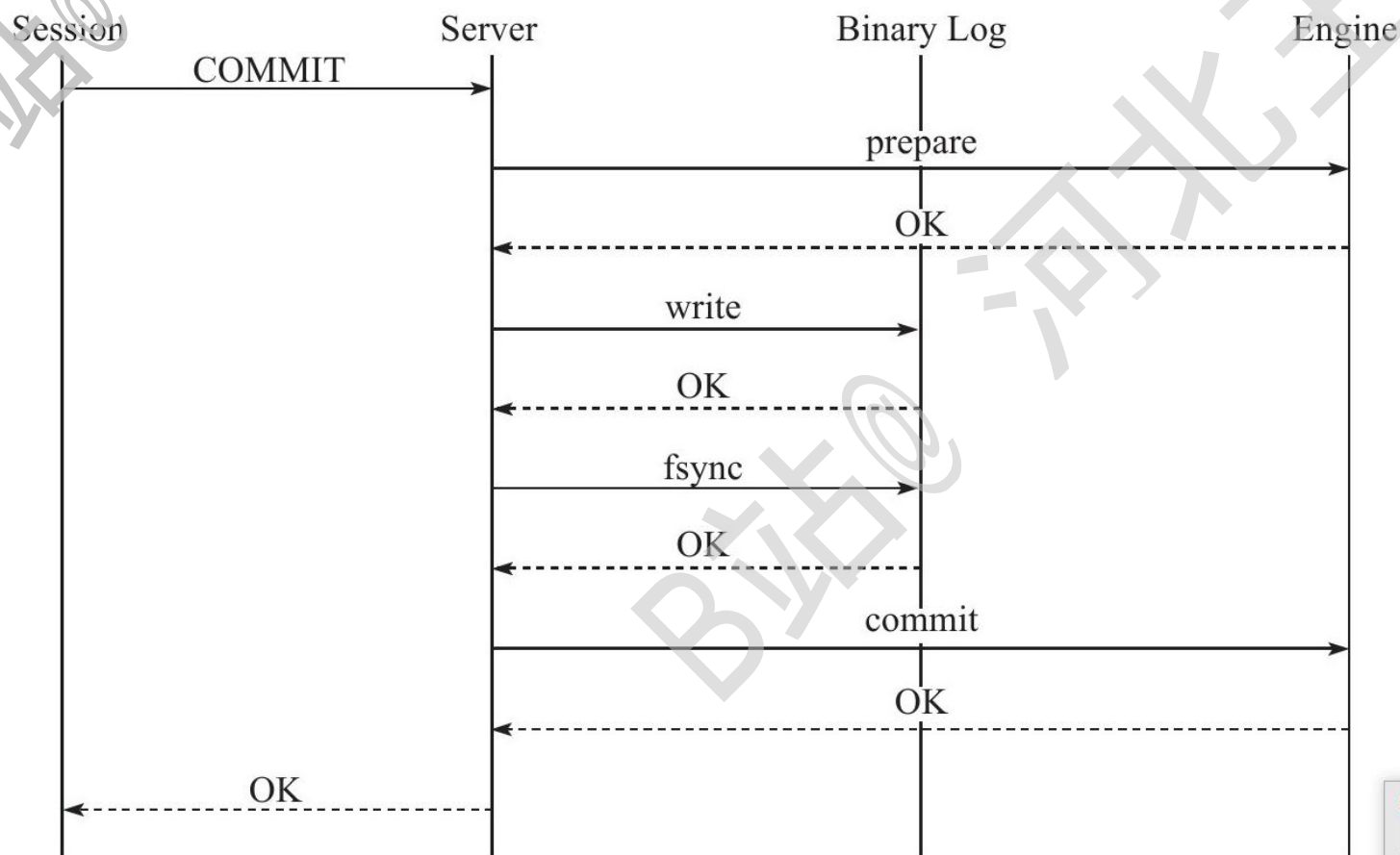
- 1) 修改内存中事务对应的信息，并且将日志写入重做日志缓冲。
- 2) 调用fsync将确保日志都从重做日志缓冲写入磁盘。

步骤2) 相对步骤1) 是一个较慢的过程，这是因为存储引擎需要与磁盘打交道。但当有事务进行这个过程时，其他事务可以进行步骤1) 的操作，正在提交的事物完成提交操作后，再次进行步骤2) 时，可以将多个事务的重做日志通过一次fsync刷新到磁盘，这样就大大地减少了磁盘的压力，从而提高了数据库的整体性能。对于写入或更新较为频繁的操作，group commit的效果尤为明显。

然而在InnoDB1.2版本之前，在开启二进制日志后，InnoDB存储引擎的group commit功能会失效，从而导致性能的下降。并且在线环境多使用replication环境，因此二进制日志的选项基本都为开启状态，因此这个问题尤为显著。导致这个问题的原因是在开启二进制日志后，为了保证存储引擎层中的事务和二进制日志的一致性，二者之间使用了两阶段事务，其步骤如下：

- 1) 当事务提交时InnoDB存储引擎进行prepare操作。
- 2) MySQL数据库上层写入二进制日志。
- 3) InnoDB存储引擎层将日志写入重做日志文件。
  - a) 修改内存中事务对应的信息，并且将日志写入重做日志缓冲。
  - b) 调用fsync将确保日志都从重做日志缓冲写入磁盘。

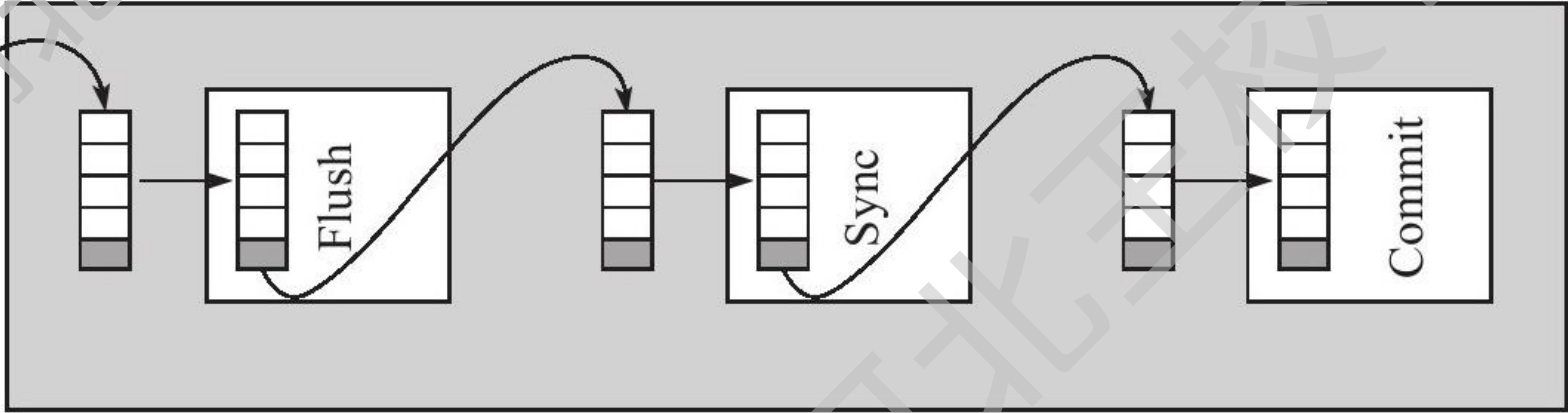
一旦步骤2) 中的操作完成，就确保了事务的提交，即使在执行步骤3) 时数据库发生了宕机。此外需要注意的是，每个步骤都需要进行一次fsync操作才能保证上下两层数据的一致性。步骤2) 的fsync由参数sync\_binlog控制，步骤3) 的fsync由参数innodb\_flush\_log\_at\_trx\_commit控制。因此上述整个过程如下图所示。



为了保证MySQL数据库上层二进制日志的写入顺序和InnoDB层的事务提交顺序一致，MySQL数据库内部使用了prepare\_commit\_mutex这个锁。但是在启用这个锁之后，步骤3) 中的步骤a) 步不可以在其他事务执行步骤b) 时进行，从而导致了group commit失效。

这个问题最早在2010年的MySQL数据库大会中提出，Facebook MySQL技术组，Percona公司都提出过解决方案。最后由MariaDB数据库的开发人员Kristian Nielsen完成了最终的“完美”解决方案。在这种情况下，不但MySQL数据库上层的二进制日志写入是group commit的，InnoDB存储引擎层也是group commit的。此外还移除了原先的锁prepare\_commit\_mutex，从而大大提高了数据库的整体性。MySQL 5.6采用了类似的实现方式，并将其称为Binary Log Group Commit (BLGC)。

MySQL 5.6 BLGC的实现方式是将事务提交的过程分为几个步骤来完成，如下图所示。



在MySQL数据库上层进行提交时首先按顺序将其放入一个队列中，队列中的第一个事务称为leader，其他事务称为follower，leader控制着follower的行为。**BLGC的步骤分为以下三个阶段：**

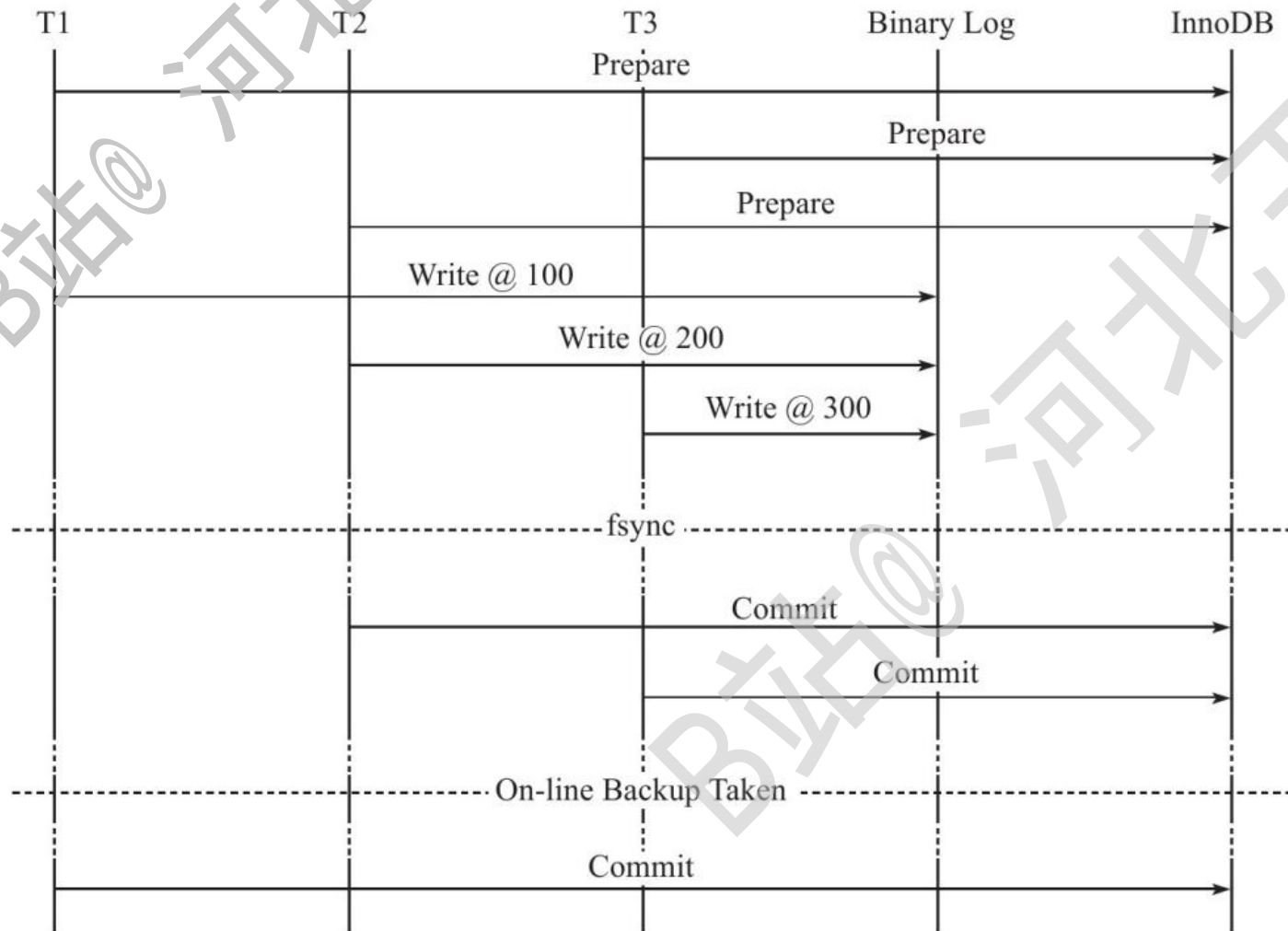
- ❑ Flush阶段，将每个事务的二进制日志写入内存中。
- ❑ Sync阶段，将内存中的二进制日志刷新到磁盘，若队列中有多个事务，那么仅一次fsync操作就完成了二进制日志的写入，这就是BLGC。
- ❑ Commit阶段，leader根据顺序调用存储引擎层事务的提交，InnoDB存储引擎本就支持group commit，因此修复了原先由于锁prepare\_commit\_mutex导致group commit失效的问题。

参数binlog\_max\_flush\_queue\_time用来控制Flush阶段中等待的时间，即使之前的一组事务完成提交，当前一组的事务也不马上进入Sync阶段，而是至少需要等待一段时间。这样做的好处是group commit的事务数量更多，然而这也可能会导致事务的响应时间变慢。该参数的默认值为0，且推荐设置依然为0。除非用户的MySQL数据库系统中有着大量的连接（如100个连接），并且不断地在进行事务的写入或更新操作。



## 36. 为什么需要保证MySQL数据库上层二进制日志的写入顺序和InnoDB层的事务提交顺序一致呢？

因为备份及恢复的需要，例如通过工具xtrabackup或者ibbackup进行备份，并用来建立replication，如下图所示。



可以看到若通过在线备份进行数据库恢复来重新建立replication，事务T1的数据会产生丢失。因为在InnoDB存储引擎层会检测最后一次的事务T3在上下两层都完成了提交，不需要再进行恢复，故认为之前的 T1 ,T2 也都完成了提交。

因此通过锁 `prepare_commit_mutex` 以串行的方式来保证顺序性，然而这会使group commit无法生效



## 37. Mysql 事务的隔离级别

SQL标准定义四个隔离级别为：

- ❑ READ UNCOMMITTED （导致脏读）
- ❑ READ COMMITTED （导致幻读）
- ❑ REPEATABLE READ （默认使用，避免幻读，也能避免脏读）
- ❑ SERIALIZABLE （更高级别隔离，避免幻读，避免脏读）

InnoDB存储引擎默认支持的隔离级别是REPEATABLE READ，但是与标准SQL不同的是，InnoDB存储引擎在REPEATABLE READ事务隔离级别下，使用Next-Key Lock锁的算法，因此避免幻读的产生。这与其他数据库系统（如Microsoft SQL Server数据库）是不同的。所以说，InnoDB存储引擎在默认的REPEATABLE READ的事务隔离级别下已经能完全保证事务的隔离性要求，即达到SQL标准的SERIALIZABLE隔离级别。

隔离级别越低，事务请求的锁越少或保持锁的时间就越短。这也是为什么大多数数据库系统默认的事务隔离级别是READ COMMITTED。

在InnoDB存储引擎中，可以使用以下命令来设置当前会话或全局的事务隔离级别：

```
SET[GLOBAL|SESSION]TRANSACTION ISOLATION LEVEL
{
    READ UNCOMMITTED
    |READ COMMITTED
    |REPEATABLE READ
    |SERIALIZABLE
}
```

在SERIALIZABLE的事务隔离级别，InnoDB存储引擎会对每个SELECT语句后自动加上LOCK IN SHARE MODE，即为每个读取操作加一个共享锁。因此在这个事务隔离级别下，读占用了锁，对一致性的非锁定读不再予以支持。

## 38. MySQL的分布式事务有了解吗

InnoDB存储引擎提供了对**XA事务**的支持，并通过**XA事务来支持分布式事务的实现**。分布式事务指的是允许多个独立的事务资源（transactional resources）参与到一个全局的事务中。事务资源通常是关系型数据库系统，但也可以是其他类型的资源。全局事务要求在其中的所有参与的事务要么都提交，要么都回滚，这对于事务原有的ACID要求又有了提高。另外，在**使用分布式事务时，InnoDB存储引擎的事务隔离级别必须设置为SERIALIZABLE**。

**XA事务允许不同数据库之间的分布式事务**，如一台服务器是MySQL数据库的，另一台是Oracle数据库的，又可能还有一台服务器是SQL Server数据库的，只要参与在全局事务中的每个节点都支持XA事务。分布式事务可能在银行系统的转账中比较常见，如用户David需要从上海转10 000元到北京的用户Mariah的银行卡中：

```
#Bank@Shanghai:
```

```
UPDATE account SET money=money-10000 WHERE user='David';
```

```
#Bank@Beijing
```

```
UPDATE account SET money=money+10000 WHERE user='Mariah';
```

在这种情况下，一定需要使用分布式事务来保证数据的安全。如果发生的操作不能全部提交或回滚，那么任何一个结点出现问题都会导致严重的结果。要么是David的账户被扣款，但是Mariah没收到，又或者是David的账户没有扣款，Mariah却收到钱了。

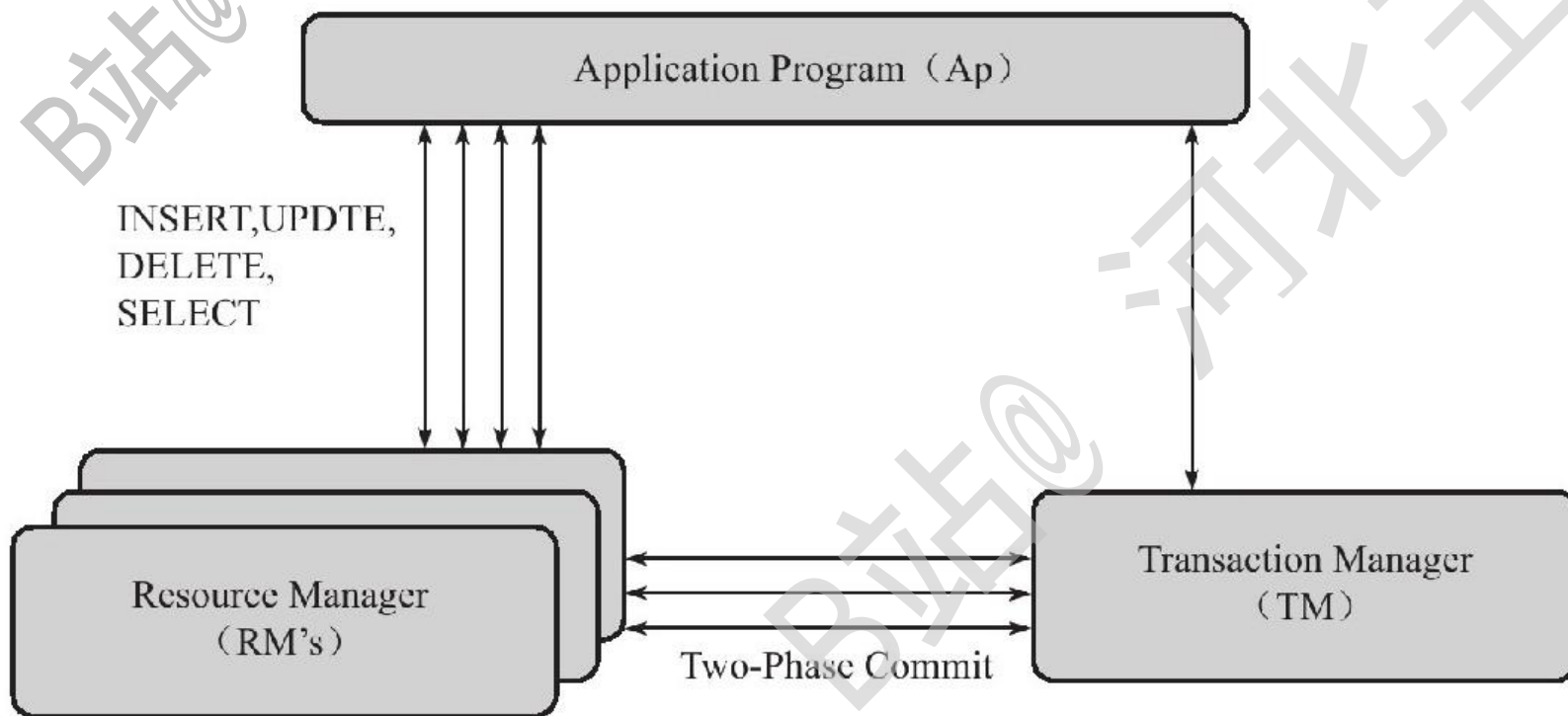
XA事务由一个或多个资源管理器（Resource Managers）、一个事务管理器（Transaction Manager）以及一个应用程序（Application Program）组成。

❑ **资源管理器**：提供访问事务资源的方法。通常一个数据库就是一个资源管理器。

❑ **事务管理器**：协调参与全局事务中的各个事务。需要和参与全局事务的所有资源管理器进行通信。

❑ **应用程序**：定义事务的边界，指定全局事务中的操作。

在MySQL数据库的分布式事务中，**资源管理器就是MySQL数据库，事务管理器为连接MySQL服务器的客户端**。下图显示了一个分布式事务的模型。



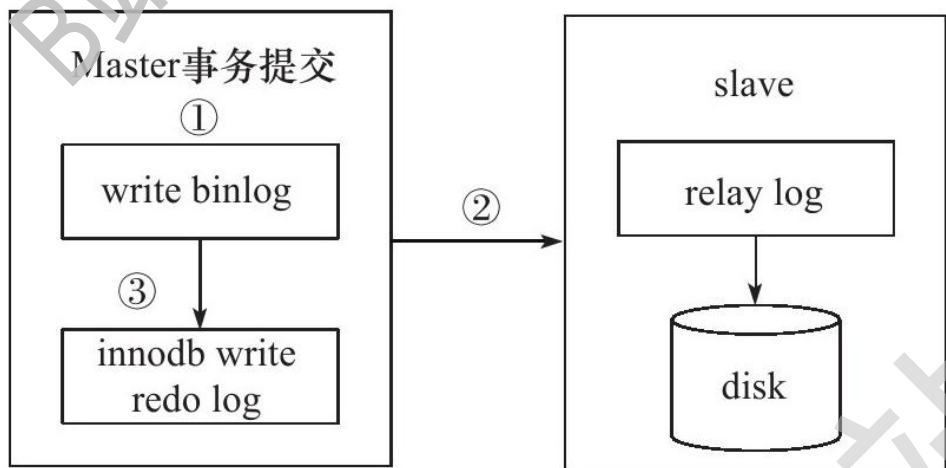
分布式事务使用**两段式提交（two-phase commit）**的方式。在第一阶段，所有参与全局事务的节点都开始准备（PREPARE），告诉事务管理器它们准备好提交了。在第二阶段，事务管理器告诉资源管理器执行ROLLBACK还是COMMIT。如果任何一个节点显示不能提交，则所有的节点都被告知需要回滚。可见与本地事务不同的是，分布式事务需要多一次的PREPARE操作，待收到所有节点的同意信息后，再进行COMMIT或是ROLLBACK操作。

MySQL数据库XA事务的SQL语法如下：

```
XA{START|BEGIN}xid[JOIN|RESUME]
XA END xid[SUSPEND[FOR MIGRATE]]
XA PREPARE xid
XA COMMIT xid[ONE PHASE]
XA ROLLBACK xid
XA RECOVER
```

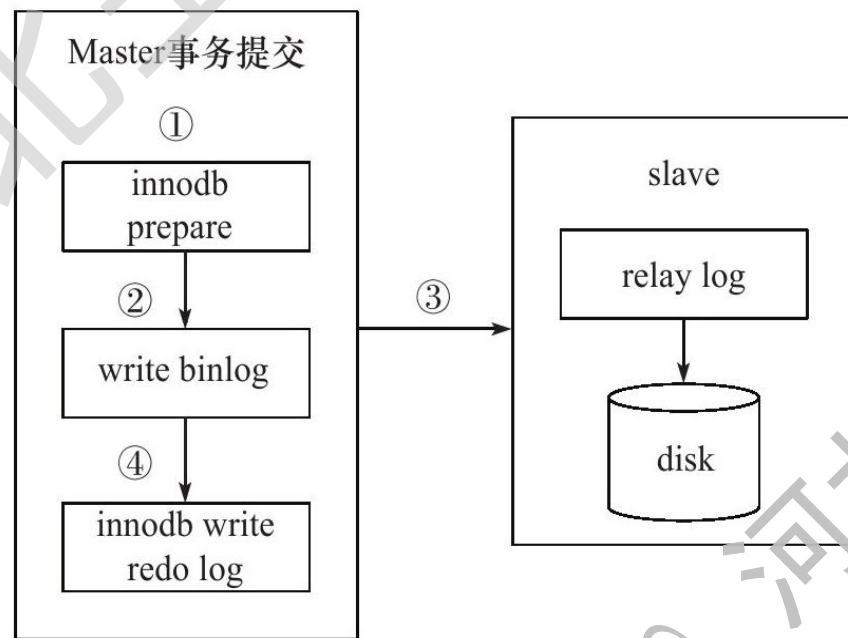
## 39. MySQL内部自身有没有需要考虑分布式事务的?

最为常见的内部XA事务存在于binlog与InnoDB存储引擎之间。由于复制的需要，因此目前绝大多数数据库都开启了binlog功能。在事务提交时，先写二进制日志，再写InnoDB存储引擎的重做日志。对上述两个操作的要求也是原子的，即二进制日志和重做日志必须同时写入。若二进制日志先写了，而在写入InnoDB存储引擎时发生了宕机，那么slave可能会接收到master传过去的二进制日志并执行，最终导致了主从不一致的情况。



宕机导致replication主从不一致的情况

如果执行完①、②后在步骤③之前MySQL数据库发生了宕机，则会发生主从不一致的情况。为了解决这个问题，MySQL数据库在binlog与InnoDB存储引擎之间采用XA事务。当事务提交时，InnoDB存储引擎会先做一个PREPARE操作，将事务的xid写入，接着进行二进制日志的写入。



MySQL数据库通过内部XA事务保证主从数据一致

如果在InnoDB存储引擎提交前，MySQL数据库宕机了，那么MySQL数据库在重启后会先检查准备的XID事务是否已经提交，若没有，则在存储引擎层再进行一次提交操作。

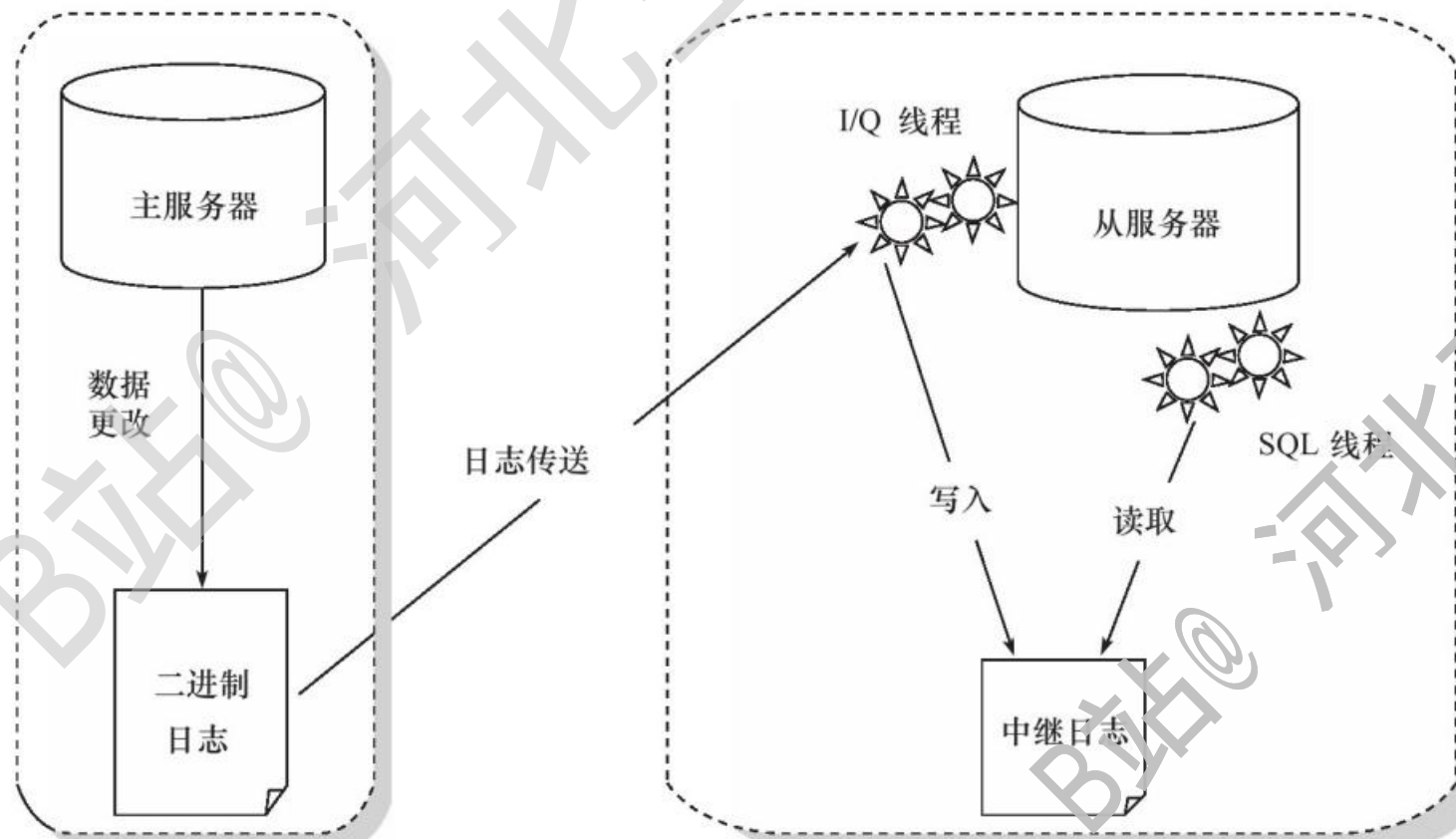


## 40. 说说主从复制 (replication) 的工作原理

复制 (replication) 是MySQL数据库提供了一种高可用高性能的解决方案，一般用来建立大型的应用。总体来说，replication的工作原理分为以下3个步骤：

- 1) 主服务器 (master) 把数据更改记录到二进制日志 (binlog) 中。
- 2) 从服务器 (slave) 把主服务器的二进制日志复制到自己的中继日志 (relay log) 中。
- 3) 从服务器重做中继日志中的日志，把更改应用到自己的数据库上，以达到数据的最终一致性。

复制的工作原理并不复杂，其实就是一个完全备份加上二进制日志备份的还原。不同的是这个二进制日志的还原操作基本上实时在进行中。这里特别需要注意的是，复制不是完全实时地进行同步，而是异步实时。这中间存在主从服务器之间的执行延时，如果主服务器的压力很大，则可能导致主从服务器延时较大。复制的工作原理如右图：



从服务器有2个线程，一个是IO线程，负责读取主服务器的二进制日志，并将其保存为中继日志；另一个是SQL线程，复制执行中继日志。MySQL4.0版本之前，从服务器只有1个线程，既负责读取二进制日志，又负责执行二进制日志中的SQL语句。这种方式不符合高性能的要求，目前已淘汰。



## 41. 主从复制bin log 日志有几种记录方式，说说各自的优缺点

Replication之所以能够工作，主要还是归结于binlog（binary log），所以在 Replication 模式下必须开启 binlog 功能；slave 从 masters 上增量获取 binlog 信息，并在本地应用日志中的变更操作（即“重放”）。变更操作将根据选定的格式类型写入 binlog 文件，目前支持三种 format：

**statement-based Replication（SBR）**：master将SQL statements语句写入binlog，slave 也将 statements 复制到本地执行；简单而言，就是在 master 上执行的 SQL 变更语句，也同样在 slaves 上执行。SBR 模式是 MySQL 最早支持的类型，也是 Replication 默认类型。

**row-based Replication（RBR）**：master将每行数据的变更信息写入binlog，每条 binlog 信息表示一行（row）数据的变更内容，对于 slaves 而言将会复制 binlog 信息，然后单条或者批量执行变更操作；

**mix-format Replication**：混合模式，在这种模式下，master将根据根据存储引擎、变更操作类型等，从SBR、RBR中来选择更合适的日志格式，默认为 SBR；具体选择那种格式，这取决于变更操作发生的存储引擎、statement 的类型以及特征，优先选择“数据一致性”最好的方式（RBR），然后才兼顾性能，比如 statement 中含有“不确定性”方法或者批量变更，那么将选择 RBR 方式，其他的将选择 SBR 以减少 binlog 的大小。我们建议使用 mix 方式。

SBR 和 RBR 都有各自的优缺点，对于大部分用而言，mix 方式在兼顾数据完整性和性能方面是最佳的选择。

## SBR的优点：

- ❑ 因为 binlog 中只写入了变更操作的 statements，所以日志量将会很小；
- ❑ 当使用 SQL 语句批量更新、删除数据时，只需要在 binlog 中记录 statement 即可，可以大大减少 log 文件对磁盘的使用
- ❑ 当然这也意味着 slave 复制信息量也更少，以及通过 binlog 恢复数据更加快速；

**SBR的缺点：**有些变更操作使用 SBR 方式会带来数据不一致的问题，一些结果具有不确定性的操作使用 **SBR 将会引入数据不一致的问题。**

- ❑ statement 中如果使用了 UDF(User Definition Fuction)，UDF 的计算结果可能依赖于 SQL 执行的时机和系统变量，这可能在 slave 上执行的结果与 master 不同，此外如果使用了 trigger，也会带来同样的问题；
- ❑ statement 中如果使用了如下函数的（举例）：UUID()，SYSDATE()，RAND() 等，不过 NOW () 函数可以正确的被 Replication（但在 UDF 或者触发器中则不行）；这些函数的特点就是它们的值依赖于本地系统，RAND () 本身就是随机所以值是不确定的。如果 statement 中使用了上述函数，那么将会在日志中输出 warning 信息；
- ❑ 对于“INSERT ... SELECT”语句，SBR 将比 RBR 需要更多的行锁。（主要是为了保障数据一致性，需要同时锁定受影响的所有行，而 RBR 则不必要）；
- ❑ 对于 InnoDB，使用“AUTO\_INCREMENT”的 insert 语句，将会阻塞其他“非冲突”的 INSERT。（因为 AUTO\_INCREMENT，为了避免并发导致的数据一致性问题，只能串行，但 RBR 则不需要）；（参考面试题274，前后呼应）
- ❑ 对于复杂的SQL语句，在 slaves 上仍然需要评估（解析）然后才能执行，而对于 RBR，SQL 语句只需要直接更新相应的行数据即可；在 slave 上评估、执行 SQL 时可能会发生错误，这种错误会随着时间的推移而不断累加，数据一致性的问题或许会不断增加。

## RBR的优点：

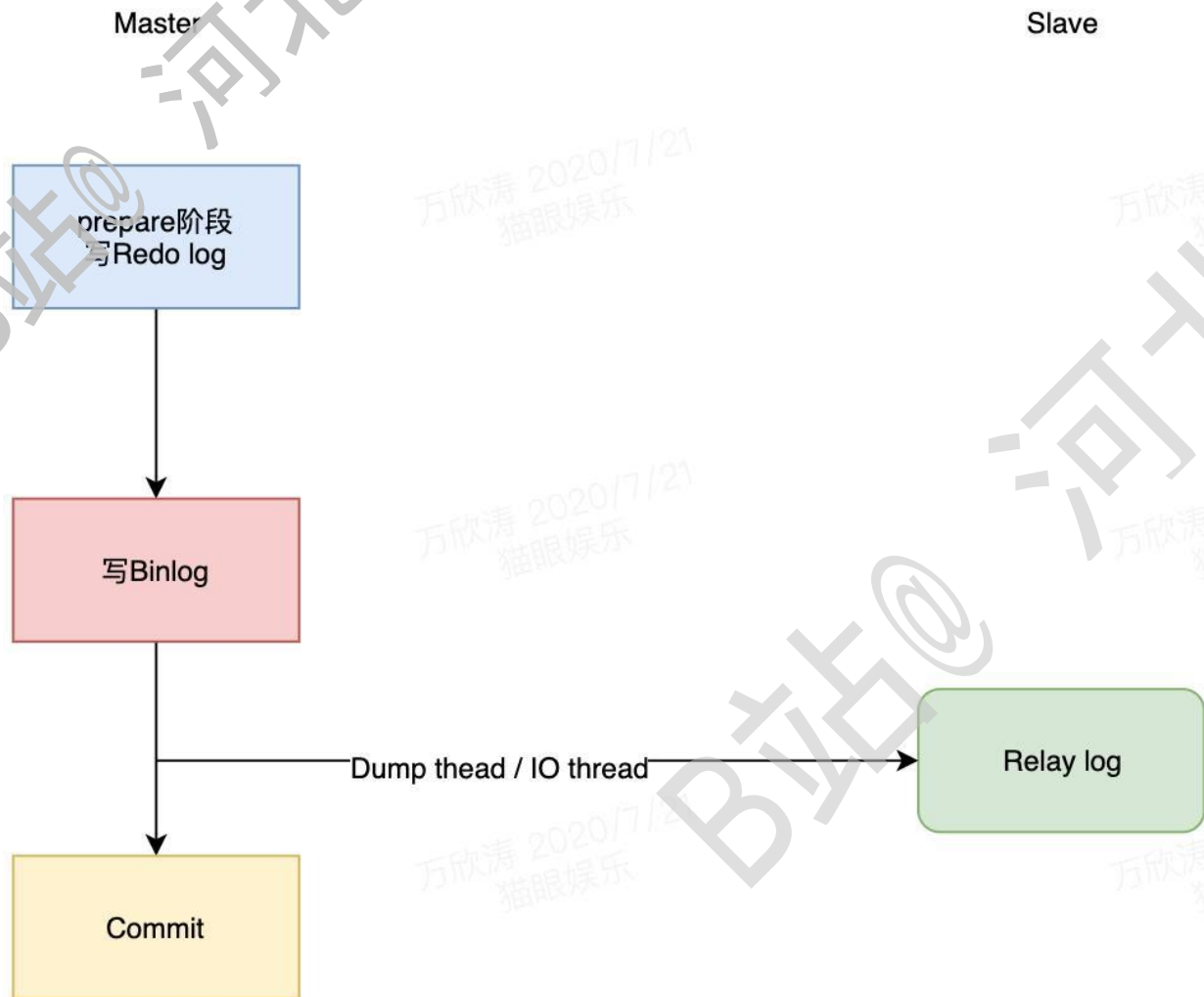
- ❑ 所有的变更操作，都可以被正确的 Replication，这事最安全的方式；
- ❑ 对于“INSERT... SELECT”、包含“AUTO\_INCREMENT”的 inserts、没有使用索引的 UPDATE/DELETE，相对于 SBR 将需要更少的行锁。（意味着并发能力更强）；

## RBR的缺点：

- ❑ **最大的缺点：**就是 RBR 需要更多的日志量。任何数据变更操作都将被写入 log，受影响的每行都要写入日志，日志包含此行所有列的值（即使没有值变更的列）；因此 RBR 的日志条数和尺寸都将会远大于 SBR，特别是在批量的 UPDATE/DELETE 时，可能会产生巨大的 log 量，反而对性能带来影响，尽管这确实保障了数据一致性，确导致 Replication 的效率较低；

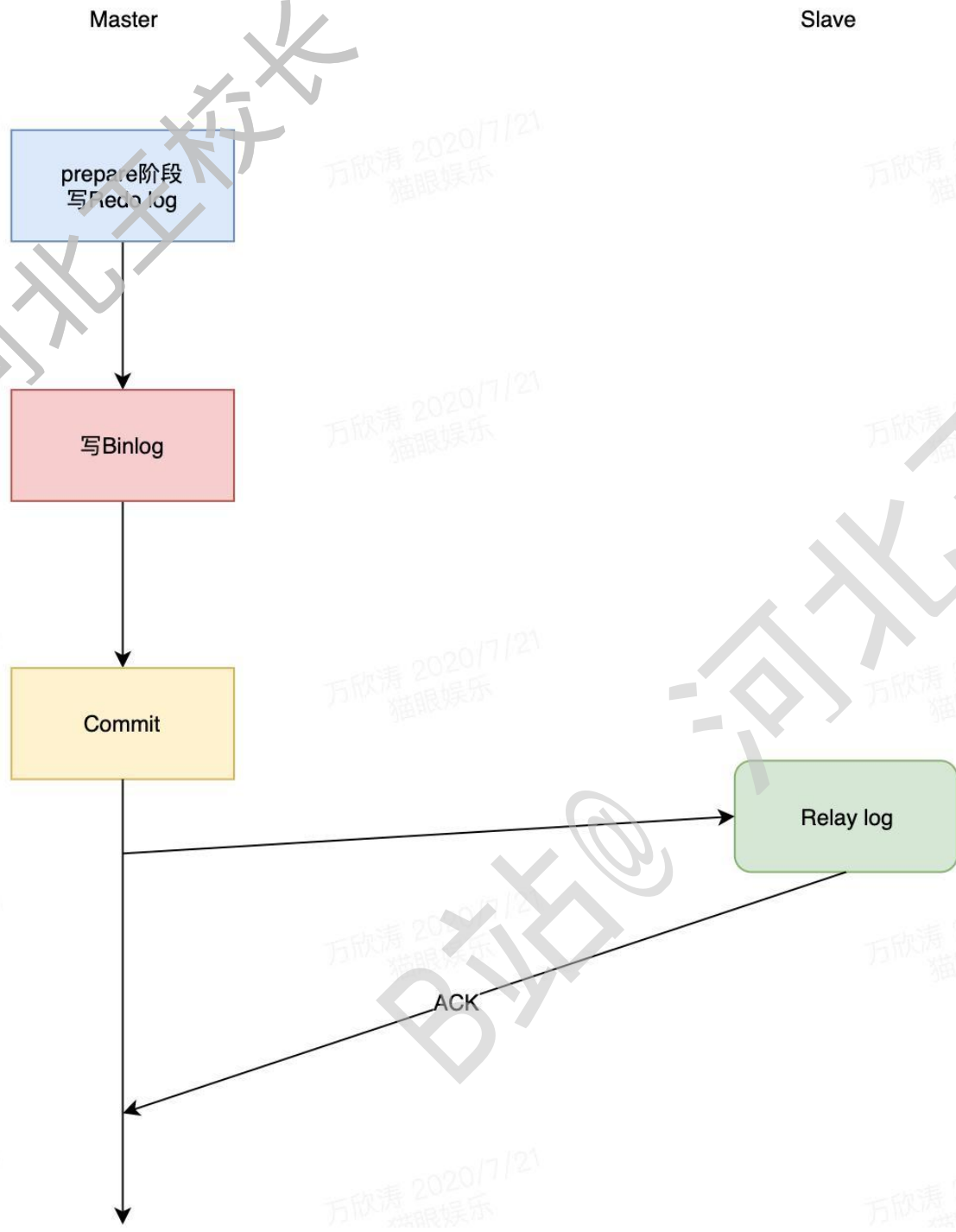
## 42. 主从复制有几种方式？分别说说。

### 异步复制



MySQL 默认的复制策略，Master 处理事务过程中，将其写入 Binlog 就会通知 Dump thread 线程处理，然后完成事务的提交，不会关心是否成功发送到任意一个 slave 中

## 半同步复制



Master 处理事务过程中，提交完事务后，必须等至少一个 Slave 将收到的 binlog 写入 relay log 返回 ack 才能继续执行处理用户的事务。

### 相关配置

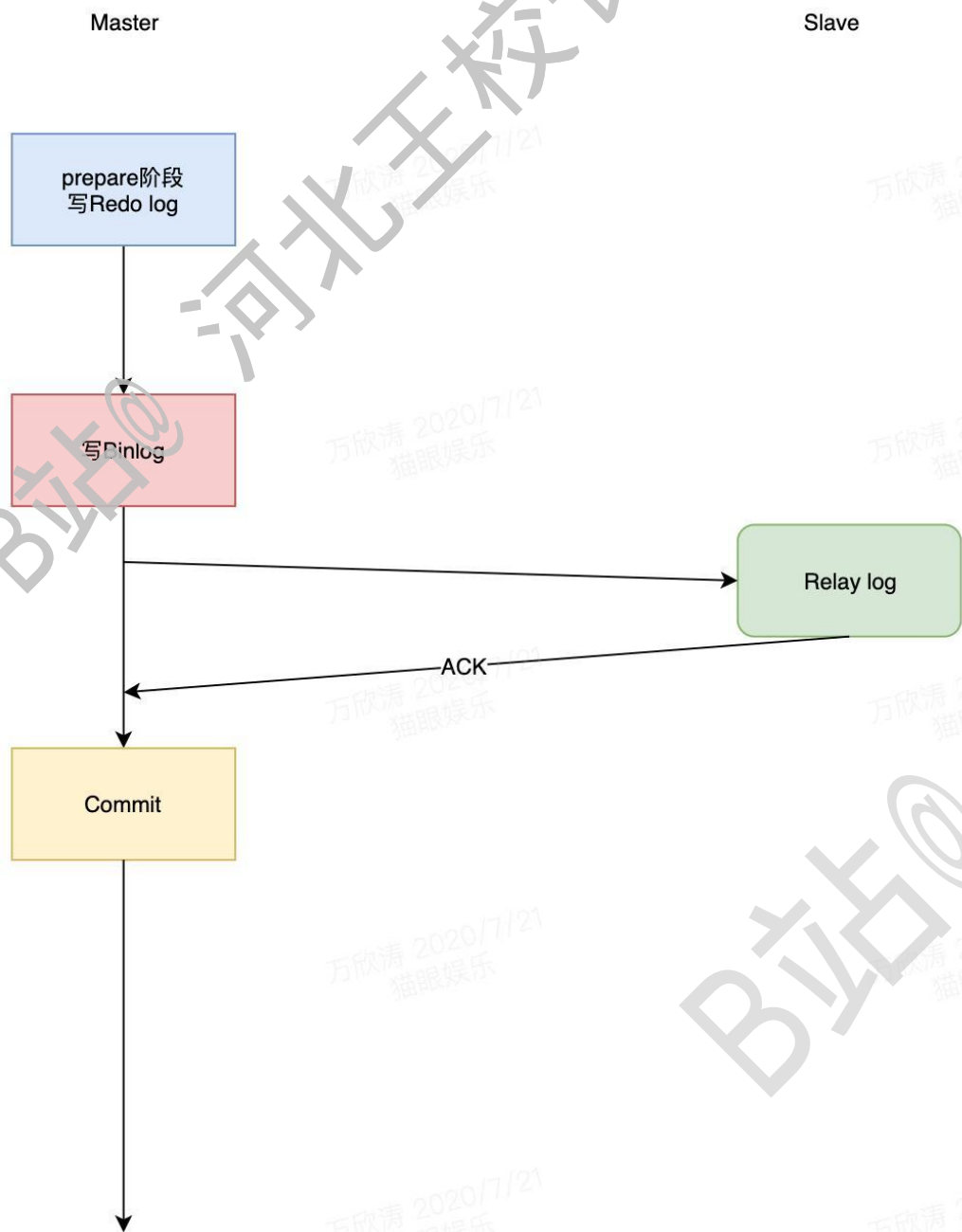
`rpl_semi_sync_master_wait_point = AFTER_COMMIT` 【这里 MySQL 5.5 并没有这个配置，MySQL 5.7 为了解决半同步的问题而设置的】

`rpl_semi_sync_master_wait_for_slave_count = 1` （最低必须收到多少个 slave 的 ack）

`rpl_semi_sync_master_timeout = 100` （等待 ack 的超时时间）



## 增强半同步复制



增强半同步和半同步不同是，等待 ACK 时间不同

`rpl_semi_sync_master_wait_point = AFTER_SYNC` (唯一区别)

半同步的问题是因为等待 ACK 的点是 Commit 之后，此时 Master 已经完成数据变更，用户已经可以看到最新数据，当 Binlog 还未同步到 Slave 时，发生主从切换，那么此时从库是没有这个最新数据的，用户又看到老数据。

增强半同步将等待 ACK 的点放在提交 Commit 之前，此时数据还未被提交，外界看不到数据变更，此时如果发送主从切换，新库依然还是老数据，不存在数据不一致的问题。

## 43. 主从复制有什么好处呢？

- ❑ 在从服务器可以执行查询工作，降低主服务器压力；（主库写，从库读，降压）读写分离
- ❑ 在从主服务器进行备份，避免备份期间影响主服务器服务；容灾
- ❑ 当主服务器出现问题时，可以切换到从服务器。提高可用性

## 44. 说说InnoDB 和 mylsam 存储引擎的主要区别

1. **是否支持行级锁**：MyISAM 只有表级锁 (table-level locking)，而 InnoDB 支持行级锁 (row-level locking) 和表级锁，默认为行级锁。
2. **是否支持事务和崩溃后的安全恢复**：MyISAM 强调的是性能，每次查询具有原子性，其执行速度比 InnoDB 类型更快，但是不提供事务支持。但是 InnoDB 提供事务支持事务，外部键等高级数据库功能。具有事务 (commit)、回滚 (rollback) 和崩溃修复能力 (crash recovery capabilities) 的事务安全 (transaction-safe (ACID compliant)) 型表。
3. **是否支持外键**：MyISAM 不支持，而 InnoDB 支持。
4. **是否支持 MVCC**：仅 InnoDB 支持。应对高并发事务，MVCC 比单纯的加锁更高效；MVCC 只在 READ COMMITTED 和 REPEATABLE READ 两个隔离级别下工作；MVCC 可以使用乐观 (optimistic) 锁和悲观 (pessimistic) 锁来实现；各数据库中 MVCC 实现并不统一。

....

## MyISAM

MyISAM 引擎是 MySQL 5.1 及之前版本的默认引擎，它的特点是：

- 不支持行锁，读取时对需要读到的所有表加锁，写入时则对表加排它锁
- 不支持事务
- 不支持外键
- 不支持崩溃后的安全恢复
- 在表有读取查询的同时，支持往表中插入新纪录
- 支持 **BLOB** 和 **TEXT** 的前 500 个字符索引，支持全文索引
- 支持延迟更新索引，极大提升写入性能
- 对于不会进行修改的表，支持压缩表，极大减少磁盘空间占用

## InnoDB

InnoDB 在 MySQL 5.5 后成为默认索引，它的特点是：

- 支持行锁，采用 MVCC 来支持高并发
- 支持事务
- 支持外键
- 支持崩溃后的安全恢复
- 不支持全文索引

## 45. mysql单表优化有什么经验吗？

### 字段

- 尽量使用 `TINYINT`、`SMALLINT`、`MEDIUM_INT` 作为整数类型而非 `INT`，如果非负则加上 `UNSIGNED`
- `VARCHAR` 的长度只分配真正需要的空间
- 使用枚举或整数代替字符串类型
- 尽量使用 `TIMESTAMP` 而非 `DATETIME`，
- 单表不要有太多字段，建议在 20 以内
- 避免使用 `NULL` 字段，很难查询优化且占用额外索引空间
- 用整型来存 IP

有时候可以使用枚举列代替常用的字符串类型。枚举列可以把一些不重复的字符串存储成一个预定义的集合。MySQL 在存储枚举时非常紧凑，会根据列表值的数量压缩到一个或者两个字节中。MySQL 在内部会将每个值在列表中的位置保存为整数，并且在表的 .frm 文件中保存“数字 - 字符串”映射关系的“查找表”。如 `enum ('man', 'woman')` 实际存储为 1 和 2，减少数据占用空间。

`TIMESTAMP` 占用 4 字节，`DATETIME` 占用 8 字节，且 `TIMESTAMP` 在多数场景下容易转换。

可为 `NULL` 的列会使索引、索引统计和值比较都更复杂。可为 `NULL` 的列会使用更多的存储空间，在 MySQL 里也需要做特殊处理。当可为 `NULL` 的列被索引时，每个索引记录需要一个额外的字节，在 MyISAM 里面甚至可能导致固定大小的索引（例如只有一个整数列的索引）变成可变大小的索引。

整型字段的比较比字符串效率高很多，这也符合一项优化原则：字段类型定义使用最合适（最小）、最简单的数据类型。

`inet_aton()` 算法，其实借用了国际上对各国 IP 地址的区分中使用的 ip number。



索引优化，读写分离不予多说，之前有很多相关面试题

## 查询 SQL

- 可通过开启慢查询日志来找出较慢的 SQL
- 不做列运算：SELECT id WHERE age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数、计算表达式等等，查询时要尽可能将操作移至等号右边
- sql 语句尽可能简单：一条 sql 只能在一个 cpu 运算；大语句拆小语句，减少锁时间；一条大 sql 可以堵死整个库
- 不用 SELECT \*
- OR 改写成 IN：OR 的效率是 n 级别，IN 的效率是 log (n) 级别，in 的个数建议控制在 200 以内
- 不用函数和触发器，在应用程序实现
- 避免 %xxx 式查询
- 少用 JOIN
- 使用同类型进行比较，比如用 '123' 和 '123' 比，123 和 123 比
- 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
- 对于连续数值，使用 BETWEEN 不用 IN：SELECT id FROM t WHERE num BETWEEN 1 AND 5
- 列表数据不要拿全表，要使用 LIMIT 来分页，每页数量也不要太大

## 表分区

MySQL 在 5.1 版引入的分区是一种简单的水平拆分，用户需要在建表的时候加上分区参数，对应用是透明的无需修改代码

对用户来说，分区表是一个独立的逻辑表，但是底层由多个物理子表组成，实现分区的代码实际上是通过一组底层表的对象封装，但对 SQL 层来说是一个完全封装底层的黑盒子。MySQL 实现分区的方式也意味着索引也是按照分区的子表定义，没有全局索引

用户的 SQL 语句是需要针对分区表做优化，SQL 条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，可以通过 `EXPLAIN PARTITIONS` 来查看某条 SQL 语句会落在那些分区上，从而进行 SQL 优化，如下图 5 条记录落在两个分区上：

```
mysql> explain partitions select count(1) from user_partition where id in (1,2,3,4,5);
```

id	select_type	table	partitions	type	possible_keys	key	key_len
1	SIMPLE	user_partition	p1,p4	range	PRIMARY	PRIMARY	8

```
1 row in set (0.00 sec)
```

## 46. 表分区有什么优缺点

分区的好处是：

- 可以让单表存储更多的数据
- 分区表的数据更容易维护，可以通过清楚整个分区批量删除大量数据，也可以增加新的分区来支持新插入的数据。另外，还可以对一个独立分区进行优化、检查、修复等操作
- 部分查询能够从查询条件确定只落在少数分区上，速度会很快
- 分区表的数据还可以分布在不同的物理设备上，从而搞笑利用多个硬件设备
- 可以使用分区表来避免某些特殊瓶颈，例如 InnoDB 单个索引的互斥访问、ext3 文件系统的 inode 锁竞争
- 可以备份和恢复单个分区

分区的限制和缺点：

- 一个表最多只能有 1024 个分区
- 如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来
- 分区表无法使用外键约束



## 47. 表分区有几种方式

分区的类型:

- RANGE 分区: 基于属于一个给定连续区间的列值, 把多行分配给分区
- LIST 分区: 类似于按 RANGE 分区, 区别在于 LIST 分区是基于列值匹配一个离散值集合中的某个值来进行选择
- HASH 分区: 基于用户定义的表达式的返回值来进行选择的分区, 该表达式使用将要插入到表中的这些行的列值进行计算。这个函数可以包含 MySQL 中有效的、产生非负整数值的任何表达式
- KEY 分区: 类似于按 HASH 分区, 区别在于 KEY 分区只支持计算一列或多列, 且 MySQL 服务器提供其自身的哈希函数。必须有一列或多列包含整数值

分区适合的场景有:

- 最适合的场景数据的时间序列性比较强, 则可以按时间来分区

## 48. 如果单表存储数据太大，分区也不能支持，怎么办？

### 垂直拆分

垂直分库是根据数据库里面的数据表的相关性进行拆分，比如：一个数据库里面既存在用户数据，又存在订单数据，那么垂直拆分可以把用户数据放到用户库、把订单数据放到订单库。垂直分表是对数据表进行垂直拆分的一种方式，常见的是把一个多字段的大表按常用字段和非常用字段进行拆分，每个表里面的数据记录数一般情况下是相同的，只是字段不一样，使用主键关联

比如原始的用户表是：

Users

ID	username	email	first	last	location
1	jsmith	smith@example.com	John	Smith	Seattle, WA
2	ramsey	ramsey@example.com	Ramsey	White	Mt. Airy, MD
3	siegfried	crossbow@example.com	Siegfried	Faust	Rostock, Germany
4	amber	amber@example.com	Amber	Simpson	Las Vegas, NV

垂直拆分后是：

Users

ID	username	email
1	jsmith	smith@example.com
2	ramsey	ramsey@example.com
3	siegfried	crossbow@example.com
4	amber	amber@example.com
5	burke.j	jburke@example.com
6	owenh	harper@example.com

UsersExtra

ID	first	last	location
1	John	Smith	Seattle, WA
2	Ramsey	White	Mt. Airy, MD
3	Siegfried	Faust	Rostock, Germany
4	Amber	Simpson	Las Vegas, NV
5	Juliet	Burke	Miami, FL
6	Owen	Harper	Cardiff, UK



垂直拆分的优点是：

- 可以使得行数据变小，一个数据块 (Block) 就能存放更多的数据，在查询时就会减少 I/O 次数 (每次查询时读取的 Block 就少)
- 可以达到最大化利用 Cache 的目的，具体在垂直拆分的时候可以将不常变的字段放一起，将经常改变的放一起
- 数据维护简单

缺点是：

- 主键出现冗余，需要管理冗余列
- 会引起表连接 JOIN 操作 (增加 CPU 开销) 可以通过在业务服务器上进行 join 来减少数据库压力
- 依然存在单表数据量过大的问题 (需要水平拆分)
- 事务处理复杂

## 水平拆分

### 概述

水平拆分是通过某种策略将数据分片来存储，分库内分表和分库两部分，每片数据会分散到不同的 MySQL 表或库，达到分布式的效果，能够支持非常大的数据量。前面的表分区本质上也是一种特殊的库内分表

库内分表，仅仅是单纯的解决了单一表数据过大的问题，由于没有把表的数据分布到不同的机器上，因此对于减轻 MySQL 服务器的压力来说，并没有太大的作用，大家还是竞争同一个物理机上的 IO、CPU、网络，这个就要通过分库来解决

前面垂直拆分的用户表如果进行水平拆分，结果是：

Users\_A\_M

ID	username	email	first	last	location
1	jsmith	smith@example.com	John	Smith	Seattle, WA
4	amber	amber@example.com	Amber	Simpson	Las Vegas, NV
5	burke.j	jburke@example.com	Juliet	Burke	Miami, FL

Users\_N\_Z

ID	username	email	first	last	location
2	ramsey	ramsey@example.com	Ramsey	White	Mt. Airy, MD
3	siegfried	crossbow@example.com	Siegfried	Faust	Rostock, Germany
6	owenh	harper@example.com	Owen	Harper	Cardiff, UK

水平拆分的优点是:

- 不存在单库大数据和高并发的性能瓶颈
- 应用端改造较少
- 提高了系统的稳定性和负载能力

缺点是:

- 分片事务一致性难以解决
- 跨节点 Join 性能差, 逻辑复杂
- 数据多次扩展难度跟维护量极大



## 49. 分片（分库分表）过程中有什么需要注意的吗

### 分片原则

- 能不分就不分，参考单表优化
- 分片数量尽量少，分片尽量均匀分布在多个数据结点上，因为一个查询 SQL 跨分片越多，则总体性能越差，虽然要好于所有数据在一个分片的结果，只在必要的时候进行扩容，增加分片数量
- 分片规则需要慎重选择做好提前规划，分片规则的选择，需要考虑数据的增长模式，数据的访问模式，分片关联性问题，以及分片扩容问题，最近的分片策略为范围分片，枚举分片，一致性 Hash 分片，这几种分片都有利于扩容
- 尽量不要在一个事务中的 SQL 跨越多个分片，分布式事务一直是个不好处理的问题
- 查询条件尽量优化，尽量避免 Select \* 的方式，大量数据结果集下，会消耗大量带宽和 CPU 资源，查询尽量避免返回大量结果集，并且尽量为频繁使用的查询语句建立索引。
- 通过数据冗余和表分区降低跨库 Join 的可能

这里特别强调一下分片规则的选择问题，如果某个表的数据有明显的时间特征，比如订单、交易记录等，则他们通常比较合适用时间范围分片，因为具有时效性的数据，我们往往关注其近期的数据，查询条件中往往带有时间字段进行过滤，比较好的方案是，当前活跃的数据，采用跨度比较短的时间段进行分片，而历史性的数据，则采用比较长的跨度存储。

总体来说，分片的选择是取决于最频繁的查询 SQL 的条件，因为不带任何 Where 语句的查询 SQL，会遍历所有的分片，性能相对最差，因此这种 SQL 越多，对系统的影响越大，所以我们要尽量避免这种 SQL 的产生。

## 分片（分库分表）后，如何保证全局的唯一主键id呢？

生成全局 id 有下面这几种方式：

**UUID**：不适合作为主键，因为太长了，并且无序不可读，查询效率低。比较适合用于生成唯一的名字的标示比如文件的名字。

**数据库自增 id**：两台数据库分别设置不同步长，生成不重复 ID 的策略来实现高可用。这种方式生成的 id 有序，但是需要独立部署数据库实例，成本高，还会有性能瓶颈。

**利用 redis 生成 id**：性能比较好，灵活方便，不依赖于数据库。但是，引入了新的组件造成系统更加复杂，可用性降低，编码更加复杂，增加了系统成本。



## 50. MySQL 语句执行原理

