2020 面试题总结——MySQL 篇

1、数据库隔离级别有哪些，各自的含义是什么，MySQL 默认的隔离级别是多少。

隔离级别有四种。

- 未提交读：是最低的隔离级别，其含义是允许一个事务读取另外一个事务没有提交的数据。会出现脏读。
- 读写提交：是指一个事务只能读取另一个事务已经提交的数据，不能读取未提交的数据。克服了脏读，但会出现不可重复读现象。
- 可重复读：克服读写提交中出现的不可重复读现象。但会出现幻读现象。
- 串行化：数据库中最高的隔离级别，她会要求所有的 SQL 都会按照顺序执行，这样就可以克服上述隔离级别出现的各种问题，所以它能完全保证数据的一致性。

MySQL 默认的隔离级别是可重复读。

2、什么是幻读。

幻读是指在同一个事务下，连续执行两次同样的 SQL 语句可能导致不同的结果，第二次的 SQL 语句可能会返回之前不存在的行。

事务 A 读取与搜索条件相匹配的若干行，事务 B 以插入或删除行等方式来修改事务 A 的结果集，然后再提交，就会发生幻读。例如第一个事务对一个表中的数据进行了修改，比如这种修改涉及到表中的“全部数据行”。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入“一行新数据”。那么，以后就会发生操作第一个事务的用户发现表中还存在没有修改的数据行，就好象发生了幻觉一样。

在默认的事务隔离级别下，即 REPEATABLE READ（可重复读）下，InnoDB 存储引擎采用 **Next-Key Locking** 机制来避免幻读。

具体看《MySQL 技术内幕-InnoDB 存储引擎》的 6.4.2 小节。

3、MySQL 有哪些存储引擎，各自优缺点。

MySQL 支持 InnoDB、MyISAM、MEMORY 等存储引擎。

InnoDB 引擎（MySQL5.5 以后默认使用）：

- 灾难恢复性好
- 支持事务
- 使用行级锁和表级锁，能支持更多的并发量

- 查询不加锁
- 支持外键关联
- 支持热备份
- 实现缓冲管理

MyISAM 引擎：

- 不支持事务
- 使用表级锁，并发性差
- 主机宕机后，MyISAM 表易损坏，灾难恢复性不佳
- 可以配合锁，实现操作系统下的复制备份、迁移
- 只缓存索引
- 数据紧凑存储，因此可获得更小的索引和更快的全表扫描性能

两者主要区别：

- InnoDB 支持事务，MyISAM 不支持事务处理等高级处理。
- InnoDB 支持行级锁，而 MyISAM 仅支持表级锁。
- MyISAM 类型的表强调的是性能，其执行速度比 InnoDB 类型更快。
- MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用。
- InnoDB 支持外键，MyISAM 不支持。
- MyISAM 支持全文搜索，而 InnoDB 1.2.x 版本后才支持。
- 对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引。

4、高并发下，如何做到安全的修改同一行数据。

- 使用悲观锁。本质是当前只有一个线程执行操作，排斥外部请求的修改。遇到加锁的状态，就必须等待。结束了唤醒其他线程进行处理。但是，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。
- FIFO（先进先出）缓存队列思路，直接将请求放入队列中，这样就不会导致某些请求永远获取不到锁。有点强行把多线程变成单线程的感觉。
- 使用乐观锁。相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。

5、乐观锁和悲观锁是什么，InnoDB 的标准行级锁有哪两种，解释其含义。

悲观锁和乐观锁是两种常见的资源并发锁设计思路。

悲观锁：它指的是对数据被外界（包括当前系统的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁是使用数据库内部的锁（排他锁）对记录进行加锁，从而使得其他事务等待以保证数据的一致性。**通常通过常用的 select ... for update 操作来实现悲观锁**，在 SQL 的最后加入 for update 语句，就可以在数据库事务执行过程中，锁定查询出来的数据，其他事务将不能再对其进行读写操作，这样避免了数据的不一致，单个请求直至数据库事务完成，才会释放这个锁。

乐观锁：是指一种不使用数据库锁和不阻塞线程并发的思路。它的特点是先进行业务操作，不到万不得已不去拿"锁"。即"乐观"的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。乐观锁在数据库的实现完全是逻辑的，不需要数据库提供特殊的支持，一般的做法是在需要锁的数据上增加一个版本号，或者时间戳。

InnoDB 存储引擎实现了两种标准的行级锁：

- 共享锁（S Lock），允许事务读一行数据，可以多个事务同时获取，也成为锁兼容。但阻止其他事务获得相同数据集的排他锁。
- 排他锁（X Lock），允许事务删除或更新一行数据，组织其他事务取得相同数据集的共享读锁和排他写锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁，这两种意向锁都是表级别的锁。

- 意向共享锁（IS Lock），事务想要获得一张表中某几行的共享锁，事务再给一个数据行加共享锁之前必须先取得该表的 IS 锁。
- 意向排他锁（IX Lock），事务想要获得一张表中某几行的排他锁，事务在给一个数据行加排他锁之前必须先取得该表的 IX 锁。

简洁来说，悲观锁就是用共享锁和排他锁；而乐观锁，实际通过版本号，从而实现 CAS 原子性更新。

5、SQL 优化的一般步骤是什么，怎么看执行计划，如何理解其中各个字段的含义。

SQL 优化步骤一般是：

- 通过 show status 命令了解各种 SQL 的执行频率
- 定位执行效率较低的 SQL 语句
- 通过 EXPLAIN 分析较低 SQL 的执行计划
- 通过 show profile 分析 SQL
- 通过 trace 分析优化器如何选择执行计划

- 确定问题并采取相应的优化措施

执行计划是 SQL 在数据库执行时的表现情况，通常用于 SQL 性能分析、优化等场景。在 MySQL 中使用 explain 关键字来查看。

参考链接： SQL 优化的一般步骤是什么，怎么看执行计划，如何理解其中各个字段的含义。

6、数据库会死锁吗，举一个死锁的例子，mysql 是怎么解决死锁的。

数据库会出现死锁。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相互等待的现象。

举个例子，一个用户 A 访问表 A（锁住了表 A），然后又访问表 B；另一个用户 B 访问表 B（锁住了表 B），然后企图访问表 A；这时用户 A 由于用户 B 已经锁住表 B，它必须等待用户 B 释放表 B 才能继续，同样用户 B 要等用户 A 释放表 A 才能继续，这就死锁就产生了。

解决死锁最简单的方式是不要有等待，将任何的等待都转换为回滚，并且事务重新开启。但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。这个方法不适用。

另一个简单方法是超时，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。

除了超时机制外，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，这是一种更为主动的死锁检测方式，InnoDB 存储引擎中也采用这种方式。

7、MySQL 的索引原理，索引的类型有哪些，如何创建合理的索引，索引如何优化。

MySQL 中索引采用的数据结构主要是 B+Tree，Hash，平衡二叉树等。

索引的类型可以分为：

- 普通索引（INDEX），最基本的索引，没有任何的约束。INDEX index_name (name)
- 唯一索引（UNIQUE），与普通索引类似，但索引列的值必须唯一，但允许有控制（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。UNIQUE index_name (name)
- 全文索引（ FULLTEXT ），MyISAM 表全系支持，InnoDB 1.2.x 后支持。FULLTEXT (content)
- 主键索引（PRIMARY KEY），特殊的唯一索引，一个表只能有一个，不允许有空值。

- 复合索引，将多个列组合在一起创建索引，可以覆盖多个列。

索引如何优化：

- 非空字段 NOT NULL，Mysql 很难对空值作查询优化
- 区分度高，离散度大，作为索引的字段值尽量不要有大量相同值
- 索引的长度不要太长（比较耗费时间）

参考链接：mysql 索引总结(1)-mysql 索引类型以及创建 和 MySQL 的索引原理，索引的类型有哪些，如何创建合理的索引，索引如何优化。

8、聚集索引和非聚集索引的区别。

非聚集索引也称之为辅助索引。聚集索引与辅助索引不同的是，叶子节点存放的是否是一整行的数据。聚集索引叶子节点存放的即为整张表的行记录数据；而辅助索引叶子节点除了包含键值以外，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。**所以，聚集索引一般比辅助索引体积大。

由于实际的数据页只能按照一颗 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只能找到数据。聚集索引的好处在于，它对于主键的排序查找和范围查找速度都非常快，叶子节点的数据就是用户所要查询的数据。需要注意的是：

- 如果一个主键被定义了，那么这个主键就是作为聚集索引。
- 如果没有主键被定义，那么该表的第一个唯一非空索引被作为聚集索引。
- 如果没有主键也没有合适的唯一索引，那么 innodb 内部会生成一个隐藏的主键作为聚集索引，这个隐藏的主键是一个 6 个字节的列，改列的值会随着数据的插入自增。

另外，切记的是，**聚集索引的存储并不是物理上连续的，而是逻辑上连续的。**这其中有两点：

- 每个表的数据页通过双向链表链接，页按照主键的顺序排序；
- 每个页的记录也是通过双向链表进行维护的，物理存储上可以同样不按照主键存储。

辅助索引的存在并不影响数据在聚集索引的组织，所以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（也就是聚集索引）的主键，然后在通过主键索引来找到一个完整的行记录。

以上说的都是 InnoDB 存储引擎场景下，而对于 MyISAM 引擎，索引文件和数据文件是分离的，索引文件仅保存数据记录的地址，具体不细究。

具体看《MySQL 技术内幕-InnoDB 存储引擎》的 5.4 小节。

9、select for update 是什么含义，会锁表还是锁行还是其他。

select for update 会锁定查询出来的数据，其他事务将不能再对其进行读写操作，这样避免了数据的不一致，单个请求直至数据库事务完成，才会释放这个锁。**记住，for update 是排他锁。**

当使用 select ... for update ...where ...时，mysql 进行 row lock 还是 table lock 只取决于是否有明确的指定主键，能则为行锁，否则为表锁；未查到数据则无锁。

10、为什么要用 Btree 实现，它是怎么分裂的，什么时候分裂，为什么是平衡的。

具体看《MySQL 技术内幕-InnoDB 存储引擎》

11、数据库的 ACID 是什么。

数据库系统引入事务的主要目的在于，事务会把数据库从一种一致性状态转换为另一种一致状态。

数据库事务具有以下 4 个基本特征，也就是著名的 ACID。

- Atomic（原子性）：事务中包含的操作被看作是一个整体的业务单元，这个业务单元中的操作要么全部成功，要么全部失败，不会出现部分失败、部分成功的场景。
- Consistency（一致性）：事务在完成时，必须使所有的数据都保持一致状态，在数据库中所有的修改都基于事务，保证了数据的完整性。
- Isolation（隔离性）：数据库定义了隔离级别的概念，通过它的选择，可以在不同程度上压制丢失更新的发生。
- Durability（持久性）：事务结束后，所有的数据会固化到一个地方，如保存到磁盘中，即使断电重启后也可以提供给应用程序访问。

12、某一个表有近千万的数据，CRUD 比较慢，如何优化。

- 可以做表拆分，减少单表字段数量，优化表结构。
- 在保证主键有效的情况下，检查主键索引的字段顺序，使得查询语句中条件的字段顺序和主键索引的字段顺序保持一致。
- 建立合理的索引。
- 可以结合 Redis、Memcache 等缓存服务，把复杂的 SQL 进行拆分，充分利用二级缓存，减少数据库 IO 操作。

参考链接： 某个表有近千万数据，CRUD 比较慢，如何优化？

13、MySQL 是怎么优化 table scan 的。

对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

避免全表扫描的优化方案：

- 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
- 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 子句中使用 or 来连接条件，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=10 or num=20。也可以这样查询： select id from t where num=10 union all select id from t where num=20
- in 和 not in 也要慎用，否则会导致全表扫描，如：select id from t where num in(1,2,3) 。可以用 between 来代替 in，如 select id from t where num between 1 and 3
- 应尽量避免在 where 子句中对字段进行表达式操作，这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/2=100。应改为：select id from t where num=100*2

14、如何写 SQL 能够有效地使用到复合索引。

复合索引也叫组合索引，用户可以在多个列上建立索引,这种索引叫做复合索引(组合索引)。复合索引在数据库操作期间所需的开销更小,可以代替多个单一索引。

创建复合索引：CREATE INDEX columnId ON table1(col1,col2,col3) ;

使用复合索引：select * from table1 where col1= A and col2= B and col3 = C

对于复合索引,在查询使用时,最好将条件顺序按找索引的顺序,这样效率最高。

复合索引可以用到多个 where 条件查询下，比如查询年龄是 12 和性别是男的所有学生。这样避免了多一次的排序操作。

参考链接： 如何写 sql 能够有效的使用到复合索引

另外，**联合索引具有最左匹配原则，即最左优先。**比如，我们建立了一个 2 列的联合索引(col1,col2),实际上已经建立了两个联合索引(col1)、(col1,col2），解释如下。

B+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+树是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较 name 来确定下一步的所搜方向，如果 name 相同再依次比较 age 和 sex，最后得到检索的数据；但当(20,F)这样的没有 name 的数据来的时候，b+树就不知道第一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用 name 来指定搜索方向，但下一个字段 age 的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是 F 的数据了（这种情况下无法使用联合索引）。

联合索引的意义在于：

- 一个顶三个。建了一个(a,b,c)的复合索引，那么实际等于建了(a),(a,b),(a,b,c)三个索引，因为每多一个索引，都会增加写操作的开销和磁盘空间的开销。
- 作为覆盖索引。同样的有复合索引（a,b,c），如果有如下的 sql: select a,b,c from table where a=1 and b = 1。那么 MySQL 可以直接通过遍历索引取得数据，而无需回表，这减少了很多的随机 io 操作。
- 索引列越多，通过索引筛选出的数据越少。

参考链接：Mysql 中联合索引的最左匹配原则

15、MySQL 中 in 和 exists 的区别。

in 和 exists 主要用在子查询：

```
select * from A where id in (select id from B);
select * from A where exists (select 1 from B where A.id=B.id);
```

两者区别在于：

- exists 是对外表做 loop 循环，每次 loop 循环再对内表（子查询）进行查询，那么因为对内表的查询使用的索引（内表效率高，故可用大表），而外表有多大都需要遍历，不可避免（尽量用小表），故内表大的使用 exists，可加快效率。
- in 是把外表和内表做 hash 连接，先查询内表，再把内表结果与外表匹配，对外表使用索引（外表效率高，可用大表），而内表多大都需要查询，不可避免，故外表大的使用 in，可加快效率。

外层查询表小于子查询表，则用 exists，外层查询表大于子查询表，则用 in。

16、数据库自增主键可能的问题。

自增长是一个很常见的数据属性，在 MySQL 中我很喜欢让一个自增长属性的字段（比如 ID）当作一个主键。特别是 InnoDB，因为 InnoDB 的聚集索引特性，使用自增长属性的字段当主键性能更好。但是也存在一些问题。

首先是 MyISAM 引擎下，由于该引擎是表锁设计，所以自增长不用考虑并发插入的问题。

然后来到较为复杂的 InnoDB 引擎情况下。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。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。这个实现方式称作 AUTO-INC Locking，这种锁其实是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，插入性能较差，事务必须等待前一个插入的完成。其次，对于 INSERT...SELECT 的大数据量的插入会影响插入的性能，因为另一个事务的插入会被阻塞。

从 MySQL 5.1.22 版本开始，InnoDB 引擎提供了一种轻量级互斥量的自增长实现机制，提高了自增长插入的性能。

另外，在 InnoDB 引擎中，自增长值的列必须是索引，同时必须是索引的第一列，如果不是会抛出异常，而 MyISAM 引擎没有这个问题。

参考《MySQL 技术内幕-InnoDB 存储引擎》的 6.3.4 小节-自增长与锁

另外，在多机数据库设计中，自增长主键 ID 会有重复现象，这也导致了系统设计时单点数据库不能拆库，因为 ID 会重复。

17、MVCC 的含义，如何实现的。

MVCC(Multi Version Concurrency Control 的简称)，代表多版本并发控制。与 MVCC 相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。

MVCC，是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。 这样在读操作不用阻塞写操作，写操作不用阻塞读操作的同时，避免了脏读和不可重复读。

MVCC 最大的优势：读不加锁，读写不冲突。在读多写少的 OLTP 应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能。

MVCC 是通过在每行记录后面保存两个隐藏的列来实现的。这两个列，一个保存了行的创建时间，一个保存行的过期时间（或删除时间）。当然存储的并不是实际的时间值，而是系统版本号（system version number)。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行记录的版本号进行比较。

参考链接：MVCC 的含义，如何实现的？

18、你做过的项目里遇到分库分表了吗？怎么做的，有用到中间件么，比如 sharding、jdbc 等，他们的原理知道吗。

没有遇到过。

19、MySQL 的主从延迟怎么解决。

（没用过主从数据库，这个问题可以不回答了^-^）

主从数据库复制存在的问题：

- 主库宕机后，数据可能丢失；
- 主从同步延迟。

主从延迟解决方法：

- 优化网络
- 升级 Slave 硬件配置
- Slave 调整参数，关闭 binlog，修改 innodb_flush_log_at_trx_commit 参数值
- 升级 MySQL 版本到 5.7，使用并行复制

20、什么是回表，覆盖索引有什么作用。

回表是指，数据库根据索引找到了指定的记录所在行后，还需要根据 rowid 再次到数据块里取数据的操作。在执行计划中，先索引扫描，再通过 rowid 去取索引中未能提供的数据，即为回表。

避免回表的方法就是将需要的字段放在索引中去，查询的时候避免回表。也就是覆盖索引。

覆盖索引：一个索引内包含（或覆盖）所有需要查询的字段的值，即只需扫描索引而无须回表，减少了 IO 操作，提高了效率。

21、B+Tree 索引和 Hash 索引区别？

在 InnoDB 存储引擎使用哈希算法对字典进行查找，其冲突机制采用链表方式（与 JDK1.7 的 HashMap 一样）。

- 哈希索引是自适应索引，InnoDB 存储引擎会根据表的使用情况自动为表生成哈希索引，是无法人为干预的。
- 哈希索引适合等值查询，但是无法进行范围查询。
- 哈希索引没办法利用索引完成排序。
- 哈希索引不支持多列联合索引的最左匹配规则。
- 如果有大量重复键值的情况下，哈希索引的效率会很低，因为存在哈希碰撞问题

22、在 Mybatis 中，占位符$和#的区别（防止 SQL 注入）。

先举例来说明，看一个很简单的 MySQL 的 Mapper。

```
<select id="getById" resultType="Blog" parameterType=”int”>
        SELECT id,title,author,content
        FROM blog
WHERE id=#{id}
</select>
```

这里，parameterType 表示了输入的参数类型，resultType 表示了输出的参数类型。回应上文，如果我们想防止 SQL 注入，理所当然地要在输入参数上下功夫。上面代码中 WHERE id=#{id}即输入参数在 SQL 中拼接的部分，传入参数后，打印出执行的 SQL 语句，会看到 SQL 是这样的：

```
SELECT id,title,author,content FROM blog WHERE id = ?
```

不管输入什么参数，打印出的 SQL 都是这样的。这是因为 MyBatis 启用了预编译功能，在 SQL 执行前，会先将上面的 SQL 发送给数据库进行编译；执行时，直接使用编译好的 SQL，替换占位符“?”就可以了。因为 SQL 注入只能对编译过程起作用，所以这样的方式就很好地避免了 SQL 注入的问题。

【底层实现原理】MyBatis 是如何做到 SQL 预编译的呢？其实在框架底层，是 JDBC 中的 PreparedStatement 类在起作用，PreparedStatement 是我们很熟悉的 Statement 的子类，它的对象包含了编译好的 SQL 语句。这种“准备好”的方式不仅能提高安全性，而且在多次执行同一个 SQL 时，能够提高效率。原因是 SQL 已编译好，再次执行时无需再编译。

话说回来，是否我们使用 MyBatis 就一定可以防止 SQL 注入呢？当然不是，请看下面的代码：

```
<select id="getById" resultType="Blog" parameterType=”int”>
        SELECT id,title,author,content
        FROM blog
WHERE id=${id}
</select>
```

仔细观察，内联参数的格式由“#{xxx}”变为了“${xxx}”。如果我们给参数“id”赋值为“3”，将 SQL 打印出来是这样的：

```
SELECT id,title,author,content FROM blog WHERE id = 3
```

显然，这样是无法阻止 SQL 注入的。在 MyBatis 中，“${xxx}”这样格式的参数会直接参与 SQL 编译，从而不能避免注入攻击。但涉及到动态表名和列名时，只能使用“${xxx}”这样的参数格式。所以，这样的参数需要我们在代码中手工进行处理来防止注入。比如

```
<select id="order" resultType="Blog" parameterType=”map”>
        SELECT id,title,author,content
        FROM blog
ORDER BY ${orderParam}
</select>
```

如果我们给参数“orderParam”赋值为“id”，将 SQL 打印出来是这样的：

```
SELECT id,title,author,content FROM blog ORDER BY id
```

在编写 MyBatis 的映射语句时，尽量采用“#{xxx}”这样的格式。若不得不使用“${xxx}”这样的参数，要手工地做好过滤工作，来防止 SQL 注入攻击。

- #{}：相当于 JDBC 中的 PreparedStatement。
- ${}：是输出变量的值。

简单说，#{}是经过预编译的，是安全的；${}是未经过预编译的，仅仅是取变量的值，是非安全的，存在 SQL 注入。

如果我们 order by 语句后用了${}，那么不做任何处理的时候是存在 SQL 注入危险的，必须要事先手动处理过滤一下输入的内容。

参考文章：Mybatis 如何防止 SQL 注入

23、SQL 的注入攻击是什么，如何防范。

SQL 注入攻击，就是通过把 SQL 命令插入到 Web 表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。

解决 SQL 注入问题的关键是对所有可能来自用户输入的数据进行严格的检查，对数据库配置使用最小权限原则。有几种防范方法：

- 在 JDBC 中，使用 PreparedStatement 来拼接动态字符串，可以保证输入的数据被视作 SQL 中纯粹的字符串，而不会当作 SQL 语法来解释。比如。

```
String url="jdbcUrl";
String userName="userName";
String passWord="passWord";
Connection conn= DriverManager.getConnection(url);
PreparedStatement stmt=conn.prepareStatement("SELECT * FROM user_table
WHERE username=? AND password=?");
stmt.setString(1,userName);
```

```
stmt.setString(2,passWord);
```

- 对进入数据库的特殊字符（'"\尖括号&*;等）进行转义处理，或编码转换。或者直接禁止用户向参数中写入特殊字符。
- 网站每个数据库的编码统一，建议全部使用 utf-8 编码，上下层编码不一致有可能导致一些过滤模型被绕过。
- 严格限制网站用户对数据库的操作权限，给此用户提供仅仅能够满足其工作的权限，从而最大限度地减少注入攻击对数据库的损坏。

24、什么时候添加 B+树索引。

如果每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+树索引最合适。

怎么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（索引基数）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。故在访问高选择性属性地字段并从表中去除很少一部分数据时，对这个字段添加 B+树索引是非常有必要的。

另外，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。当然，并不是每次发生 INSERT 和 UPDATE 时就去更新 Cardinality 信息。

25、MySQL 的嵌套事务。

（这道题主要为了纪念当时华为的面试题。。。）

在 InnoDB 引擎下，MySQL 是支持嵌套事务的。嵌套事务是一个层次结构框架，由一个顶层事务控制各个层次的事务。顶级事务之下嵌套的事务被称为子事务，其控制着每一个局部的变换。需要注意的是：

- 子事务既可以提交也可以回滚，但它的提交操作并不马上生效，除非其父事务已经提交。因此可以推断出，任务子事务都在顶层事务提交后才真正的提交。
- 任何一个父事务的回滚会引起它所有的子事务一同回滚，故子事务仅保留 A、C、I 特征，不具有 D 特征（一致性）。

具体看《MySQL 技术内幕-InnoDB 存储引擎》7.1.2 小节中的嵌套事务部分。

26、给出一个学生成绩 studuent 表，写一个 SQL 语句，统计每个学生所有成绩平均分大于 80 分的结果。

这个问题主要考察 GROUP BY 和 HAVING 语句的联合使用，答案如下。

```
SELECT id, COUNT(course) as numcourse, AVG(score) as avgscore
FROM student
GROUP BY id
HAVING AVG(score)>=80;
```

在 select 语句中可以使用 groupby 子句将行划分成较小的组，然后，使用聚组函数返回每一个组的汇总信息，另外，可以使用 having 子句限制返回的结果集。HAVING 语句的存在弥补了 WHERE 关键字不能与聚合函数联合使用的不足。

HAVING 子句对 GROUP BY 子句设置条件的方式与 WHERE 和 SELECT 的交互方式类似。WHERE 搜索条件在进行分组操作之前应用；而 HAVING 搜索条件在进行分组操作之后应用。

HAVING 语法与 WHERE 语法类似，但 HAVING 可以包含聚合函数。HAVING 子句可以引用选择列表中显示的任意项。

27、MySQL 中一条 SQL 语句的执行过程。

SQL 是一套标准，是用来完成和数据库之间的通信的编程语言，SQL 语言是脚本语言，直接运行在数据库上。同时，SQL 语句与数据在数据库上的存储方式无关，只是不同的数据库对于同一条 SQL 语句的底层实现不同罢了，但结果相同。SQL 语句如下，序号则为实际执行顺序：

```
(7) SELECT
(8) DISTINCT <select_list>
(1) FROM <left_table>
(3) <join_type> JOIN <right_table>
(2) ON <join_condition>
(4) WHERE <where_condition>
(5) GROUP BY <group_by_list>
(6) HAVING <having_condition>
(9) ORDER BY <order_by_condition>
(10) LIMIT <limit_number>
```

建议直接去看这篇文章：SQL 查询之执行顺序解析

28、MySQL 中 int(11)中的 11 代表什么含义。

nt(11) 中的 11 ，不影响字段存储的范围，只影响展示效果。当数字不足 11 位时，前面会用 0 补齐。

mysql 中 int 长度并不影响数据的存储精度，长度只与显示有关。无论是 int(3) 还是 int(9)，存储的都是 4 字节无符号整数，也就是 0~2^32。

29、InnoDB 中为什么采用 B+树结构，而不是平衡树。

数据库文件很大，不可能全部存储在内存中，故要存储到磁盘上。索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数。索引采用 B/+Tree 而不是二

叉查找树，关键因素就是磁盘 I/O 次数。这是一种**多路搜索树**，而不是简单的二叉树。

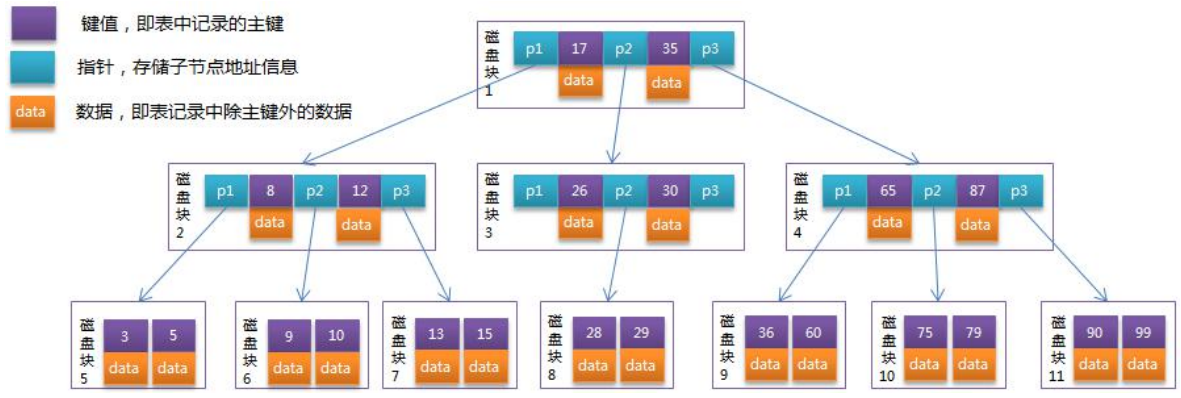
BTree 是为磁盘等外存储设备设计的一种平衡查找树。因此在讲 B-Tree 之前先了解下磁盘的相关知识。

- 系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。InnoDB 存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。
- InnoDB 存储引擎中默认每个页的大小为 16 KB，可通过参数 innodb_page_size 将页的大小设置为 4K、8K、16K，在 MySQL 中可通过如下命令查看页的大小：

```
mysql> show variables like 'innodb_page_size';
```

- 而系统一个磁盘块的存储空间往往没有这么大，因此 InnoDB 每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小 16KB 。InnoDB 在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘 I/O 次数，提高查询效率。

BTree 结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述 BTree，首先定义一条记录为一个二元组 [key, data] ，key 为记录的键值，对应表中的主键值，data 为一行记录中除主键外的数据。对于不同的记录，key 值互不相同。BTree 结构如下图所示。



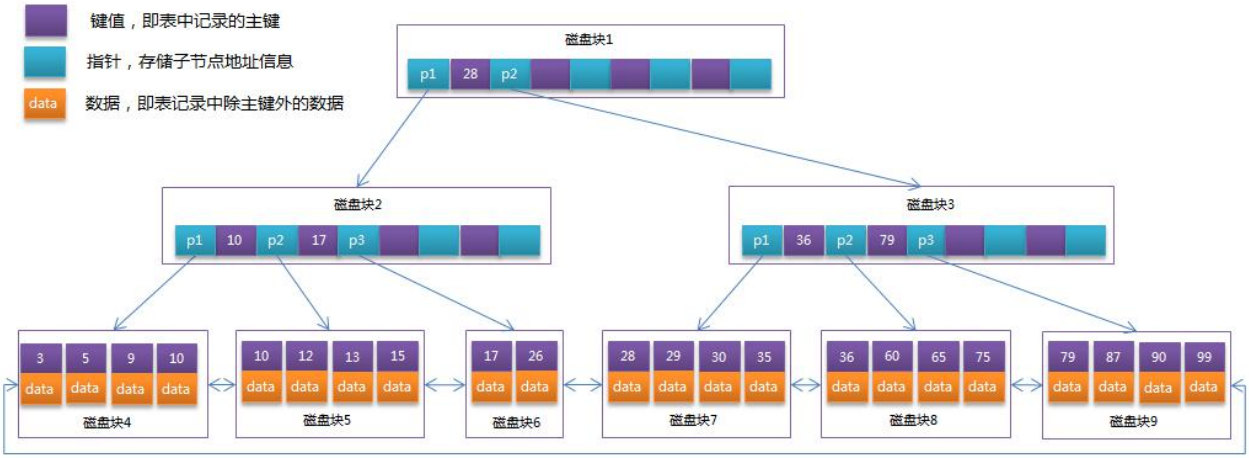
B+Tree 是在 B-Tree 基础上的一种优化，使其更适合实现外存储索引结构，InnoDB 存储引擎就是用 B+Tree 实现其索引结构。从刚刚的 B-Tree 结构图中可以看到，每个节点中不仅包含数据的 key 值，还有 data 值。而每一个页的存储空间是有限的，如果 data 数据较大时将会导致每个节点（即一个页）能存储的 key 的数量很小，当存储的数据量很大时同样会导致 B-Tree 的深度较大，增大查询时的磁盘 I/O 次数，进而影响查询效率。在 B+Tree 中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存

储 key 值信息，这样可以大大加大每个节点存储的 key 值数量，降低 B+Tree 的高度。

B+Tree 相对于 BTree 有几点不同：

- 非叶子节点只存储键值信息。
- 所有叶子节点之间都有一个链指针。
- 数据记录都存放在叶子节点中。

将 B-Tree 优化，由于 B+Tree 的非叶子节点只存储键值信息，假设每个磁盘块能存储 4 个键值及指针信息，则变成 B+Tree 后其结构如下图所示：



总结一波：因为 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。

参考文章：为什么数据库选 B-tree 或 B+tree 而不是二叉树作为索引结构

30、MySQL 索引的“创建”原则。

可以结合 24 题一起来看。主要在以下几种条件下，推荐创建索引。

- 最适合索引的列是出现在 WHERE 子句中的列，或连接子句中的列，而不是出现在 SELECT 关键字后的列。
- 索引列的基数 Cardinality 越大，索引效果越好。
- 根据情况创建复合索引，复合索引可以提高查询效率。因为复合索引的基数会更大。
- 避免创建过多的索引，索引会额外占用磁盘空间，降低写操作效率。
- 主键尽可能选择较短的数据类型，可以有效减少索引的磁盘占用提高查询效率。
- 对字符串进行索引，应该定制一个前缀长度，可以节省大量的索引空间。

31、为什么官方建议使用自增长主键作为索引。

结合 B+Tree 的特点，自增主键是连续的，在插入过程中尽量减少页分裂，即使要进行页分裂，也只会分裂很少一部分。并且能减少数据的移动，每次插入都是插入到最后。总之就是减少分裂和移动的频率。

32、MySQL 主从复制的作用和原理。

主从复制，是用来建立一个和主数据库完全一样的数据库环境，称为从数据库。主数据库一般是准实时的业务数据库。

主从复制的好处是从数据库可以作为数据的热备份，作为后备数据库，主数据库服务器故障后，可切换到从数据库继续工作，避免数据丢失。还可以支持读写分离，在主数据库上进行写入工作，在从数据库执行查询工作，支持更大的并发性能。

先介绍两个**重要概念**。

- 主库 bin-log：二进制日志，记录主库发生的修改事件。
- 从库 relay-log：中继日志，存储所有主库 TP 过来的 bin-log 事件。

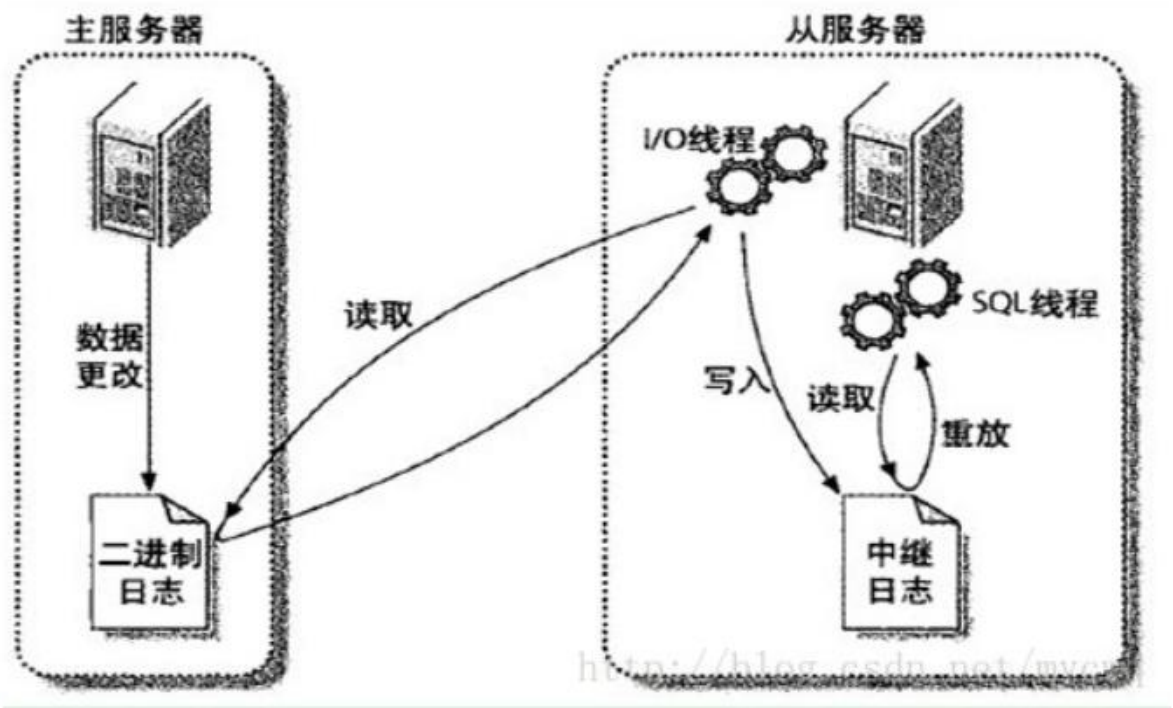
主从复制库的原理：

- 数据库有一个 bin-log 的二进制文件，记录了所有的 SQL 语句。
- 目标就是把主数据库的 bin-log 文件的 SQL 语句复制过来。
- 让其在从数据库的 relay-log 重做日志文件中再执行一次这些 SQL 语句即可。

具体操作过程需要三个线程。

- bin-log 输出线程：每当从库连接到主库的时候，主库都会创建一个线程，然后发送 bin-log 内容到从库。在从库里，当复制开始的时候，从库就会创建两个线程进行处理。
- 从库 I/O 线程：当 START SLAVE 语句在从库开始执行之后，从库创建一个 I/O 线程，该线程连接到主库并请求主库发送到 bin-log 里面的更新记录到从库中，从库 I/O 线程读取主库的 bin-log 输出线程发送的更新并拷贝这些更新到本地文件，其中包括 relay-log 文件（中继日志）。
- 从库的 SQL 线程：从库创建一个 SQL 线程，这个线程读取从库 I/O 线程写到 relay log 的更新事件并执行。

从上可知，对于每一个主从复制的连接，都有三个线程。拥有多个从库的主库为每一个连接到主库的从库创建一个 bin-log 输出线程，每一个从库都有它自己的 I/O 线程和 SQL 线程。



33、MySQL 事务日志。

innodb 事务日志包括 redo log 和 undo log。redo log 是重做日志，提供前滚操作；undo log 是回滚日志，提供回滚操作。

- redo log 通常是物理日志，记录的是数据页的物理修改，而不是某一行或某几行修改成怎样怎样，它用来恢复提交后的物理数据页(恢复数据页，且只能恢复到最后一次提交的位置)。又叫做重做日志文件，用于记录事务操作的变化，记录的是数据修改之后的值，不管事务是否提交都会记录下来。如数据库掉电后，InnoDB 存储引擎会使用 redo log 来恢复到掉电前的时刻，以此来保证数据的完整性。
- undo 用来回滚行记录到某个版本。undo log 一般是逻辑日志，根据每行记录进行记录。

redo log

事务中所有操作会先写到 redo log 中，然后再同步到数据库文件中。所以数据库文件进行事务操作修改时，redo log 肯定已经记录了所有事务操作，此时即使数据库挂掉，事务操作也都已经持久化到 redo log 中了，数据库恢复后可以继续执行剩下操作。它保证了事务的**一致性**。

undo log

undo log 有两个作用：

- 提供回滚

- 多个行版本控制(MVCC)

undo log 和 redo log 记录物理日志不一样，它是逻辑日志。可以认为当 delete 一条记录时，undo log 中会记录一条对应的 insert 记录，反之亦然，当 update 一条记录时，它记录一条对应相反的 update 记录。

undo log 是采用**段(segment)的方式**来记录的，每个 undo 操作在记录的时候占用一个 undo log segment。

它保证了事务的**原子性**。

34、JOIN 的用途。

JOIN 按照功能大致分为如下三类：

- INNER JOIN（内连接,或等值连接）：获取两个表中字段匹配关系的记录。
- LEFT JOIN（左连接）：获取左表所有记录，即使右表没有对应匹配的记录。
- RIGHT JOIN（右连接）： 与 LEFT JOIN 相反，用于获取右表所有记录，即使左表没有对应匹配的记录。

35、PreparedStatement 和 Statement 的区别。

- PreparedStatement 接口继承于 Statement，PreparedStatement 是**预编译**的，实例中包含的是已编译好的 SQL 语句，执行速度要快于 Statement 对象。以后每当执行同一个 PreparedStatement 对象时，预编译的命令是可以重复使用的。
- PreparedStatement 可以**防止 SQL 注入式攻击**，在使用参数化查询的情况下，数据库系统不会将参数的内容视为 SQL 指令的一部分来处理，而是在数据库完成 SQL 指令的编译后，才套用参数运行，因此就算参数中含有破坏性的指令，也不会被数据库所运行。

36、数据库第一、第二、第三范式的理解。

第一范式，是指没有重复的列，表示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。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

第二范式，是指属性完全依赖主键，要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。例如员工信息表中加上了员工编号（emp_id）列，因为每个员工的员工编号是惟一的，因此每个员工可以被惟一区分。这个惟一属性列被称为主关键字或主键、主码。

第三范式，是要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept_id）、部门名称、部门简介等信息。那么在的员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性。 也就是说， 如果存在非主属性对于码的传递函数依赖，则不符合 3NF 的要求。

37、MySQL 半同步复制原理。

MySQL 主从复制分为**异步、同步和半同步复制**，区别主要如下：

- 异步复制（Asynchronous replication），MySQL 默认的复制是异步的，主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理。原理最简单，性能最好，但是主从之间数据不一致的概率很大。
- 全同步复制（Fully synchronous replication），指当主库执行完一个事务，所有的从库都执行了该事务才返回给客户端。因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。
- 半同步复制（Semisynchronous replication），介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到并写到 relay log 中才返回给客户端。相对于异步复制，半同步复制牺牲了一定的性能，提高了数据的安全性。

半同步复制原理：

默认情况下，MySQL 的主从复制是异步的，异步复制可以提供最佳的性能，主库把 binlog 日志发送给从库，然后将结果返回给客户端，并不会验证从库是否接收完毕。这也就意味着有可能出现当主库或从库发生故障的时候，从库没有接收到主库发送过来的 binlog 日志，导致主库和从库的数据不一致，甚至在恢复时造成数据的丢失。为了解决上述出现的问题，MySQL 5.5 引入了一种半同步复制模式。该模式可以确保从库接收完主库发送的 binlog 日志文件并写入到自己的中继日志 relay log 里，然后会给主库一个反馈，告诉主库已经接收完毕，这时主库才返回结果给客户端告知操作完成。当出现从库响应超时情况时，主库会暂时切换到异步复制模式，直到下一次同步没有超时转为半同步复制为止。（master 的 dump 线程除了发送 binlog 数据到 slave，还承担了接收 slave 的 ack 工作。如果出现异常，没有收到 ack，那么将自动降为普通的异步复制，直到异常修复）

38、MySQL 中 Distinct 与 Group by 的区别。

distinct 简单来说就是用来去重的，而 group by 的设计目的则是用来聚合统计的，两者在能够实现的功能上有些相同之处。

单纯的去重操作使用 distinct，速度是快于 group by 的。