MySQL

1、关系型和非关系型数据库的区别你了解多少?

- 关系型数据库的优点
    - 容易理解。因为它采用了关系模型来组织数据。
    - 可以保持数据的一致性。
    - 数据更新的开销比较小。
    - 支持复杂查询（带where子句的查询）
- 非关系型数据库的优点
    - 不需要经过SQL层的解析，读写效率高。
    - 基于键值对，数据的扩展性很好。
    - 可以支持多种类型数据的存储，如图片，文档等等。

2、什么是非关系型数据库?

非关系型数据库也叫NOSQL，采用键值对的形式进行存储。

它的读写性能很高，易于扩展，可分为内存性数据库以及文档型数据库，比如 Redis，Mongodb，HBase等等。

适合使用非关系型数据库的场景：

- 日志系统
- 地理位置存储
- 数据量巨大
- 高可用

3、为什么使用索引?

- 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
- 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
- 帮助服务器避免排序和临时表
- 将随机IO变为顺序IO。
- 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

4、Innodb为什么要用自增id作为主键?

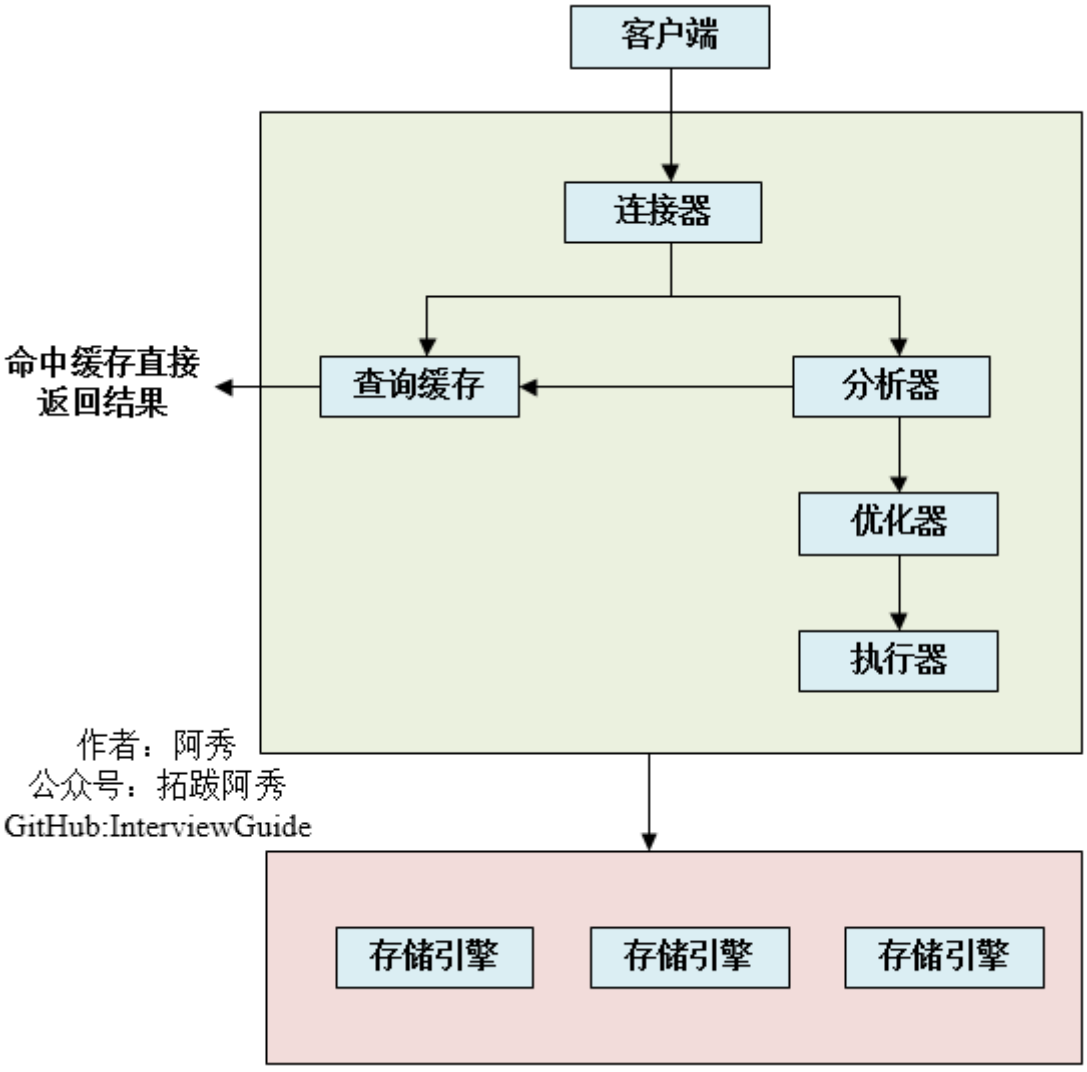
如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。
如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置， 频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE（optimize table）来重建表并优化填充页面。

5、MyISAM和InnoDB实现B树索引方式的区别是什么?

- MyISAM，B+Tree叶节点的data域存放的是数据记录的地址，在索引检索的时候，首先按照B+Tree搜索算法搜索索引，如果指定的key存在，则取出其data域的值，然后以data域的值为地址读取相应的数据记录，这被称为“非聚簇索引”
- InnoDB，其数据文件本身就是索引文件，相比MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按B+Tree组织的一个索引结构，树的节点data域保存了完整的数据记录，这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引，这被称为“聚簇索引”或者聚集索引，而其余的索引都作为辅助索引，辅助索引的data域存储相应记录主键的值而不是地址，这也是和MyISAM不同的地方。

  在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。因此，在设计表的时候，不建议使用过长的字段为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。

6、说一下MySQL是如何执行一条SQL的？具体步骤有哪些?



Server层按顺序执行sql的步骤为：

1. 客户端请求->
2. 连接器（验证用户身份，给予权限） ->
3. 查询缓存（存在缓存则直接返回，不存在则执行后续操作）->
4. 分析器（对SQL进行词法分析和语法分析操作） ->
5. 优化器（主要对执行的sql优化选择最优的执行方案方法） ->
6. 执行器（执行时会先看用户是否有执行权限，有才去使用这个引擎提供的接口）->

7. 去引擎层获取数据返回（如果开启查询缓存则会缓存查询结果）

简单概括：

- **连接器**：管理连接、权限验证；
- **查询缓存**：命中缓存则直接返回结果；
- **分析器**：对SQL进行词法分析、语法分析；（判断查询的SQL字段是否存在也是在这步）
- **优化器**：执行计划生成、选择索引；
- **执行器**：操作引擎、返回结果；
- **存储引擎**：存储数据、提供读写接口。

7、你了解MySQL的内部构造吗？一般可以分为哪两个部分？

可以分为服务层和存储引擎层两部分，其中：

服务层包括连接器、查询缓存、分析器、优化器、执行器等，涵盖MySQL的大多数核心服务功能，以及所有的内置函数（如日期、时间、数学和加密函数等），所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图等。

存储引擎层负责数据的存储和提取。其架构模式是插件式的，支持InnoDB、MyISAM、Memory等多个存储引擎。现在最常用的存储引擎是InnoDB，它从MySQL 5.5.5版本开始成为了默认的存储引擎。

8、说一说Drop、Delete与Truncate的共同点和区别

第一种回答

Drop、Delete、Truncate都表示删除，但是三者有一些差别：
Delete用来删除表的全部或者一部分数据行，执行delete之后，用户需要提交(commmit)或者回滚(rollback)来执行删除或者撤销删除，会触发这个表上所有的delete触发器。
Truncate删除表中的所有数据，这个操作不能回滚，也不会触发这个表上的触发器，TRUNCATE比delete更快，占用的空间更小。
Drop命令从数据库中删除表，所有的数据行，索引和权限也会被删除，所有的DML触发器也不会被触发，这个命令也不能回滚。

因此，在不再需要一张表的时候，用Drop；在想删除部分数据行时候，用Delete；在保留表而删除所有数据的时候用Truncate。

第二种回答

- Drop直接删掉表;
- Truncate删除表中数据，再插入时自增长id又从1开始 ;
- Delete删除表中数据，可以加where字句。

具体解析

1. DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

2. 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。
3. 一般而言，drop > truncate > delete
4. 应用范围。TRUNCATE 只能对TABLE；DELETE可以是table和view
5. TRUNCATE 和DELETE只删除数据，而DROP则删除整个表（结构和数据）。
6. truncate与不带where的delete ： 只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。
7. delete语句为DML（Data Manipulation Language),这个操作会被放到 rollback segment中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。
8. truncate、drop是DDL（Data Define Language),操作立即生效，原数据不放到 rollback segment中，不能回滚
9. 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用delete且注意结合where来约束影响范围。回滚段要足够大。要删除表用drop;若想保留表而将表中数据删除，如果与事务无关，用truncate即可实现。如果和事务有关，或老是想触发trigger,还是用delete。
10. Truncate table 表名 速度快,而且效率高,因为: truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。
11. TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。
12. 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

9、MySQL优化了解吗？说一下从哪些方面可以做到性能优化?

- 为搜索字段创建索引
- 避免使用 Select *，列出需要查询的字段
- 垂直分割分表
- 选择正确的存储引擎

10、数据库隔离级别

- **未提交读**，事务中发生了修改，即使没有提交，其他事务也是可见的，比如对于一个数A原来50修改为100，但是我还没有提交修改，另一个事务看到这个修改，而这个时候原事务发生了回滚，这时候A还是50，但是另一个事务看到的A是100.**可能会导致脏读、幻读或不可重复读**
- **提交读**，对于一个事务从开始直到提交之前，所做的任何修改是其他事务不可见的，举例就是对于一个数A原来是50，然后提交修改成100，这个时候另一个事务在A提交修改之前，读取的A是50，刚读取完，A就被修改成100，这个时候另一个事务再进行读取发现A就突然变成100了；**可以阻止脏读，但是幻读或不可重复读仍有可能发生**
- **重复读**，就是对一个记录读取多次的记录是相同的，比如对于一个数A读取的话一直是A，前后两次读取的A是一致的；**可以阻止脏读和不可重复读，但幻读仍有可能发生**
- **可串行化读**，在并发情况下，和串行化的读取的结果是一致的，没有什么不同，比如不会发生脏读和幻读；**该级别可以防止脏读、不可重复读以及幻读**

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED 未提交读	√	√	√
READ-COMMITTED 提交读	×	√	√
REPEATABLE-READ 重复读	×	×	√
SERIALIZABLE 可串行化读	×	×	×

MySQL InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ**（可重读）

这里需要注意的是：与 SQL 标准不同的地方在于InnoDB 存储引擎在 REPEATABLE-READ（可重读）事务隔离级别 下使用的是**Next-Key Lock 锁**算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server)是不同的。所以 说InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读） 已经可以完全保证事务的隔离性要 求，即达到了 SQL标准的SERIALIZABLE(可串行化)隔离级别。

因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是READ-COMMITTED(读取提交内 容):，但是你要知道的是InnoDB 存储引擎默认使用 **REPEATABLE-READ（可重读）并不会有任何性能损失。**

InnoDB 存储引擎在分布式事务 的情况下一般会用到SERIALIZABLE(可串行化)隔离级别。

11、都知道数据库索引采用B+树而不是B树，原因也有很多，主要原因是什么?

主要原因：B+树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而B树只能中序遍历所有节点，效率太低。

12、文件索引和数据库索引为什么使用B+树?（第9个问题的详细回答）

文件与数据库都是需要较大的存储，也就是说，它们都不可能全部存储在内存中，故需要存储到磁盘上。而所谓索引，则为了数据的快速定位与查找，那么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，因此B+树相比B树更为合适。数据库系统巧妙利用了局部性原理与磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，而红黑树这种结构，高度明显要深的多，并且由于逻辑上很近的节点(父子)物理上可能很远，无法利用局部性。

最重要的是，B+树还有一个最大的好处：方便扫库。

B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持，这是数据库选用B+树的最主要原因。

B+树查找效率更加稳定，B树有可能在中间节点找到数据，稳定性不够。

B+tree的磁盘读写代价更低：B+tree的内部结点并没有指向关键字具体信息的指针(红色部分)，因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一块盘中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多，相对来说IO读写次数也就降低了；

B+tree的查询效率更加稳定：由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引，所以，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；

13、听说过视图吗？那游标呢？

视图是一种虚拟的表，通常是有一个表或者多个表的行或列的子集，具有和物理表相同的功能
游标是对查询出来的结果集作为一个单元来有效的处理。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

14、MySQL中为什么要有事务回滚机制？

而在 MySQL 中，恢复机制是通过回滚日志（undo log）实现的，所有事务进行的修改都会先记录到这个回滚日志中，然后在对数据库中的对应行进行写入。 当事务已经被提交之后，就无法再次回滚了。

回滚日志作用：
1)能够在发生错误或者用户执行 ROLLBACK 时提供回滚相关的信息
2) 在整个系统发生崩溃、数据库进程直接被杀死后，当用户再次启动数据库进程时，还能够立刻通过查询回滚日志将之前未完成的事务进行回滚，这也就需要回滚日志必须先于数据持久化到磁盘上，是我们需要先写日志后写数据库的主要原因。

15、数据库引擎InnoDB与MyISAM的区别

InnoDB

- 是 MySQL 默认的事务型存储引擎，只有在需要它不支持的特性时，才考虑使用其它存储引擎。
- 实现了四个标准的隔离级别，默认级别是可重复读(REPEATABLE READ)。在可重复读隔离级别下，通过多版本并发控制(MVCC)+ 间隙锁(Next-Key Locking)防止幻影读。
- 主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。
- 内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。
- 支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

MyISAM

- 设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。
- 提供了大量的特性，包括压缩表、空间数据索引等。
- 不支持事务。
- 不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入(CONCURRENT INSERT)。

总结

- 事务: InnoDB 是事务型的，可以使用 `Commit` 和 `Rollback` 语句。
- 并发: MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 外键: InnoDB 支持外键。
- 备份: InnoDB 支持在线热备份。
- 崩溃恢复: MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。
- 其它特性: MyISAM 支持压缩表和空间数据索引。

16、数据库并发事务会带来哪些问题?

数据库并发会带来脏读、幻读、丢弃更改、不可重复读这四个常见问题，其中：

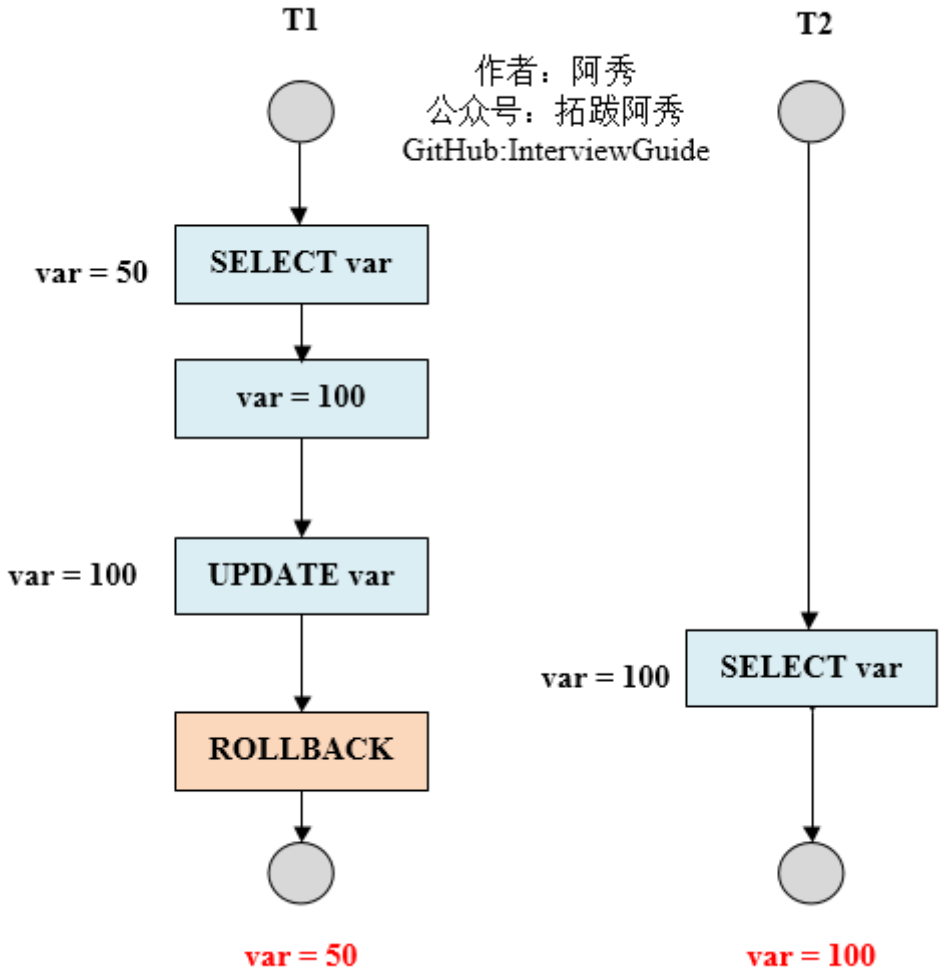
脏读：在第一个修改事务和读取事务进行的时候，读取事务读到的数据为100，这是修改之后的数据，但是之后该事务满足一致性等特性而做了回滚操作，那么读取事务得到的结果就是脏数据了。

幻读：一般是T1在某个范围内进行修改操作（增加或者删除），而T2读取该范围导致读到的数据是修改之间的了，强调范围。

丢弃修改：两个写事务T1 T2同时对A=0进行递增操作，结果T2覆盖T1，导致最终结果是1 而不是2，事务被覆盖

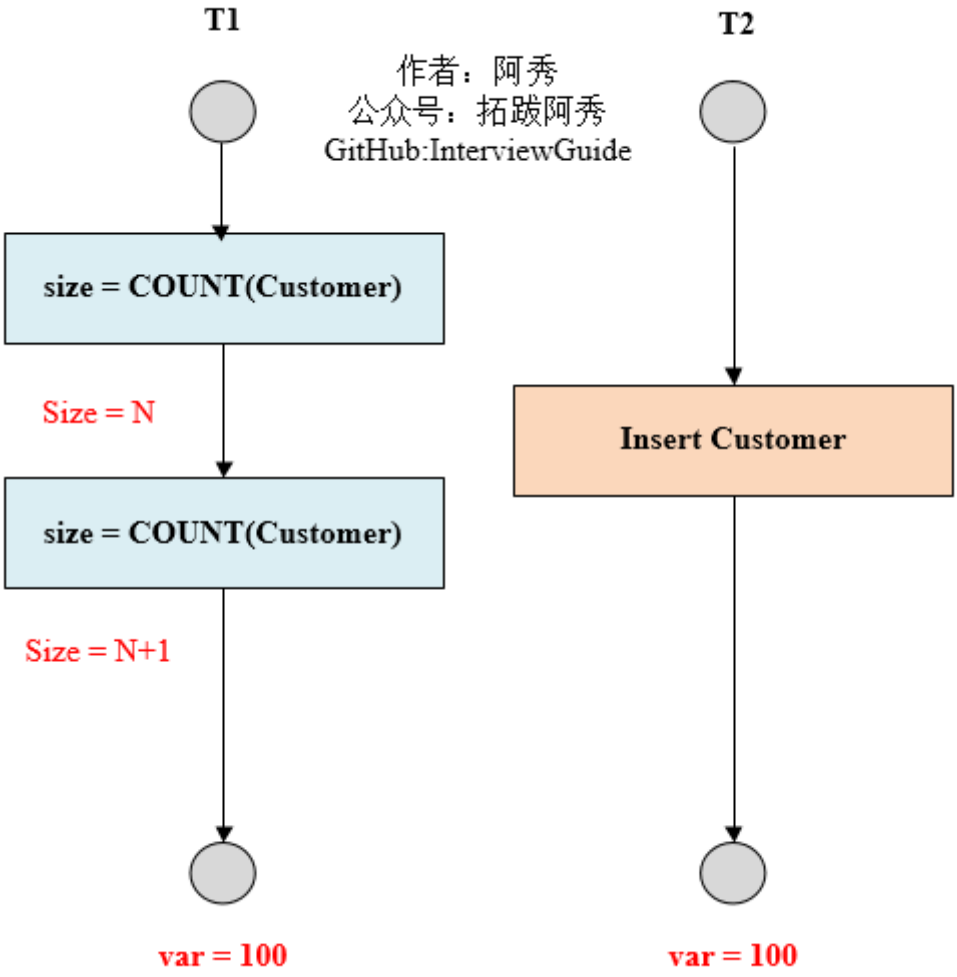
不可重复读：T2 读取一个数据，然后T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。

脏读



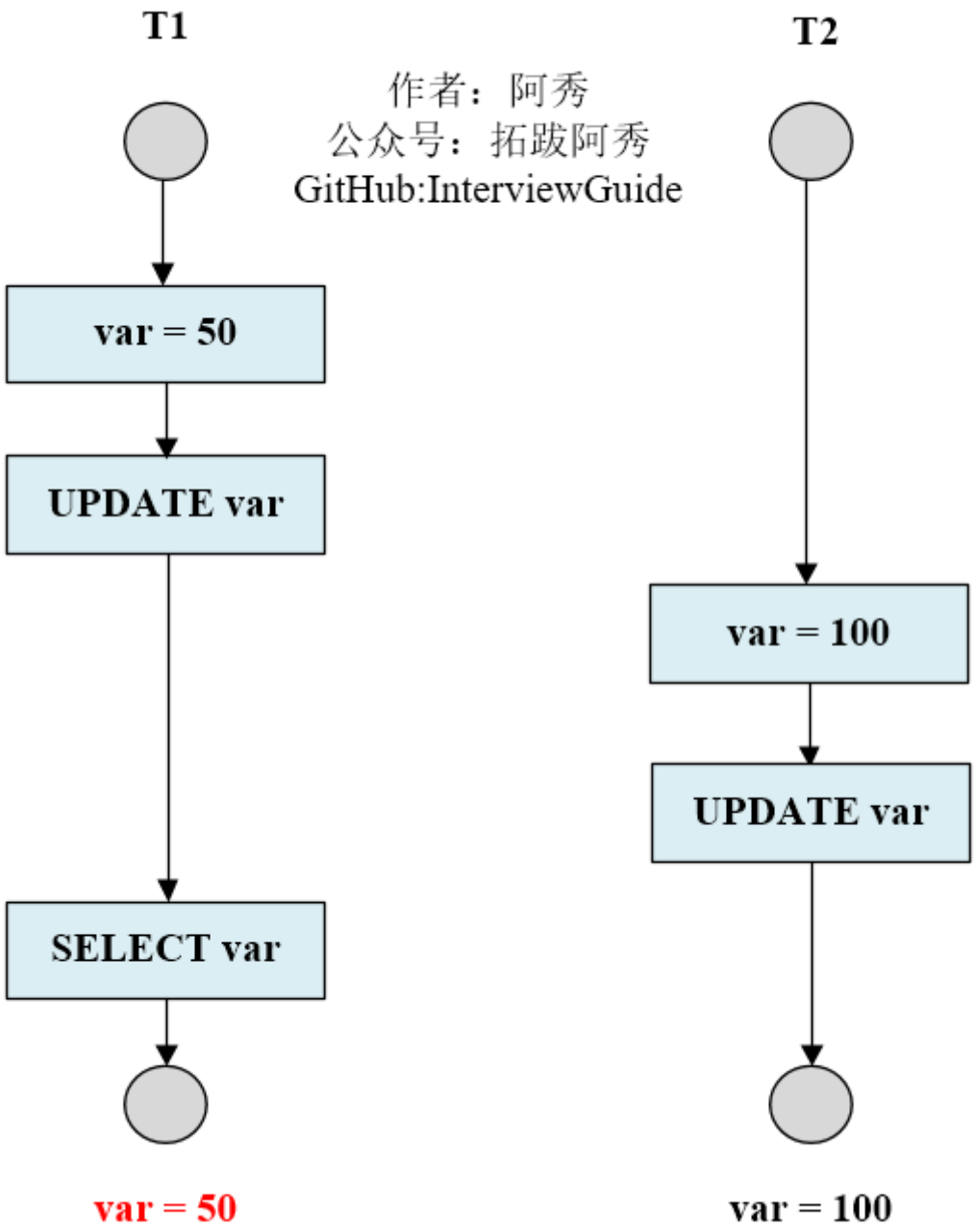
第一个事务首先读取var变量为50，接着准备更新为100的时，并未提交，第二个事务已经读取var为100，此时第一个事务做了回滚。最终第二个事务读取的var和数据库的var不一样。

幻读（幻影读）



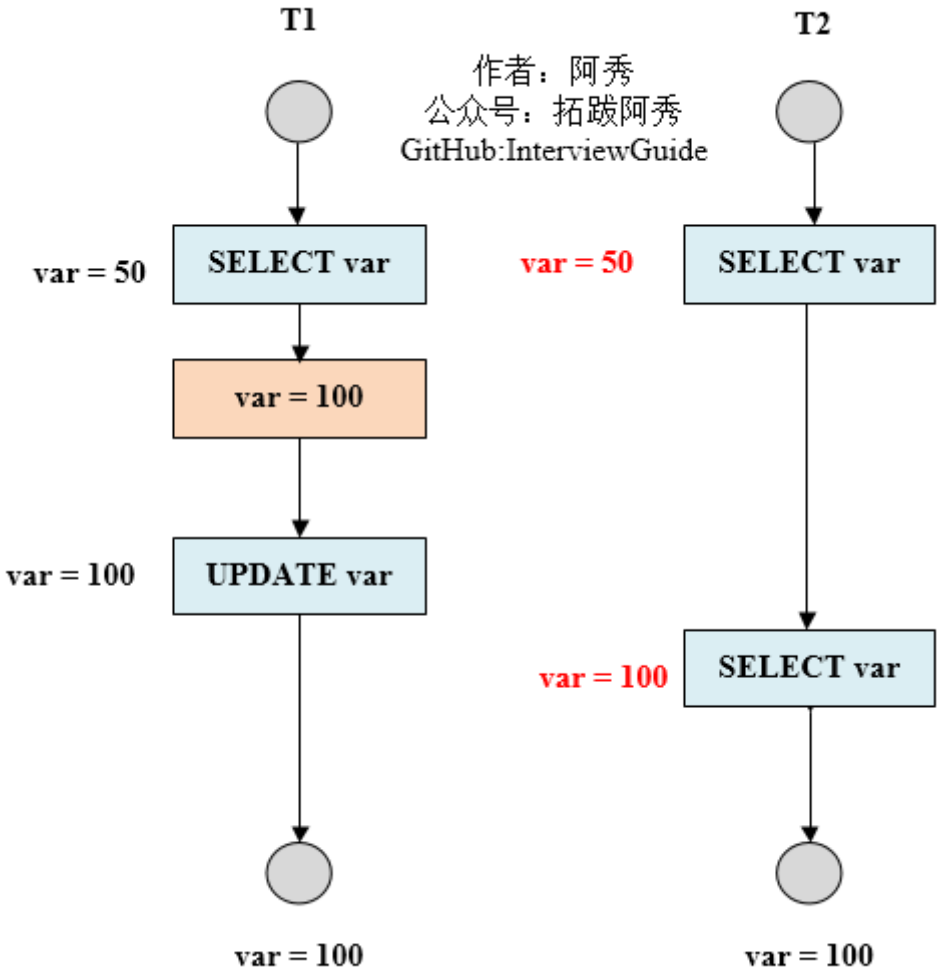
T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。

丢弃修改



T1 和 T2 两个事务都对一个数据进行修改，T1 先修改，T2 随后修改，T2 的修改覆盖了 T1 的修改。例如：事务1读取某表中的数据A=50，事务2也读取A=50，事务1修改A=A+50，事务2也修改A=A+50，最终结果A=100，事务1的修改被丢失。

不可重复读



T2 读取一个数据，T1 对该数据做了修改。如果 T2 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。

17、数据库悲观锁和乐观锁的原理和应用场景分别有什么?

悲观锁，先获取锁，再进行业务操作，一般就是利用类似 SELECT … FOR UPDATE 这样的语句，对数据加锁，避免其他事务意外修改数据。
当数据库执行SELECT … FOR UPDATE时会获取被select中的数据行的行锁，select for update获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

乐观锁，先进行业务操作，只在最后实际更新数据时进行检查数据是否被更新过。Java 并发包中的 AtomicFieldUpdater 类似，也是利用 CAS 机制，并不会对数据加锁，而是通过对比数据的时间戳或者版本号，来实现乐观锁需要的版本判断。

18、MySQL索引主要使用的两种数据结构是什么?

- **哈希索引**，对于哈希索引来说，底层的数据结构肯定是哈希表，因此**在绝大多数需求为单条记录查询**的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引
- **BTree索引**，Mysql的BTree索引使用的是B树中的B+Tree，BTREE索引就是一种将索引值按一定的算法，存入一个树形的数据结构中（二叉树），每次查询都是从树的入口root开始，依次遍历node，获取leaf。

  但对于主要的两种存储引擎（MyISAM和InnoDB）的实现方式是不同的。

19、数据库为什么要进行分库和分表呢？都放在一个库或者一张表中不可以吗？

分库与分表的目的在于，减小数据库的单库单表负担，提高查询性能，缩短查询时间。

通过分表，可以减少数据库的单表负担，将压力分散到不同的表上，同时因为不同的表上的数据量少了，起到提高查询性能，缩短查询时间的作用，此外，可以很大的缓解表锁的问题。
分表策略可以归纳为垂直拆分和水平拆分：
水平分表：取模分表就属于随机分表，而时间维度分表则属于连续分表。
如何设计好垂直拆分，我的建议：将不常用的字段单独拆分到另外一张扩展表. 将大文本的字段单独拆分到另外一张扩展表, 将不经常修改的字段放在同一张表中，将经常改变的字段放在另一张表中。
对于海量用户场景，可以考虑取模分表，数据相对比较均匀，不容易出现热点和并发访问的瓶颈。

库内分表，仅仅是解决了单表数据过大的问题，但并没有把单表的数据分散到不同的物理机上，因此并不能减轻 MySQL 服务器的压力，仍然存在同一个物理机上的资源竞争和瓶颈，包括 CPU、内存、磁盘 IO、网络带宽等。

分库与分表带来的分布式困境与应对之策
数据迁移与扩容问题----一般做法是通过程序先读出数据，然后按照指定的分表策略再将数据写入到各个分表中。
分页与排序问题----需要在不同的分表中将数据进行排序并返回，并将不同分表返回的结果集进行汇总和再次排序，最后再返回给用户。

20、不可重复读和幻读区别是什么？可以举个例子吗？

不可重复读的重点是修改，幻读的重点在于新增或者删除。

- 例1（同样的条件, 你读取过的数据, 再次读取出来发现值不一样了 ）：事务1中的A先生读取自己的工资为 1000的操作还没完成，事务2中的B先生就修改了A的工资为2000，导致A再读自己的工资时工资变为 2000；这就是不可重复读。
- 例2（同样的条件, 第1次和第2次读出来的记录数不一样 ）：假某工资单表中工资大于3000的有4人，事务1读取了所有工资大于3000的人，共查到4条记录，这时事务2 又插入了一条工资大于3000的记录，事务1再次读取时查到的记 录就变为了5条，这样就导致了幻读。

21、MySQL中有四种索引类型，可以简单说说吗？

- **FULLTEXT**：即为全文索引，目前只有MyISAM引擎支持。其可以在CREATE TABLE ，ALTER TABLE ，CREATE INDEX 使用，不过目前只有 CHAR、VARCHAR ，TEXT 列上可以创建全文索引。
- **HASH**：由于HASH的唯一（几乎100%的唯一）及类似键值对的形式，很适合作为索引。 HASH索引可以一次定位，不需要像树形索引那样逐层查找,因此具有极高的效率。但是，这种高效是有条件的，即只在“=”和“in”条件下高效，对于范围查询、排序及组合索引仍然效率不高。
- **BTREE**：BTREE索引就是一种将索引值按一定的算法，存入一个树形的数据结构中（二叉树），每次查询都是从树的入口root开始，依次遍历node，获取leaf。这是MySQL里默认和最常用的索引类型。
- **RTREE**：RTREE在MySQL很少使用，仅支持geometry数据类型，支持该类型的存储引擎只有MyISAM、BDb、InnoDb、NDb、Archive几种。 相对于BTREE，RTREE的优势在于范围查找。

22、视图的作用是什么？可以更改吗？

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的 sql 操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by 则对视图再次order by 将被覆盖。

创建视图：create view xxx as xxxx

对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

23、为什么说B+tree比B 树更适合实际应用中操作系统的文件索引和数据库索引？

B+tree的磁盘读写代价更低，B+tree的查询效率更加稳定
数据库索引采用B+树而不是B树的主要原因：B+树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而B树只能中序遍历所有节点，效率太低。

B+树的特点

- 所有关键字都出现在叶子结点的链表中(稠密索引)，且链表中的关键字恰好是有序的;
- 不可能在非叶子结点命中;
- 非叶子结点相当于是叶子结点的索引(稀疏索引)，叶子结点相当于是存储(关键字)数据的数据层;

24、一道场景题：假如你所在的公司选择MySQL数据库作数据存储，一天五万条以上的增量，预计运维三年，你有哪些优化手段？

- 设计良好的数据库结构，允许部分数据冗余，尽量避免join查询，提高效率。
- 选择合适的表字段数据类型和存储引擎，适当的添加索引。
- MySQL库主从读写分离。
- 找规律分表，减少单表中的数据量提高查询速度。
- 添加缓存机制，比如Memcached，Apc等。
- 不经常改动的页面，生成静态页面。
- 书写高效率的SQL。比如 SELECT * FROM TABEL 改为 SELECT field_1, field_2, field_3 FROM TABLE。

25、什么时候需要建立数据库索引呢？

在最频繁使用的、用以缩小查询范围的字段,需要排序的字段上建立索引。
不宜：
1）对于查询中很少涉及的列或者重复值比较多的列
2）对于一些特殊的数据类型，不宜建立索引，比如文本字段（text）等。

26、覆盖索引是什么?

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”。

我们知道在InnoDB存储引 擎中，如果不是主键索引，叶子节点存储的是主键+列值。最终还是要“回表”，也就是要通过主键再查找一次,这样就 会比较慢。覆盖索引就是把要查询出的列和索引是对应的，不做回表操作!

27、数据库中的主键、超键、候选键、外键是什么？（很棒）

- **超键**：在关系中能唯一标识**元组的属性集**称为关系模式的超键
- **候选键**：不含有**多余属性的超键**称为候选键。也就是在候选键中，若再删除属性，就不是键了!
- **主键**：**用户选作元组标识的一个候选键程序主键**
- **外键**：如果关系模式**R中属性K是其它模式的主键**，那么**k在模式R中称为外键。**

举例：

学号	姓名	性别	年龄	系别	专业
20020612	李辉	男	20	计算机	软件开发
20060613	张明	男	18	计算机	软件开发
20060614	王小玉	女	19	物理	力学
20060615	李淑华	女	17	生物	动物学
20060616	赵静	男	21	化学	食品化学
20060617	赵静	女	20	生物	植物学

1. 超键：于是我们从例子中可以发现 学号是标识学生实体的唯一标识。那么该元组的超键就为学号。除此之外我们还可以把它跟其他属性组合起来，比如：(`学号`，`性别`)，(`学号`，`年龄`)
2. 候选键：根据例子可知，学号是一个可以唯一标识元组的唯一标识，因此学号是一个候选键，实际上，候选键是超键的子集，比如 （学号，年龄）是超键，但是它不是候选键。因为它还有了额外的属性。
3. 主键：简单的说，例子中的元组的候选键为学号，但是我们选定他作为该元组的唯一标识，那么学号就为主键。
4. 外键是相对于主键的，比如在学生记录里，主键为学号，在成绩单表中也有学号字段，因此学号为成绩单表的外键，为学生表的主键。

主键为候选键的子集，候选键为超键的子集，而外键的确定是相对于主键的。

28、数据库三大范式精讲

第一范式

在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。 所谓第一范式（1NF）是指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。

如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。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

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）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。这个惟一属性列被称为主关键字或主键、主码。 第二范式（2NF）要求实体的属性完全依赖于主关键字。

所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。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。

简而言之，**第二范式就是非主属性非部分依赖于主关键字。**

第三范式

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）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。

29、数据库三大范式精要总结

（1）简单归纳：

第一范式（1NF）：字段不可分；
第二范式（2NF）：有主键，非主键字段依赖主键；
第三范式（3NF）：非主键字段不能相互依赖。

（2）解释：

1NF：原子性。 字段不可再分,否则就不是关系数据库;；
2NF：唯一性 。一个表只说明一个事物；
3NF：每列都与主键有直接关系，不存在传递依赖。

30、MySQL常见的存储引擎InnoDB、MyISAM的区别？适用场景分别是?

1）事务：MyISAM不支持，InnoDB支持
2）锁级别： MyISAM 表级锁，InnoDB 行级锁及外键约束
3）MyISAM存储表的总行数；InnoDB不存储总行数；
4）MyISAM采用非聚集索引，B+树叶子存储指向数据文件的指针。InnoDB主键索引采用聚集索引，B+树叶子存储数据

适用场景：
MyISAM适合： 插入不频繁，查询非常频繁，如果执行大量的SELECT，MyISAM是更好的选择， 没有事务。
InnoDB适合： 可靠性要求比较高，或者要求事务； 表更新和查询都相当的频繁， 大量的INSERT或UPDATE

31、事务四大特性（ACID）原子性、一致性、隔离性、持久性?

第一种回答

原子性：一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。
。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
一致性：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
隔离性：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
持久性：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

第二种回答

原子性（Atomicity）

- 原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性（Consistency）

- 事务开始前和结束后，数据库的完整性约束没有被破坏。比如A向B转账，不可能A扣了钱，B却没收到。

隔离性（Isolation）

- 隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。
关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。　　持久性（Durability）

- 持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

32、SQL中的NOW()和CURRENT_DATE()两个函数有什么区别?

NOW（）命令用于显示当前年份，月份，日期，小时，分钟和秒。
CURRENT_DATE（）仅显示当前年份，月份和日期。

33、什么是聚合索引？

聚簇索引就是按照拼音查询，非聚簇索引就是按照偏旁等来进行查询。

其实，我们的汉语字典的正文本身就是一个聚集索引。比如，我们要查"安"字，就会很自然地翻开字典的前几页，因为"安"的拼音是"an"，而按照拼音排序 汉字的字典是以英文字母"a"开头并以"z"结尾的，那么"安"字就自然地排在字典的前部。如果您翻完了所有以"a"开头的部分仍然找不到这个字，那么就 说明您的字典中没有这个字；同样的，如果查"张"字，那您也会将您的字典翻到最后部分，因为"张"的拼音

是"zhang"。也就是说，字典的正文部分本身 就是一个目录，您不需要再去查其他目录来找到您需要找的内容。

我们把这种**正文内容本身就是一种按照一定规则排列的目录称为"聚集索引"**

34、什么是非聚合索引?

如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而 需要去根据"偏旁部首"查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结合"部首目录"和"检字表"而查到的字的排序并不是 真正的正文的排序方法，比如您查"张"字，我们可以看到在查部首之后的检字表中"张"的页码是672页，检字表中"张"的上面是"驰"字，但页码却是63 页，"张"的下面是"弩"字，页面是390页。很显然，这些字并不是真正的分别位于"张"字的上下方，现在您看到的连续的"驰、张、弩"三字实际上就是他 们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后 再翻到您所需要的页码。

我们把**这种目录纯粹是目录，正文纯粹是正文的排序方式称为"非聚集索引"。**

35、聚集索引与非聚集索引的区别是什么?

非聚集索引和聚集索引的区别在于， 通过聚集索引可以查到需要查找的数据， 而通过非聚集索引可以查到记录对应的主键值 ， 再使用主键的值通过聚集索引查找到需要的数据。
聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致。

聚集索引（Innodb）的叶节点就是数据节点，而非聚集索引(MyisAM)的叶节点仍然是索引节点，只不过其包含一个指向对应数据块的指针。

36、创建索引时需要注意什么?

非空字段：应该指定列为NOT NULL，除非你想存储NULL。在 MySQL 中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；

取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；

索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。
唯一、不为空、经常被查询的字段 的字段适合建索引

37、MySQL中CHAR和VARCHAR的区别有哪些?

- char的长度是不可变的，用空格填充到指定长度大小，而varchar的长度是可变的。
- char的存取数度还是要比varchar要快得多
- char的存储方式是：对英文字符（ASCII）占用1个字节，对一个汉字占用两个字节。varchar的存储方式是：对每个英文字符占用2个字节，汉字也占用2个字节。

38、MySQL 索引使用的注意事项

MySQL 索引通常是被用于提高 WHERE 条件的数据行匹配时的搜索速度，在索引的使用过程中，存在一些使用细节和注意事项。

函数，运算，否定操作符，连接条件，多个单列索引，最左前缀原则，范围查询，不会包含有NULL值的列，like 语句不要在列上使用函数和进行运算

1）不要在列上使用函数，这将导致索引失效而进行全表扫描。

```
select * from news where year(publish_time) < 2017
```

为了使用索引，防止执行全表扫描，可以进行改造。

```
select * from news where publish_time < '2017-01-01'
```

还有一个建议，不要在列上进行运算，这也将导致索引失效而进行全表扫描。

```
select * from news where id / 100 = 1
```

为了使用索引，防止执行全表扫描，可以进行改造。

```
select * from news where id = 1 * 100
```

2）尽量避免使用 != 或 not in或 <> 等否定操作符
应该尽量避免在 where 子句中使用 != 或 not in 或 <> 操作符，因为这几个操作符都会导致索引失效而进行全表扫描。尽量避免使用 or 来连接条件
应该尽量避免在 where 子句中使用 or 来连接条件，因为这会导致索引失效而进行全表扫描。

```
select * from news where id = 1 or id = 2
```

3）多个单列索引并不是最佳选择
MySQL 只能使用一个索引，会从多个索引中选择一个限制最为严格的索引，因此，为多个列创建单列索引，并不能提高 MySQL 的查询性能。
假设，有两个单列索引，分别为 news_year_idx(news_year) 和 news_month_idx(news_month)。现在，有一个场景需要针对资讯的年份和月份进行查询，那么，SQL 语句可以写成：

```
select * from news where news_year = 2017 and news_month = 1
```

事实上，MySQL 只能使用一个单列索引。为了提高性能，可以使用复合索引 news_year_month_idx(news_year, news_month) 保证 news_year 和 news_month 两个列都被索引覆盖。

4）复合索引的最左前缀原则
复合索引遵守“最左前缀”原则，即在查询条件中使用了复合索引的第一个字段，索引才会被使用。因此，在复合索引中索引列的顺序至关重要。如果不是按照索引的最左列开始查找，则无法使用索引。
假设，有一个场景只需要针对资讯的月份进行查询，那么，SQL 语句可以写成：

```
select * from news where news_month = 1
```

此时，无法使用 news_year_month_idx(news_year, news_month) 索引，因为遵守“最左前缀”原则，在查询条件中没有使用复合索引的第一个字段，索引是不会被使用的。

5）覆盖索引的好处

如果一个索引包含所有需要的查询的字段的值，直接根据索引的查询结果返回数据，而无需读表，能够极大的提高性能。因此，可以定义一个让索引包含的额外的列，即使这个列对于索引而言是无用的。

6）范围查询对多列查询的影响

查询中的某个列有范围查询，则其右边所有列都无法使用索引优化查找。

举个例子，假设有一个场景需要查询本周发布的资讯文章，其中的条件是必须是启用状态，且发布时间在这周内。那么，SQL 语句可以写成：

```
select * from news where publish_time >= '2017-01-02' and publish_time <= '2017-01-08' and enable = 1
```

这种情况下，因为范围查询对多列查询的影响，将导致 news_publish_idx(publish_time, enable) 索引中 publish_time 右边所有列都无法使用索引优化查找。换句话说，news_publish_idx(publish_time, enable) 索引等价于 news_publish_idx(publish_time) 。

对于这种情况，我的建议：对于范围查询，务必要注意它带来的副作用，并且尽量少用范围查询，可以通过曲线救国的方式满足业务场景。

例如，上面案例的需求是查询本周发布的资讯文章，因此可以创建一个news_weekth 字段用来存储资讯文章的周信息，使得范围查询变成普通的查询，SQL 可以改写成：

```
select * from news where news_weekth = 1 and enable = 1

```

然而，并不是所有的范围查询都可以进行改造，对于必须使用范围查询但无法改造的情况，我的建议：不必试图用 SQL 来解决所有问题，可以使用其他数据存储技术控制时间轴，例如 Redis 的 SortedSet 有序集合保存时间，或者通过缓存方式缓存查询结果从而提高性能。

7）索引不会包含有NULL值的列

只要列中包含有 NULL 值都将不会被包含在索引中，复合索引中只要有一列含有 NULL值，那么这一列对于此复合索引就是无效的。

因此，在数据库设计时，除非有一个很特别的原因使用 NULL 值，不然尽量不要让字段的默认值为 NULL。

8）隐式转换的影响

当查询条件左右两侧类型不匹配的时候会发生隐式转换，隐式转换带来的影响就是可能导致索引失效而进行全表扫描。下面的案例中，date_str 是字符串，然而匹配的是整数类型，从而发生隐式转换。

```
select * from news where date_str = 201701

```

因此，要谨记隐式转换的危害，时刻注意通过同类型进行比较。

9）like 语句的索引失效问题

like 的方式进行查询，在 like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。所以，根据业务需求，考虑使用 ElasticSearch 或 Solr 是个不错的方案。

39、MySQL中有哪些索引？有什么特点?

- **普通索引**：仅加速查询
- **唯一索引**：加速查询 + 列值唯一（可以有null）
- **主键索引**：加速查询 + 列值唯一（不可以有null）+ 表中只有一个
- **组合索引**：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并
- **全文索引**：对文本的内容进行分词，进行搜索
- **索引合并**：使用多个单列索引组合搜索
- **覆盖索引**：select的数据列只用从索引中就能够取得，不必读取数据行，换句话说查询列要被所建的索引覆盖
- **聚簇索引**：表数据是和主键一起存储的，主键索引的叶结点存储行数据(包含了主键值)，二级索引的叶结点存储行的主键值。使用的是B+树作为索引的存储结构，非叶子节点都是索引关键字，但非叶子节点中的关键字中不存储对应记录的具体内容或内容地址。叶子节点上的数据是主键与具体记录(数据内容)

40、既然索引有那么多优点，为什么不对表总的每一列创建一个索引呢?

- 当对表中的数据进行增加、删除和修改的时候，**索引也要动态的维护**，这样就降低了数据的维护速度。
- **索引需要占物理空间**，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立簇索引，那么需要的空间就会更大。
- **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加

41、索引如何提高查询速度的

将无序的数据变成相对有序的数据（就像查有目的一样）

42、使用索引的注意事项

- 在经常需要搜索的列上，可以加快搜索的速度；
- 在经常使用在where子句中的列上面创建索引，加快条件的判断速度。
- **将打算加索引的列设置为NOT NULL，否则将导致引擎放弃使用索引而进行全表扫描**
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间
- 避免where子句中对字段施加函数，这会造成无法命中索引
- 在中到大型表索引都是非常有效的，但是特大型表的维护开销会很大，不适合建索引，建立用逻辑索引
- 在经常用到连续的列上，这些列主要是由一些外键，可以加快连接的速度
- 与业务无关时多使用逻辑主键，也就是自增主键在使用InnoDB时使用与业务无关的自增主键作为主键，即使用逻辑主键，而不要使用业务主键。
- 删除长期未使用的索引，不用的索引的存在会造成不必要的性能损耗
- 在使用limit offset查询缓存时，可以借助索引来提高性能。

43、增加B+树的路数可以降低树的高度，那么无限增加树的路数是不是可以有最优的查找效率?

不可以。因为这样会形成一个有序数组，文件系统和数据库的索引都是存在硬盘上的，并且如果数据量大的话，不一定能一次性加载到内存中。有序数组没法一次性加载进内存，这时候B+树的多路存储威力就出来了，可以每次加载B+树的一个结点，然后一步步往下找，

44、说一下数据库表锁和行锁吧

表锁

不会出现死锁，发生锁冲突几率高，并发低。

MyISAM在执行查询语句（select）前，会自动给涉及的所有表加读锁，在执行增删改操作前，会自动给涉及的表加写锁。

MySQL的表级锁有两种模式：表共享读锁和表独占写锁。

读锁会阻塞写，写锁会阻塞读和写

- 对MyISAM表的读操作，不会阻塞其它进程对同一表的读请求，但会阻塞对同一表的写请求。只有当读锁释放后，才会执行其它进程的写操作。
- 对MyISAM表的写操作，会阻塞其它进程对同一表的读和写操作，只有当写锁释放后，才会执行其它进程的读写操作。

MyISAM不适合做写为主表的引擎，因为写锁后，其它线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永远阻塞。

行锁

会出现死锁，发生锁冲突几率低，并发高。

在MySQL的InnoDB引擎支持行锁，与Oracle不同，MySQL的行锁是通过索引加载的，也就是说，行锁是加在索引响应的行上的，要是对应的SQL语句没有走索引，则会全表扫描，行锁则无法实现，取而代之的是表锁，此时其它事务无法对当前表进行更新或插入操作。

行锁的实现需要注意：

- 行锁必须有索引才能实现，否则会自动锁全表，那么就不是行锁了。
- 两个事务不能锁同一个索引。
- insert，delete，update在事务中都会自动默认加上排它锁。

行锁的适用场景：

A用户消费，service层先查询该用户的账户余额，若余额足够，则进行后续的扣款操作；这种情况查询的时候应该对该记录进行加锁。

否则，B用户在A用户查询后消费前先一步将A用户账号上的钱转走，而此时A用户已经进行了用户余额是否足够的判断，则可能会出现余额已经不足但却扣款成功的情况。

为了避免此情况，需要在A用户操作该记录的时候进行for update加锁

45、SQL语法中内连接、自连接、外连接（左、右、全）、交叉连接的区别分别是什么?

内连接：只有两个元素表相匹配的才能在结果集中显示。
外连接： 左外连接: 左边为驱动表，驱动表的数据全部显示，匹配表的不匹配的不会显示。
右外连接:右边为驱动表，驱动表的数据全部显示，匹配表的不匹配的不会显示。
全外连接：连接的表中不匹配的数据全部会显示出来。
交叉连接： 笛卡尔效应，显示的结果是链接表数的乘积。

46、你知道哪些数据库结构优化的手段?

- **范式优化**： 比如消除冗余（节省空间。。）
- **反范式优化**：比如适当加冗余等（减少join）
- **限定数据的范围**： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。
- **读/写分离**： 经典的数据库拆分方案，主库负责写，从库负责读；
- **拆分表**：分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘I/O，一个精心设置的分区可以将数据传输对磁盘I/O竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。

47、数据库优化中有一个比较常用的手段就是把数据表进行拆分，关于拆分数据表你了解哪些?

拆分其实又分**垂直拆分**和**水平拆分**

案例： 简单购物系统暂设涉及如下表：

1.产品表（数据量10w，稳定）

2.订单表（数据量200w，且有增长趋势）

3.用户表 （数据量100w，且有增长趋势）

以 MySQL 为例讲述下水平拆分和垂直拆分，MySQL能容忍的数量级在百万静态数据可以到千万

垂直拆分

解决问题：表与表之间的io竞争

不解决问题：单表中数据量增长出现的压力

方案： 把产品表和用户表放到一个server上 订单表单独放到一个server上

水平拆分

解决问题：单表中数据量增长出现的压力

不解决问题：表与表之间的io争夺

方案：**用户表** 通过性别拆分为男用户表和女用户表，**订单表** 通过已完成和完成中拆分为已完成订单和未完成订单，**产品表** 未完成订单放一个server上，已完成订单表盒男用户表放一个server上，女用户表放一个server上(女的爱购物 哈哈)。

48、为什么MySQL索引要使用B+树，而不是B树或者红黑树?

我们在MySQL中的数据一般是放在磁盘中的，读取数据的时候肯定会有访问磁盘的操作，磁盘中有两个机械运动的部分，分别是盘片旋转和磁臂移动。盘片旋转就是我们市面上所提到的多少转每分钟，而磁盘移动则是在盘片旋转到指定位置以后，移动磁臂后开始进行数据的读写。那么这就存在一个定位到磁盘中的块的过程，而定位是磁盘的存取中花费时间比较大的一块，毕竟机械运动花费的时候要远远大于电子运动的时间。当大规模数据存储到磁盘中的时候，显然定位是一个非常花费时间的过程，但是我们可以通过B树进行优化，提高磁盘读取时定位的效率。

为什么B类树可以进行优化呢？我们可以根据B类树的特点，构造一个多阶的B类树，然后在尽量多的在结点上存储相关的信息，**保证层数（树的高度）尽量的少**，以便后面我们可以更快的找到信息，**磁盘的I/O操作也少一些**，而且B类树是平衡树，每个结点到叶子结点的高度都是相同，这也保证了每个查询是稳定的。

特别地：**只有B-树和B+树，这里的B-树是叫B树，不是B减树，没有B减树的说法。**

49、为什么MySQL索引适用用B+树而不用hash表和B树?

- 利用Hash需要把数据全部**加载到内存中**，如果数据量大，是一件很**消耗内存**的事，而采用B+树，是基于**按照节点分段加载，由此减少内存消耗。**
- 和业务场景有段，**对于唯一查找**（查找一个值），Hash确实更快，**但数据库中经常查询多条数据**，这时候由于B+数据的有序性，与叶子节点又有链表相连，他的查询效率会比Hash快的多。
- b+树的**非叶子节点不保存数据**，**只保存子树的临界值**（最大或者最小），所以同样大小的节点，**b+树相对于b树能够有更多的分支，使得这棵树更加矮胖，查询时做的IO操作次数也更少。**

50、既然Hash比B+树更快，为什么MySQL用B+树来存储索引呢?

MySQL中存储索引用到的数据结构是B+树，B+树的查询时间跟树的高度有关，是log(n)，如果用hash存储，那么查询时间是O(1)。

采用Hash来存储确实要更快，但是采用B+树来存储索引的原因主要有以下两点：

一、**从内存角度上说**，数据库中的索引一般是在磁盘上，数据量大的情况可能无法一次性装入内存，B+树的设计可以允许数据分批加载。

二、**从业务场景上说**，如果只选择一个数据那确实是hash更快，但是数据库中经常会选中多条，这时候由于B+树索引有序，并且又有链表相连，它的查询效率比hash就快很多了。

51、关系型数据库的四大特性在得不到保障的情况下会怎样?

ACID，原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)

我们以从A账户转账50元到B账户为例进行说明一下ACID这四大特性。

原子性

原子性是指一个事务是一个不可分割的工作单位，**其中的操作要么都做，要么都不做。**即要么转账成功，要么转账失败，是不存在中间的状态！

如果无法保证原子性会怎么样?

OK，就会出现数据不一致的情形，A账户减去50元，而B账户增加50元操作失败。系统将无故丢失50元~

一致性

一致性是指事务执行前后，数据处于一种合法的状态，这种状态是语义上的而不是语法上的。 那什么是合法的数据状态呢？这个状态是满足预定的约束就叫做合法的状态，再通俗一点，这状态是由你自己来定义的。**满足这个状态，数据就是一致的，不满足这个状态，数据就是不一致的！**

如果无法保证一致性会怎么样?

- 例一:A账户有200元，转账300元出去，此时A账户余额为-100元。你自然就发现了此时数据是不一致的，为什么呢？因为你定义了一个状态，余额这列必须大于0。
- 例二:A账户200元，转账50元给B账户，A账户的钱扣了，但是B账户因为各种意外，余额并没有增加。你也知道此时数据是不一致的，为什么呢？因为你定义了一个状态，要求A+B的余额必须不变。

隔离性

隔离性是指**多个事务并发执行的时候，事务内部的操作与其他事务是隔离的**，并发执行的各个事务之间不能互相干扰。

如果无法保证隔离性会怎么样?

假设A账户有200元，B账户0元。A账户往B账户转账两次，金额为50元，分别在两个事务中执行。如果无法保证隔离性，A可能就会出现扣款两次的情形，而B只加款一次，凭空消失了50元，依然出现了数据不一致的情形！

持久性

根据定义，**持久性是指事务一旦提交，它对数据库的改变就应该是永久性的。**接下来的其他操作或故障不应该对其有任何影响。

如果无法保证持久性会怎么样?

在MySQL中，为了解决CPU和磁盘速度不一致问题，MySQL是将磁盘上的数据加载到内存，对内存进行操作，然后再回写磁盘。好，假设此时宕机了，在内存中修改的数据全部丢失了，持久性就无法保证。

设想一下，系统提示你转账成功。但是你发现金额没有发生任何改变，此时数据出现了不合法的数据状态，我们将这种状态认为是**数据不一致**的情形。

52、数据库如何保证一致性?

分为两个层面来说。

- **从数据库层面**，数据库通过原子性、隔离性、持久性来保证一致性。也就是说ACID四大特性之中，C(一致性)是目的，A(原子性)、I(隔离性)、D(持久性)是手段，是为了保证一致性，数据库提供的手段。**数据库必须要实现AID三大特性，才有可能实现一致性。**例如，原子性无法保证，显然一致性也无法保证。
- **从应用层面**，通过代码判断数据库数据是否有效，然后决定回滚还是提交数据！

53、数据库如何保证原子性?

主要是利用 Innodb 的**undo log。 undo log**名为回滚日志，是实现原子性的关键，当事务回滚时能够撤销所有已经成功执行的 SQL语句，他需要记录你要回滚的相应日志信息。 例如

- 当你delete一条数据的时候，就需要记录这条数据的信息，回滚的时候，insert这条旧数据
- 当你update一条数据的时候，就需要记录之前的旧值，回滚的时候，根据旧值执行update操作
- 当年insert一条数据的时候，就需要这条记录的主键，回滚的时候，根据主键执行delete操作

undo log记录了这些回滚需要的信息，当事务执行失败或调用了**rollback**，导致事务需要回滚，便可以利用**undo log**中的信息将数据回滚到修改之前的样子。

54、数据库如何保证持久性?

主要是利用Innodb的**redo log**。重写日志， 正如之前说的，MySQL是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再写回到磁盘上。如果此时突然宕机，内存中的数据就会丢失。 怎么解决这个问题？ 简单啊，事务提交前直接把数据写入磁盘就行啊。 这么做有什么问题?

- 只修改一个页面里的一个字节，就要将整个页面刷入磁盘，太浪费资源了。毕竟一个页面16kb大小，你只改其中一点点东西，就要将16kb的内容刷入磁盘，听着也不合理。
- 毕竟一个事务里的SQL可能牵涉到多个数据页的修改，而这些数据页可能不是相邻的，也就是属于随机IO。显然操作随机IO，速度会比较慢。

于是，决定采用**redo log**解决上面的问题。当做数据修改的时候，不仅在内存中操作，还会在**redo log**中记录这次操作。当事务提交的时候，会将**redo log**日志进行刷盘(**redo log**一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据**undo log**和**binlog**内容决定回滚数据还是提交数据。

采用redo log的好处?

其实好处就是将**redo log**进行刷盘比对数据页刷盘效率高，具体表现如下:

- **redo log**体积小，毕竟只记录了哪一页修改了啥，因此体积小，刷盘快。
- **redo log**是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快。

55、数据库高并发是我们经常会遇到的，你有什么好的解决方案吗?

- 在web服务框架中加入缓存。在服务器与数据库层之间加入缓存层，将高频访问的数据存入缓存中，减少数据库的读取负担。
- 增加数据库索引，进而提高查询速度。（不过索引太多会导致速度变慢，并且数据库的写入会导致索引的更新，也会导致速度变慢）
- 主从读写分离，让主服务器负责写，从服务器负责读。
- 将数据库进行拆分，使得数据库的表尽可能小，提高查询的速度。
- 使用分布式架构，分散计算压力。

参考文献

> 《高性能MySQL》：https://item.jd.com/11220393.html
>
> 《 MySQL是怎样运行的 从根儿上理解MySQL》:https://item.jd.com/13009316.html
>
> 《 MySQL技术内幕：InnoDB存储引擎（第2版）》:https://item.jd.com/11252326.html
>
> 极客时间专栏-《MySQL实战45讲》：https://time.geekbang.org/column/intro/100020801
>
> https://blog.csdn.net/BEYOA/article/details/115829327
>
> https://segmentfault.com/a/119000003984710
>
> https://blog.csdn.net/FL63Zv96950w/article/details/11577443
>
> https://segmentfault.com/a/1190000039848
>
> https://blog.csdn.net/wypblog/article/details/1158432
>
> https://segmentfault.com/q/101000003971

> https://blog.csdn.net/wei6569/article/details/11585679
>
> https://blog.csdn.net/dog250/article/details/115783
>
> https://segmentfault.com/q/101000421003971
>
> https://blog.csdn.net/prograer_editor/article/details/11572561
>
> https://segmentfault.com/q/10100004134471
>
> https://csdnnews.blog.csdn.net/article/details/11574389
>
> https://segmentfault.com/q/101000714155354