1、TPS(每秒传输的事物处理个数)= (COM_COMMIT + COM_ROLLBACK)/UPTIME，即提交的事务数与回滚的事务数之和除以时间就是每秒事务处理个数。

2、QPS(每秒查询数处理)=QUESTIONS/UPTIME，即查询数除以时间就是每秒处理查询个数。

3、mysql安装目录下的my.ini文件中可以设置mysql系统环境变量参数。设置开启Ferderated引擎就可以加参数 ferderated=1（1代表是，0代表否，所有参数如此）。

3、mysql缓存。sql缓存默认开启，数据缓存需要手动开启。 show variables like '%query_cache_type%' 语句查询缓存是否开启。注意数据缓存开启以后还需要设置缓存大小，缓存太小的话相当于没有缓存。 show variables like '%query_cache_size%' 查询缓存大小。

4、mysql缓存生产环境一般不开启，因为开启缓存后对mysql压力很大。我们可以选择redis等其他缓存策略。

5、Mysql优化器：

6、MyISAM 存储引擎由MYD和MYI组成。MYD文件存储数据(data)，MYI文件存储索引(index)。非聚集索引（因为数据和索引是分文件放的）。MyISAM压缩表空间之后不能进行删除和插入操作，但是可以查询。可以恢复被压缩的表空间。Innodb是聚集索引。

7、MyISAM适合只读类和空间类应用（空间函数，坐标），它不支持事务，但是查询比较快。

8、独立表空间可以用OPTIMIZE TABLE product_info2语句收缩 （收缩表空间），而系统表空间无法收缩。当进行删除操作以后数据文件大小并没有变，使用上边语句收缩之后文件大小才会变，相当于磁盘整理。

9、MyISAM关注的是性能，innodb关注的是事务。

10、show VARIABLES语句可以查看mysql的系统环境变量及其值，比如show VARIABLES like '%datadir%' 查看数据库路径。设置环境变量set global slow_query_log = 1;

11、Memory引擎数据存在内存中，容易丢失，所以要求数据可再生。

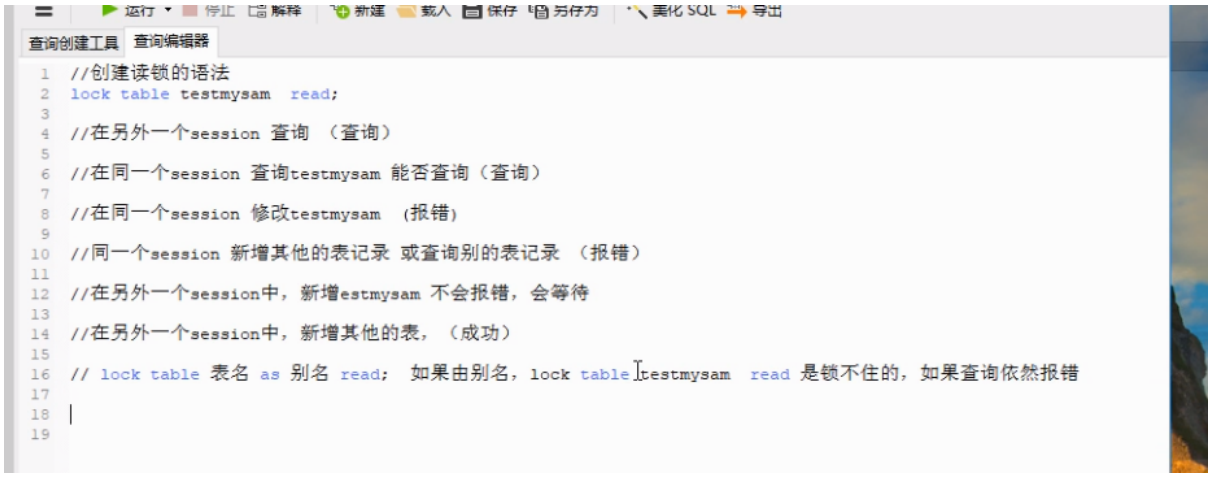
12、Memory存储引擎表和临时表的区别：

- Memory引擎数据存储在内存中，重启数据库服务器后数据就会丢失查不到了。
- 临时表是针对一次会话来说，新建的临时表只在当前会话有效，其他会话查不到。

13、Ferderated存储引擎表远程和本地的表结构必须一样。本地相当于一个代理。

14、MyISAM表级锁的两种模式（1、表共享读锁；2、表独占写锁）：

- 读锁语法



15、事务隔离级别及影响：

事务隔离级别的影响

事务隔离级别	脏读	不可重复读	幻读
读未提交 | read uncommitted	会	会	会
读已提交 | read committed	不会	会	会
可重复读 | repeatable read	不会	不会	会
串行化 | serializable	不会	不会	不会

16、何为脏读、不可重复读、幻读：

脏读、幻读、不可重读的区别

现在，我们是不是更容易把脏读、幻读、不可重读混淆了呢?

我们一起来总结一下它们之前的区别：

脏读：当前事务可以查看到别的事务未提交的数据（侧重点在于别的事务未提交）。

幻读：幻读的表象与不可重读的表象都让人"憋逼"，很容易搞混，但是如果非要细分的话，幻读的侧重点在于新增和删除。表示在同一事务中，使用相同的查询语句，第二次查询时，莫名的多出了一些之前不存在数据，或者莫名的不见了一些数据。

不可重读：不可重读的侧重点在于更新修改数据。表示在同一事务中，查询相同的数据范围时，同一个数据资源莫名的改变了。

所谓不可重复读就是在同一个事务，对同一表数据的读取，多次查询数据结果并不一样。

17、隔离级别（可重复读）：

隔离级别：可重读

我们先来总结一下可重读隔离级别的特性，仍然以刚才文章开头的示例为例，下图中，再回话1与回话2中同时开启两个事务，在事务1的事务中修改了t1表的数据以后（将第二条数据的t1str的值修改为test），事务2中查看到的数据仍然是事务1修改之前的数据，即使事务1提交了，在事务2没有提交之前，事务2中查看到的数据都是相同的，比如t1表中的第2条数据，不管事务1是否提交，在事务2没有提交之前，这条数据对于事务2来说一直都是没有发生改变的，这条数据在事务2中是可以重复的被读到，所以，这种隔离级别被称为"可重读"。

18、读已提交和可重复读的区别是。当前session开启事务后读取被其他事务修改的数据结果是不一样的。假如表中数据amt=500，开启A、B两个事务，A事务将执行amt-50操作。此时B事务读取amt的值，当A事务没提交之前两种隔离级别下B读取到的都是500，但当A事务提交以后，读已提交情况下B事务读取到的是450，而可重复读情况下B事务读取到的是500。

19、串行化一次只能一个人操作表(所有操作都会锁表)。

19、解决幻读要么从jdk锁层面解决。要么数据库事务隔离级别用串行化（但是这种方式的话数据库效率太低，不推荐）。

19、当索引失效时行锁会升级成表锁。（此处有疑问需要后续注意）

20、完全符合范式化的设计有时并不能得到良好得SQL查询性能。

21、反范式化允许数据存在少量冗余，思想就是以空间换时间。

22、datatime（8字节）与时区无关，而timestamp（4字节）与时区有关。中国在东八区。

23、慢查询日志不止记录查询记录，insert语句也会记录。只要执行时间超过设定阈值的都会记录。

24、mysql全局变量和会话变量，当set global的时候发现当前会话中并没有生效：

```
mysql> set global wait_timeout=10;

mysql> show global variables like 'wait_timeout';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| wait_timeout              | 10    |
+---------------------------+-------+
```

这里一个容易把人搞蒙的地方是如果查询时使用的是show variables的话，会发现设置好像并没有生效，这是因为单纯使用show variables的话就等同于使用的是show session variables，查询的是会话变量，只有使用show global variables，查询的才是全局变量。

网络上很多人都抱怨说他们set global之后使用show variables查询没有发现改变，原因就在于混淆了会话变量和全局变量，如果仅仅想修改会话变量的话，可以使用类似set wait_timeout=10;或者set session wait_timeout=10;这样的语法。

截图(Alt + A)

25、执行计划个属性含义：

1 简要说明

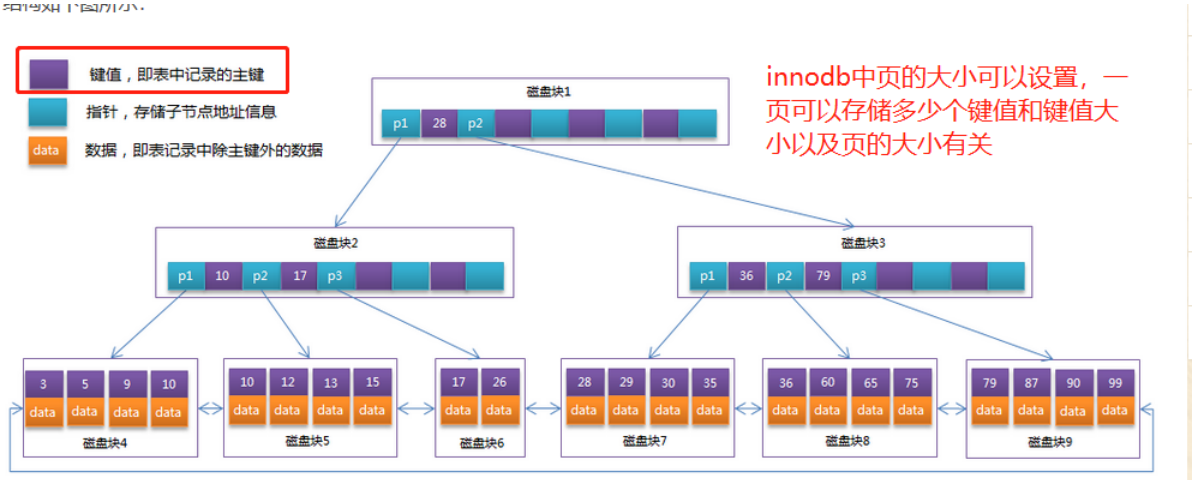
```
+----+-------------+---------+------------+-------+---------------+---------------+---------+-------------------------------------+-------+----------+-----------------------------------------------------+
| id | select_type | table   | partitions | type  | possible_keys | key           | key_len | ref                                 | rows  | filtered | Extra                                               |
+----+-------------+---------+------------+-------+---------------+---------------+---------+-------------------------------------+-------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | Extent1 | NULL       | range | IX_CreatedDate | IX_CreatedDate | 6       | NULL                                | 66074 |     5.00 | Using index condition; Using where; Using filesort |
|  1 | SIMPLE      | Extent3 | NULL       | eq_ref | PRIMARY       | PRIMARY       | 4       | ycf_stock_online.Extent1.ProviderId |     1 |   100.00 | NULL                                                |
|  1 | SIMPLE      | Extent4 | NULL       | eq_ref | PRIMARY       | PRIMARY       | 4       | ycf_stock_online.Extent1.ProductId  |     1 |   100.00 | NULL                                                |
|  1 | SIMPLE      | Extent2 | NULL       | eq_ref | PRIMARY       | PRIMARY       | 4       | ycf_stock_online.Extent1.ItemId     |     1 |   100.00 | NULL                                                |
+----+-------------+---------+------------+-------+---------------+---------------+---------+-------------------------------------+-------+----------+-----------------------------------------------------+
```

id	表格查询的顺序编号。	降序查看，id相同的从上到下查查看。 id可以为null，当table为(union ,m,n)类型的时候，id为null，这个时候，id的顺序为 m跟n的后面。
select_type	查询的方式	下文详细说明。
table	表格名称	表名，别名，(union m,n) 。
partitions	分区名称	查询使用到表分区的分区名。
type	表连接的类型	下文详细说明。
possible_keys	可能使用到的索引	这里的索引只是可能会有到，实际不一定会用到。
key	使用到的索引	实际使用的索引。
key_len	使用到索引的长度	比如多列索引，只用到最左的一列，那么使用到索引的长度则为该列的长度，故该值不一定等于 key 列索引的长度。
ref	谓词的关联信息	当 join type 为 const、eq_ref 或者 ref 时，谓词的关联信息。 可能为：null（非 const \ eq_ref \ ref join type 时）、const（常量）、关联的谓词列名。
rows	扫描的行数	该表格扫描到的行数。这里注意在mysql里边是嵌套链接，所以，需要把所有rows相乘就会得到查询数据行关联的次数
filtered	实际显示行数占扫描rows的比例	实际显示的行数 = rows * filtered / 100
extra	特性使用	

26、B+Tree动态演示

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html;

27、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，B+Tree非叶子节点只存储键值(即主键的值)：



28、组合索引最左原则：

组合索引规则：
1、最左原则：索引是key index (a,b,c). 可以支持a | a,b| a,b,c 3种组合进行查找，但不支持 b,c进行查找 .当最左侧字段是常量引用时，索引就十分有效。（电话簿中利用姓名查找人，姓和名分别是不同的列，知道姓电话簿有用，知道姓知道名电话簿有用，知道名不知道姓电话簿无用）

29、聚合索引与非聚合索引是一种存储方式，而不是一种单独的索引类型。(https://blog.csdn.net/qq_32679835/article/details/94166747)

30、InnoDB引擎表是基于B+树的索引组织表(IOT)，一般采用自增序列做主键是因为减少页的分裂，每次新增数据的时候直接在已有页后追加，追加满了就新增页：

> 4、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
>
> 5、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，此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、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来重建表并优化填充页面。

31、对于mysql索引的一些疑惑（https://ask.csdn.net/questions/774069）；

32、InnoDB 引擎下主键索引（聚簇索引）、非聚集索引（二级索引）、覆盖索引的实现原理。（https://blog.csdn.net/lixiangda/article/details/88959300）：

- **聚簇索引：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。主键索引的数据结构是一颗“B+树”，树的子节点存储索引节点信息及关联关系，树的叶子节点存储一行数据本身，一颗树的叶子节点按照主键索引有序排列开来，查找起来更便捷。**
- **非聚集索引：我们常用的普通索引或者复合索引。非聚集索引存储的具体的索引字段信息，而叶子节点存储主键值，查找数据时，需要先在非聚集索引中找到对应的主键，再根据主键索引，查找对应的行数据，等于走了两遍B+树进行搜索，效率上肯定比主键索引低。**
- **覆盖索引：查询的列中能全部匹配辅助索引关键字，则直接返回对应辅助索引的数据，那么此时就不用在根据辅助索引查找主键然后在聚集索引中再查找一遍，等于覆盖索引能减少IO操作，所以我们在某些特殊场景下，如果只查索引列的字段，建议可以使用覆盖索引，能大幅度提供查询性能。但是也不能将全表字段都设置为索引，这样索引表和数据表就一样大了，反而会造成性能下降。所以将经常查询的字段设置为索引是最佳的。**

33、innodb表一般采用自增序列做主键是为了生成聚簇索引，当我们使用非聚簇索引查询数据的时候mysql会通过两次B+Tree搜索到我们需要的值。第一次根据辅助索引查询到主键值，然后依据主键通过主索引（聚簇索引）查询所需数据。

34、innodb无论设置唯一主键还是设置复合主键，都需采用一个自增列，这样方便innodb生成聚簇索引（主索引），当我们使用联合主键的时候第一列应该是自增列，因为联合主键生成主索引时先按照第一列进行排序，然后在第一列排好序的基础上再对第二列排序。避免使用UUID。

34、UUID和自增序列的优缺点

（https://blog.csdn.net/rocling/article/details/83116950）：

以默认的innodb存储引擎为例：

做为主键时，uuid和自增相比较，自增更适合。

原因：

1 uuid是无序的， 插入数据时，页的位置会发生变化，页分裂，**速度慢。**

2 uuid占的空间大，并且innodb中，别的索引还都要包含主键的值，那么每个索引的空间也都会增大，占的空间大，需要读数据时一般会认为需要的**io次数多。**

35、为什么执行计划中type属性值排序为

（system>const>eq_ref>ref>range>index>ALL）：

- **eq_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描。本质通过索引树（B+Tree）搜索，一个索引下边只挂载一行数据。**
- **ref：非唯一性索引扫描，它可能会找到多个符合条件的行，然后扫描这些行匹配某个单独行的值，所以它属于查找和扫描的混合体。本质通过索引树（B+Tree）搜索，一个索引下边挂载多行数据。**
- **index：它并没有通过索引树（B+Tree）搜索。它是全表扫描，只不过扫描的是索引库，而没有去扫描数据库。**
- **all：不通过索引树（B+Tree）搜索。全表扫描数据表。**

35、复合索引须遵循最左前缀原则的底层原理：

最左前缀原理

假设联合索引是state/city/zipCode

那么state就是第一关，city是第二关，zipCode就是第三关

必须匹配了第一关，才能匹配第二关，匹配了第一关和第二关，才能匹配第三关

你不能直接到第二关的

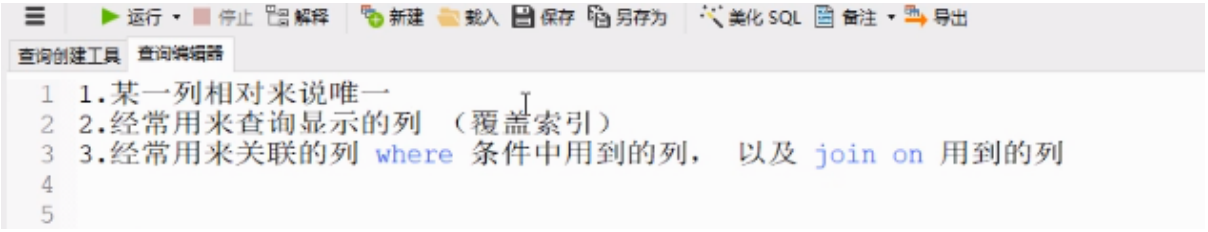
索引的格式就是第一层是state，第二层才是city

多列索引是先按照第一列进行排序，然后在第一列排好序的基础上再对第二列排序，如果没有第一列的话，直接访问第二列，那第二列肯定是无序的，直接访问后面的列就用不到索引了

36、无论采用单主键还是复合主键，只要有自增列，则生成聚簇索引（主索引）时都不会有页的分裂，直接往后追加页就行了。当我们用辅助索引查询的时候查询出的主键第一列也是自增的，这样第二次在搜索主索引的时候也很快。

37、MyISAM 和 InnoDB 索引的区别

38、适合建索引的列（https://www.cnblogs.com/chenhaoyu/p/8761305.html）：



39、mysql查询优化器。假如复合索引为（a，b），当查询条件为where b = '' and a = ''时，看似不符合最左原则，但是mysql会自动优化成where a = '' and b = ''，然后通过索引进行查找。

40、第四节课程的家庭作业需百度了解一下。

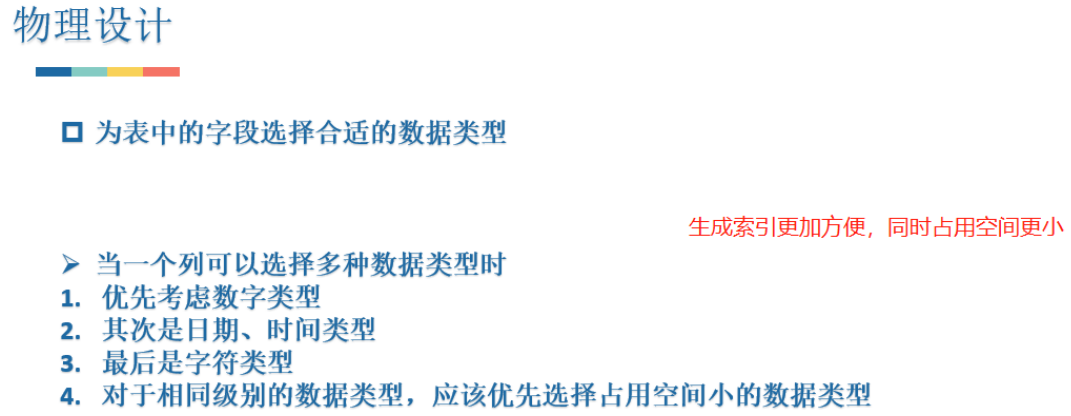
41、行锁必须有索引才能实现

（https://blog.csdn.net/djrm11/article/details/96499817）：

https://blog.csdn.net/silyvin/article/details/79332879

那么答案也出来了，InnoDB之所以可以锁行，是因为Innodb的主索引结构上，既存储了主键值，又直接存储了行数据，可以方便的锁住行数据，而Mylsam索引指向另一片数据文件，没有办法精确锁住数据段

42、物理设计表规则：



43、key_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的（假如当定义为长度为10 的char类型的字段为索引而实际存储值为"ljq"占3*3=9个字节【gbk编码一个字符占三个字节】，此时key_len显示的值为3*10=30，所以此值并非实际使用长度而是我们定义的长度）。在不损失精确性的情况下，长度越短越好。所以blog、text类型的字段不适合做索引。当我们将字段设置是否为null也会影响key_len的长度，为null时长度会加1。varchar类型的字段需要额外加一个字节，当长度设置大于255时需要加两个字节。额外的这两个字节存储varchar的实际长度，大于255需要两个字节是因为一个字节占8位，最大值只能表示2^8即256，而varchar的最大长度为65535，即2^16次方，所以需要两位来存储。

44、索引字段最好不要为NULL，因为NULL让统计更加复杂并且需要额外的存储空间。

44、复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之和，这也可以用来判定复合索引是否部分使用，还是全部使用。例如复合索引（name【10】，age【10】，sex【10】），当所有索引生效时key_len=30，而当key_len=20时表明只有name、age生效了。

45、执行计划中key表示命中的是哪个索引，根据key_len可以判断命中的复合索引中的那几个字段被有效使用了。

45、避免Extra中的Using filesort问题（https://blog.csdn.net/qq_15037231/article/details/88601699）。要么将order by后边的字段全加在复合索引中，要么将字段去掉，通过java程序进行排序。可以看下第五个视频2:18:00处的例子。例子中在order by后边加desc【降序排序】时查询效率还是很低，并且Extra的属性值包括Using filesort，即对查询结果又进行重新排序了。er将desc改为asc时查询效率很高是因为asc是默认的排序方式，不需要重新进行排序。

```
AND (
  (
    a.date_str >= '2018-07-01'
    AND a.date_str <= '2018-08-30'
  )
)
ORDER BY
  a.date_str,a.shopCode,a.add_car_pv

//里面已经维护了顺序，这个时候，我排序的时候直接用里面索引的顺序就好了，而如果用desc,那么需要自己进行排序
```

46、sql优化时主要通过type、key、key_len、ref、rows、extra综合考虑。

47、sql优化策略：

1. **尽量匹配索引全值。**
2. **最佳左前缀原则。**
3. **不在索引列上做任何操作（计算、函数、(自动or手动)类型转换）。**
4. **范围条件放最后（因为复合索引是按1、2、3...列的顺序生成B+Tree的，当把范围条件例如【>22】放到where中间时，按索引列查询的时候查到范围的条件后的索引就会失效），in查询也属于范围查询。**
5. **覆盖索引尽量使用。**
6. **不等于要慎用。（假如表A有组合索引（a、b、c），当执行sql语句select a，b，c from A where a <> 'ljq' 时索引不会失效，虽然这个sql中使用了不等于，但是这**

个sql也存在覆盖索引，所以索引没失效。即通过覆盖索引解决不等于到时索引失效的问题）。

7. 将索引字段【a】定义为null或者not null对索引可能有影响（可通过覆盖索引解决此种情况下索引失效问题）：

- 将字段a定义为not null的时候，在sql中使用【a is null】或者【a is not null】时会导致索引失效。
- 将字段a自定义为null或者不定义的时候，在sql中使用【a is null】不会导致索引失效，但使用【a is not null】会导致索引失效。

8. like查询要当心，以通配符('%abc...')开头的会导致索引失效（也可以通过覆盖索引解决问题）。具体原因我认为主索引生成是根据某个字段的全值或者前面几个字符的值生成，当使用'%abc..'或者'%abc...%'时它的字符值是无法匹配的。

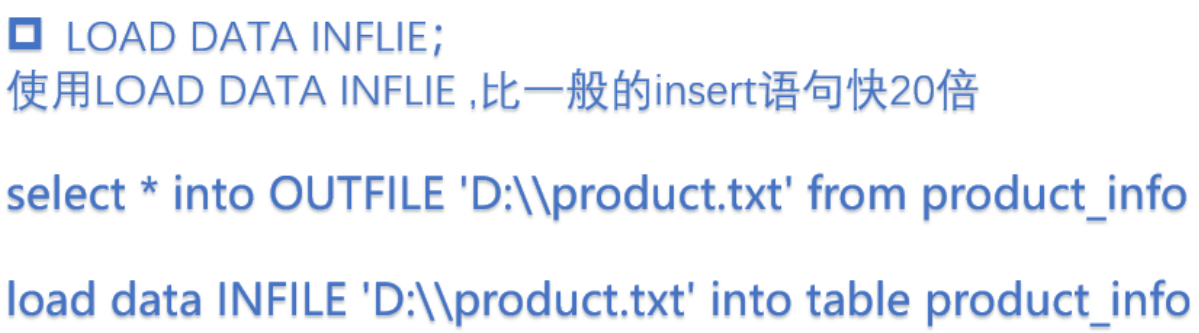
9. 字符类型加引号。

10. OR改UNION效率高。

48、批量插入优化：

1. 常用的插入语句NSERT INTO insert_table (datetime，uid， content， type) VALUES ('0'， 'userid_0'， 'content_0'， 0)； INSERT INTO insert_table (datetime， uid， content， type) VALUES ('1'， 'userid_1'， 'content_1'， 1)；修改成INSERT INTO insert_table (datetime，uid，content，type) VALUES ('0'，'userid_0'，'content_0'，0)，('1'，'userid_1'，'content_1'，1)；

2. 使用load data file：



49、mysql使用全文索引进行文档查询，全文索引原理及使用（https://www.cnblogs.com/shen-qian/p/11883442.html）。

50、innodb辅助索引为什么存储主键键值而不是原数据的可能原因（https://blog.csdn.net/moakun/article/details/81813994）。

51、一张表只能有一个主键，所谓的多个主键是联合主键（即多个列组成一个主键，其实主键（PRIMARY KEY）只有一个）。

52、当使用union all替换or、in的时候条件列必须为索引，不然union all反而会比较耗时，因为要多次扫表，看union的用法就明白了。（替换or、in的时候是拆分成多个select查询）

```
Select * FROM `article` Where article_category IN (2,3) ORDER BY article_id DESC LIMIT 5 扫
(select * from article where article_category=2 order by article_id desc limit 5)
UNION ALL (select * from article where article_category=3 order by article_id desc limit 5)
ORDER BY article_id desc
```

53、一切优化的原则都在与条件列为索引，优化只是为了命中这些索引。而如果条件列都不是索引，那么何谈优化。一切优化都在于将索引命中率提高或者新加索引。

16、mysql的case、when用法有两种（写法有所不同）：

- **简单Case函数 （case后边为条件）**

```
CASE [col_name] WHEN [value1] THEN [result1]…ELSE [default] END
```

col_name为列名，value1为值

- **Case搜索函数（case后边什么都没有，when后边是表达式）**

```
CASE WHEN [expr] THEN [result1]…ELSE [default] END
```

expr为表达式，例如col>2

54、mysql索引下推和回表（https://www.cnblogs.com/lonelyxmas/p/12630085.html）。

myisam的写锁优先级比读锁高（防止读线程比较多的情况下写线程饥饿）

55、mysql锁和索引：

- **https://segmentfault.com/a/1190000019619667**
- **https://juejin.im/post/5b55b842f265da0f9e589e79#heading-13**

- https://zhuanlan.zhihu.com/p/29150809

innodb行锁是加在索引项上边，不是在数据上边加锁。所谓索引项就是表中的字段名，索引值就是字段对应的值

MySQL中的for update 仅适用于InnoDB（因为是只有此引擎才有行级锁），并且必须开启事务，在begin与commit之间才生效。for update是在数据库中上锁用的，可以为数据库中的行上一个排它锁。当一个事务的操作未完成时候，其他事务可以对这行读取但是不能写入或更新，只能等该事务Rollback, Commit, Lost connection...

MVCC：MVCC其实是乐观锁的一种实现方式，只不过有版本链。
https://baijiahao.baidu.com/s?id=1629409989970483292&wfr=spider&for=pc
https://www.codercto.com/a/88775.html

在**表锁中我们读写是阻塞**的，基于提升并发性能的考虑，**MVCC一般读写是不阻塞的**(所以说MVCC很多情况下避免了加锁的操作)，但是MVCC读到的数据可能不是最新数据（在快照读下读到的可能是历史数据，而在当前读下读到的是最新数据。）

切记：MVCC（多版本并发控制）是通过在数据行加列来记录历史版本而解决读写互斥的（即读、写同时进行，读不需要加锁而是去读取历史版本），它并不能解觉写写互斥，因为写是必须要加锁的。

在innodb中如果单纯依赖数据库锁（S锁、X锁）的方式实现事务隔离级别的话那数据库的并发度将会很低（因为当对数据加S锁的时候，其他事务也只能加S锁，但不能加X锁，即多个事务可以同时读相同数据，但是读、写不能共存，这就是所谓的锁机制下的读写阻塞），虽然innodb的行锁已经在并发度方面提升很多了，但是行锁也是读写阻塞（读写互斥）的，即当有事务加了X锁以后其他事务去读取数据的时候需要加S锁，但此时会加锁失败，而MVCC就是用来处理读写冲突的手段，目的在于提高数据库高并发场景下的吞吐性能，这就使得在事务并发过程中别的事务可以修改这条记录，反正每次修改都会在版本链中记录。SELECT可以去版本链中拿记录，这就实现了读-写，写-读的并发执行，提升了系统的性能。SELECT 操作可以不加锁而是通过 MVCC 机制读取指定的版本历史记录，并通过一些手段（ReadView）保证保证读取的记录值符合事务所处的隔离级别，从而解决并发场景下的读写冲突。说白了并发事务处理大大增加了数据库资源的利用率，但是并发事务处理却带来了（1、更新丢失；2、脏

读；3、不可重复读；4、幻读）四个主要问题，这四个问题中解决更新丢失可以交给应用处理，但是后三个问题需要数据库提供事务间的隔离机制来解决（即四种事务隔离级别），而实现事务隔离机制的主要方法就是以下两种，innodb是通过两种结合两种机制来实现的：

- 加读写锁：只使用读写锁会造成读写阻塞，并发能力降低。
- MVCC（一致性快照读）：对select进行无锁操作从而避免读写阻塞，即读与写可同时进行。但是写与写之间还是互斥的，即意味着写之前还是需要X锁。

InnoDb中每一行记录都有两个隐藏列data_trx_id、data_roll_pointer（如果没有主键，则还会多一个隐藏的主键列）：

- data_trx_id：用来存储每次对某条记录进行修改时候的事务id。
- data_roll_pointer：每次对某行记录有修改的时候，都会把老版本写入undo日志中。这个roll_pointer就是存了一个回滚指针，它指向这条记录的上一个版本的位置，通过它可以获得上一个版本的信息。

版本链是如何组织的：在多个事务并行操作某行数据的情况下，不同事务对改行数据的update会产生多个版本，然后通过回滚指针组成一条Undo Log链。例如有三个事务A、B、C想对同一行数据进行修改，此时需要他们先获得改行的X锁，假如获得锁的顺序为A、B、C，当A获得X锁后，它先会将改行的原本数据写入到undo log中（包括data_trx_id和db_roll_pointer），然后进行更新操作，同时修改data_trx_id为当前事务（A事务）的ID，将data_roll_pointer指向刚刚拷贝到undo log链中的旧版本记录，然后A事务提交，此时B事务获得X锁，B事务会先将A事务修改之后的数据拷贝到undo log中（包括data_trx_id和db_roll_pointer），然后B事务进行更新操作并将data_trx_id修改为当前事务（B事务）的ID，将data_roll_pointer指向刚刚拷贝到undo log链中的旧版本（A版本）记录，C事务获得X锁后进行的操作亦是如此，此时【C→B→A】的undo log链已生成。

什么是ReadView：ReadView中主要就是有个列表来存储我们系统中当前活跃着的读写事务的ID，通过这个列表来判断记录的某个版本是否对当前事务可见。

MVCC是如何解决读写阻塞的：根据前文可以知道多个事务对同一行数据修改的时候会生成undo log版本链，版本链中的每个版本都记录了原数据以及当次事务的id（即隐藏列data_trx_id的值）以及指向上个版本的指针。此时如果某个事务需要查询该行数据，先会生

成ReadView，然后从最新版本开始遍历undo log链，当拿到一个版本（即准备访问的记录版本）后用这个版本的事务id去和ReadView中的事务ID比对，假如ReadView列表里的事务id为【60，100】，比对规则如下：

- 如果你要访问的记录版本（当前从undo log链表中遍历到的版本）的事务id为50，比当前列表最小的id80小，那说明这个事务在之前就提交了，所以对当前活动的事务来说是可访问的。
- 如果你要访问的记录版本的事务id为70，发现此事务在列表id最大值和最小值之间，那就再判断一下是否在列表内，如果在那就说明此事务还未提交，所以版本不能被访问。如果不在那说明事务已经提交，所以版本可以被访问。
- 如果你要访问的记录版本的事务id为110，那比事务列表最大id100都大，那说明这个版本是在ReadView生成之后才发生的，所以不能被访问。

RC和RR隔离级别下MVCC表现差异总结： RC 、 RR 两种隔离级别的事务在执行普通的读操作时，通过访问版本链的方法，使得事务间的读写操作得以并发执行，从而提升系统性能。 RC 、 RR 这两个隔离级别的不同是RC不可重读，RR可重读，底层原理是RC、RR生成 ReadView 的时间点不同， RC 在每一次 SELECT 语句前都会生成一个 ReadView ，事务期间会更新（这期间其他事务可能提交了或者回滚了），因此在其他事务提交前后所得到的 m_ids 列表可能发生变化，使得先前不可见的版本后续又突然可见了（即能读到第一次读取时还没提交但第二次读取时已经提交的数据）。而 RR 只在事务的第一个 SELECT 语句时生成一个 ReadView ，后续所有的 SELECT 都是复用这个 ReadView，事务操作期间不更新。

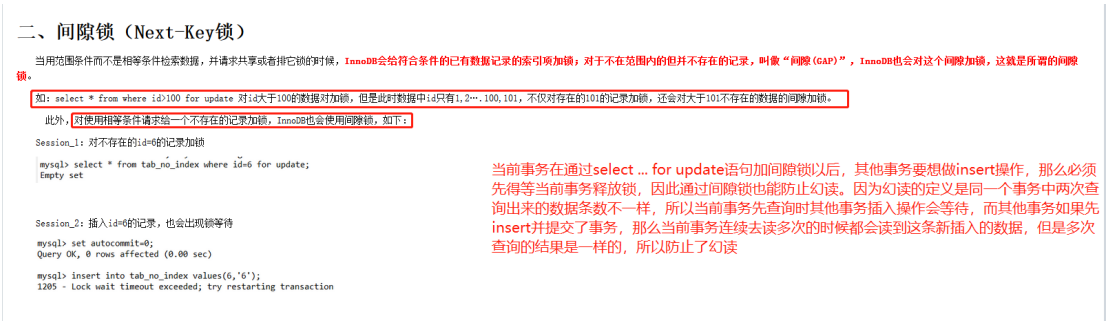
可重复读和幻读讲的是同一个事务中两次查询的结果不一致，并不是说先开启的事务并不能读取到比它晚开启但早提交的事务修改的值，举例如下：A事务先开启（事务id为100），B事务后开启（事务id为200），假如此时B事务修改了一行数据的值并提交了，此时A事务才去读取这行数据时会读取到B事务提交的结果。因为ReadView是在事务的第一次查询时生成的。

快照读和当前读的实现方式。

(https://www.cnblogs.com/wwcom123/p/10727194.html)

mysql中RR隔离级别通过MVCC和间隙锁解决了幻读问题：

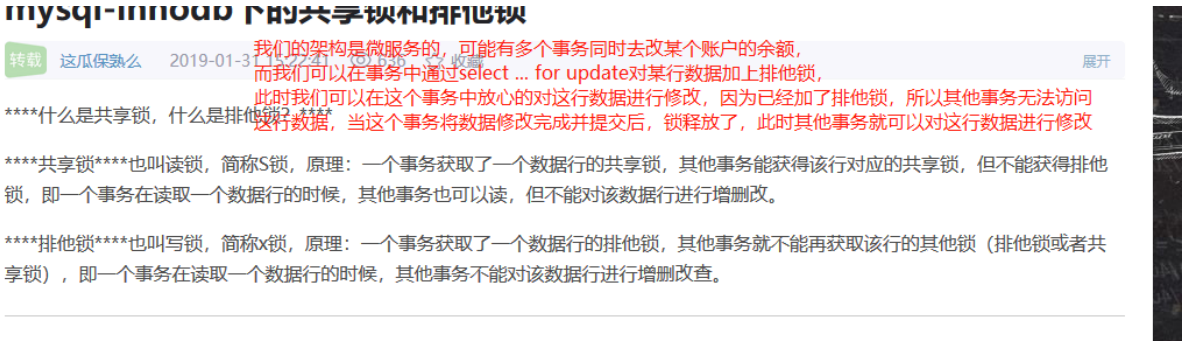
- **在快照读下没有幻读和不可重复读问题。因为快照读是通过MVCC技术实现的，即MVCC机制会避免给select语句加锁而是去读历史版本中的数据，不会读取当前其他活跃事务没提交的数据。**
- **在当前读下（当前读是通过行锁+间隙锁实现的，所以不会出现幻读）RR隔离级别通过间隙锁防止了幻读。当前读就是select ... where ... for update、update、insert、delete语句。因为在update、insert、delete是必须要获得锁才能执行，所以和MVCC机制无关，这对于select ... for update语句已经显示表明需要先获得锁，这样就相当于导致MVCC机制失效了。因此可以看出当前读语句都是需要先获得数据行的X锁，而当有事务正在修改需要读的数据时当前读语句都会阻塞（因为获取不到锁），当它能顺利执行的时候说明其他事务的修改都已经提交。**



共享锁（读锁）与排它锁（写锁）：

https://blog.csdn.net/luzhensmart/article/details/86715379

锁的前提是在不同的事务中（即当有多个事务同时操作同一条数据时才需要加锁），当某个事务获取到锁以后，事务结束了，锁也就释放了：



2.3.1悲观锁

所以，按照上面的例子。我们使用悲观锁的话其实很简单(手动加行锁就行了)：

- `select * from xxxx for update`

在select 语句后边加了 `for update` 相当于加了排它锁(写锁)，加了写锁以后，其他的事务就不能对它修改了！需要等待当前事务修改完之后才可以修改.

- 也就是说，如果张三使用 `select ... for update` ，李四就无法对该条记录修改了~

在数据库操作中，为了有效保证并发读取数据的正确性，提出的事务隔离级别。我们的数据库锁，也是为了构建这些隔离级别存在的。Innodb的行锁对事务隔离级别的构建功不可没。不同的隔离级别对锁的使用是不同的，锁的应用最终导致不同事务的隔离级别

(https://blog.csdn.net/zxjiayou1314/article/details/80351603)

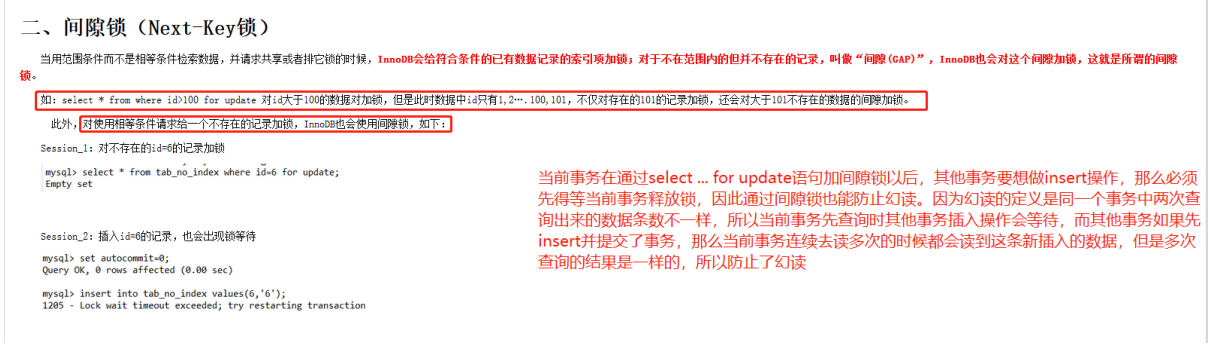
innodb实现原理：InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁!

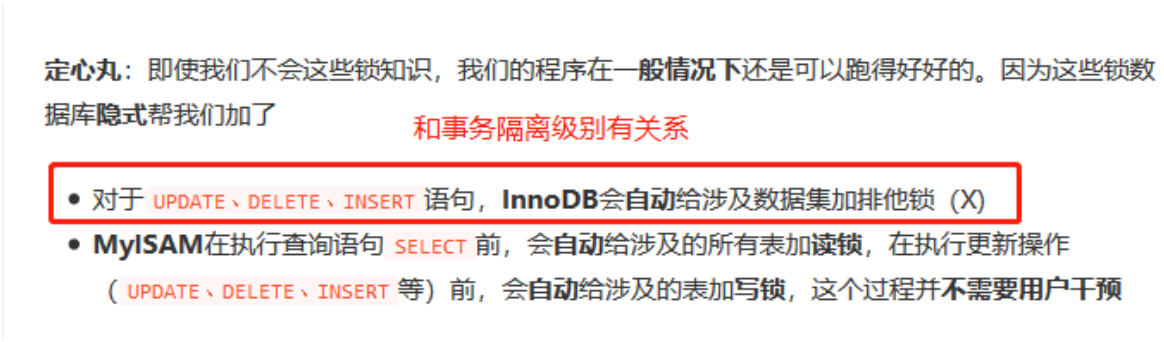
innodb已经有了行锁为什么不放弃表锁：因为行锁是基于索引实现的，当where后边的条件命中索引时才会进行行锁，当索引失效时会升级成表锁。所以表锁是有用的。索引失效以后聚集索引上的每条记录，无论是否满足条件，都会被加上X锁。但是，为了效率考量，MySQL做了优化，对于不满足条件的记录，会在判断后释放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。

innobd表锁使用场景以及间隙锁：

https://www.cnblogs.com/jian0110/p/12721924.html

https://www.jianshu.com/p/42e60848b3a6





数据库乐观锁：乐观锁不是数据库层面上的锁，是需要自己手动去加的锁。一般我们添加一个版本字段来实现。每次更新前先版本，然后更新的时候将版本放到where条件中。

意向锁存在的意义：减少逐行检查锁标志的开销。
https://www.jianshu.com/p/e83e88e2bcee
https://blog.csdn.net/jinjiniao1/article/details/99572530

间隙锁：https://blog.csdn.net/sinat_27143551/article/details/81736330

对于快照的两个级别：

- **语句级：针对于Read committed隔离级别**
- **事务级别：针对于Repeatable read隔离级别**

四种事务隔离级别的实现技术：

- **Read uncommitted（读未提交）：出现脏读的本质就是因为操作(修改)完该数据就立马释放掉锁（事务未提交之前），导致读的数据就变成了无用的或者是错误的数据。**
- **Read committed（读已提交）：就是把释放锁的位置调整到事务提交之后，此时在事务提交前，其他进程是无法对该行数据进行读取的，包括任何操作，这样就避免了脏读。但是Read committed是语句级别的快照，每次读取的都是当前数据已提交的最新版本，所以会有不可重复读的问题。**
- **Repeatable read（可重复读）：RR是事务级别的快照，每次读取的都是当前事务的版本，即使被其他事务修改了，也只会读取当前事务版本的数据。**

----------常用sql语句----------

新增列：alter table mytable add column mjmiid varchar(8) character set utf8mb4 default null。

关联查询https://www.cnblogs.com/hitechr/p/10408085.html