**Mysql数据库的个人理解和总结**

**常用的数据类型**

整数数据类型、浮点数数据类型、精确小数类型、二进制数据类型、日期/时间数据类型、字符串数据类型。

**MySQL中的存储引擎myisam和innodb**

（1）InnoDB存储引擎支持事务，而MyISAM不支持事务；

（2）InnoDB支持行级锁，而MyISAM只支持表级锁；

（3）InnoDB支持外键，而MyISAM不支持外键；

（4）InnoDB不保存数据库表中表的具体行数，而MyISAM会保存；

**MySQL中的事务，以及事务的特征**

事务是一种操作序列，一系列的操作要么都执行，要么都不执行，它们是一个整体。每个事务结束时，都保持着数据一致性。事务有四大特征：原子性、一致性、隔离性、持久性。

**数据库中的乐观锁和悲观锁**

乐观锁：假设不会发生并发冲突，只在提交的时候检查是否发生并发冲突。可以使用版本号机制和CAS算法实现。

版本号机制：一般在数据表中加一个数据版本号version字段，表示数据被修改的次数，当数据被修改时version值加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若当前读取到的version值与第一次读取到的数据库version值相等时才更新，否则重试更新操作，直到更新成功。

CAS机制：即compare and swap（比较与交换），无锁编程，在不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，因此也叫非阻塞同步。

悲观锁：假定会发生并发冲突，在读取的时候就对数据进行加锁， 在该用户读取数据的期间，其他任何用户都不能来修改该数据，但是其他用户是可以读取该数据的， 只有当自己读取完毕才释放锁。

　在数据库中可以使用Repeatable Read的隔离级别（可重复读）来实现悲观锁，它完全满足悲观锁的要求（加锁）。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

两种锁的使用场景：

乐观锁性能好，但是无法解决脏读问题

悲观锁能解决脏读问题，开销较大，而且加锁时间较长，对于并发的访问性支持不好。

　　如果冲突很少，或者冲突的后果不会很严重，那么通常情况下应该选择乐观锁，因为它能得到更好的并发性；

　　如果冲突太多或者冲突的结果对于用户来说痛苦的，那么就需要使用悲观策略，它能避免冲突的发生。

一般乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候；悲观锁适用于多写的情况，多写的情况一般会经常产生冲突。

**共享锁与排它锁**

共享锁和排它锁是具体的锁，是数据库机制上的锁。

共享锁（读锁）： 在同一个时间段内，多个用户可以读取同一个资源，读取的过程中数据不会发生任何变化。读锁之间相互不阻塞， 多个用户可以同时读，但是不能允许有人修改。

排它锁（写锁）： 在任何时候只能有一个用户写入资源，当进行写锁时会阻塞其他的读锁或者写锁操作，只能由这一个用户来写，其他用户既不能读也不能写。

加锁会有粒度问题，从粒度上从大到小可以划分为 ：

表锁：开销较小，一旦有用户访问这个表就会加锁，其他用户就不能对这个表操作了，应用程序的访问请求遇到锁等待的可能性比较高。

页锁：是MySQL中比较独特的一种锁定级别，锁定颗粒度介于行级锁定与表级锁之间，所以获取锁定所需要的资源开销，以及所能提供的并发处理能力也同样是介于上面二者之间。另外，页级锁定和行级锁定一样，会发生死锁。

行锁：开销较大，能具体的锁定到表中的某一行数据，但是能更好的支持并发处理， 会发生死锁。

**MySQL数据库的基本的索引类型**

索引是对数据库表中一或多个列的值进行排序的结构，利用索引可快速访问数据库表的特定信息。

普通索引、唯一索引、主键索引、联合索引、全文索引。

唯一索引：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

主键索引：是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。 为表定义主键将自动创建主键索引。（数据库表某列或列组合，其值唯一标识表中的每一行。该列称为表的主键。）

联合索引：指对表上的多个列做索引。只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀原则。

全文索引：主要用来查找文本中的关键字，而不是直接与索引中的值相比较。目前只有char、varchar，text 列上可以创建全文索引。

**数据表建立索引的原则：**

在最频繁使用的、用以缩小查询范围的字段上建立索引。

在频繁使用的、需要排序的字段上建立索引。

**B+树索引与哈希索引**

Hash索引和B+树索引的特点：

Hash索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位;

B+树索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问。

Hash索引与B+树索引区别？

如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；

从示意图中也能看到，如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；

同理，哈希索引也没办法利用索引完成排序，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；

哈希索引也不支持多列联合索引的最左匹配规则；

B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

**B树和B+树**

B+树是一种平衡查找树。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节点中，各叶结点指针进行连接。

（平衡二叉树AVL：首先符合二叉查找树的定义（最结点的值比根节点小，右结点的值比根结点大），其次必须满足任何节点的左右两个子树的高度最大差为1。）

B树 ：每个节点都存储key和data，所有节点组成这棵树，并且叶子节点指针为nul，叶子结点不包含任何关键字信息。

B+树：所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接，所有的非终端结点可以看成是索引部分。

**B+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**

（1）B+的磁盘读写代价更低

　　B+的内部结点并没有指向关键字具体信息的指针。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盘块中，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。

（2）B+tree的查询效率更加稳定

　由于非终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。

**聚集索引和非聚集索引**

数据库中的B+索引可以分为聚集索引和辅助聚集索引。不管是聚集索引还是非聚集的索引，其内部都是B+树的，即高度平衡的，叶节点存放着所有的数据，聚集索引与非聚集索引不同的是，叶节点存放的是否是一整行的信息。

聚集索引(clustered index)：

　　聚集索引就是按照每张表的主键构造一颗B+树，并且叶节点中存放着整张表的行记录数据，因此也让聚集索引的叶节点成为数据页。聚集索引的这个特性决定了索引组织表中数据也是索引的一部分。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。

　　聚集索引表记录的排列顺序和索引的排列顺序一致，所以查询效率快，只要找到第一个索引值记录，其余就连续性的记录在物理也一样连续存放。聚集索引对应的缺点就是修改慢，因为为了保证表中记录的物理和索引顺序一致，在记录插入的时候，会对数据页重新排序。

非聚集索引(nonclustered index)（也叫辅助索引）：

　　对于辅助索引(非聚集索引)，叶级别不包含行的全部数据。聚集索引键来告诉InnoDB存储引擎，哪里可以找到与索引相对应的行数据。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。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。

　　非聚集索引指定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致，两种索引都采用B+树结构，非聚集索引的叶子层并不和实际数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针方式。非聚集索引层次多，不会造成数据重排。

根本区别：

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。