数据库的特性

四大隔离级别

1. 读取未提交： 最低的隔离级别，允许读取尚未提交的数据变更， 可能会导致脏读、幻读或不可重复读。
2. 读取已提交： 允许读取并发事务已经提交的数据， 可以阻止脏读，但是幻读或不可重复读仍有可能发生。
3. 可重复读： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改， 可以阻止脏读和不可重复读，但幻读仍有可能发生。
4. 可串行化： 最高的隔离级别， 该级别可以防止脏读、不可重复读以及幻读。

存储引擎

索引

索引是什么

索引（Index）是帮助 MySQL 高效获取数据的数据结构，是一种排好序的数据结构

为什么要用索引

1. 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。
2. 可以大大加快 数据的检索速度（大大减少的检索的数据量），这也是创建索引的最主要的原因。
3. 帮助服务器避免排序和临时表。
4. 将随机IO变为顺序IO
5. 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。

索引这么多优点，为什么不对表中的每一个列创建一个索引呢?

1. 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。
2. 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，如果要建立聚簇索 引，那么需要的空间就会更大。
3. 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。

索引的优缺点

优点：

1. 提高数据检索效率，降低数据的IO成本
2. 降低数据排序成本，降低CPU消耗

缺点：

1. 索引也是一张表，索引本身也是要占用空间的
2. 索引虽然提高了查询速度，但是降低了更新表的速度

索引使用场景

1. 为经常出现在关键字order by. group by. distinct后面的字段 ,建立索引。
2. 在union等集合操作的结果集字段上,建立索引。
3. 为经常用作查询选择的字段,建立索引. 在经常用作表连接的属性上,建立索引。 考虑使用索引覆盖。对数据很少被更新的表,如果用户经常只查询其中的几个字段，可以考虑在这几个字段 上建立索引,从而将表的扫描改变为索引的扫描。

索引的分类

1. 单值索引
2. 唯一索引
3. 复合索引
4. 主键索引
5. 组合索引
6. 聚集索引
7. 非聚集索引
8. 覆盖索引

什么是覆盖索引

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！

为什么索引能提高查询速度

如果我们写select * from user where indexname = 'xxx'这样没有进行任何优化的sql语句，默认会这样做：

定位到记录所在的页：需要遍历双向链表，找到所在的页 从所在的页内中查找相应的记录：由于不是根据主键查询，只能遍历所在页的单链表了

没有用索引我们是需要遍历双向链表来定位对应的页，现在通过 “目录” 就可以很快地定位到对应的页上了！（二分查找，时间复杂度近似为O(logn)）

mysql的索引结构

1. B树索引
2. B+树
3. Hash索引
4. 聚簇索引与非聚簇索引

B树与B+树的区别

1. B-树的关键字和记录是放在一起的，叶子节点可以看作外部节点，不包含任何信息；B+树叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。(一次查询可能进行两次i/o操作)

2. 在B-树中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而B+树中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。从这个角度看B-树的性能好像要比B+树好，而在实际应用中却是B+树的性能要好些。因为B+树的非叶子节点不存放实际的数据，这样每个节点可容纳的元素个数比B-树多，树高比B-树小，这样带来的好处是减少磁盘访问次数。尽管B+树找到一个记录所需的比较次数要比B-树多，但是一次磁盘访问的时间相当于成百上千次内存比较的时间，因此实际中B+树的性能可能还会好些，而且B+树的叶子节点使用指针连接在一起，方便顺序遍历（例如查看一个目录下的所有文件，一个表中的所有记录等），这也是很多数据库和文件系统使用B+树的缘故。

B+树相比于B树的查询优势

1. B+树的磁盘读写代价更低　　B+树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+树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。

B+树与红黑树比较

B+树与hash索引比较

聚簇索引的好处：

按照聚簇索引排列顺序，查询显示一定范围数据的时候，由于数据都是紧密相连，数据库不用从多个数据块中提取数据，所以节省了大量的io操作。

聚簇索引的限制：

1. 对于mysql数据库目前只有innodb数据引擎支持聚簇索引，而Myisam并不支持聚簇索引。
2. 由于数据物理存储排序方式只能有一种，所以每个Mysql的表只能有一个聚簇索引。一般情况下就是该表的主键。
3. 为了充分利用聚簇索引的聚簇的特性，所以innodb表的主键列尽量选用有序的顺序id，而不建议用无序的id，比如uuid这种。（参考聚簇索引的好处。）

聚集索引

聚集索引即索引结构和数据一起存放的索引。主键索引属于聚集索引。

非聚集索引

非聚集索引即索引结构和数据分开存放的索引。

非聚集索引的优点

更新代价比聚集索引要小 。非聚集索引的更新代价就没有聚集索引那么大了，非聚集索引的叶子节点是不存放数据的

非聚集索引的缺点

跟聚集索引一样，非聚集索引也依赖于有序的数据 可能会二次查询(回表) :这应该是非聚集索引最大的缺点了。当查到索引对应的指针或主键后，可能还需要根据指针或主键再到数据文件或表中查询。

MySQL优化

索引优化

查询优化

聚集索引与非聚集索引

事务

什么是事务

事务是逻辑上的一组操作，要么都执行，要么都不执行。

数据库事务特性

1. 原子性（Atomicity）： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. 一致性（Consistency）： 执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；
3. 隔离性（Isolation）： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
4. 持久性（Durability）： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

并发事务带来什么问题

1. 脏读: 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务 读到的这个数据是"脏数据"，依据"脏数据"所做的操作可能是不正确的。

2. 不可重复读（Unrepeatableread）: 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务 两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可 重复读。

3. 幻读（Phantom read）: 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在 的记录，就好像发生了幻觉一样，所以称为幻读。

mysql引擎

MyISAM和InnoDB区别

MyISAM是MySQL的默认数据库引擎。虽然性能极佳，而且提供了大量的特性，包括全文索引、压缩、空间函数等，但MyISAM不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。不过， 5.5版本之后，MySQL引入了InnoDB（事务性数据库引擎）， MySQL 5.5版本后默认的存储引擎为InnoDB。 大多数时候我们使用的都是 InnoDB 存储引擎，但是在某些情况下使用 MyISAM 也是合适的比如读密集 的情况下。

比较9条

MyISAM和InnoDB存储引擎使用的锁：

MyISAM采用表级锁(table-level locking)。 InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

表级锁和行级锁对比：

表级锁： MySQL中锁定 粒度最大 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM和 InnoDB引擎都支持表级锁。 行级锁： MySQL中锁定 粒度最小 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

InnoDB存储引擎的锁的算法：

Record lock：单个行记录上的锁 Gap lock：间隙锁，锁定一个范围，不包括记录本身 Next-key lock：record+gap 锁定一个范围，包含记录本身

当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，一些常见的优化措施如下：

大表优化

1. 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；
2. 读/写分离 经典的数据库拆分方案，主库负责写，从库负责读；
3. 垂直分区 根据数据库里面数据表的相关性进行拆分。
4. 水平分区 保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。

垂直拆分优缺点：

垂直拆分的优点： 可以使得列数据变小，在查询时减少读取的Block数，减少I/O次数。此外，垂直分区可以简化表的结构，易于维护。 垂直拆分的缺点： 主键会出现冗余，需要管理冗余列，并会引起Join操作，可以通过在应用层进行Join来解决。此外，垂直分区会让事务变得更加复杂；

数据库分片的两种常见方案：

1. 客户端代理： 分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。
2. 中间件代理： 在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。

池化

什么是池化设计思想。

这种设计会初始预设资源，解决的问题就是抵消每次获取资源的消耗，池化设计还包括如下这些特征：池子的初始值、池子的活跃值、池子的最大值等，这些特征可以直接映射到java线程池和数据库连接池的成员属性中。这篇文章对池化设计思想介绍的还不错，直接复制过来，避免重复造轮子了。

什么是数据库连接池?为什么需要数据库连接池?

数据库连接本质就是一个 socket 的连接。数据库服务端还要维护一些缓存和用户权限信息之类的 所以占用了一些内存。我们可以把数据库连接池是看做是维护的数据库连接的缓存，以便将来需要对数据库的请求时可以重用这些连接。为每个用户打开和维护数据库连接，尤其是对动态数据库驱动的网站应用程序的请求，既昂贵又浪费资源。在连接池中，创建连接后，将其放置在池中，并再次使用它，因此不必建立新的连接。如果使用了所有连接，则会建立一个新连接并将其添加到池中。 连接池还减少了用户必须等待建立与数据库的连接的时间。

分库分表

分库分表之后,id 主键如何处理?

因为要是分成多个表之后，每个表都是从1开始累加，这样是不对的，我们需要一个全局唯一的 id 来支持。

生成全局 id 有下面这几种方式：

1. 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。
2. 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。
3. 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。

慢查询

什么是慢查询

它用来记录在MySQL中响应时间超过阀值的语句日志记录

应用

一条SQL语句执行得很慢的原因有哪些

要分两种情形：

1. 大多数情况是正常的，只是偶尔会出现很慢的情况。
    - 数据库在刷新脏页，例如 redo log 写满了需要同步到磁盘。
    - 执行的时候，遇到锁，如表锁、行锁。

2. 在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。

    - 没有用上索引
    - 数据库选错了索引

为什么数据库会选错了索引

系统在执行的时候，会进行预测，是走 c 索引扫描的行数少，还是直接扫描全表扫描的行数少呢?

扫描全表的话，那么扫描的次数就是这个表的总行数了，假设为 n；而如果走索引 c 的话，我们通过索引 c 找到主键之后，还得再通过主键索引来找我们整行的数据，需要走两次索引，而且，我们也不知道符合 这个条件的数据有多少行，万一真的是n条，那就惨了，所以系统是有可能走全表扫描而不走索引的

系统如何进行预判主要依赖于索引的区分度来判断的，一个索引上不同的值越多，意味着出现相同数值的索引越少，意味着索引的区分度越高。

这个区分度也叫做基数，系统当然是不会遍历全部来获得一个索引的基数的，代价太大了，索引系统是通过遍历部分数据，也就是通过采样的方式，来预测索引的基数的

那么出现失误的地方就是采样，比如采样的那一部分数据刚好基数很小，然后就误以为索引的基数很小。然后，系统就不走索引了，直接走全部扫描了。

主要是由于统计的失误，导致系统没有走索引，而是走了全表扫描。