MySQL数据库

- 关系型数据库特性/缺点
  - 特性
    - 1、关系型数据库，是指采用了关系模型来组织数据的数据库；
    - 2、关系型数据库的最大特点就是事务的一致性；
    - 3、简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。
  - 关系型数据库的缺点
    - 1、为了维护一致性所付出的巨大代价就是其读写性能比较差；
    - 2、固定的表结构；
    - 3、高并发读写需求；
    - 4、海量数据的高效率读写；
- 事务（ACID）
  - 事务提供一种机制将一个活动涉及的**所有操作纳入到一个不可分割的执行单元**，组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交，只要其中任一操作执行失败，都将导致整个事务的回滚。
  - 简单地说，事务提供一种“ 要么什么都不做，要么做全套（All or Nothing）”机制。
  - ACID
  - 原子性Atomicity —— 事务操作的整体性
    - 整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被 回滚（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Consistency —— 事务操作下数据的正确性
    - 在事务开始之前和事务结束以后，数据库数据的一致性约束没有被破坏。
  - 隔离性 Isolation —— 事务并发操作下数据的正确性
    - 数据库允许多个并发事务同时对数据进行读写和修改的能力，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响。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。
  - 持久性 Durability —— 事务对数据修改的可靠性
    - 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。
  - 锁：是实现事务的关键，所可以保证事务的完整性和并发行，与现实生活中的锁一样，可以使某些数据的拥有者，在某段时间内不能使用某些数据或者数据结构。
- 事务的崩溃恢复（日志）

- Undo Log:
  - Undo Log是为了实现事务的原子性，在MySQL数据库InnoDB存储引擎中，还用了Undo Log来实现多版本并发控制(简称：MVCC)。
  - Undo Log的原理很简单，为了满足事务的原子性，**在操作任何数据之前，首先将数据备份到一个地方**（这个存储数据备份的地方称为UndoLog）。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态。
  - 之所以能同时保证原子性和持久化，是因为以下特点：
    - 更新数据前记录Undo log。
    - 为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。
  - 缺陷
    - 每个事务提交前将数据和Undo Log写入磁盘，这样会导致大量的磁盘IO，因此性能很低。
- 如果能够将数据缓存一段时间，就能减少IO提高性能。但是这样就会丧失事务的持久性。因此引入了另外一种机制来实现持久化，即Redo Log。
- Redo Log:
  - 原理和Undo Log相反，Redo Log记录的是**新数据的备份。在事务提交前，只要将Redo Log持久化即可**，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据Redo Log的内容，将所有数据恢复到最新的状态。
- binlog
  - 二进制日记记录数据库的变化情况，内容报错数据库所有的更新操作，ddl和dml。数据库管理员可以通过二进制日志查看数据库过去某一时刻发生了哪些变化，必要时可以使用二进制日志恢复数据库。二进制文件内容为二进制信息

- 隔离级别 /事务传播
  - 从理论上来说, 事务应该彼此完全隔离, 以避免并发事务所导致的问题，然而, 那样会对性能产生极大的影响, 因为事务必须按顺序运行， 在实际开发中, 为了提升性能, 事务会以较低的隔离级别运行
  - 事务的并发问题
    - 脏读
      - 事务B读取事务A还没有提交的数据
    - 不可重复读
      - 事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果因此本事务先后两次读到的数据结果会不一致
    - 幻读
      - 不可重复读相比指更新数据，幻读指插入或者删除数据

        - mysql 的幻读并非什么读取两次返回结果集不同，而是事务在插入事先检测不存在的记录时，惊奇的发现这些数据已经存在了，之前的检测读获取到的数据如同鬼影一般。
- 事务的隔离级别
    - 1、读未提交：另一个事务修改了数据，但尚未提交，而本事务中的SELECT会读到这些未被提交的数据脏读
    - 2、读已提交：事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果因此本事务先后两次读到的数据结果会不一致。
    - 3、可重复读（**MySQL默认**）：在同一个事务里，SELECT的结果是事务开始时时间点的状态，因此，同样的SELECT操作读到的结果会是一致的。但是，会有幻读现象
    - 4、串行化：最高的隔离级别，在这个隔离级别下，不会产生任何异常。事务串行化执行，就像事务是在一个个按照顺序执行一样
- 如何实现可重复读
    - 一般都不修改这个隔离级别，需要清楚是怎么实现的
    - MySQL是通过MVCC机制来实现的，就是多版本并发控制，multi-version concurrency control。
    - 当我们使用innodb存储引擎，会在每行数据的最后加两个隐藏列，一个保存行的创建时间，一个保存行的删除时间，但是这儿存放的不是时间，而是**事务id**，事务id是mysql自己维护的自增的，全局唯一。
    - 事务id，在mysql内部是全局唯一递增的

id	name	创建事务id	删除事务id
1	张三	120	122
2	李四	119	空
2	小李四	122	空

    - 在一个事务内查询的时候，mysql只会**查询创建时间的事务id小于等于当前事务id的行**，这样可以确保这个行是在当前事务中创建，或者是之前创建的；
    - 同时一个行的删除时间的事务id要么没有定义（就是没删除），要么是必当前事务id大（在事务开启之后才被删除）；满足这两个条件的数据都会被查出来。
    - 那么如果某个事务执行期间，别的事务**更新了一条数据**呢？这个很关键的一个实现，其实就是**在innodb中，是插入了一行记录，**然后将新插入的记录的创建时间设置为新的事务的id，同时将这条记录之前的那个版本的删除时间设置为新的事务的id。
- **索引**
    - 什么是索引
        - 索引是帮助mysql高效获取数据的排好序的数据结构

- 索引的优点和缺点
  - 优点:
    - 1.大大加快了查询速度
    - 2.加速表和表之间的连接
    - 3.分组和排序时候速度更快
  - 缺点:
    - 1.时间方面,创建和维护都要花更多时间
    - 2.空间方面,索引要占用更多空间.
- 聚集索引和非聚集索引
  - 聚集索引：表数据文件本身就是**按B+Tree组织的一个索引结构文件，**数据按索引顺序存储，叶子结点存储真实的物理数据
  - 非聚集索引：存储指向真正数据行的指针，记录的是逻辑顺序
- InnoDB索引和MyISAM索引
  - 一是主索引的区别，InnoDB的**数据文件本身就是索引文件。而MyISAM的索引和数据是分开的。**
  - 二是辅助索引的区别：InnoDB的辅助索引data域存储相应记录的主键
- B+树索引和其他索引比较（为什么选择B+树）
  - 二叉树
    - 从算法逻辑来说，二叉查找树的查找速度和比较次数都是最少的
    - 但是我们要逐一加载每一个磁盘页，磁盘页对应着索引数的节点。IO次数会比较多
  - B-树
    - 多路平衡查找树
    - 1.根结点至少有两个子女。
    - 2.每个中间节点都包含k-1个元素和k个孩子，其中 m/2 <= k <= m
    - 3.每一个叶子节点都包含k-1个元素，其中 m/2 <= k <= m
    - 4.所有的叶子结点都位于同一层。
    - 5.每个节点中的元素从小到大排列，节点当中k-1个元素正好是k个孩子包含的元素的值域分划。
    - B-树查询次数比二叉树多，可是相比于磁盘io来说耗时几乎可以忽略不记。节点内部元素多一点仅仅是多几次内存交互，只要不超过磁盘页的大小即可。只要树的高度比较低io次数就会比较少
  - B+树索引
    - 1.有k个子树的中间节点包含有k个元素（B树中是k-1个元素），每个元素**不保存数据，只用来索引**，所有数据都保存在叶子节点。

        - 2.所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
        - B+树与B-树相比的优势
        - 1.非叶子节点不存储data，只存储索引key，一个节点可以放更多索引，减少iO。指针数量变多（有些资料也称为扇出）能减少树的高度
        - 2.叶子结点中包含了全部元素的信息，指针连接起来，关键字的大小自小而大顺序链接。（适合范围查询）
        - 3.更稳定,每次查询到叶子节点
    - hash
        - 等值查询效率高，不能排序，不能进行范围查询
    - 红黑树
        - 增加，删除，红黑树会进行频繁的调整，来保证红黑树的性质，浪费时间
- 为什么InnoDB表必须有主键，
    - 主键索引是InnoDB存储的结构，所以必须有
    - 如果我们定义了主键(PRIMARY KEY)，那么InnoDB会选择主键作为聚集索引、
    - 如果没有显式定义主键，则InnoDB会选择第一个不包含有NULL值的唯一索引作为主键索引、
    - 如果也没有这样的唯一索引，则InnoDB会选择内置6字节长的ROWID作为隐含的聚集索引(ROWID随着行记录的写入而主键递增，这个ROWID不像ORACLE的ROWID那样可引用，是隐含的)。
- 为什么推荐使用整型的自增主键?
    - 索引中存在大量比较（先转换为ASCII码），用整形比较会更快，用整形占用的存储空间更小
    - 自增新增的索引元素一定是比前面大的，不需要重新维护（分裂和平衡）
    - 数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放
    - 因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）
    - 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，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。
- 为什么**非主键索引结构叶子节点存储的是主键值**?

        - 一致性（节省维护成本）和节省存储空间
    - 联合索引的底层存储结构
        - 索引最左前缀原理
        - 每个比较的单位key有多个字段（联合索引的字段）组成，比较大小是逐个字段比较，先比较第一个字段然后是其他的。
    - InnoDB数据页
        - Mysql把页作为管理存储空间的基本单位，是Innodb磁盘管理的最小单位。
        - （在计算机中磁盘存储数据最小单元是扇区，一个扇区的大小是512字节，而文件系统（例如XFS/EXT4）的最小单元是块，一个块的大小是4k，InnoDB中一个页的大小是16K。）
        - 一个页的大小一般是16KB，记录是被储存在页中的
        - 一个页中能存储多少行数据呢？假设一行数据的大小是1k，那么一个页可以存放16行这样的数据。
        - B-树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；
    - 一般来说，应该在这些列上创建索引：
        - （1）在经常需要搜索的列上，可以加快搜索的速度；
        - （2）在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；
        - （3）在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；
        - （4）在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；
        - （5）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；
        - （6）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。
- 三大范式
    - 1.列不可分割 (确保每列保持原子性)
    - 2.不可部分依赖 (确保表中的每列都和主键相关)
    - 3.不可以传递依赖(确保每列都和主键列直接相关,而不是间接相关)
- 触发器、函数、视图、存储过程
    - 触发器
        - 触发器是一种特殊的存储过程，主要是通过事件来触发而被执行的。
        - 使用触发器可以定制用户对表进行【增、删、改】操作时前后的行为,触发器无法由用户直接调用，而知由于对表的【增/删/改】操作被动引发的
    - 函数
        - 是MySQL数据库提供的内部函数(当然也可以自定义函数)。这些内部函数可以帮助用户更加方便的处理表中的数据

- 视图
  - 是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有一个表或者多个表的行或列的子集
  - 对视图的修改会影响基本表。它使得我们获取数据更容易，相比多表查询。（用户通过简单的查询可以从复杂查询中得到结果。）
- 存储过程
  - 存储过程就像我们编程语言中的函数一样
  - 是一组**预编译（效率高）**的SQL语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。
  - 如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。
  - 缺点是移植性差

- MySQL中的日志
  - 持久性：redo log；原子性undo log
  - binlog
    - binlog记录了数据库表结构和表数据变（可以简单理解为：存储着每条变更的SQL语句）
    - 作用：主从复制和恢复数据
  - 重做日志redo
    - 如果在内存中把数据改了，还没来得及落磁盘，而此时的数据库挂了怎么办?
    - 内存写完了，然后会写一份redo log，这份redo log记载着这次在某个页上做了什么修改。顺序IO，写入数据快
    - redo log的作用是为持久化而生的。记录着数据的变更，如果数据已经刷到了磁盘，redo log中的数据就无效了。
  - 回滚日志undo
    - undo log主要有两个作用：回滚和多版本控制(MVCC)
    - undo log主要存储的也是逻辑日志，比如我们要insert一条数据了，那undo log会记录的一条对应的delete日志。我们要update一条记录时，它会记录一条对应相反的update记录。
  - binlog和redolog的比较
    - binlog记载的是update/delete/insert这样的SQL语句，而redo log记载的是物理修改的内容（xxxx页修改了xxx）
    - redo log 记录的是数据的物理变化，binlog 记录的是数据的逻辑变化
    - redo log事务开始的时候，就开始记录每次的变更信息，而binlog是在事务提交的时候才记录。
- 主从复制
  - 主从复制是指一台服务器充当主数据库服务器，另一台或多台服务器充当从数据库服务器，主服务器中的数据自动复制到从服务器之中。

- 读操作可以在所有的服务器上面进行，而写操作只能在主服务器上面进行。
- **异步复制（默认）**
  - 1、master将改变的数 记录在本地的 二进制日志中（binary log）；该过程 称之为：二进制日志事件
  - 2、slave服务器上面启动一个I/O thread（实际上就是一个主服务器的客户端进程），连接到主服务器上面请求读取二进制日志，
  - 3、Slave的IO进程接收到信息后，将接收到的日志内容依次添加到Slave端的relay-log文件的最末端，并将读取到的Master端的 bin-log的文件名和位置记录到master-info文件中，以便在下一次读取的时候从需要的bin-log位置开始往后的日志内容
  - 3、中继日志事件，slave服务器上面开启一个SQL thread定时检查Realy log，如果发现有更改立即把更改的内容在本机上面执行一遍。
- 半同步复制
  - 半同步复制则一定程度上保证提交的事务已经传给了至少一个备库。仅仅保证事务的已经传递到备库上，但是并不确保已经在备库上执行完成了。
  - 在以前的异步（asynchronous）复制中，主库在执行完一些事务后，是不会管备库的进度的。如果备库处于落后，而更不幸的是主库此时又出现Crash（例如宕机），这时备库中的数据就是不完整的。
  - 在主库发生故障的时候，我们无法使用备库来继续提供数据一致的服务了。
- 同步复制
- 主从复制类型
  - 基于语句的复制：
    - 主服务器上面执行的语句在从服务器上面再执行一遍，在MySQL-3.23版本以后支持。
    - 存在的问题：时间上可能不完全同步造成偏差，执行语句的用户也可能是不同一个用户。
  - 基于行的复制：
    - 把主服务器上面改编后的内容直接复制过去，而不关心到底改变该内容是由哪条语句引发的，在MySQL-5.0版本以后引入。
    - 存在的问题：比如一个工资表中有一万个用户，我们把每个用户的工资+1000，那么基于行的复制则要复制一万行的内容，由此造成的开销比较大，而基于语句的复制仅仅一条语句就可以了。
  - 混合类型的复制：
    - MySQL默认使用基于语句的复制，当基于语句的复制会引发问题的时候就会使用基于行的复制，MySQL会自动进行选择。
- slave不开启两个线程会怎么样
  - 首先，一个进程会使复制bin-log日志和解析日志并在自身执行的过程成为一个串行的过程，性能受到了一定的限制，**异步复制的延迟也会比较长**

    - 丢失数据：Slave端从Master端获取bin-log过来之后，需要接着解析日志内容，然后在自身执行。在这个过程中，Master端可能又产生了大量变化并新增了大量的日志。如果在这个阶段Master端的存储出现了无法修复的错误，那么在这个阶段所产生的所有变更都将永远无法找回。
- master的写操作，slaves被动的进行一样的操作，保持数据一致性，那么slave是否可以主动的进行写操作?
    - 假设slave可以主动的进行写操作，slave又无法通知master，这样就导致了master和slave数据不一致了。因此slave不应该进行写操作，至少是slave上涉及到复制的数据库不可以写。实际上，这里已经揭示了读写分离的概念。
- 主从复制中，可以有N个slave,可是这些slave又不能进行写操作，要他们干嘛?
    - 实现数据备份:
    - 类似于高可用的功能，一旦master挂了，可以让slave顶上去，同时slave提升为master。
    - 异地容灾:比如master在北京，地震挂了，那么在上海的slave还可以继续。
    - 主要用于实现scale out,分担负载,可以将读的任务分散到slaves上。
- 主从复制中有master,slave1,slave2,...等等这么多MySQL数据库，那比如一个JAVA WEB应用到底应该连接哪个数据库?
    - 我们在应用程序中可以这样，insert/delete/update这些更新数据库的操作，用connection(for master)进行操作，
    - select用connection(for slaves)进行操作。那我们的应用程序还要完成怎么从slaves选择一个来执行select，例如使用简单的轮循算法。
- 如果MySQL proxy , direct , master他们中的某些挂了怎么办?
    - ***一般都会弄个副***，以防不测。同样的，可以给这些关键的节点来个备份。
- 当master的二进制日志每产生一个事件，都需要发往slave，如果我们有N个slave,那是发N次，还是只发一次?
    - 显 然，应该发N次。实际上，在MySQL master内部，维护N个线程，每一个线程负责将二进制日志文件发往对应的slave。

- 读写分离（分散读写操作压力）
    - 原理：通过数据冗余将读写操作分散到不同的节点上面
    - 基本实现
        - 1）一主一从/一主多从
        - 2）主机负责读写操作，从机负责读操作
        - 3）主机复制数据到从机，每台机器都存储了所有的业务数据
        - 4）写操作发给主机，读操作发给从机
    - master选举
        - zookeerper集群中的某一个zk客户端创建一个临时节点\master，主要是用来存放Master数据库的IP的，然后其他的机器连接\master中的IP的数据库，即为主数据库

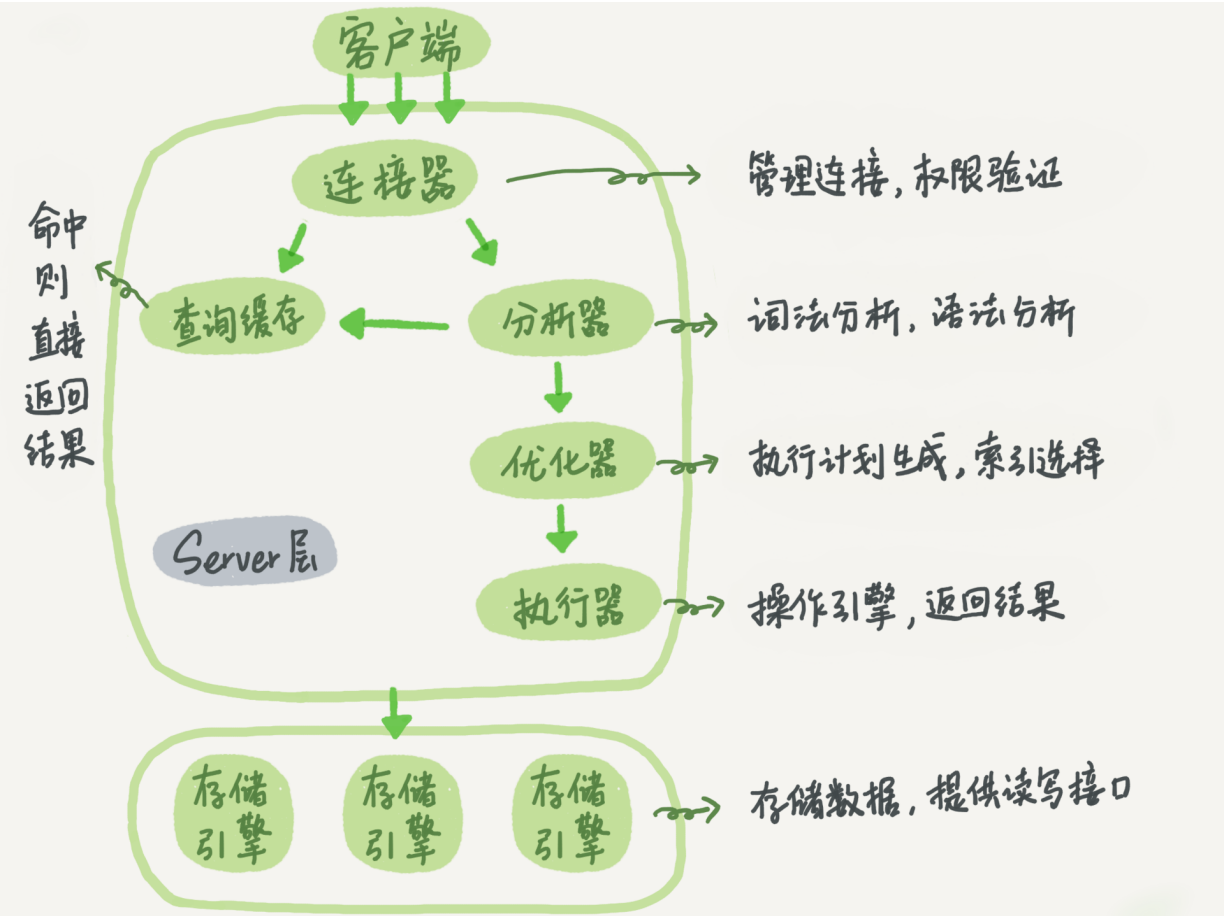
        - 例如当前部署zookeeper1的数据库为主数据库，当因为某些原因导致zookeeper1断开连接，此时临时节点会因为断开连接而消失
        - 其他客户端监到该节点删除的事件后，立马争抢创建\master节点，谁抢到了就将自己的IP放进去，这样新抢的就成了主。
        - 之前挂掉的此时如果重新连接到集群中，也只能是从，等待主挂掉之后进行主的争抢。
    - 使用场景
        - 并发量大，单机不能处理该数量的并发请求
        - 读大于写
        - 实时性要求不严格
    - 造成问题
        - 1.主从复制延迟分配机制
- 分库分表（分散存储压力）
    - 思想：将一个数据库切分为多个部分放到不同的数据库上面，从而缓解单一数据库的性能问题
    - 分库
        - 一个健康的单库并发值你最好保持在每秒 1000 左右，不要太大。那么你可以将一个库的数据拆分到多个库中，访问的时候就访问一个库好了。
    - 分表
        - 单表数据量太大，会极大影响你的 sql 执行的性能，到了后面你的 sql 可能就跑的很慢了。一般来说，就以我的经验来看，单表到几百万的时候，性能就会相对差一些了，你就得分表了。
        - 分表是啥意思？就是把一个表的数据放到多个表中，然后查询的时候你就查一个表。比如按照用户 id 来分表，将一个用户的数据就放在一个表中。然后操作的时候你对一个用户就操作那个表就好了。
    - 单台服务器的瓶颈问题
        - 1.数据量太大，读写性能下降
        - 2.数据库文件大导致数据备份和恢复需要很长时间
        - 3.丢失数据的风险太大
    - 没有其他方案时才使用
        - 1.硬件优化
        - 2.数据库调优
        - 3.引入缓存
        - 4.数据库表优化，重构
        - 5.读写分离或者分库分表
- 分库分表之后id

- 这是分库分表之后你必然要面对的一个问题，就是 id 咋生成？因为要是分成多个表之后，每个表都是从 1 开始累加，那肯定不对啊，需要一个**全局唯一**的 id 来支持。
- 数据库自增id
  - 特意使用一个数据库，系统里每次得到一个 id，都是往一个库的一个表里插入一条没什么业务含义的数据，然后获取一个数据库自增的一个 id。拿到这个 id 之后再往对应的分库分表里去写入。
  - 这个方案的好处就是方便简单，谁都会用；缺点就是单库生成自增 id，要是高并发的话，就会有瓶颈的；如果你硬是要改进一下，那么就专门开一个服务出来，这个服务每次就拿到当前 id 最大值，然后自己递增几个 id，一次性返回一批 id，然后再把当前最大 id 值修改成递增几个 id 之后的一个值；但是无论如何都是基于单个数据库。
  - **适合的场景**：你分库分表就俩原因，**要不就是单库并发太高，要不就是单库数据量太大**；除非是你并发不高，但是数据量太大导致的分库分表扩容，你可以用这个方案，因为可能每秒最高并发最多就几百，那么就走单独的一个库和表生成自增主键即可。
- UUID
  - 好处就是本地生成，不要基于数据库来了；不好之处就是，UUID 太长了、占用空间大，作为主键性能太差了；更重要的是，UUID 不具有有序性，影响B+树索引
- 获取系统当前时间
  - 这个就是获取当前时间即可，但是问题是，并发很高的时候，比如一秒并发几千，会有重复的情况，这个是肯定不合适的。基本就不用考虑了。
  - 适合的场景：一般如果用这个方案，是将当前时间跟很多其他的业务字段拼接起来，作为一个 id，如果业务上你觉得可以接受，那么也是可以的。你可以将别的业务字段值跟当前时间拼接起来，组成一个全局唯一的编号。
- **snowflake算法**
  - 是把一个 64 位的 long 型的 id，1 个 bit 是不用的，用其中的 41 bit 作为毫秒数，用 10 bit 作为工作机器 id，12 bit 作为序列号
    - 1 bit：不用，为啥呢？因为二进制里第一个 bit 为如果是 1，那么都是负数，但是我们生成的 id 都是正数，所以第一个 bit 统一都是 0。
    - 41 bit：表示的是时间戳，单位是毫秒。41 bit 可以表示的数字多达 2^41 - 1，也就是可以标识 2^41 - 1 个毫秒值，换算成年就是表示69年的时间。
    - 10 bit：记录工作机器 id，代表的是这个服务最多可以部署在 2^10台机器上哪，也就是1024台机器。但是 10 bit 里 5 个 bit 代表机房 id，5 个 bit 代表机器 id。意思就是最多代表 2^5个机房（32个机房），每个机房里可以代表 2^5 个机器（32台机器）。
    - 12 bit：这个是用来记录同一个毫秒内产生的不同 id，12 bit 可以代表的最大正整数是 2^12 - 1 = 4096，也就是说可以用这个 12 bit 代表的数字来区分同一个毫秒内的 4096 个不同的 id。
  - 利用这个 snowflake 算法，你可以开发自己公司的服务，甚至对于机房 id 和机器 id，反正给你预留了 5 bit + 5 bit，你换成别的有业务含义的东西也可以的。

        - 这个 snowflake 算法相对来说还是比较靠谱的，所以你要真是搞分布式 id 生成，如果是高并发啥的，那么用这个应该性能比较好，一般每秒几万并发的场景，也足够你用了。
- 语言分类
    - SQL语言共分为四大类：
    - 1. 数据查询语言DQL
        - 数据查询语言DQL基本结构是由SELECT子句，FROM子句，WHERE子句组成的查询块：
        - SELECT、FROM、WHERE
    - 2 .数据操纵语言DML
        - 数据操纵语言DML主要有三种形式：
        - 1) 插入：INSERT
        - 2) 更新：UPDATE
        - 3) 删除：DELETE
    - 3. 数据定义语言DDL
        - 数据定义语言DDL用来创建数据库中的各种对象-----表、视图、索引、同义词、聚簇等如：
        - CREATE TABLE/VIEW/INDEX/SYN/CLUSTER
        - 表 视图 索引 同义词 簇
        - DDL操作是隐性提交的！不能rollback
    - 4. 数据控制语言DCL
        - 数据控制语言DCL用来授予或回收访问数据库的某种特权，并控制数据库操纵事务发生的时间及效果，对数据库实行监视等。如：
        - 1) GRANT：授权。2) ROLLBACK 回滚
        - 3) COMMIT [WORK]：提交。
- 具体语句
    - inner join, outer join, cross join
        - 以A，B两张表为例
        - 1、A left join B
            - 选出A的所有记录，B表中没有的以null 代替
            - right join 同理
        - 2.inner join
            - A,B的所有记录都选出，没有的记录以null代替
        - 3.cross join (笛卡尔积)
            - A中的每一条记录和B中的每一条记录生成一条记录

        - 例如A中有4条，B中有4条，cross join 就有16条记
    - drop、truncate、 delete区别
        - 最基本：
            - 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。（可以回滚）
        - 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。
        - drop语句将删除表的结构
        - 一般而言，drop > truncate > delete
    - varchar和char的区别和使用场景?
        - 1.char的长度是不可变的，而varchar的长度是可变的。定义一个char[10]和varchar[10]。
            - 如果存进去的是‘csdn’,那么char所占的长度依然为10，除了字符‘csdn’外，后面跟六个空格，varchar就立马把长度变为4了，取数据的时候，char类型的要用trim()去掉多余的空格，而varchar是不需要的。
        - 2.char的存取数度还是要比varchar要快得多，因为其长度固定，方便程序的存储与查找。
            - char也为此付出的是空间的代价，因为其长度固定，所以难免会有多余的空格占位符占据空间，可谓是以空间换取时间效率。
            - varchar是以空间效率为首位。
        - 3.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。
        - 4.两者的存储数据都非unicode的字符数据。
- MyISAM和InnoDB引擎的比较
    - MyISAM
        - 不支持外键事务；只支持表级锁，插入数据时，锁定整个表，查表总行数时，不需要全表扫描；其优势是访问的速度快
        - 索引文件和数据文件分开，这样在内存里可以缓存更多的索引，对查询的性能会更好，适用于那种少量的插入，大量查询的场景。
        - 强调的是性能
    - InnoDB（默认）

    - 支持外键，行锁，页级锁(默认)，事务；写的效率差一些，并且会占据更多的磁盘空间。查表总行数时，全表扫描
    - 适合频繁修改以及涉及到安全性问题的应用
    - INNODB的行级锁是有条件的。在where条件没有使用主键时，照样会锁全表。
  - 使用场景
    - MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。
    - InnoDB用于事务处理应用程序，具有众多特性，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。
- 一条mysql语句的执行过程
  - 图解



  - Server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存储引擎。

- MySQL锁
  - 操作类型：
    - a.读锁（共享锁）： 对同一个数据（衣服），多个读操作可以同时进行，互不干扰。（加读锁的时候不能加写锁）
    - b.写锁（互斥锁）： 如果当前写操作没有完毕（买衣服的一系列操作），则无法进行其他的读操作、写操作
  - 操作范围：
    - a.表锁： 一次性对一张表整体加锁。
      - 一般myisam会加表锁，执行查询的时候，会默认加一个表共享锁，也就是表读锁。这个时候其他线程可以来查但是不能写；myisam写的时候会加一个表独占锁，也就是表写锁，其他线程既不能读也不能写。
      - 开销小、加锁快；无死锁；但锁的范围大，容易发生锁冲突、并发度低。
    - b.行锁： 一次性对一条数据加锁。（InnoDB默认）
      - 行锁有共享锁、排他锁
      - insert、update、delete时会自动给那一行加行级排他锁
      - select时什么都不加，因为实现了可重复读，也就是mvcc机制
      - InnoDB存储引擎使用行锁，开销大，加锁慢；容易出现死锁；锁的范围较小，不易发生锁冲突，并发度高
    - c.页锁
      - 表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录
  - mysql死锁
    - 两个线程分别在持有一把锁的时候去获取对方的锁。
    - 排查
      - 查找死锁日志
  - MySQL表级锁的锁模式
    - MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（DML）前，会自动给涉及的表加写锁。
    - 所以对MyISAM表进行操作，会有以下情况：
      - a、对MyISAM表的读操作（加读锁），不会阻塞其他进程（会话）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。
      - b、对MyISAM表的写操作（加写锁），会阻塞其他进程（会话）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。
- MySQL优化/执行计划

- 编写过程：
  - select dinstinct ..from ..join ..on ..where ..group by ...having ..order by ..limit ..
- 解析过程：
  - from .. on.. join ..where ..group by ....having ...select dinstinct ..order by limit ...
- explain模拟优化
  - explain + sql语句
  - 使用EXPLAIN关键字可以模拟优化器执行SQL语句，分析你的查询语句或是结构的性能瓶颈在 select 语句之前增加 explain 关键字，MySQL 会在查询上设置一个标记，执行查询会返回执行计划的信息，而不是执行这条SQL
- explain中的列
  - id : 编号
    - id的顺序是按 select 出现的顺序增长的。
    - id值越大越优先查询
      - 本质：在嵌套子查询时，先查内层 再查外层
    - id相同则从上往下执行
      - 因数据的个数改变而改变：笛卡尔积
      - 数据量少的优先执行
  - **select_type： 查询类型**
  - table： 表
    - 表示 explain 的一行正在访问哪个表
  - **type ： 类型（索引类型）**
    - **要对type进行优化的前提：有索引**
    - 关联类型或访问类型，即MySQL决定如何查找表中的行，查找数据行记录的大概范围。
    - 依次从最优到最差分别为：system > const > eq_ref > ref > range > index > ALL
    - system,const只是理想情况；一般来说，得保证查询达到range级别，最好达到ref
    - system（忽略）
      - **只有一条数据的系统表**；或 衍生表只有一条数据的主查询
    - const
      - 仅仅能查到一条数据的SQL ,用于Primary key 或unique索引 （类型 与索引类型有关）
    - eq_ref
      - 唯一性索引：对于每个索引键的查询，返回匹配唯一行数据（有且只有1个，不能多 、**不能0**），常见于唯一索引或者主键索引
    - ref

        - 非唯一性索引，**对于每个索引键的查询**，返回匹配的所有行（0，多）
    - range
        - 检索指定**范围**的行 ,where后面是一个范围查询(between ,> < >=
        - 特殊:in有时候会失效 ，从而转为 无索引all)
    - index
        - 查询全部索引中数据
        - explain select tid from teacher ; --tid 是索引， 只需要扫描索引表，不需要所有表中的所有数据
    - all
        - 查询全部表中的数据
        - explain select cid from course ; --cid不是索引，需要全表所有，即需要所有表中的所有数据
    - NULL
        - mysql能够在优化阶段分解查询语句，在执行阶段用不着再访问表或索引。例如：在索引列中选取最小值，可以单独查找索引来完成，不需要在执行时访问表
- possible_keys 预测用到的索引
    - possible_keys： 可能用到的索引，是一种预测，不准。
- **key ： 实际使用的索引**
    - 显示mysql实际采用哪个索引来优化对该表的访问。
    - 如果没有使用索引，则该列是 NULL
- key_len： 实际使用索引的长度
    - 用于判断复合索引是否被完全使用 （a,b,c）
    - 在索引里使用的字节数，通过这个值可以算出具体使用了索引中的哪些列
    - 在utf8：1个字符站3个字节 ； gbk:1个字符2个字节；latin:1个字符1个字节
    - 如果索引字段可以为Null,则会使用1个字节用于标识。2个字节表示可变长度（varchar：22）
- ref :表之间的引用
    - 指明当前表所 参照的 字段。
    - 常量，const
- rows： 实际通过索引而查询到的 数据个数
    - 被索引优化查询的数据个数 (实际通过索引而查询到的 数据个数)
    - 估计要读取并检测的行数，注意这个不是结果集里的行数。
- **Extra :额外的信息**

- (i).using filesort： 性能消耗大；需要“额外”的一次排序（查询） 。常见于 **order by** 语句中。
  - 排序的前提：先查询
  - 单索引，如果排序和查找不是同一个字段，则会出现using filesort；
  - select * from table where a1 ='1' order by a2 ;
  - 对于联合索引，不能跨列（最佳左前缀）
  - explain select *from test02 where a1='' order by a3 ; --using filesort
  - explain select *from test02 where a2='' order by a3 ; --using filesort
  - explain select *from test02 where a1='' order by a2 ;
  - explain select *from test02 where a2='' order by a1 ; --using filesort
- (ii). using temporary:性能损耗大 ，用到了临时表。一般出现在**group by** 语句中。
  - explain select a1 from test02 where a1 in ('1','2') group by a1 ;
  - explain select a1 from test02 where a1 in ('1','2') group by a2 ; --using temporary
  - 避免：查询那些列，就根据那些列 group by .
- (iii). using index :性能提升; 索引覆盖（覆盖索引）。
  - 原因：不读取原文件，**只从索引文件中获取数据** （不需要回表查询）。只要使用到的列 全部都在索引中，就是索引覆盖using index
- (iii).using where （需要回表查询）
  - 假设age是索引列
  - 但查询语句select age,name from ...where age =...,
  - 此语句中必须回原表查Name，因此会显示using where.
- (iv). impossible where： where子句永远为false

- 索引失效场景
  - 优化
    - 单表优化
      - 根据SQL实际解析的顺序，调整索引的顺序
    - 两表优化（多表）
      - 索引往哪张表加？ -小表驱动大表 小表.id = 大表.id
      - 左表连接给左表加索引，右表连接给右表加索引
      - for循环中外循环小内循环大会更快
      - a.小表驱动大表 b.索引建立在经常查询的字段上
  - 索引失效的原因
    - 1、查询条件中有or、not in、not exist、不等于!=等
    - 2、在 where 子句中对字段进行 null 值判断，将导致引擎放弃使用索引而进行全表扫描

- 3、like查询是以%开头
- 4、在索引上进行任何操作（计算、函数、类型转换），索引失效
- 5、复合索引，跨列或无序使用（违背最佳左前缀原则）
- 6、如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引
- 7、使用类型转换（显示、隐式），否则索引失效

- mysql语句优化
  - 1、exist和in
    - 如果**主查询的数据集大**，则使用In ,效率高。
    - 如果**子查询的数据集大**，则使用exist,效率高
    - in是遍历一次子查询
    - exist语法： 将主查询的结果，放到子查需结果中进行条件校验（看子查询是否有数据，如果有数据 则校验成功） ，如果 复合校验，则保留数据；
  - 2、子查询变成left join
  - 3、用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤
  - 4、对多个字段进行等值查询时，联合索引
  - 5、limit 分布优化，先利用ID定位，再分页
  - 6、or条件优化，多个or条件可以用union all对结果进行合并（union all结果可能重复）
  - 7、不必要的排序
  - 8、避免嵌套查询
  - 9、order by优化
- 慢查询日志
  - MySQL提供的一种日志记录，用于记录MySQL种响应时间超过阀值的SQL语句（long_query_time，默认10秒）
  - 慢查询日志默认是关闭的；建议：开发调优是 打开，而 最终部署时关闭。
  - 开启
    - 临时开启：set global slow_query_log = 1；--在内存种开启
    - 永久开启：/etc/my.cnf 中追加配置：slow_query_log_file=/var/lib/mysql/localhost-slow.log
  - 慢查询阈值
    - show variables like '%long_query_time%' ;
    - 临时设置阀值：
      - set global long_query_time = 5 ; --设置完毕后，重新登陆后起效 （不需要重启服务）

        - 永久设置阀值：
            - /etc/my.cnf 中追加配置：long_query_time=3
    - 查询超过阀值的SQL
        - 1、show global status like '%slow_queries%' ;
        - 2、通过日志 查看具体的慢SQL。
            - cat /var/lib/mysql/localhost-slow.log
- 防止sql注入
    - SQL注入是一种代码注入技术，用于攻击数据驱动的应用，恶意的SQL语句被插入到执行的实体字段中
    - mybatis
        - #{}：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。