# MySQL

## 写在前面

MySQL已经是后端面试中的必问知识点，核心知识点集中在索引、SQL、事务、锁、调优、主从复制、分库分表等等，理解MySQL的原理，无论是应对面试，还是日常开发，都非常有用。

本文将对面试中的常考MySQL知识点做一些整理，并且尽可能附上我自己的答案，答案随着我掌握知识的深度而进行不断更新，当然内容难免有误，欢迎指正。

## 基础篇

### Innodb和MyISAM的区别

* MyISAM是非事务安全的，而InnoDB是事务安全的
* MyISAM锁的粒度是表级的，而InnoDB支持行级锁
* MyISAM支持全文类型索引，而InnoDB不支持全文索引
* MyISAM相对简单，效率上要优于InnoDB，小型应用可以考虑使用MyISAM
* MyISAM表保存成文件形式，跨平台使用更加方便
1. MyISAM管理非事务表，提供高速存储和检索以及全文搜索能力，如果再应用中执行大量select操作，应该选择MyISAM
2. InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量insert和update操作，应该选择InnoDB

mysql 数据库引擎: http://www.cnblogs.com/0201zcr/p/5296843.html

MySQL存储引擎－－MyISAM与InnoDB区别: https://segmentfault.com/a/1190000008227211

### char和varchar的区别

todo

### MySQL用什么编码

utf8mb4

todo

### timestamp和datetime的区别

todo

## 索引篇

### 为什么要使用索引

避免全表扫描查找数据，提升查询效率

http://blog.codinglabs.org/articles/theory-of-mysql-index.html

https://tech.meituan.com/2014/06/30/mysql-index.html

### 什么样的信息能成为索引

有一定区分度的字段

### 索引的数据结构

从二叉查找数上阵 - 时间复杂度O(logn)，容易退化成O(n) - 深度每增加1，就会增加一次IO，红黑树也这样

B-tree - 跟节点至少包括两个孩子 - 树中每个节点最多含有m个孩子（m>=2） - 除跟节点和叶节点外，其他每个节点至少有ceil(m/2)个孩子 - 所有叶子都位于同一层

让每个索引快尽可能存储更多信息

B+-Tree - 非叶子节点的子树指针与关键字个数相同 - 非叶子节点的子树指针P[i]，指向关键字值（K[i],K[i+1]）的子树 - 非叶子节点仅用来索引，数据都保存在叶子节点中 - 所有叶子节点均有一个指针指向下一个节点（方便做范围统计）

B+Tree更适合用来做存储索引 - B+树的磁盘读写代价更低（非叶子节点不存数据，能读的量就越多） - B+树的查询效率更加稳定（任何关键字查找一定要走到叶子节点的路） - B+树更有利于对数据库的扫描（范围查询有更高的性能）

Hash索引 - 优点：效率高 - 缺点： - 仅仅满足=、 in，不能使用范围查询 - 无法被用来避免数据的排序操作 - 不能利用部分索引键查询（组合索引） - 不能避免表扫描 - 遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高

### 密集索引和稀疏索引的区别

Innodb - 若一个主键被定义，该主键则作为密集索引 - 如没有主键被定义，该表的第一个唯一非空索引则作为密集索引 - 若不满足以上条件，innodb内部会生成一个隐藏主键（密集索引） - 非主键索引存储相关键位和其对应的主键值，包含两次查找

myIsam，只要一次，跟innodb不同

inndob的索引和数据是存一块的，myIsam是分开的

### 联合索引的最左匹配原则

* 最左前缀匹配原则：mysql会一直向右匹配直到遇到范围查询（>,<,between,like）就停止匹配，比如a=3 and b=4 and c>5 and d=6，如果建立（a，b,c,d）顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整
* =和in可以乱序，比如a=1 and b=2 and c=3 建立（a,b,c）索引可以任意顺序，mysql的查询优化器会帮你优化所以可以识别的形式

### 索引是建立的越多越好吗

* 数据量小的表不需要建立索引，建立会增加额外的索引开销
* 数据变更需要维护索引，因此更多的索引意味着更多的维护成本
* 更多的索引意味着也需要更多的空间

### 为什么不建议用UUID为主键

* 数字的比较比字符串要快很多
* 非自增

### 什么样的查询语句用不到索引

* 函数计算
* 数据类型转换
* like %开头
* or运算
* 复合索引，没有左边字段

### 索引如何优化

* 经常被查询的区分度高的列做索引（比如status，区分度低，降级为全表扫描）（20-40%左右，70%没意义）
* 做左原则（a=3 and b>4 and c=5 是用到a和b，c不能用在范围之后。a=3 and b like "kk%" and c=5使用到 a,b,c）
* 回盘排序（排序列联合索引）
* 覆盖索引(select a,b from , 减少一跳)
* 小表驱动大表

### 索引形态分类

* 聚簇索引：直接通过索引路由找到数据。聚簇索引是有一个
* 非聚簇索引：通过磁盘寻址去找到数据。非聚簇索引有很多

不会无限制增加非聚簇索引，增加insert,delete开销

### 数据库索引，为什么不使用二叉树：

1. 平衡二叉树必须满足（所有节点的左右子树高度差不超过1）。执行插入还是删除操作，只要不满足上述条件，就要通过旋转来保持平衡，而旋转是非常耗时的，所以AVL树适合用于查找多的情况。
2. 二叉树的数据结构，会导致“深度”，比较深，这种“瘦高”的特性，加大了平均查询的磁盘IO次数，随着数据量的增多，查询效率也会受到影响；

### B+ 树和 B 树在构造和查询性能上有什么差异呢？

B+ 树的中间节点并不直接存储数据。 - 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+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节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，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。 - 由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

### 联合索引的结构

todo

## SQL篇

### 如何定位并优化慢sql

* 根据慢查询日志定位慢查询sql
* 使用explain等工具分析sql
* 修改sql或者尽量让sql走索引

long\_query\_time show\_query\_log show\_query\_log\_file

explain关键字段

type: system>const>eq\_ref>ref>fulltext>ref\_or\_null>index\_merge>unique\_subquery>index\_subquery>range>index>all

all表明走的是全表扫描，需要优化

extra: 出现下面两个意味着MySQL根本不能使用索引，效率会受到重大影响。应尽可能对此进行优化。 - using filesort:表示MySQL会对结果使用一个外部索引排序，而不是从表里按索引次序读到相关内容。可能在内存或者磁盘上进行排序。MySQL中无法利用索引完成的排序操作称为“文件排序” - using temporary:表示MySQL在对查询结果排序时使用临时表。常见于排序order by和分组查询group by

### count(\*),count(1),count(字段)的区别

todo

### 写SQL的题目

* group by
* having
* 统计相关：count,sum,max,min,avg

group by - 满足select子句中的列名必须为分组列或列函数 - 列函数对于group by子句定义的每个组各返回一个结果 - 如果用group by，那么你的select语句中的列要么是你group by里用到的列，要么就是带有之前我们说的如sum,min等列函数的列 - select student\_id, count(course\_id), sum(score) from score group by student\_id - 要注意：select里面不能有group by里面没出现的列 - select s.student\_id, stu.name, count(s.count\_id), sum(s.score) from score s, student stu where s.student\_id = stu.student.id group by s.student\_id - group by里出现某个表的子弹，select里面的列要么是该group by里出现的列，要么是别的表的列或者带有函数的列。

Having - 通常与group by子句一起使用 - where过滤行，having过滤组 - 出现在同一SQL的顺序：where > group by > having - 查询平均成绩大于60分的同学的学号和平均成绩 - select student\_id,avg(score) from score group by student\_id having avg(socre) > 60 - 查询没有学全所有可的同学的学号、姓名 - select stu.student\_id,stu.name from student stu, score s where stu.student\_id=s.student\_id group by s.student\_id having count(*) < (select count(*) from course)

### 内连接，外连接

todo

### 外键的危害

形成闭环结构，数据难以维护

## 事务篇

### MySQL事务的四大特性

ACID - 原子性（Atomic） - 一致性（Consistencey） - 隔离性（Isolation）：事务之间不互相影响 - 持久性（Durability）：事务一旦提交，对数据库的修改是永久的

### MySQL如何实现事务

* 原子性：通过undo log实现的。每条数据变更都伴随一条undo log日志的生成，当系统发生错误或执行回滚根据undo log做逆向操作
* 持久性：通过redo log实现的。redo log记录了数据的修改日志。数据持久化到磁盘，先是储存到缓冲池里，然后缓冲池中的数据定期同步到磁盘中，如果系统宕机，可能会丢失数据，系统重启后会读取redo log恢复数据
* 隔离性：mysql数据库通过MVCC + next-key机制实现了隔离性
* 一致性：以上3大特性，保障了事务的一致性

### 事务隔离级别以及各级别下的并发访问问题

事务并发访问引起的问题以及如何避免 - 更新丢失：MySQL所有事务隔离级别在数据库层面上均可避免 - 脏读——READ-COMMITED，事务隔离级别以上可避免。事务允许读到未提交的事务，前一个事务如果回滚，会出错 - 不可重复度——REPEATABLE-READ，事务隔离级别以上可避免。一个事务读的时候，另一个事务在修改，导致前面的事务读的两个数据不一致 - 幻读——SERIALIZABLE，事务隔离级别可避免。一个事务在更新的时候，如果另一个事务插入或删除行，第一个事务会发现更新的数目多了

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事务隔离级别（从低到高） | 更新丢失 | 脏读 | 不可重复读 | 幻读 |
| 未提交读 | 避免 | 发生 | 发生 | 发生 |
| 已提交读（Oracle默认） | 避免 | 避免 | 发生 | 发生 |
| 可重复读 （MySQL默认） | 避免 | 避免 | 避免 | 发生 |
| 串行化 | 避免 | 避免 | 避免 | 避免 |

出于性能考虑，事务隔离级别越高，越容易降低并发度

事务隔离级越高，串行化越高，并发度越低

彻底理解事务 http://www.hollischuang.com/archives/898

### 什么是MVCC

全称是Multi-Version Concurrent Control，即多版本并发控制，在MVCC协议下，每个读操作会看到一个一致性的snapshot，并且可以实现非阻塞的读。MVCC允许数据具有多个版本，这个版本可以是时间戳或者是全局递增的事务ID，在同一个时间点，不同的事务看到的数据是不同的。

MySQL的innodb引擎是如何实现MVCC的 innodb会为每一行添加两个字段，分别表示该行创建的版本和删除的版本，填入的是事务的版本号，这个版本号随着事务的创建不断递增。在repeated read的隔离级别（事务的隔离级别请看这篇文章）下，具体各种数据库操作的实现：

select：满足以下两个条件innodb会返回该行数据： 该行的创建版本号小于等于当前版本号，用于保证在select操作之前所有的操作已经执行落地。 该行的删除版本号大于当前版本或者为空。删除版本号大于当前版本意味着有一个并发事务将该行删除了。 insert：将新插入的行的创建版本号设置为当前系统的版本号。 delete：将要删除的行的删除版本号设置为当前系统的版本号。 update：不执行原地update，而是转换成insert + delete。将旧行的删除版本号设置为当前版本号，并将新行insert同时设置创建版本号为当前版本号。 其中，写操作（insert、delete和update）执行时，需要将系统版本号递增。

由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb会开启一个后台线程执行清理工作，具体的规则是将删除版本号小于当前系统版本的行删除，这个过程叫做purge。

通过MVCC很好的实现了事务的隔离性，可以达到repeated read级别，要实现serializable还必须加锁。

参考：MVCC浅析http://blog.csdn.net/chosen0ne/article/details/18093187

### InnoDB可重复读隔离级别下如何避免幻读

表象：快照读（非阻塞读）——伪MvCC

内在：next-key锁（行锁+gap锁）

在可重复读的情况下，InnoDB 可以通过 Next-Key 锁 +MVCC 来解决幻读问题。

在读已提交的情况下，即使采用了 MVCC 方式也会出现幻读。如果我们同时开启事务 A 和事务 B，先在事务 A 中进行某个条件范围的查询，读取的时候采用排它锁，在事务 B 中增加一条符合该条件范围的数据，并进行提交，然后我们在事务 A 中再次查询该条件范围的数据，就会发现结果集中多出一个符合条件的数据，这样就出现了幻读。

出现幻读的原因是在读已提交的情况下，InnoDB 只采用记录锁（Record Locking）。这里要介绍下 InnoDB 三种行锁的方式： - 记录锁：针对单个行记录添加锁。 - 间隙锁（Gap Locking）：可以帮我们锁住一个范围（索引之间的空隙），但不包括记录本身。采用间隙锁的方式可以防止幻读情况的产生。 - Next-Key 锁：帮我们锁住一个范围，同时锁定记录本身，相当于间隙锁 + 记录锁，可以解决幻读的问题。

在隔离级别为可重复读时，InnoDB 会采用 Next-Key 锁的机制，帮我们解决幻读问题。

还是这个例子，我们能看到当我们想要插入球员艾利克斯·伦（身高 2.16 米）的时候，事务 B 会超时，无法插入该数据。这是因为采用了 Next-Key 锁，会将 height>2.08 的范围都进行锁定，就无法插入符合这个范围的数据了。然后事务 A 重新进行条件范围的查询，就不会出现幻读的情况。

https://time.geekbang.org/column/article/120351

### redo log，undo log，bin log

* binlog是二进制文件，记录了对数据库执行更改的所有操作，不包括 select、show，因为这两个操作没有对数据本身做修改。但是若操作了数据，但是数据没有发生变化，也会记录到binlog。常用来数据恢复，数据备份。
* redo log又叫做重做日志文件，记录了事务的修改，不管事务是否提交都记录下来。在实例和介质失败时，InnoDB存储引擎会使用redo log恢复到之前的状态，保证数据的完整性

## 锁篇

### MyISAM与Innodb关于锁方面的区别是什么

* MyISAM默认用的是表级锁，不支持行级锁
* Innodb默认用的是行级锁，也支持表级锁

myISAM里查询的时候，会对表加上表级的读锁

### MyISAM的适合场景

* 频繁执行全表的count语句
* 对数据进行增删改查的频率不高，查询非常频繁
* 没有事务

### Innodb适合场景

* 数据增删改查都相当频繁。增删改都是某些行被锁，避免了阻塞。而不是MyISAM，每次增删改查都是锁整张表
* 可靠性要求比较高，要求支持事务

### 数据库锁的分类

* 按锁的粒度划分，可分为表级锁，行级锁，页级锁
* 按锁级别划分，可分为共享锁，排他锁
* 按加锁方式划分，可分为自动锁，显示锁
* 按操作划分，可分为DML锁（增删改查），DDL锁
* 按使用方式划分，可分为乐观锁，悲观锁（程序中也常见）

悲观锁：保守机制，先取锁再访问

乐观锁：提交时才对数据的冲突进行检测（版本号，时间戳） - 先读取数据，得到的version值为versionValue select version from test where id =2 - 每次更新表里的字段时，为了防止发生冲突，先去检查version再做更新，更新成功的话version+1 update test set money = 123,version = 0+1 where version=0 and id=2

### 乐观锁和悲观锁

* 悲观锁是先获取锁再进行操做。一锁二查三更新。select for update
* 乐观锁先修改，更新的时候发现数据已经变了就回滚。check and set
* 使需要根据响应速度、冲突频率、重试代价来判断使用哪一种

乐观锁与悲观锁的具体区别: http://www.cnblogs.com/Bob-FD/p/3352216.html

### MYSQL的读写方式有哪些

1. 快照读

select \* from user where \*\* group by \*\* 读的是事务开启时的数据

整个数据库都是依赖mvcc的版本机制

1. 当前读

select \* from user where \*\* group by \*\* for update

update/delete

会阻塞

### 当前读的锁机制

行锁 - 主键锁行 - 普通索引，锁普通索引的行，可能有很多 - 无索引，表锁

所有的select for update都是行锁

表锁 间隙锁

唯一索引不会产生间隙锁（推荐）

### 什么是意向锁

todo

## 主从复制篇

### 谈一下MySQL的主从复制

主从复制 - 一主多从 - 读库延迟问题处理 - 主从切换处理

主从复制默认是异步的方案，master和slave只能是最终一致性，不能是强一致性

应用层面做让步 - loading页面 - 强制路由到master上

目前主从同步基本控制在1ms以内，例外原因： - 网络 - 负载压力

主从切换 - 有风险 - 半同步方案，至少其中一个slave同步完后，才提交 - 只需dba寻找binlog最新的slave，提升为master

半同步也容易导致slave比master多一条记录的情况 - slave返回master时ack丢失

### 主从复制原理

todo

### 生产环境如何对MySQL做热切换

todo

## 分库分表篇

### 谈一下MySQL的分库分表

* 垂直拆分
* 水平拆分
* 多主多从

join的表不能做跨库

垂直拆分 - 根据业务

水平拆分 - 路由位 - 时间戳

### MySQL多主多从

* 数据分片
* 分片维度
* 分片冗余一致性保障
* 无迁移扩展

数据分片 - hash+mode分片

MySQL数据分片维度 - 固定路由位（比如用户id，路由到唯一的数据库） - 时间自增分片（2019年所有数据放在一个数据库）

MySQL数据分片冗余 - 用户订单 - 商户订单 因为是按用户路由的，如果商户想查所有订单就要遍历所有的库，因此要冗余 - 因为通过rocketmq去制造冗余 - 监听binlog传过去

MySQL数据无迁移扩展 - mod位数据迁移 - 弹性自增（order id>xx走新规则）

### 一致性原理

* 强一致性
* 弱一致性
* 最终一致性

CAP - C：一致性 - A：可用性 - P：分片性

Base - Basic available：基本可用 - S：软状态 - E：最终一致性

mysql主从同步时有半同步的机制 分布式条件下，要保证强一致性，只有二阶段提交 事务处理接收者维护自己的状态，如果是半同步状态，外部client要查数据时会把block掉

mysql同步复制，就抱着了强一致性

### 分库分表后，ID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： - UUID：不适合作为主键，因为太⻓了，并且⽆序不可读，查询效率低。⽐᫾适合⽤于⽣成唯⼀的名字的标示⽐如⽂件的名字。 - 数据库⾃增 id : 两台数据库分别设置不同步⻓，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。 - 利⽤ redis ⽣成 id : 性能⽐᫾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。 - Twitter的snowflake算法 ：Github 地址：https://github.com/twitter-archive/snowflake。 - 美团的Leaf分布式ID⽣成系统 ：Leaf 是美团开源的分布式ID⽣成器，能保证全局唯⼀性、趋势递增、单调递增、信息安全，⾥⾯也提到了⼏种分布式⽅案的对⽐，但也需要依赖关系数据库、Zookeeper等中间件。感觉还不错。美团技术团队的⼀篇⽂章：https://tech.meituan.com/2017/04/21/mt-leaf.html 。 ......

## MySQL调优篇

### 什么是数据库连接池?为什么需要数据库连接池?

池化设计应该不是⼀个新名词。我们常⻅的如java线程池、jdbc连接池、redis连接池等就是这类设计的代表实现。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，如创建线程的开销，获取远程连接的开销等。就好⽐你去⻝堂打饭，打饭的⼤妈会先把饭盛好⼏份放那⾥，你来了就直接拿着饭盒加菜即可，不⽤再临时⼜盛饭⼜打菜，效率就⾼了。除了初始化资源，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。

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。

### MySQL的调优手段

业务需求 - 不合理的需求，可能造成很多问题 - 拨乱反正

系统架构 - 做架构设计的时候，应充分考虑业务的实际情况，考虑好数据库的各种选择 - 读写分离？高可用？实例个数？分库分表？用什么数据库？

SQL及索引 - 根据需求编写良好的SQL, 并去创建足够高效的索引

表结构 - 设计良好的表结构

数据库参数设置 - 设置合理的数据库性能参数

系统配置 - 操作系统提供了各种资源使用策略，设置合理的配置，以便于数据库充分利用资源

硬件 - 选用什么配置的机器

### explain用法

todo

* system：仅一行
* const：主键or唯一键的常量等值查询
* eq\_ref:主键or唯一键的扫描或关联查询
* ref:非唯一索引的常量等值查询
* range:索引的范围查询
* index:索引全查询
* all:遍历表查询

优化到至少range范围

### 如何做索引优化

最左前缀原则

需要创建索引的场景 - select语句，频繁作为where条件的字段 - update/delete语句的where条件 - 需要分组、排序的字段 - distinct所使用的字段 - 字段的值有唯一性约束 - 对于多表查询，连接字段应创建索引

不建议创建索引的场景 - where子句里用不到的字段 - 表的记录非常少 - 有大量重复数据，选择性低 - 频繁更新的字段，如果创建索引要考虑其索引维护开销

### 索引失效和解决方案

* 索引字段不独立
	+ 索引字段进行了表达式计算
	+ 索引字段是函数的参数
* 使用了左模糊
* 使用or查询的部分字段没有索引（分别为两个字段创建索引）
* 字符串条件未使用''引起来（规范地编写SQL）
* 不符合最左前缀原则的查询
* 索引字段建议添加NOT NUll约束
	+ 单列索引无法存储null值，复合索引无法存储全为null的值
	+ 查询时，采用is null条件时，不能利用到索引，只能全表扫描
* 隐式转换导致索引失效

### 索引调优技巧

#### 长字段的调优技巧

* 引入hash字段，作为索引
* 使用前缀索引

#### 单列索引 VS 组合索引

* SQL存在多个条件，多个单列索引，会使用索引合并
* 如果出现索引合并，往往说明索引不够合理
* 如果SQL暂时没有性能问题，暂时可以不管
* 组合索引要注意索引列顺序【最左前缀原则】

#### 覆盖索引

尽量只返回想要的字段 - 使用覆盖索引 - 减少网络传输的开销

#### 重复索引、冗余索引、未使用的索引

重复索引 - 在相同的列上按照相同的顺序创建的索引 - 尽量避免重复索引，如果发现重复索引应该删除

冗余索引 - 如果已经存在索引index(A,B)，又创建了index(A)，那么index(A)就是index(A，B)的冗余索引

未使用的索引 - 某个索引根本未曾使用

#### JOIN优化

驱动表vs被驱动表 - 外层循环的表是驱动表，内层循环的表是被驱动表

Join调优原则 - 用小表驱动大表 - 一般无需人工考虑，关联查询优化器会自动选择最优的执行顺序 - 如果优化器抽风，可使用STRAIGHT\_JOIN - 如果有where条件，应当要能够使用索引，并尽可能地减少外层循环的数据量 - join的字段尽量创建索引 - join字段的类型要保持一致 - 尽量减少扫描的行数（explain-rows） - 尽量控制在百万以内（经验之谈，仅供参考） - 参与join的表不要太多 - 阿里编程规约建议不超过3张 - 不要以编写复杂SQL为荣，在业务代码里处理 - 如果被驱动表的join字段用不了索引，且内存较为充足，可以考虑把join buffer设置得大一些

#### limit优化

limit 300000, 10 rows 299999

方案1：覆盖索引 方案2：覆盖索引+join 方案3：覆盖索引+子查询 方案4：范围查询+limit语句 方案5：如果能获得起始主键值&结束主键值 方案6：禁止传入过大的页码

#### count优化

当没有非主键索引时，会使用主键索引 如果存在非主键索引的话，会使用非主键索引 如果存在多个非主键索引，会使用一个最小的非主键索引

count(*)和count(1)没有区别 count(*)不会排除为null的行，而count(字段)会排除

如果没有特殊需求，尽量用count(\*)

* 创建一个更小的非主键索引
* 把数据库引擎换成MyISAM——实际项目用的很少，一般不会修改数据库引擎
* 汇总表 table[table\_name, count]
	+ 好处：结果比较准确
	+ 缺点：增加了维护成本
* 缓存
	+ 优点：性能比较高，结果比较准确，有误差但是比较小
	+ 缺点：引入了额外的组件，增加了架构的复杂度

#### group by语句调优

todo

### 表结构设计优化

第一范式：原子性 第二范式：互不依赖 第三范式：不存在传递依赖

表设计原则 - 字段少而精，建议20个以内，超过可以拆分 - 大字段独立出去 - 尽量用小型字段 - 避免使用允许为NULL的字段 - 合理平衡范式与冗余 - 如果数据量非常大，考虑分库分表

### 提高MySQL读写性能的方法

单机数据库 - 查询优化 - 批量写 - 索引优化 - innodb相关优化

查询优化 - 主键查询：千万条记录 1-10ms - 唯一索引：千万条记录 10-100ms - 非唯一索引：千万条记录 100-1000ms - 无索引：百万条记录 1000ms+

批量写 - for each(insert into table values(1))——效率低 - execute once insert into table values 1,2,3,4 - sql编译N次和1次的时间和空间复杂度 - 网络消耗的时间复杂度 - 磁盘寻址的复杂度

单机配置优化 - max\_connection=1000 增加最大连接数，默认为100 - innodb\_file\_per\_table=1 可以存储每个innodb表和他的索引在自己的文件中 - innodb\_buffer\_pool\_size=1G 缓存池大小，设置为当前数据库服务内存的60%-80% - innodb\_log\_file\_size=256m 一般取256m可以兼顾性能和recovery的速度，写满后只能切换日志靠buffer存储 - innodb\_log\_buffer\_size=16m - innodb\_flush\_log\_at\_trx\_commit=2（这个经常考） - 1：日志缓冲写到日志文件，对日志文件做到磁盘操作的刷新。Truly ACID。速度慢。 - 2：日志缓冲写到系统缓冲，但不对日志文件做到磁盘操作的刷新。然后根据innodb\_flush\_log\_at\_timeout（默认为1s）时间flush disk只有操作系统崩溃或者停电才会删除最后一秒的事务，不然不会丢失事务 - 0时，效率更高，但安全性差。每秒才write日志，任何mysqld进程的崩溃会删除崩溃前最后一秒的事务 - innodb\_data\_file\_path=ibdata1:1G;ibdata2:1G;ibdata3:1G:autoextend 指定表数据和索引存储的空间，可以使一个或者多个文件

### 分布式应用的性能优化

主从扩展 - 开启bin\_log - 设置主从同步账号，配置主从同步

任何MySQL主从切换不靠谱，必定会比主库慢，就算阿里也不敢切

主从作用： - 备份 - 读写分离

MySQL提供了半同步的机制，至少一台返回确认，才成功