什么是索引

索引有什么好处

- 1.提高数据的检索速度，降低数据库IO成本：使用索引的意义就是通过缩小表中需要查询的记录的数目从而加快搜索的速度
- 2.降低数据排序成本，降低CPU消耗：索引之所以查的快，是因为先将数据排好序，弱该字段正好需要排序，则正好降低排序成本

索引有什么缺点

- 占用存储空间：索引实际上也是一张表，记录了主键与索引字段，一般以索引文件的形式存储在磁盘上。
- 降低更新表的速度：表的数据发生了变化，对应的索引页需要一起变更，从而降低更新速度，否则索引指向的物理数据可能不对，这也是索引失效的原因之一

索引的类型

- 1、普通索引：最基本的索引，没有任何约束
- 2、唯一索引：与普通索引类似，没有任何约束
- 3、主键索引：特殊的唯一索引，不允许有空值
- 4、复合索引：将多个列组装在一起，可以覆盖多个列
- 5、外键索引：只有InnoDB类型的表才可以使用外键索引，保证数据的一致性，完整性和实现级联操作
- 6、全文索引：MySQL 自带的全文索引只能用于 InnoDB、MyISAM ，并且只能对英文进行全文检索，一般使用全文索引引擎。

MySQL 索引的“使用”注意事项？

- 1、应尽量避免在WHERE自居中使用!=或<>操作符，否则将引擎放弃使用索引二进行扫描，优化器将无法通过索引来确定将要命中的行数，因此需要搜索该表的所有行（is null 也是不可以使用索引）
- 2、应尽量避免在WHERE字句中使用OR来链接条件，否则将导致四年轻放弃使用索引而进行全表扫面
- 3、应尽量避免在WHERE自居中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描
- 4、应尽量避免在WHERE自居中对字段进行函数操作，浙江导致引擎放弃使用索引而进行全表扫描
- 5、不要在WHERE字句中的=左边进行函数、算数运算或其他表达式运算，否则系统将可能无法正确使用索引
- 6、符合索引遵循前缀原则
- 7、如果 MySQL 评估使用索引比全表扫描更慢，会放弃使用索引。如果此时想要索引，可以在语句中添加强制索引。
- 8、列类型是字符串类型，查询时一定要给值加引号，否则索引失效。
- 9、LIKE 查询，%不能在前，因为无法使用索引。如果需要模糊匹配，可以使用全文索引。

以下三条 SQL 如何建索引，只建一条怎么建？

```
WHERE a = 1 AND b = 1
WHERE b = 1
WHERE b = 1 ORDER BY time DESC
```

- 以顺序 b , a, time 建立复合索引，CREATE INDEX table1_b_a_time ON index_test01(b, a, time)。
- 对于第一条 SQL，因为最新MySQL版本会优化WHERE子句后面的列顺序，以匹配复合索引顺序。

MySQL 索引的原理？

解释 MySQL 索引的原理，篇幅会比较长，并且网络上已经有靠谱的资料可以看，所以芳芳这里整理了几篇，胖友可以对照着看。

- 《MySQL 索引原理》
- 《深入理解 MySQL 索引原理和实现 —— 为什么索引可以加速查询？》
- 《MySQL BTree 索引和 hash 索引的区别》 。

什么是 B-Tree 索引？

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

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，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb_page_size将页的大小设置为4K、8K、16K，在MySQL中可通过如下命令查看页的大小：

```
mysql> show variables like 'innodb_page_size';
```

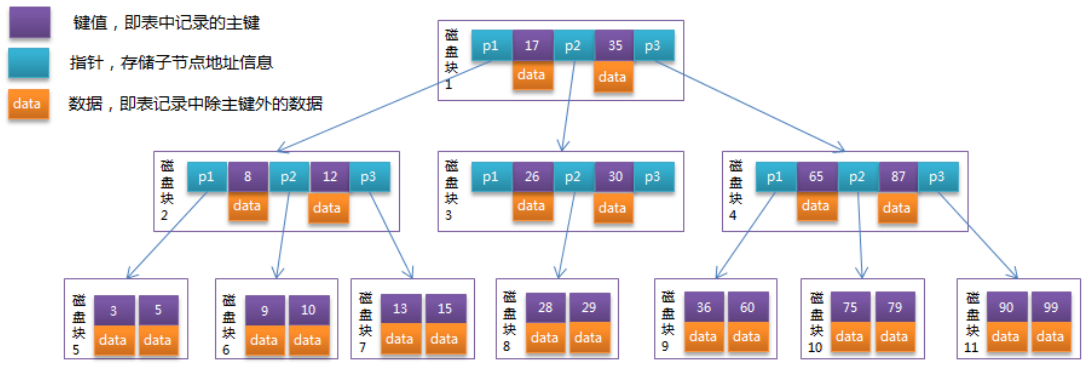
而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data] ，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

一棵m阶的B-Tree有如下特性：

- i. 每个节点最多有m个孩子。
- ii. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。
- iii. 若根节点不是叶子节点，则至少有2个孩子
- iv. 所有叶子节点都在同一层，且不包含其它关键字信息
- v. 每个非终端节点包含n个关键字信息（P0,P1,...Pn, k1,...kn）
- vi. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1
- vii. ki(i=1,...n)为关键字，且关键字升序排序。
- viii. Pi(i=1,...n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)77

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：



每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

- 1、根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
- 2、比较关键字29在区间（17,35），找到磁盘块1的指针P2。
- 3、根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
- 4、比较关键字29在区间（26,30），找到磁盘块3的指针P2。
- 5、根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
- 6、在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

什么是 B+Tree 索引？

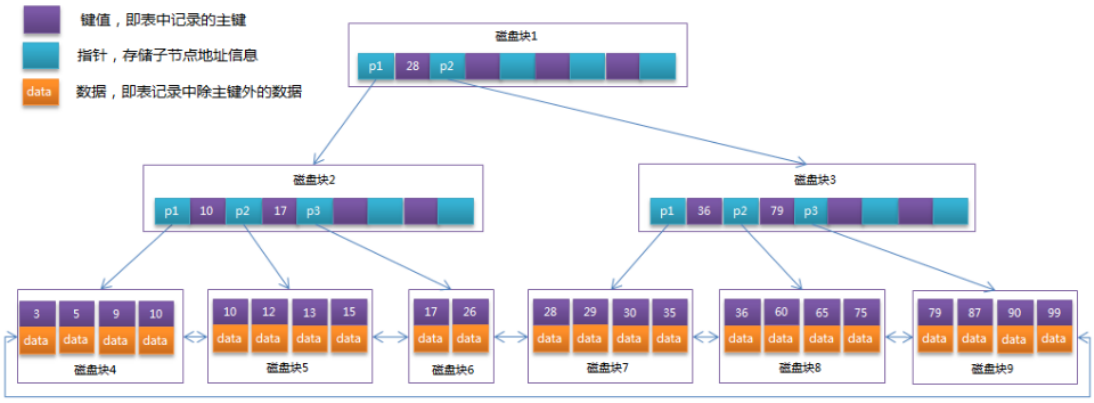
B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

- 1、非叶子节点只存储键值信息。
- 2、所有叶子节点之间都有一个链指针。
- 3、数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：



通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^{3}）。也就是说一个深度为3的B+Tree索引可以维护10^3 * 10^3 * 10^3 = 10亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2_4层。MySQL的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1^{3}次磁盘I/O操作。

数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

聚簇索引的注意点有哪些？

聚簇索引表最大限度地提高了I/O密集型应用的性能，但它也有以下几个限制：

- 1、插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于 InnoDB 表，我们一般都会定义一个自增的 ID 列为主键。
- 2、更新主键的代价很高，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义主键为不可更新。
- 3、二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。
- 4、主键 ID 建议使用整型。因为，每个主键索引的 B+Tree 节点的键值可以存储更多主键 ID ，每个非主键索引的 B+Tree 节点的数据可以存储更多主键 ID 。

什么是索引的最左匹配特性？

当 B+Tree 的数据项是复合的数据结构，比如索引 (name, age, sex) 的时候，B+Tree 是按照从左到右的顺序来建立搜索树的。

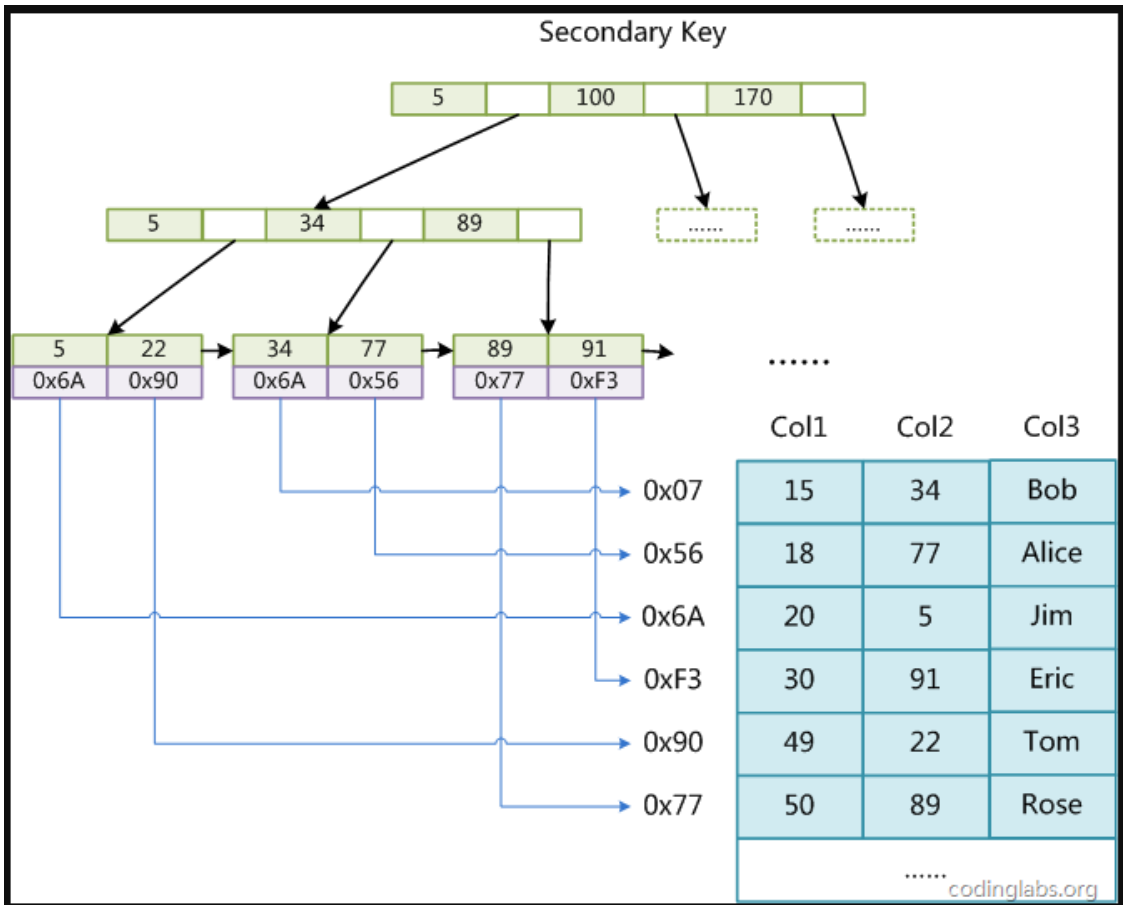
- 1、比如当 (张三, 20, F) 这样的数据来检索的时候，B+Tree 会优先比较 name 来确定下一步的所搜方向，如果 name 相同再依次比较 age 和 sex ，最后得到检索的数据。
- 2、但当 (20, F) 这样的没有 name 的数据来的时候，B+Tree 就不知道下一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。
- 3、比如当 (张三, F) 这样的数据来检索时，B+Tree 可以用 name 来指定搜索方向，但下一个字段 age 的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是 F 的数据了。

MyISAM 索引(非聚簇索引)实现？

MyISAM 索引的实现，和 InnoDB 索引的实现是一样使用 B+Tree ，差别在于 MyISAM 索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。

> 主键索引

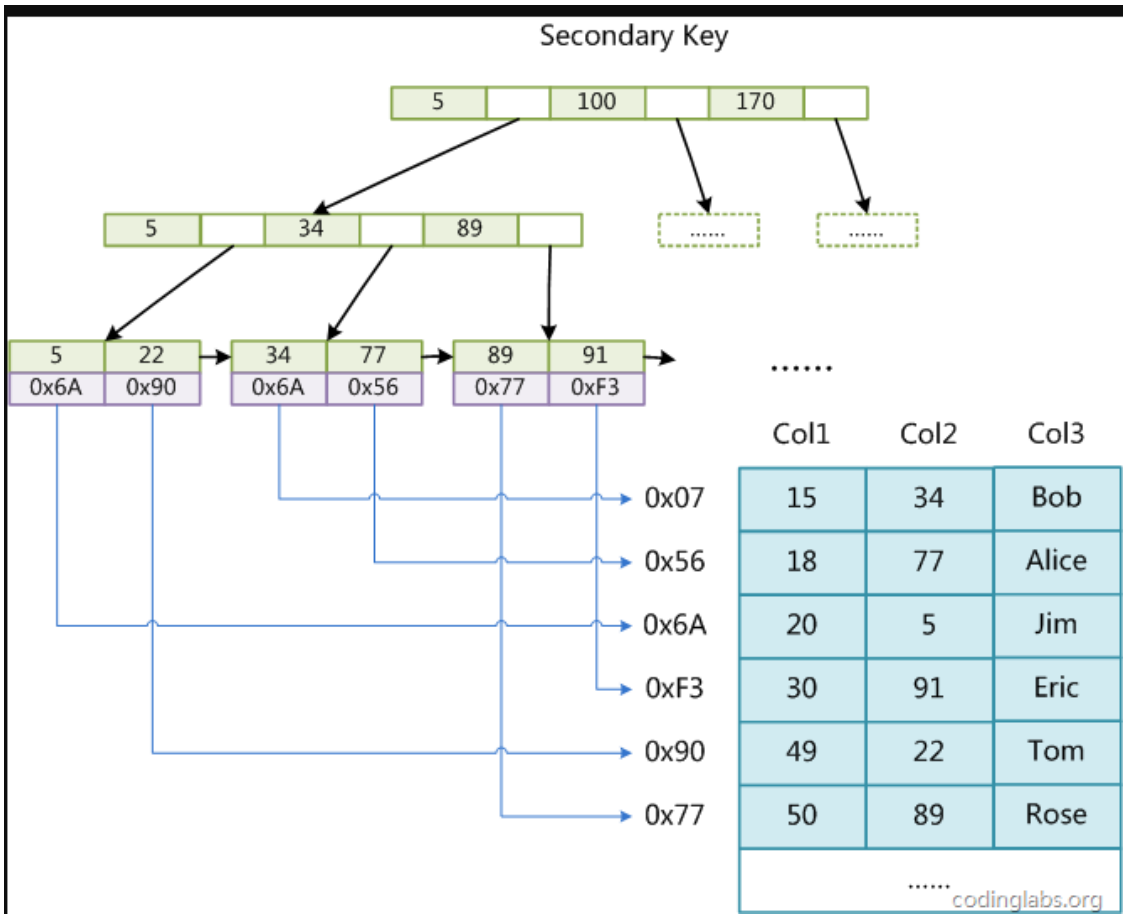
MyISAM引 擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM主键索引的原理图：



这里设表一共有三列，假设我们以 Col1 为主键，上图是一个 MyISAM 表的主索引（Primary key）示意。可以看出 MyISAM 的索引文件仅仅保存数据记录的地址。

> 辅助索引：

在 MyISAM 中，主索引和辅助索引在结构上没有任何区别，只是主索引要求 key 是唯一的，而辅助索引的 key 可以重复。**如果我们在 Col2 上建立一个辅助索引，则此索引的结构如下图所示：



同样也是一颗 B+Tree ，data 域保存数据记录的地址。因此，MyISAM 中索引检索的算法为首先按照 B+Tree 搜索算法搜索索引，如果指定的 Key 存在，则取出其 data 域的值，然后以 data 域的值为地址，读取相应数据记录。

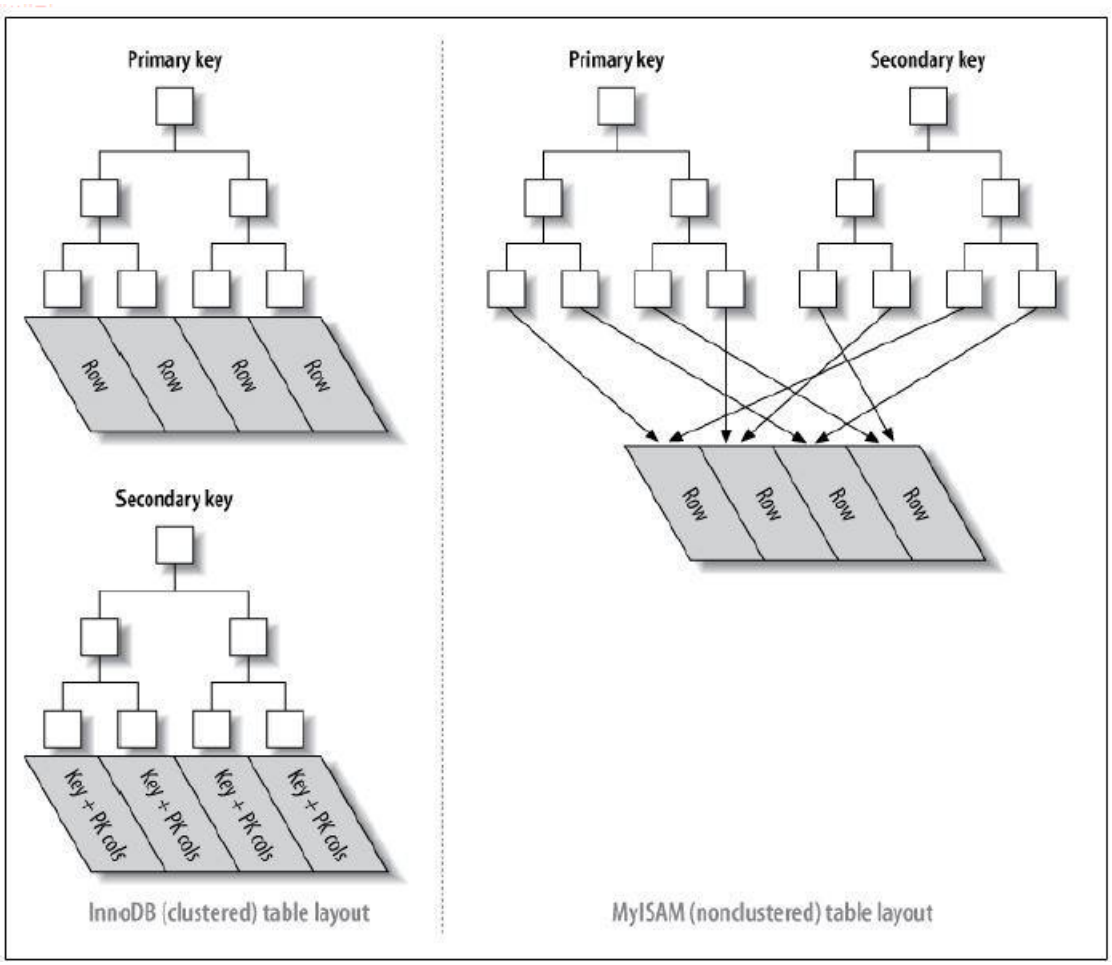
最开始我一直不懂既然非聚簇索引的主索引和辅助索引指向相同的内容，为什么还要辅助索引这个东西呢，后来才明白索引不就是用来查询的吗，用在那些地方呢，不就是WHERE和ORDER BY 语句后面吗，那么如果查询的条件不是主键怎么办呢，这个时候就需要辅助索引了

InnoDB 索（聚簇索引）引的实现？

- 聚簇索引的主索引的叶子结点存储的是键值对应的数据本身，辅助索引的叶子结点存储的是键值对应的数据的主键键值。因此主键的值长度越小越好，类型越简单越好。
- 聚簇索引的数据和主键索引存储在一起。
- 聚簇索引的数据是根据主键的顺序保存。因此适合按主键索引的区间查找，可以有更少的磁盘I/O，加快查询速度。但是也是因为这个原因，聚簇索引的插入顺序最好按照主键单调的顺序插入，否则会频繁的引起页分裂，严重影响性能。
- 在InnoDB中，如果只需要查找索引的列，就尽量不要加入其它的列，这样会提高查询效率。

MyISAM 索引与 InnoDB 索引的区别？

下图可以形象的说明聚簇索引和非聚簇索引的区别:



- InnoDB 索引是聚簇索引，MyISAM 索引是非聚簇索引。
- InnoDB 的主键索引的叶子节点存储着行数据，因此主键索引非常高效。
- MyISAM 索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。
- InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

MySQL的四种事务隔离级别？

事物就是对一系列的数据库操作进行统一的提交或回滚操作，如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前所有的操作，这样可以防止出现脏数据，防止数据库数据出现问题。

如果异常为runtimeException（uncheckedException，不可解决异常）则会回滚，如果是checkedException则不会回滚在客户端可解决异常。

事物特性指的是？

指的是 ACID ，如下图所示：

- 原子性Atomicity：一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（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。即，事务不可分割、不可约简。
- 一致性 Consistency ：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设约束、触发器、级联回滚等。
- 隔离性 Isolation ：数据库允许多个并发事务同时对其数据进行读写和修改的能力，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。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
- 持久性 Durability ：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。

事务的并发问题？

实际场景下，事务并不是串行的，所以会带来如下三个问题：

- 1、脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据。
- 2、不可重复读：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，结果不一致。
- 3、幻读：系统管理员 A 将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

MySQL 的锁机制？

表锁是日常开发中的常见问题，因此也是面试当中最常见的考察点，当多个查询同一时刻进行数据修改时，就会产生并发控制的问题。MySQL 的共享锁和排他锁，就是读锁和写锁。

- 共享锁：不堵塞，多个用户可以同时读一个资源，互不干扰。
- 排他锁：一个写锁会阻塞其他的读锁和写锁，这样可以只允许一个用户进行写入，防止其他用户读取正在写入的资源。

锁的粒度？

- 表锁：系统开销最小，会锁定整张表，MyIsam 使用表锁。
- 行锁：最大程度的支持并发处理，但是也带来了最大的锁开销，InnoDB 使用行锁。

什么是悲观锁？什么是乐观锁？

> 1)悲观锁

它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

在悲观锁的情况下，为了保证事务的隔离性，就需要一致性锁定读。读取数据时给加锁，其它事务无法修改这些数据。修改删除数据时也要加锁，其它事务无法读取这些数据。

> 2)乐观锁

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

什么是死锁？

多数情况下，可以认为如果一个资源被锁定，它总会在以后某个时间被释放。而死锁发生在当多个进程访问同一数据库时，其中每个进程拥有的锁都是其他进程所需的，由此造成每个进程都无法继续下去。简单的说，进程 A 等待进程 B 释放他的资源，B 又等待 A 释放他的资源，这样就互相等待就形成死锁。

虽然进程在运行过程中，可能发生死锁，但死锁的发生也必须具备一定的条件，死锁的发生必须具备以下四个必要条件：

- 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
- 请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
- 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
- 环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合 {P0，P1，P2，•••，Pn} 中的 P0 正在等待一个 P1 占用的资源；P1 正在等待 P2 占用的资源，……，Pn 正在等待已被 P0 占用的资源。

MySQL 查询执行顺序？

具体的，可以看看 《SQL 查询之执行顺序解析》 文章。

MySQL SQL 优化？

- 《PHP 面试之 MySQL 查询优化》
- 《【面试】【MySQL常见问题总结】【03】》 第 078、095、105 题

另外，除了从 SQL 层面进行优化，也可以从服务器硬件层面，进一步优化 MySQL 。具体可以看看 《MySQL 数据库性能优化之硬件优化》 。

MySQL 数据库 CPU 飙升到 500% 的话，怎么处理？

当 CPU 飙升到 500% 时，先用操作系统命令 top 命令观察是不是 mysqld 占用导致的，如果不是，找出占用高的进程，并进行相关处理。(如果此时是 IO 压力比较大，可以使用 iostat 命令，定位是哪个进程占用了磁盘 IO 。)

如果是 mysqld 造成的，使用 show processlist 命令，看看里面跑的 Session 情况，是不是有消耗资源的 SQL 在运行。找出消耗高的 SQL ，看看执行计划是否准确，index 是否缺失，或者实在是数据量太大造成。一般来说，肯定要 kill 掉这些线程(同时观察 CPU 使用率是否下降)，等进行相应的调整(比如说加索引、改 SQL 、改内存参数)之后，再重新跑这些 SQL。

也有可能是每个 SQL 消耗资源并不多，但是突然之间，有大量的 Session 连进来导致 CPU 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等。

Mysql执行count优化

在开发系统的时候，你可能经常需要计算一个表的行数，比如一个交易系统的所有变更记录总数。这时候你可能会想，一条 select count(*) from t 语句不就解决了吗？

但是，你会发现随着系统中记录数越来越多，这条语句执行得也会越来越慢。然后你可能就想了，MySQL 怎么这么笨啊，记个总数，每次要查的时候直接读出来，不就好了吗。

那么今天，我们就来聊聊 count(*) 语句到底是怎样实现的，以及 MySQL 为什么会这么实现。然后，我会再和你说说，如果应用中有这种频繁变更并需要统计表行数的需求，业务设计上可以怎么做。

count(*) 的实现方式

你首先要明确的是，在不同的 MySQL 引擎中，count(*) 有不同的实现方式。

- MyISAM 引擎把一个表的总行数存在了磁盘上，因此执行 count(*) 的时候会直接返回这

个数，效率很高；

- 而 InnoDB 引擎就麻烦了，它执行 count(*) 的时候，需要把数据一行一行地从引擎里面读出来，然后累积计数。

这里需要注意的是，我们在这篇文章里讨论的是没有过滤条件的 count(*)，如果加了 where 条件的话，MyISAM 表也是不能返回得这么快的。

在前面的文章中，我们一起分析了为什么要使用 InnoDB，因为不论是在事务支持、并发能力还是在数据安全方面，InnoDB 都优于 MyISAM。我猜你的表也一定是用了 InnoDB 引擎。这就是当你的记录数越来越多的时候，计算一个表的总行数会越来越慢的原因。

那**为什么 InnoDB 不跟 MyISAM 一样，也把数字存起来呢？**

这是因为即使是在同一个时刻的多个查询，由于多版本并发控制（MVCC）的原因，InnoDB

表“应该返回多少行”也是不确定的。这里，我用一个算 count(*) 的例子来为你解释一下。

假设表 t 中现在有 10000 条记录，我们设计了三个用户并行的会话。

- 会话 A 先启动事务并查询一次表的总行数；
- 会话 B 启动事务，插入一行后记录后，查询表的总行数；
- 会话 C 先启动一个单独的语句，插入一行记录后，查询表的总行数。

我们假设从上到下是按照时间顺序执行的，同一行语句是在同一时刻执行的。

会话A	会话B	会话C
begin;		
select count(*) from t;		
		insert into t (插入一行);
	begin;	
	insert into t (插入一行);	
select count(*) from t; （返回10000）	select count(*) from t; （返回10002）	select count(*) from t; （返回10001）

图 1 会话 A、B、C 的执行流程

图 1 会话 A、B、C 的执行流程

你会看到，在最后一个时刻，三个会话 A、B、C 会同时查询表 t 的总行数，但拿到的结果却不同。

这和 InnoDB 的事务设计有关系，可重复读是它默认的隔离级别，在代码上就是通过多版本并发控制，也就是 MVCC 来实现的。每一行记录都要判断自己是否对这个会话可见，因此对于 count(*) 请求来说，InnoDB 只好把数据一行一行地读出依次判断，可见的行才能够用于计算“基于这个查询”的表的总行数。

> 备注：如果你对 MVCC 记忆模糊了，可以再回顾下第 3 篇文章《事务隔离：为什么你改了我还看不见？》和第 8 篇文章《事务到底是隔离的还是不隔离的？》中的相关内容。

当然，现在这个看上去笨笨的 MySQL，在执行 count(*) 操作的时候还是做了优化的。

你知道的，InnoDB 是索引组织表，主键索引树

的叶子节点是数据，而普通索引树的叶子节点是主键值。所以，普通索引树比主键索引树小很多。对于 count(*) 这样的操作，遍历哪个索引树得到的结果逻辑上都是一样的。因此，MySQL 优化器会找到最小的那棵树来遍历。**在保证逻辑正确的前提下，尽量减少扫描的数据量，是数据库系统设计的通用法则之一。**

如果你用过 show table status 命令的话，就会发现这个命令的输出结果里面也有一个 TABLE_ROWS 用于显示这个表当前有多少行，这个命令执行挺快的，那这个 TABLE_ROWS 能代替 count(*) 吗？

你可能还记得在第 10 篇文章《MySQL 为什么有时候会选错索引？》中我提到过，索引统计的值是通过采样来估算的。实际上，TABLE_ROWS 就是从这个采样估算得来的，因此它也很不准。有多不准呢，官方文档说误差可能达到 40% 到 50%。**所以，show table status 命令显示的行数也不能直接使用。**

到这里我们小结一下：

- MyISAM 表虽然 count(*) 很快，但是不支持事务；
- show table status 命令虽然返回很快，但是不准确；
- InnoDB 表直接 count(*) 会遍历全表，虽然结果准确，但会导致性能问题。

那么，回到文章开头的问题，如果你现在有一个页面经常要显示交易系统的操作记录总数，到底应该怎么办呢？答案是，我们只能自己计数。

接下来，我们讨论一下，看看自己计数有哪些方法，以及每种方法的优缺点有哪些。

这里，我先和你说一下这些方法的基本思路：你需要自己找一个地方，把操作记录表的行数存起来。

用缓存系统保存计数

对于更新很频繁的库来说，你可能会第一时间想到，用缓存系统来支持。

你可以用一个 Redis 服务来保存这个表的总行数。这个表每被插入一行 Redis 计数就加 1，每被删除一行 Redis 计数就减 1。这种方式下，读和更新操作都很快，但你再想一下这种方式存在什么问题吗？

没错，缓存系统可能会丢失更新。

Redis 的数据不能永久地留在内存里，所以你会找一个地方把这个值定期地持久化存储起来。但即使这样，仍然可能丢失更新。试想如果刚刚在数据表中插入了一行，Redis 中保存的值也加了 1，然后 Redis 异常重启了，重启后你要从存储 redis 数据的地方把这个值读回来，而刚刚加 1 的这个计数操作却丢失了。

当然了，这还是有解的。比如，Redis 异常重启以后，到数据库里面单独执行一次 count(*) 获

取真实的行数，再把这个值写回到 Redis 里就可以了。异常重启毕竟不是经常出现的情况，这一次全表扫描的成本，还是可以接受的。

但实际上，**将计数保存在缓存系统中的方式，还不只是丢失更新的问题。即使 Redis 正常工作，这个值还是逻辑上不精确的。**

你可以设想一下有这么一个页面，要显示操作记录的总数，同时还要显示最近操作的 100 条记录。那么，这个页面的逻辑就需要先到 Redis 里面取出计数，再到数据表里面取数据记录。

我们是这么定义不精确的：

1. 一种是，查到的 100 行结果里面有最新插入记录，而 Redis 的计数里还没加 1；
2. 另一种是，查到的 100 行结果里没有最新插入的记录，而 Redis 的计数里已经加了 1。

这两种情况，都是逻辑不一致的。

我们一起来看看这个时序图。

时刻	会话A	会话B
T1		
T2	插入一行数据R;	
T3		读Redis计数; 查询最近100条记录;
T4	Redis 计数加1;	

图 2 会话 A、B 执行时序图

图 2 中，会话 A 是一个插入交易记录的逻辑，往数据表里插入一行 R，然后 Redis 计数加 1；会话 B 就是查询页面显示时需要的数据。

在图 2 的这个时序里，在 T3 时刻会话 B 来查询的时候，会显示出新插入的 R 这个记录，但是 Redis 的计数还没加 1。这时候，就会出现我们说的数据不一致。

你一定会说，这是因为我们执行新增记录逻辑时候，是先写数据表，再改 Redis 计数。而读的时候是先读 Redis，再读数据表，这个顺序是相反的。那么，如果保持顺序一样的话，是不是就没问题了？我们现在把会话 A 的更新顺序换一

下，再看看执行结果。

时刻	会话A	会话B
T1		
T2	Redis 计数加1;	
T3		读Redis计数; 查询最近100条记录;
T4	插入一行数据R;	
T5		

图 3 调整顺序后，会话 A、B 的执行时序图

你会发现，这时候反过来了，会话 B 在 T3 时刻查询的时候，Redis 计数加了 1 了，但还查不到新插入的 R 这一行，也是数据不一致的情况。

在并发系统里面，我们是无法精确控制不同线程的执行时刻的，因为存在图中的这种操作序列，所以，我们说即使 Redis 正常工作，这个计数值还是逻辑上不精确的。

在数据库保存计数

根据上面的分析，用缓存系统保存计数有丢失数据和计数不精确的问题。那么，**如果我们把这个**

据和计数不精确的问题。那么，**如果我们把这个计数直接放到数据库里单独的一张计数表 C 中，又会怎么样呢？**

首先，这解决了崩溃丢失的问题，InnoDB 是支持崩溃恢复不丢数据的。

> 备注：关于 InnoDB 的崩溃恢复，你可以再回顾一下第 2 篇文章《日志系统：一条 SQL 更新语句是如何执行的？》中的相关内容。

然后，我们再看看能不能解决计数不精确的问题。

你会说，这不一样吗？无非就是把图 3 中对 Redis 的操作，改成了对计数表 C 的操作。只要出现图 3 的这种执行序列，这个问题还是无解的吧？

这个问题还真不是无解的。

我们这篇文章要解决的问题，都是由于 InnoDB

要支持事务，从而导致 InnoDB 表不能把 count(*) 直接存起来，然后查询的时候直接返回形成的。

所谓以子之矛攻子之盾，现在我们就利用“事务”这个特性，把问题解决掉。

时刻	会话A	会话B
T1		
T2	begin; 表C中计数值加1;	
T3		begin; 读表C计数值; 查询最近100条记录; commit;
T4	插入一行数据R commit;	

图 4 会话 A、B 的执行时序图

我们来看下现在的执行结果。虽然会话 B 的读操作仍然是在 T3 执行的，但是因为这时候更新事务还没有提交，所以计数值加 1 这个操作对会话 B 还不可见。

因此，会话 B 看到的结果里， 查计数值和“最近 100 条记录”看到的结果，逻辑上就是一致的

100 条记录看到的结果，逻辑上就是一致的。

不同的 count 用法

在前面文章的评论区，有同学留言问到：在 select count(?) from t 这样的查询语句里面，count(*)、count(主键 id)、count(字段) 和 count(1) 等不同用法的性能，有哪些差别。今天谈到了 count(*) 的性能问题，我就借此机会和你详细说明一下这几种用法的性能差别。

需要注意的是，下面的讨论还是基于 InnoDB 引擎的。

这里，首先你要弄清楚 count() 的语义。count() 是一个聚合函数，对于返回的结果集，一行行地判断，如果 count 函数的参数不是 NULL，累计值就加 1，否则不加。最后返回累计值。

所以，count(*)、count(主键 id) 和 count(1) 都表示返回满足条件的结果集的总行数；而 count(字段），则表示返回满足条件的数据行里面，参数“字段”不为 NULL 的总个数。

面，参数字段不为 NULL 的总个数。

至于分析性能差别的时候，你可以记住这么几个原则：

1. server 层要什么就给什么；
2. InnoDB 只给必要的值；
3. 现在的优化器只优化了 count(*) 的语义为“取行数”，其他“显而易见”的优化并没有做。

这是什么意思呢？接下来，我们就一个个地来看看。

对于 count(主键 id) 来说，InnoDB 引擎会遍历整张表，把每一行的 id 值都取出来，返回给 server 层。server 层拿到 id 后，判断是不可能为空的，就按行累加。

对于 count(1) 来说，InnoDB 引擎遍历整张表，但不取值。server 层对于返回的每一行，放一个数字“1”进去，判断是不可能为空的，按行累加。

单看这两个用法的差别的话，你能对比出来，count(1) 执行得要比 count(主键 id) 快。因为从引擎返回 id 会涉及到解析数据行，以及拷贝字段值的操作。

对于 count(字段) 来说：

1. 如果这个“字段”是定义为 not null 的话，一行行地从记录里面读出这个字段，判断不能为 null，按行累加；
2. 如果这个“字段”定义允许为 null，那么执行的时候，判断到有可能是 null，还要把值取出来再判断一下，不是 null 才累加。

也就是前面的第一条原则，server 层要什么字段，InnoDB 就返回什么字段。

但是 count(*) 是例外，并不会把全部字段取出来，而是专门做了优化，不取值。count(*) 肯定不是 null，按行累加。

看到这里，你一定会说，终于结束了...

看到这里，你一定会说，优化器就不能自己判断一下吗，主键 id 肯定非空啊，为什么不能按照 count(*) 来处理，多么简单的优化啊。

当然，MySQL 专门针对这个语句进行优化，也不是不可以。但是这种需要专门优化的情况太多了，而且 MySQL 已经优化过 count(*) 了，你直接使用这种用法就可以了。

所以结论是：按照效率排序的话，count(字段)<count(主键 id)<count(1)≈count(*)，所以我建议你，尽量使用 count(*)。

小结

今天，我和你聊了聊 MySQL 中获得表行数的两种方法。我们提到了在不同引擎中 count(*) 的实现方式是不一样的，也分析了用缓存系统来存储计数值存在的问题。

其实，把计数放在 Redis 里面，不能够保证计数和 MySQL 表里的数据精确一致的原因，是**这两个不同的存储构成的系统，不支持分布式事务**

不同的存储构成的系统，不支持分布式事务，**无法拿到精确一致的视图。**而把计数值也放在 MySQL 中，就解决了一致性视图的问题。

InnoDB 引擎支持事务，我们利用好事务的原子性和隔离性，就可以简化在业务开发时的逻辑。这也是 InnoDB 引擎备受青睐的原因之一。

最后，又到了今天的思考题时间了。

在刚刚讨论的方案中，我们用了事务来确保计数准确。由于事务可以保证中间结果不被别的事务读到，因此修改计数值和插入新记录的顺序是不影响逻辑结果的。但是，从并发系统性能的角度考虑，你觉得在这个事务序列里，应该先插入操作记录，还是应该先更新计数表呢？

你可以把你的思考和观点写在留言区里，我会在