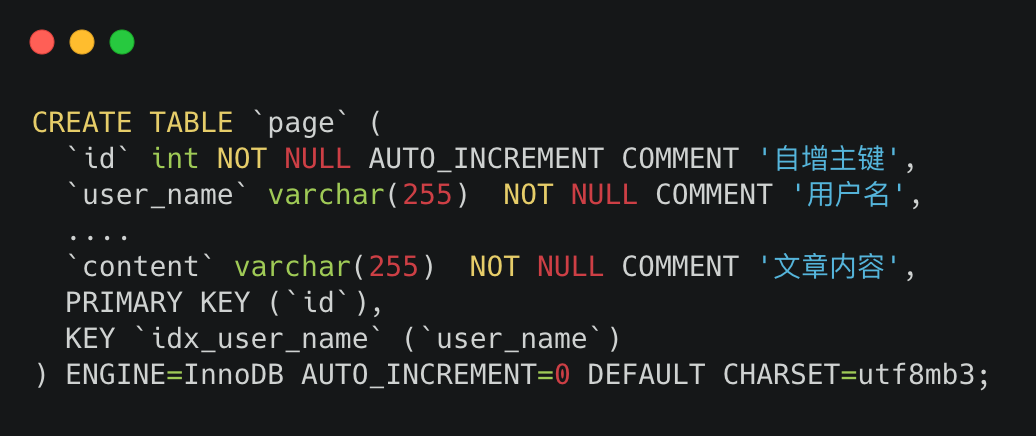
# 一、mysql查询 limit 1000,10 和limit 10 速度一样快吗？

<https://ost.51cto.com/posts/12620>

假设我们的建表sql是这样的

mysql建表sql

建表sql大家也不用扣细节，只需要知道id是主键，并且在user\_name建了个非主键索引就够了，其他都不重要。

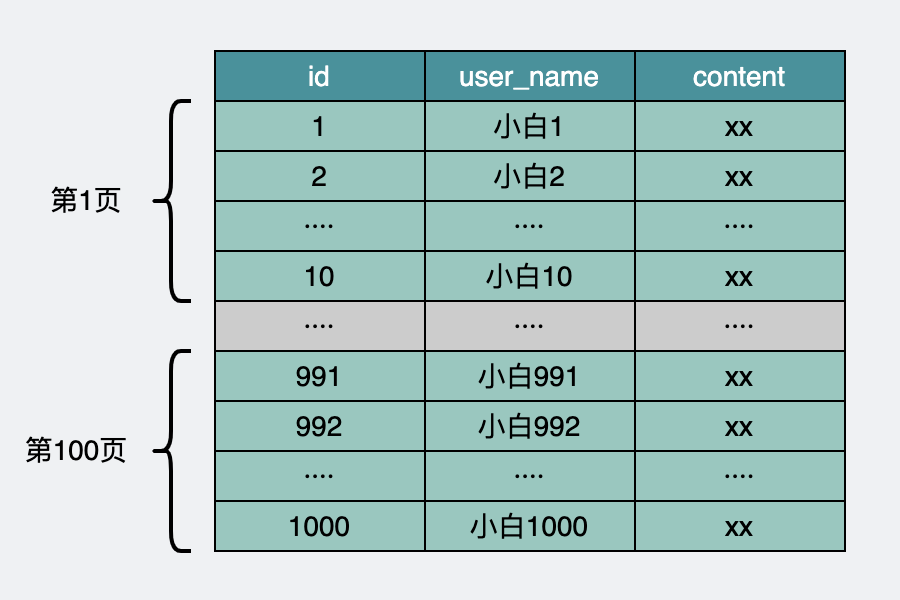
为了实现分页。

很容易联想到下面这样的sql语句。

select \* from page order by id limit offset, size;

* 1.

复制

比如一页有10条数据。

user表数据库原始状态

第一页就是下面这样的sql语句。

select \* from page order by id limit 0, 10;

第一百页就是

select \* from page order by id limit 990, 10;

那么问题来了。

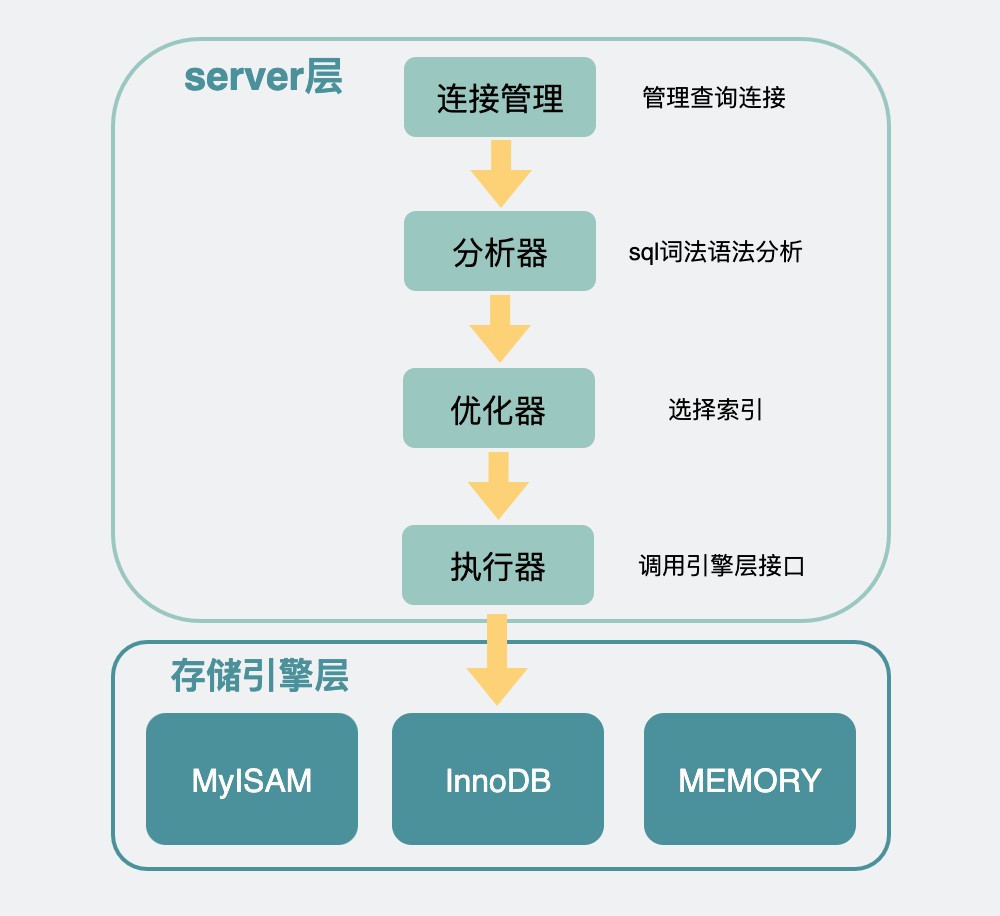
用这种方式，同样都是拿10条数据，查第一页和第一百页的查询速度是一样的吗?为什么?

**两种limit的执行过程**

上面的两种查询方式。对应 limit offset, size 和 limit size 两种方式。

而其实 limit size ，相当于 limit 0, size。也就是从0开始取size条数据。

也就是说，两种方式的区别在于offset是否为0。

我们先来看下limit sql的内部执行逻辑。

Mysql架构

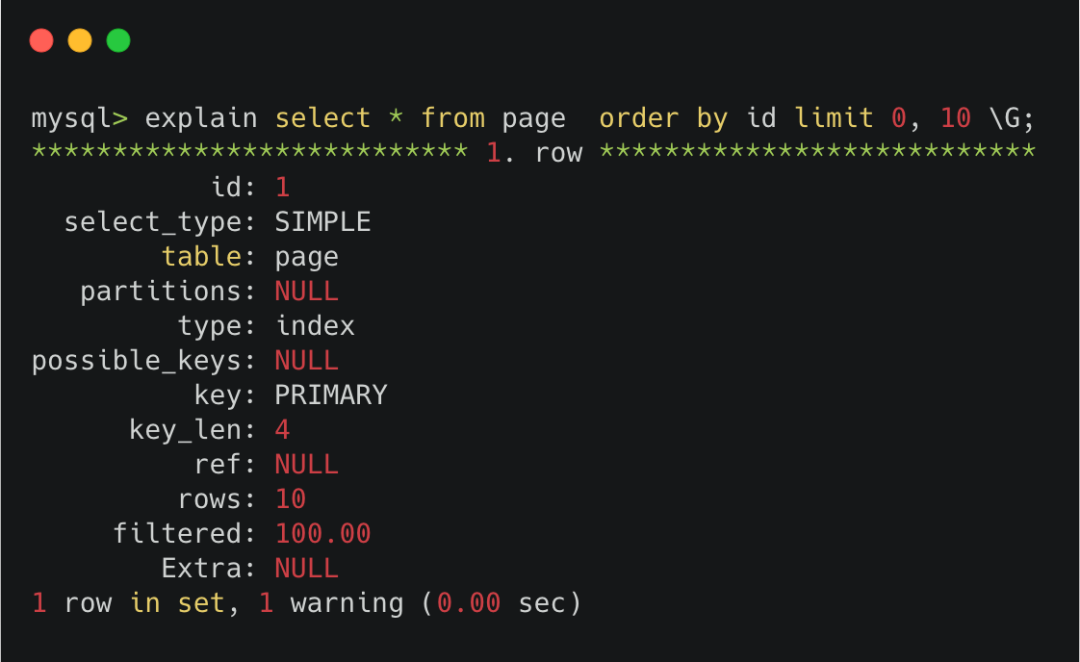
mysql内部分为server层和存储引擎层。一般情况下存储引擎都用innodb。

server层有很多模块，其中需要关注的是执行器是用于跟存储引擎打交道的组件。

执行器可以通过调用存储引擎提供的接口，将一行行数据取出，当这些数据完全符合要求(比如满足其他where条件)，则会放到结果集中，最后返回给调用mysql的客户端(go、java写的应用程序)。

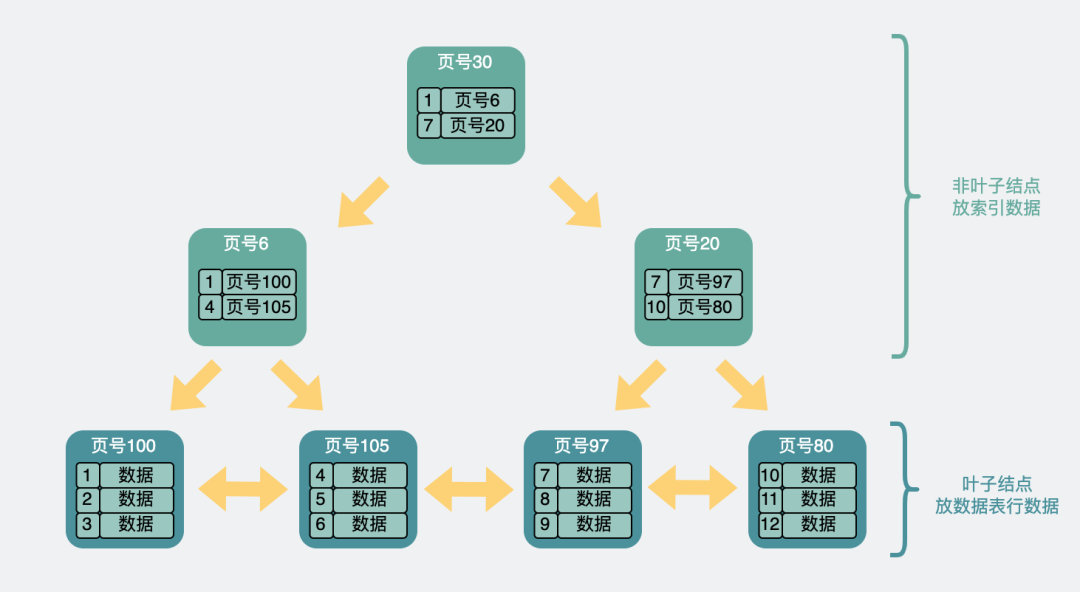
我们可以对下面的sql先执行下 explain。

explain select \* from page order by id limit 0, 10;

可以看到，explain中提示 key 那里，执行的是PRIMARY，也就是走的主键索引。

分页查询offset=0

主键索引本质是一棵B+树，它是放在innodb中的一个数据结构。

我们可以回忆下，B+树大概长这样。

B+树结构

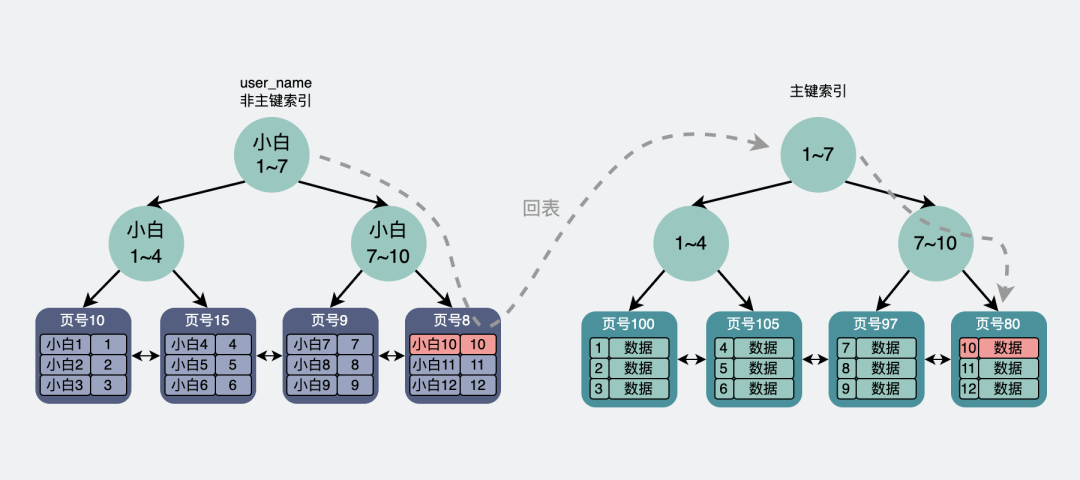
在这个树状结构里，我们需要关注的是，最下面一层节点，也就是叶子结点。而这个叶子结点里放的信息会根据当前的索引是主键还是非主键有所不同。

* 如果是主键索引，它的叶子节点会存放完整的行数据信息。
* 如果是非主键索引，那它的叶子节点则会存放主键，如果想获得行数据信息，则需要再跑到主键索引去拿一次数据，这叫回表。

比如执行

select \* from page where user\_name = "小白10";

会通过非主键索引去查询user\_name为"小白10"的数据，然后在叶子结点里找到"小白10"的数据对应的主键为10。

此时回表到主键索引中做查询，最后定位到主键为10的行数据。

回表

但不管是主键还是非主键索引，他们的叶子结点数据都是有序的。比如在主键索引中，这些数据是根据主键id的大小，从小到大，进行排序的。

**基于主键索引的limit执行过程**

那么回到文章开头的问题里。

当我们去掉explain，执行这条sql。

select \* from page order by id limit 0, 10;

上面select后面带的是星号\*，也就是要求获得行数据的所有字段信息。

server层会调用innodb的接口，在innodb里的主键索引中获取到第0到10条完整行数据，依次返回给server层，并放到server层的结果集中，返回给客户端。

而当我们把offset搞离谱点，比如执行的是

select \* from page order by id limit 6000000, 10;

server层会调用innodb的接口，由于这次的offset=6000000，会在innodb里的主键索引中获取到第0到(6000000 + 10)条完整行数据，返回给server层之后根据offset的值挨个抛弃，最后只留下最后面的size条，也就是10条数据，放到server层的结果集中，返回给客户端。

可以看出，当offset非0时，server层会从引擎层获取到很多无用的数据，而获取的这些无用数据都是要耗时的。

因此，我们就知道了文章开头的问题的答案，mysql查询中 limit 1000,10 会比 limit 10 更慢。原因是 limit 1000,10 会取出1000+10条数据，并抛弃前1000条，这部分耗时更大

**那这种case有办法优化吗?**

可以看出，当offset非0时，server层会从引擎层获取到很多无用的数据，而当select后面是\*号时，就需要拷贝完整的行信息，拷贝完整数据跟只拷贝行数据里的其中一两个列字段耗时是不同的，这就让原本就耗时的操作变得更加离谱。

因为前面的offset条数据最后都是不要的，就算将完整字段都拷贝来了又有什么用呢，所以我们可以将sql语句修改成下面这样。

select \* from page where id >=(select id from page order by id limit 6000000, 1) order by id limit 10;

上面这条sql语句，里面先执行子查询 select id from page order by id limit 6000000, 1, 这个操作，其实也是将在innodb中的主键索引中获取到6000000+1条数据，然后server层会抛弃前6000000条，只保留最后一条数据的id。

但不同的地方在于，在返回server层的过程中，只会拷贝数据行内的id这一列，而不会拷贝数据行的所有列，当数据量较大时，这部分的耗时还是比较明显的。

在拿到了上面的id之后，假设这个id正好等于6000000，那sql就变成了

select \* from page where id >=(6000000) order by id limit 10;

这样innodb再走一次主键索引，通过B+树快速定位到id=6000000的行数据，时间复杂度是lg(n)，然后向后取10条数据。

这样性能确实是提升了，亲测能快一倍左右，属于那种耗时从3s变成1.5s的操作。

这······

属实有些杯水车薪，有点搓，属于没办法中的办法。

**基于非主键索引的limit执行过程**

上面提到的是主键索引的执行过程，我们再来看下基于非主键索引的limit执行过程。

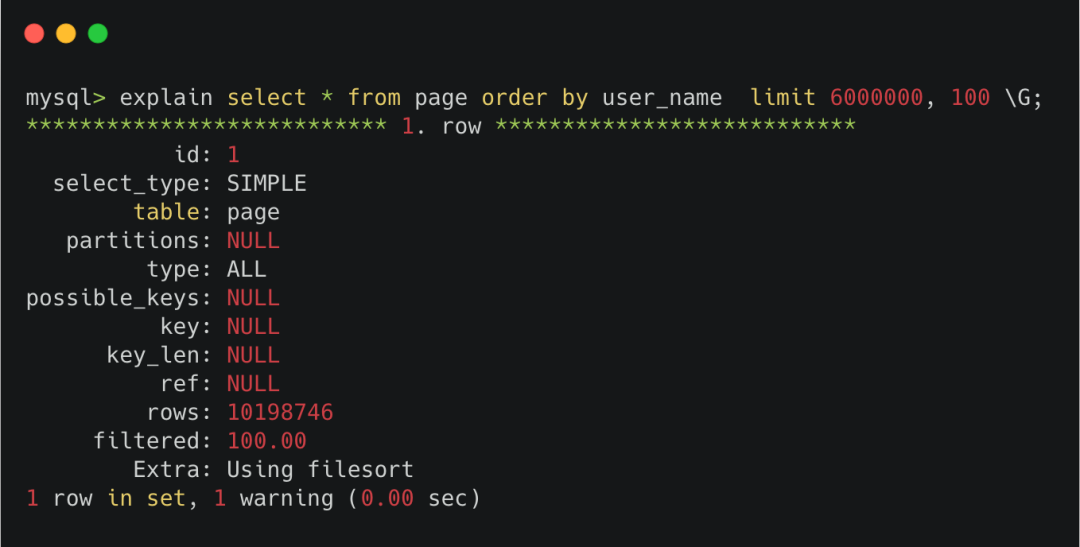
比如下面的sql语句

select \* from page order by user\_name limit 0, 10;

server层会调用innodb的接口，在innodb里的非主键索引中获取到第0条数据对应的主键id后，回表到主键索引中找到对应的完整行数据，然后返回给server层，server层将其放到结果集中，返回给客户端。

而当offset>0时，且offset的值较小时，逻辑也类似，区别在于，offset>0时会丢弃前面的offset条数据。

也就是说非主键索引的limit过程，比主键索引的limit过程，多了个回表的消耗。

但当offset变得非常大时，比如600万，此时执行explain。

非主键索引offset值超大时走全表扫描

可以看到type那一栏显示的是ALL，也就是全表扫描。

这是因为server层的优化器，会在执行器执行sql语句前，判断下哪种执行计划的代价更小。

很明显，优化器在看到非主键索引的600w次回表之后，摇了摇头，还不如全表一条条记录去判断算了，于是选择了全表扫描。

因此，当limit offset过大时，非主键索引查询非常容易变成全表扫描。是真·性能杀手。

这种情况也能通过一些方式去优化。比如

select \* from page t1, (select id from page order by user\_name limit 6000000, 100) t2 WHERE t1.id = t2.id;

通过select id from page order by user\_name limit 6000000, 100。先走innodb层的user\_name非主键索引取出id，因为只拿主键id，不需要回表，所以这块性能会稍微快点，在返回server层之后，同样抛弃前600w条数据，保留最后的100个id。然后再用这100个id去跟t1表做id匹配，此时走的是主键索引，将匹配到的100条行数据返回。这样就绕开了之前的600w条数据的回表。

当然，跟上面的case一样，还是没有解决要白拿600w条数据然后抛弃的问题，这也是非常挫的优化。

像这种，当offset变得超大时，比如到了百万千万的量级，问题就突然变得严肃了。

这里就产生了个专门的术语，叫深度分页。

**深度分页问题**

深度分页问题，是个很恶心的问题，恶心就恶心在，这个问题，它其实无解。

不管你是用mysql还是es，你都只能通过一些手段去"减缓"问题的严重性。

遇到这个问题，我们就该回过头来想想。

为什么我们的代码会产生深度分页问题?

它背后的原始需求是什么，我们可以根据这个做一些规避。

**如果你是想取出全表的数据**

有些需求是这样的，我们有一张数据库表，但我们希望将这个数据库表里的所有数据取出，异构到es，或者hive里，这时候如果直接执行

select \* from page;

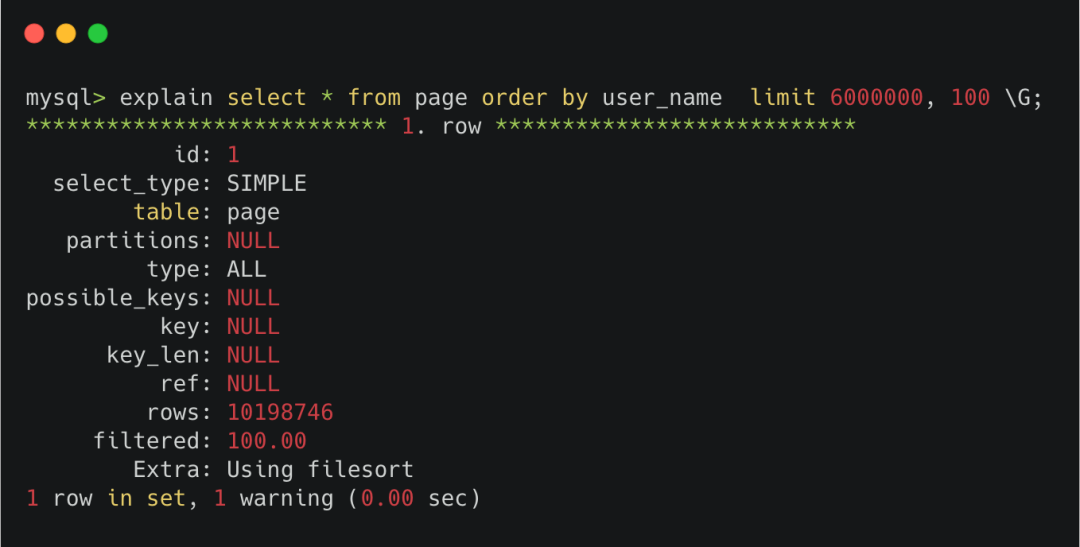
这个sql一执行，狗看了都摇头。

因为数据量较大，mysql根本没办法一次性获取到全部数据，妥妥超时报错。

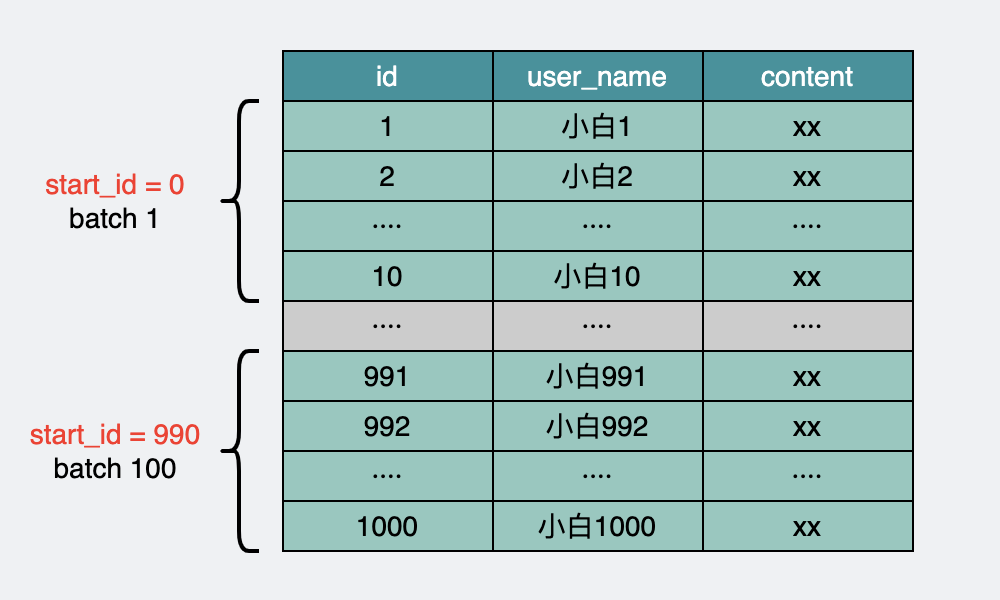
于是不少mysql小白会通过limit offset size分页的形式去分批获取，刚开始都是好的，等慢慢地，哪天数据表变得奇大无比，就有可能出现前面提到的深度分页问题。

这种场景是最好解决的。

我们可以将所有的数据根据id主键进行排序，然后分批次取，将当前批次的最大id作为下次筛选的条件进行查询。

可以看下伪代码

batch获取数据

这个操作，可以通过主键索引，每次定位到id在哪，然后往后遍历100个数据，这样不管是多少万的数据，查询性能都很稳定。

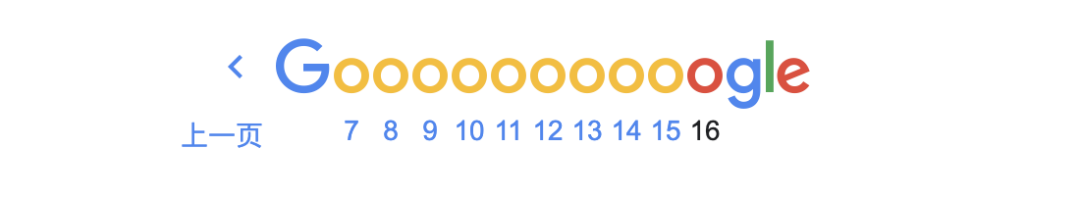
batch分批获取user表

**如果是给用户做分页展示**

如果深度分页背后的原始需求只是产品经理希望做一个展示页的功能，比如商品展示页，那么我们就应该好好跟产品经理battle一下了。

什么样的翻页，需要翻到10多万以后，这明显是不合理的需求。

是不是可以改一下需求，让它更接近用户的使用行为?

比如，我们在使用谷歌搜索时看到的翻页功能。

一般来说，谷歌搜索基本上都在20页以内，作为一个用户，我就很少会翻到第10页之后。

作为参考。

如果我们要做搜索或筛选类的页面的话，就别用mysql了，用es，并且也需要控制展示的结果数，比如一万以内，这样不至于让分页过深。

如果因为各种原因，必须使用mysql。那同样，也需要控制下返回结果数量，比如数量1k以内。

这样就能勉强支持各种翻页，跳页(比如突然跳到第6页然后再跳到第106页)。

但如果能从产品的形式上就做成不支持跳页会更好，比如只支持上一页或下一页。

上下页的形式

这样我们就可以使用上面提到的start\_id方式，采用分批获取，每批数据以start\_id为起始位置。这个解法最大的好处是不管翻到多少页，查询速度永远稳定。

听起来很挫?

怎么会呢，把这个功能包装一下。

变成像抖音那样只能上划或下划，专业点，叫瀑布流。

是不是就不挫了?

**总结**

* limit offset, size 比 limit size 要慢，且offset的值越大，sql的执行速度越慢。
* 当offset过大，会引发深度分页问题，目前不管是mysql还是es都没有很好的方法去解决这个问题。只能通过限制查询数量或分批获取的方式进行规避。
* 遇到深度分页的问题，多思考其原始需求，大部分时候是不应该出现深度分页的场景的，必要时多去影响产品经理。
* 如果数据量很少，比如1k的量级，且长期不太可能有巨大的增长，还是用limit offset, size 的方案吧，整挺好，能用就行。

# 数据库null值

注意：

（1）**负向比较**（例如：!=）会引发**全表扫描**；

（2）如果允许空值，**不等于(!=)的查询，不会将空值行(row)包含进来**，此时的结果集往往是不符合预期的，此时往往要加上一个or条件，把空值(is null)结果包含进来；

（3）or可能会导致全表扫描，此时可以优化为union查询；

（4）建表时加上**默认(default)值**，这样能避免空值的坑；

（5）**explain**工具是一个好东西；