# 背景

分页查询用的是 limit 子句，SQL 语句形态如下：

select \* from table where xxx in (1,2,3) order by id limit #offset#, 200

limit 子句的优点很明显，简单好用。缺点平时不显著，数据量一大就暴露了。数据库会完整扫描 offset 的行，然后继续扫描 200 行之后才把结果集返回。 offset 在 400W 的时候，这样的 SQL 做一次分页查询就已经至少耗时 5s 了。

为了应对数据量大的问题，可以采用另一个方案，不使用 limit，直接使用主键索引 + 左右范围查找，SQL 语句形态如下。

select \* from table where xxx in (1,2,3) and id >= #minId# and id < #maxId#

其中minId 和maxId 由代码给出，启动时先算出当前表的 id 范围，然后在此范围内以 200 为步长分页读取，即 maxId - minId = 200，有可能读不到数据，有可能读到 200 条。

在数据库里面试着跑了几条这样的查询，果然效率高了很多。但是这种实际效果还是不够理想。

在日常数据库测试的时候就发现问题了。我的数据库的 id 不是由 MySQL 自动生成的连续的自增主键，而是通过其他中间件产生的，从整体来看，整个 id 的分布比较离散，步长 200 的时候，一次查询根本查不出几条数据。如果整张表的 maxId 比 minId 大出很多很多，会产生很多次无意义的查询。要想有比较好的命中率，还需要关心表里面 id 的分布，根据分布情况调整步长。

接下来我就在 limit 和 min-max-id 两种分页方案之间纠结，开始去分析线上表的数据分布，然后考虑把大表和小表区分对待，使用不同的分页策略等等。

select \* from table where id >= #minId# and xxx in (1,2,3) limit 200

回到家之后迫不及待开始写代码，先把这条 SQL 换着参数在数据库里面一遍遍执行，感觉这种分页方式完全符合我的要求，查询使用的主键索引，虽然从执行计划看影响行数在百万级，但是实际执行的时候影响行数不过百级，还不需要考虑 id 的分布，每次都能实打实的给我捞出 200 条数据。

不过使用这样的 SQL 语句做分页，需要注意调整 minId 的值，扫表过程中需要做到不重不漏。一种可行的方案是将本次查出的结果集中的最大的 id，自增 1 后作为下次查询的 minId。

max\_id, min\_id = select min(id), max(id) from table

while min\_id <= max\_id:

objs = select \* from table where id >= min\_id and xxx in (1,2,3) limit 200

max\_id\_in\_page = max(map(lambda x: x.id, objs))

min\_id = max\_id\_in\_page + 1

整体来说，这种分页方式避免了使用 limit 时候遍历 offset 带来的无谓的性能开销，避免了对 id 使用左右范围查询时候 id 的离散分布对命中率的影响，代价是需要在内存中遍历结果集获取当前分页中 id 的最大值，局限是只能在对全表唯一的字段做分页时使用。

# 扩展

关于分页查询，不同的数据库在做 offset 的时候语法也不太一样，比如 M$ 家的语法就是 TOP n，MySQL 的语法就是 limit n。

网上的很多博客都提到一种利用子查询来提高性能的做法，大体的意思是先使用普通的 offset 语法获取目标分页的记录的 id 集合，再根据 id 集合去获取完整的目标数据，SQL 语句形态如下。

select \* from table where id in (select id from table order by id limit #offset#, #size#)

其实有些版本尤其低版本的 MySQL 是不支持直接对带 limit 的子查询的结果做 in 子句的，执行的时候会报错，信息如下。

This version of MySQL doesn’t yet support ‘LIMIT & IN/ALL/ANY/SOME subquery’

针对这个问题，他们就有了一个改进之后的子查询版本。

select \* from table where id >= (select id from table order by id limit #offset#, 1)

其实是先取出目标分页的第一条记录的id， 然后根据 id 做范围查找和条数限制。这条 SQL 语句的效率在 offset 达到百万级时相比直接 limit 有数倍的提升，但是注意到 MySQL 子查询其实是一个坑，这条语句不但没有避免遍历 offset，还做了大量的无用重复工作。

本质上，最终使用的分页方案是对这条 SQL 语句的优化，借助 id 的有序性和唯一性，使用max(map(lambda x: x.id, objs)) + 1 替代了子查询。

有人为了绕过 MySQL 不支持对子查询结果做 in 子句的限制，脑洞大开写出了如下查询。

select \* from table where id in (select id from (select id from table order by id limit #offset#, #size#) as tmp)

既然不允许直接对带 limit 的子查询做 in，那么干脆用子查询套子查询，也是醉了。这条 SQL 语句的效率还不如直接 limit。

还有一种改进方案就是传说中的“查两次”。第一次先查出目标分页的 id 集合，因为只查 id，大部分情况可以直接命中索引然后返回，速度还是可以接受的。然后第二次直接根据 id 集合做 in 子句查询，走的主键索引，这次就是秒出了。

其实查两次对性能的影响需要具体到情景来分析，不能当做是万金油。

我在最终使用的分页方案，在我的应用场景下，性能是超过上面其余几种分页方案的。

在现实世界的开发中，分页远比预料的要复杂得多。

如果 Web 页面通过记录的创建时间来分页，那么很可能我最终使用的分页方案就不能套用了，因为时间不是一个全表唯一的值，很难在不使用 limit 的情况下做到不重不漏。如果运气不错，使用的是数据库的自增主键，那么可以认为主键的变化趋势和创建时间的变化趋势是等价的，可以将对时间的分页映射成对主键的分页。

在分页的交互逻辑上，天朝的产品经理似乎偏好于电梯式的分页控件。其实还是“抛开数据量谈实现就是耍流氓”，在数据量小的时候，使用什么样的分页方式对系统性能都没什么影响，但是在数据量大到一定程度的时候，电梯式的分页对系统性能的大量消耗反而会伤害所谓的用户体验。

关于大数据量下的分页实现，业界其实已经有了好几种策略，基于不同的假设。

假设随着时间的推移，越早产生的数据，价值越小。如果假设成立，我们可以认为绝大多数用户是没有这样一个需求要去查看他在系统中产生的第一条数据的。如果第一条数据对用户真的很重要，即使再困难他也会想办法得到。

基于这样的假设，在数据量足够大的情况下，我们就可以对系统实现和交互体验做出很多优化，比如不再提供精确的电梯式分页，取而代之的是下一页、上一页；不再提供精确的总页数，而是提供一个大概的条目总数，可以通过查看 SQL 的执行计划得到。

之前看过一个关于上一页、下一页的实现技巧。假如每页显示 20 条数据，那么查询数据库的时候，用limit #offset#, 21 取出 21 条记录，页面展现20条。如果取到了 21 条，说明下一页还有数据，在页面展示下一页按钮。如果结果集数量不足 21，说明已经到了最后一页，无需显示下一页按钮了。这种方式完全避免了在分页查询时对总条目数量的查询。

还有一种策略是基于这样的假设：用户比较关心的是最近产生的一小部分数据。在用户查询的时候，我们可以一次性从数据库查询符合条件的 N 条数据缓存起来，足够用户翻个几页，这样哪怕是使用电梯式分页，在计算总页数时也无需查询数据库了。

# 总结