# 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。采用的数据库的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

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。

# 分页查询

**方法1：直接使用数据库提供的SQL语句**

语句样式：MySQL中,可用如下方法: SELECT \* FROM 表名称 LIMIT M,N

适应场景：适用于数据量较少的情况(元组百/千级)

原因/缺点：全表扫描,速度会很慢且有的数据库结果集返回不稳定(如某次返回1,2,3,另外的一次返回2,1,3)。Limit限制的是从结果集的M位置处取出N条输出，其余抛弃。

**方法2：建立主键或唯一索引，利用索引(假设每页10条)**

语句样式：MySQL中，可用如下方法： SELECT \* FROM 表名称 WHERE id\_pk > (pageNum\*10) LIMIT M

适应场景：适用于数据量多的情况(元组数上万)

原因：索引扫描，速度会很快。有人提出：因为数据查询出来并不是按照pk\_id排序的，所以会有漏掉数据的情况，只能方法3

**方法3：基于索引再排序**

语句样式：MySQL中，可用如下方法： SELECT \* FROM 表名称 WHERE id\_pk > (pageNum\*10) ORDER BY id\_pk ASC LIMIT M

适应场景：适用于数据量多的情况(元组数上万)。最好ORDER BY后的列对象是主键或唯一所以，使得ORDERBY操作能利用索引被消除但结果集是稳定的(稳定的含义，参见方法1)

原因：索引扫描,速度会很快。但MySQL的排序操作，只有ASC没有DESC(DESC是假的，未来会做真正的DESC)。

**方法4：基于索引使用prepare**

第一个问号表示pageNum，第二个？表示每页元组数

语句样式: MySQL中，可用如下方法: PREPARE stmt\_name FROM SELECT \* FROM 表名称 WHERE id\_pk > (？\* ？) ORDER BY id\_pk ASC LIMIT M

适应场景：大数据量

原因：索引扫描,速度会很快. prepare语句又比一般的查询语句快一点。

**方法5：利用MySQL支持ORDER操作可以利用索引快速定位部分元组，避免全表扫描**

比如：读第1000到1019行元组(pk是主键/唯一键).

SELECT \* FROM your\_table WHERE pk>=1000 ORDER BY pk ASC LIMIT 0,20

**方法6：利用"子查询/连接+索引"快速定位元组的位置，然后再读取元组。**

比如(id是主键/唯一键,蓝色字体时变量)

利用子查询示例：

SELECT \* FROM your\_table WHERE id <=

(SELECT id FROM your\_table ORDER BY id desc LIMIT ($page-1)\*$pagesize ORDER BY id desc

LIMIT $pagesize

利用连接示例：

SELECT \* FROM your\_table AS t1

JOIN (SELECT id FROM your\_table ORDER BY id desc LIMIT ($page-1)\*$pagesize AS t2

WHERE t1.id <= t2.id ORDER BY t1.id desc LIMIT $pagesize;

mysql大数据量使用limit分页，随着页码的增大，查询效率越低下。

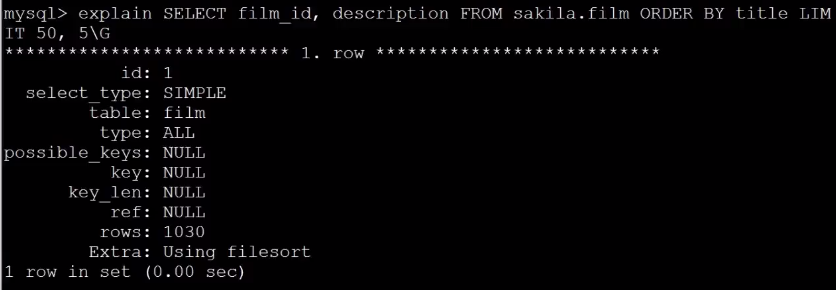
# 优化

## 单机优化

明知只有一条查询结果，那请使用 “LIMIT 1”，“LIMIT 1”可以避免全表扫描，找到对应结果就不会再继续扫描了。

limit常用于分页处理，时常会伴随order by从句使用，因此大多时候会使用filesorts这样会造成大量的IO问题。

SELECT film\_id,description FROM sakila.film ORDER BY title LIMIT 50,5;

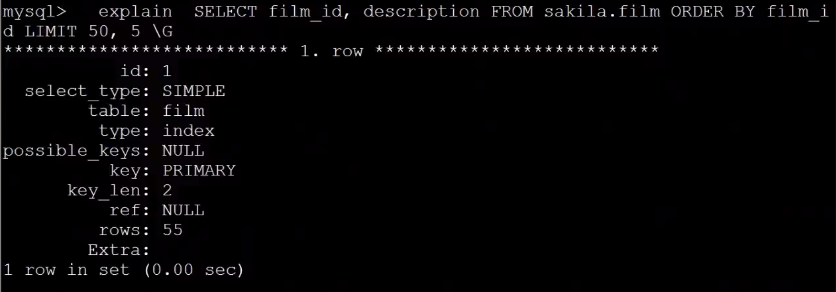


优化limit查询：

1. 使用有索引的列或主键进行order by操作

SELECT film\_id,description FROM sakila.film ORDER BY film\_id LIMIT 50,5;

注：使用主键排序，不会再使用文件排序，会使用索引，避免很多IO操作，扫描的行数比之前少了很多。



但是上述方法存在一个问题，分页查询的越大，则扫描的行越多。

1. 记录上次返回的主键，在下次查询时使用主键过滤，避免数据量大时扫描大量的记录。

SELECT film\_id,description FROM sakila.film WHERE film\_id>55 AND film\_id<=60 ORDER BY film\_id LIMIT 1,5;



注：使用这种方式的缺点就是要求主键顺序增长排序且连续的，如果出现了空缺的某几行，可能会出现最后的结果不足5行的情况。

## 分布式优化

对于单机数据库而言，以MySQL为例，功能部分，直接按照自增主键从小到大排列即可，因为自增主键的大小能够区分出数据生成的前后关系，所以功能上不存在问题。

性能部分优化，在SQL带其他过滤条件的情况下，可以将扁平的带LIMIT m，n语句优化成嵌套子查询以便让优化器做索引覆盖，避免在磁盘上遍历数据，SQL 如下所示：

SELECT \*

FROM table a JOIN(

SELECT pk FROMtable

WHERE some\_column= ?

ORDER BY pk LIMIT m, n) b ON a.pk= b.pk

**在分布式数据库中实现的难点**

对于单机数据库来说，因为存在时间序的自增主键，这个需求变得比较简单，但是对于分布式数据库而言，这个问题就变得比较复杂，主要因为主键大小和数据生成时间并没有本质关联，即使是分布式强一致数据库，保证严格时间序代价也很高，这个导致只能通过类似GMT\_CREATE这种时间字段进行排序分页，但是GMT\_CREATE可能重复，或者存在大量重复，这个导致分页按时间排序处理变得更加复杂。

另外性能层面，我们如同MySQL不能使用扁平的带LIMIT m,n语句进行分页处理，但是同时也不能优化成带子查询的分页语句，因为数据分片的原因，需要将LIMIT m,n优化成LIMIT 0,m+n，分页挪到非常大的时候，需要返回到 DRDS大量数据，再skip掉不必要的数据，即使做了倒序优化，一张拆分表性能最差的一页数据查询需要返回表中一半的数据才能满足需求，这个在拆分表有10亿或者100亿数据的时候，很难满足性能需求。

**DRDS的方案**

对这个问题仔细分析后，我们提出了一个方案。为了避免返回大量的中间结果数据，我们希望不指定LIMIT的OFFSET，而是用上一页的最大值直接跳转到下一页的起始处，因此业务做一定的妥协，功能上增加如下约束：

1、提供下一页、上一页、首页、尾页功能

2、可以在当前页相邻的几页进行跳转（例如前后10页）

3、不允许做任意页的跳转

对于单机数据库，我们可以使用如下的SQL实现上述的需求：

SELECT \*

FROM table

WHERE pk> 上一页pk最大值ORDRE BY pk LIMIT n

分布式数据库：

1、增加GMT\_CREATE字段，其默认值为CURRENT\_TIMESTAMP

2、我们仍然按照GMT\_CREATE时间字段进行第一个维度排序，但是GMT\_CREATE有重复，我们希望有一个固定顺序，所以再以主键为第二个维度排序，所以排序部分为ORDER BY gmt\_create,pk。

3、我们记住每一页GMT\_CREATE和PK的最大值作为下一页数据的起始值，但是前面提到过GMT\_CREATE有数据重复，如果通过简单AND条件拼接，会导致漏数据，所以我们将条件写为：

gmt\_create>= ?

AND(gmt\_create> ?or pk> ?)

从查询逻辑上规避掉GMT\_CREATE重复带来的可能漏数据的状况，分页SQL变成了这个：

SELECT \*

FROM page\_test

WHERE gmt\_create>= ?

AND(gmt\_create> ?

OR pk> ?)

ORDER BY gmt\_create,

pk LIMIT n

当然在MySQL 5.7上，我们还可以直接做二元组的比较（5.7之前的版本，多元组的比较MySQL无法利用到组合索引）：

SELECT \*

FROM page\_test

WHERE(gmt\_create, pk)> （?, ?）ORDER BY gmt\_create, pk LIMIT n

# 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MS的语法就是TOP n，DB 2的语法是FETCH ROW(s)，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。