【MySQL】SQL优化(九)

> 🚚MySQL学习·第九站~
> 🚩本文已收录至专栏：MySQL通关路
> ❤️‍🩹文末附全文思维导图，感谢各位点赞收藏支持~
> ⭐学习汇总贴，超详细思维导图：【MySQL】学习汇总(完整思维导图)

一.插入数据

(1) 小规模数据

如果我们需要一次性往数据库表中插入多条记录:

```
-- 例如我们需要插入大量数据
insert  into  tb_test  values(1,'tom');
insert  into  tb_test  values(2,'cat');
insert  into  tb_test  values(3,'jerry');
....
```

我们可以从以下三个方面进行优化~

(1.1) 批量插入数据

由于每次`insert`都需要**与数据库建立连接，进行网络传输导致一定的性能损失**，我们可以选择一次性插入多条数据，代替一条一条插入。

```
-- 例如上述多条插入改为
Insert  into  tb_test  values(1,'Tom'),(2,'Cat'),(3,'Jerry');
```

不过一次性插入数据不建议超过500~1000条，大批量数据我们也可以拆分为多个insert批量插入。

(1.2) 手动控制事务

由于MySQL中事务提交的方式默认是自动提交的，也就是说当我们执行完一条insert语句后，他就会自动提交事务，如此可能会涉及到频繁的事务开启与提交。因此，我们还可以手动控制事务，减少事务开关所花费的时间

```
-- 多条插入手动控制事务
start  transaction;   -- 开启事务
insert  into  tb_test  values(1,'Tom'),(2,'Cat'),(3,'Jerry');
insert  into  tb_test  values(4,'Tom'),(5,'Cat'),(6,'Jerry');
insert  into  tb_test  values(7,'Tom'),(8,'Cat'),(9,'Jerry');
commit;    -- 提交事务
```

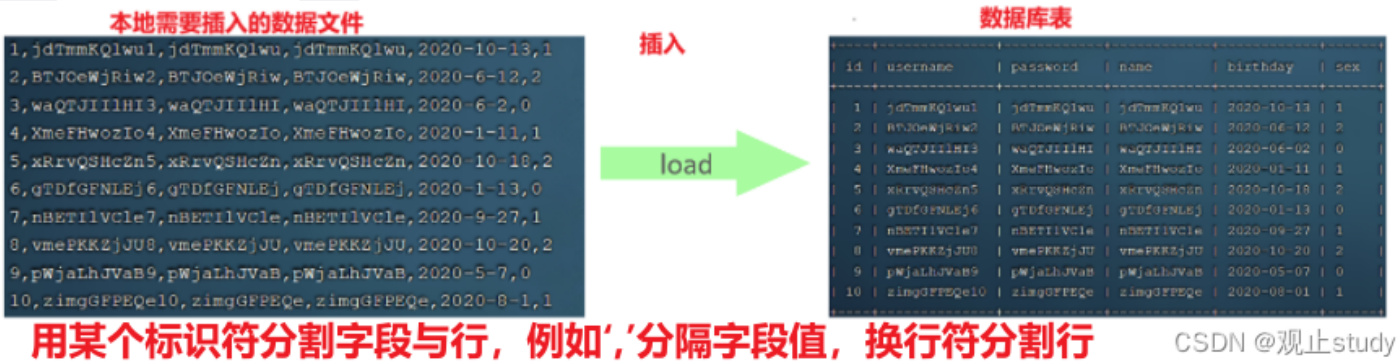
(1.3) 主键顺序插入

由于主键索引的存在，每次插入数据都可能会重新组织索引结构，因此，主键顺序插入，性能要高于乱序插入。

```
-- 主键乱序插入 : 8  1  9  21  88  2  4  15  89  5  7  3
-- 主键顺序插入 : 1  2  3  4  5  7  8  9  15  21  88  89
insert  into  tb_test  values(1,'Tom'),(2,'Cat'),(3,'Jerry');  -- 顺序插入
insert  into  tb_test  values(5,'Cat'),(6,'Jerry'),(4,'Tom');  -- 乱序插入
```

(2) 大批量数据

如果**一次性需要插入大批量数**据(比如: **几百万**的记录)，使用insert语句插入性能较低，此时可以使用MySQL数据库提供的`load`指令进行插入。通过`load`指令我们可以一次性将本地文件当中的数据全部加载进数据库表结构中。



可以执行如下指令，将数据脚本文件中的数据加载到表结构中：

- （1）客户端连接服务端时，加上参数 `--local-infile`

```
mysql --local-infile -u root -p
-- --local-infile 表示需要加载本地文件
```

- （2）设置全局参数local_infile为1，**开启从本地加载文件导入数据的开关**

```
set  global  local_infile = 1;
```

- （3）执行load指令将准备好的数据，加载到表结构中

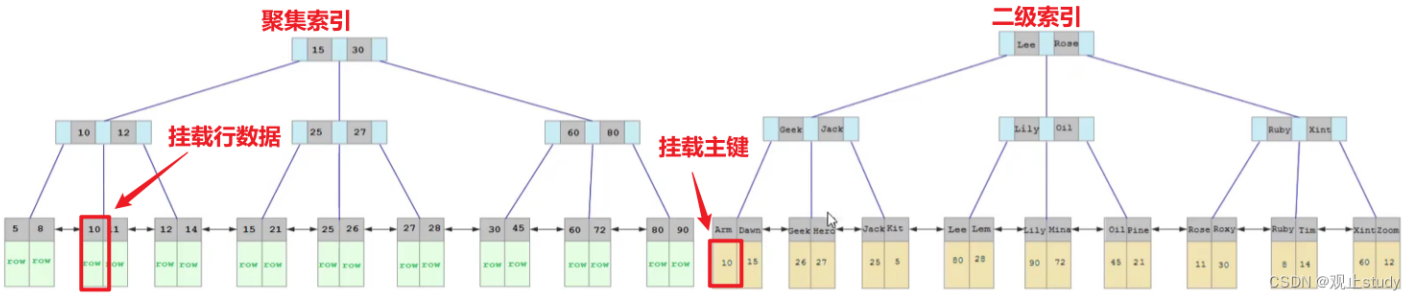
```
load  data  local  infile  '/root/sql1.log'  into  table  tb_user  fields
terminated  by  ','  lines  terminated  by  '\n' ;

--  local  infile  '需要加载的文件路径'
--  into  table  需要插入到哪张表
--  fields terminated  by  '字段分隔符'
--  lines  terminated  by  '行分隔符'
```

二.主键优化

在满足业务需求的情况下，尽量遵循以下原则设计主键。

(1) 降低主键的长度

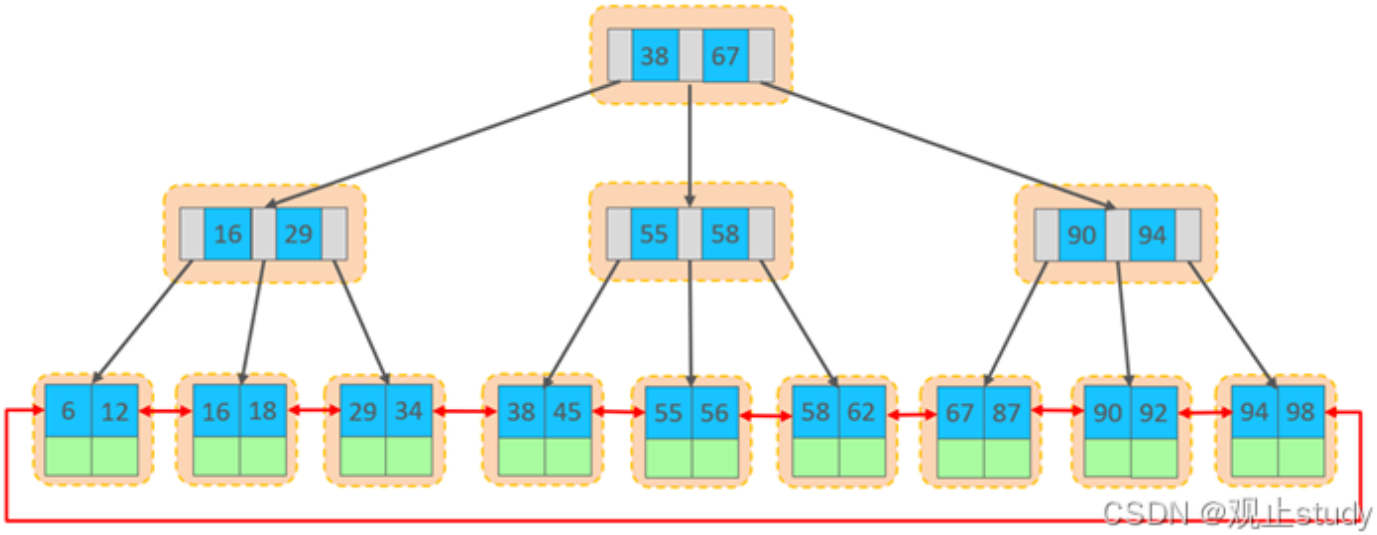
对于一张表来说主键索引只有一个，但是二级索引可能会有很多个，在二级索引的叶子节点当中挂的就是数据的主键，因此，如果主键长度比较长且二级索引比较多，将会占用大量的磁盘空间。



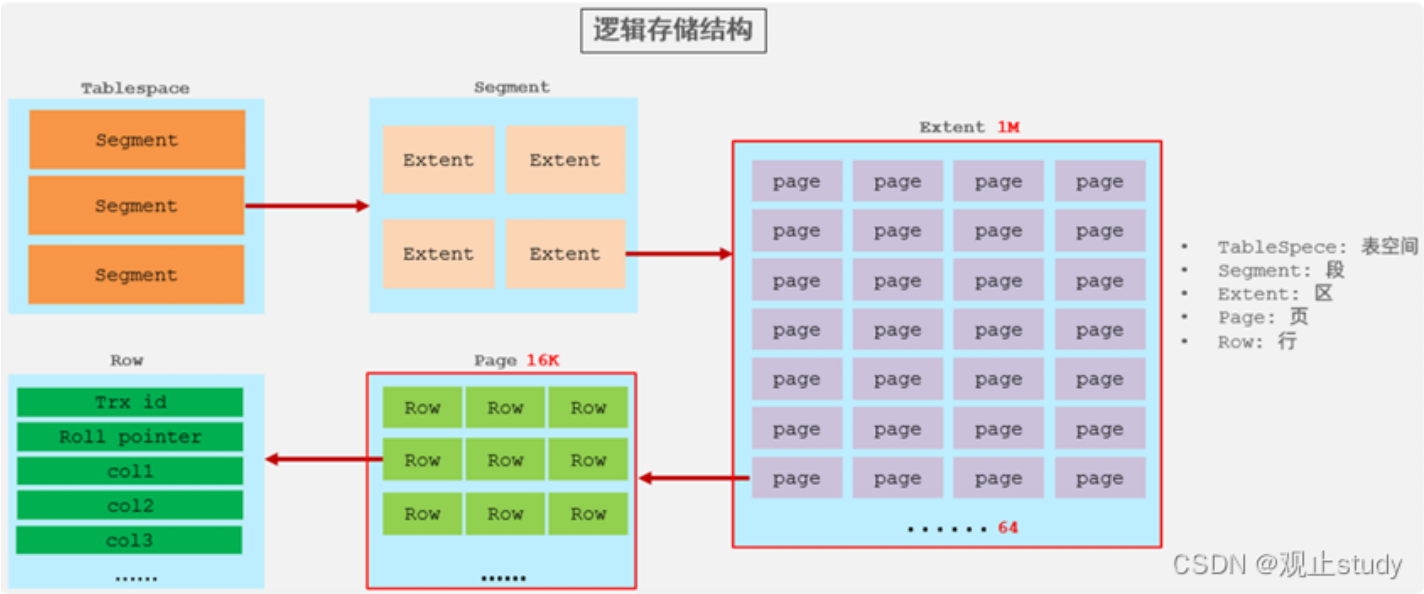
(2) 顺序插入

在条件允许的情况下，使用AUTO_INCREMENT自增主键，顺序插入数据。

在InnoDB存储引擎中，表数据都是根据主键顺序组织存放的，行数据都是存储在聚集索引的叶子节点上的。

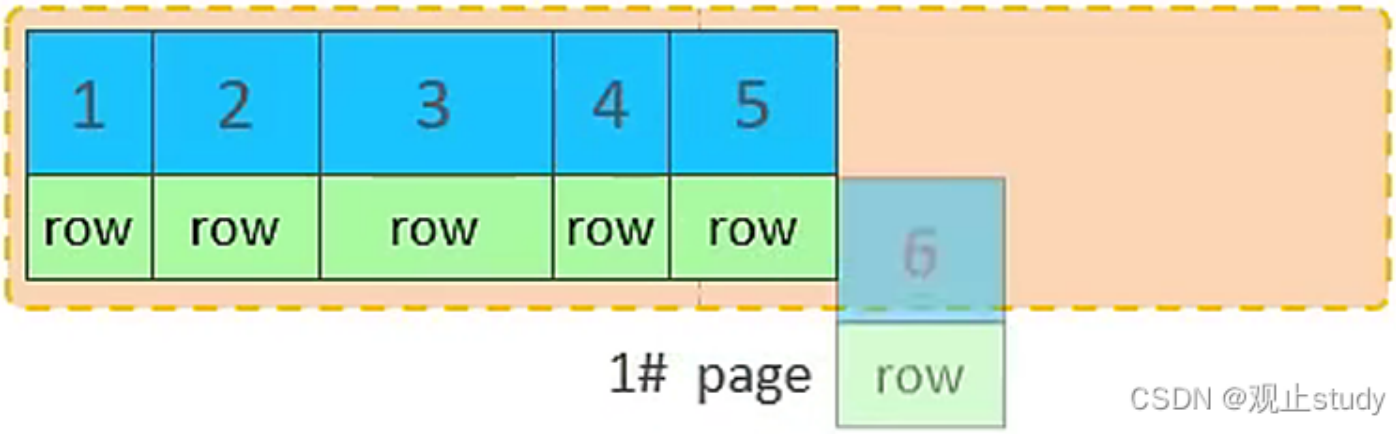


而数据行是记录在逻辑结构 page 页中的，而每一个页的大小是固定的，默认16K。 那也就意味着， 一个页中所存储的行也是有限的，如果插入的数据行row在该页存储不下，将会存储到下一个页中，页与页之间会通过指针连接。

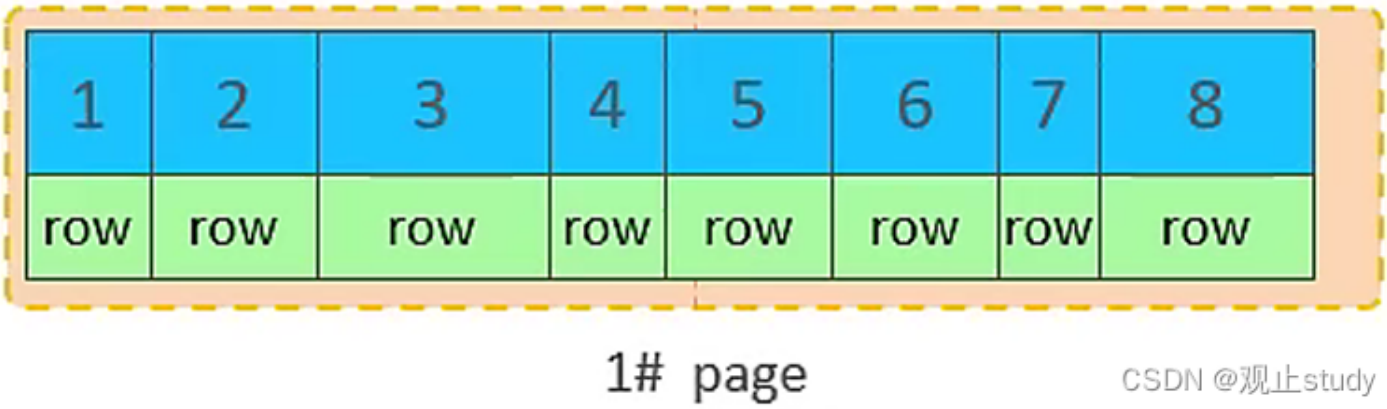


- **主键顺序插入效果**

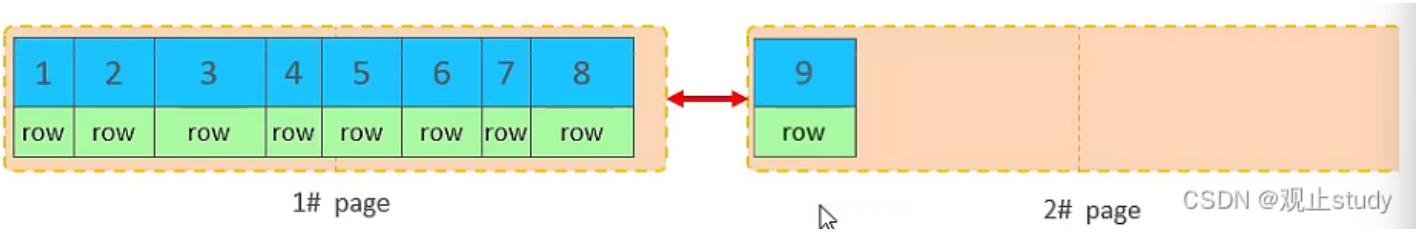
1. 从磁盘中申请页， 主键顺序插入



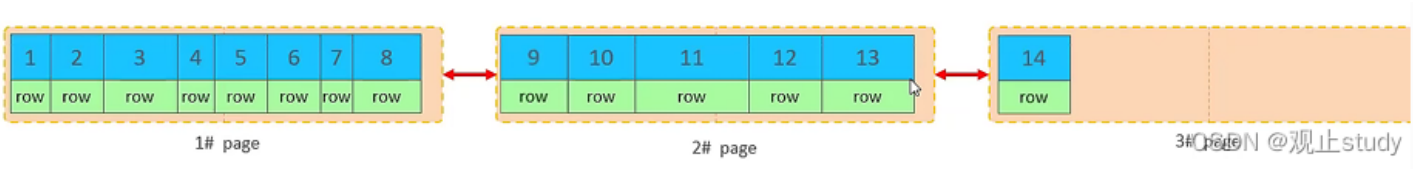
2. 第一个页没有满，继续往第一页插入



4. 当第一个也写满之后，再写入第二个页，页与页之间会通过指针连接



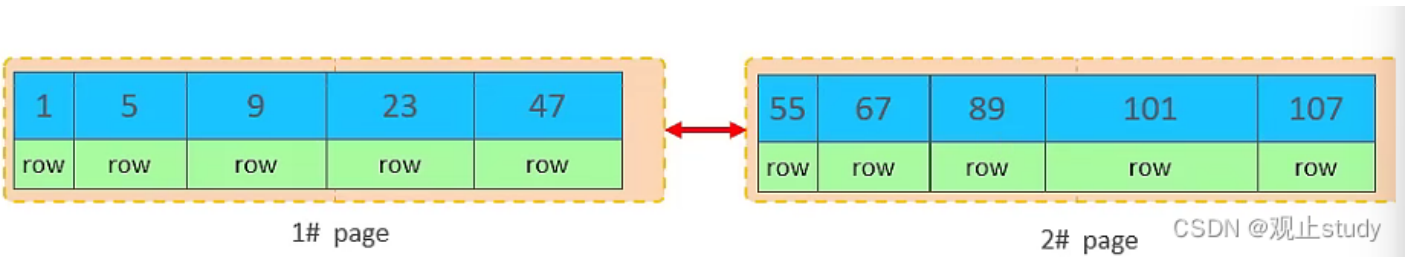
4. 当第二页写满了，再往第三页写入



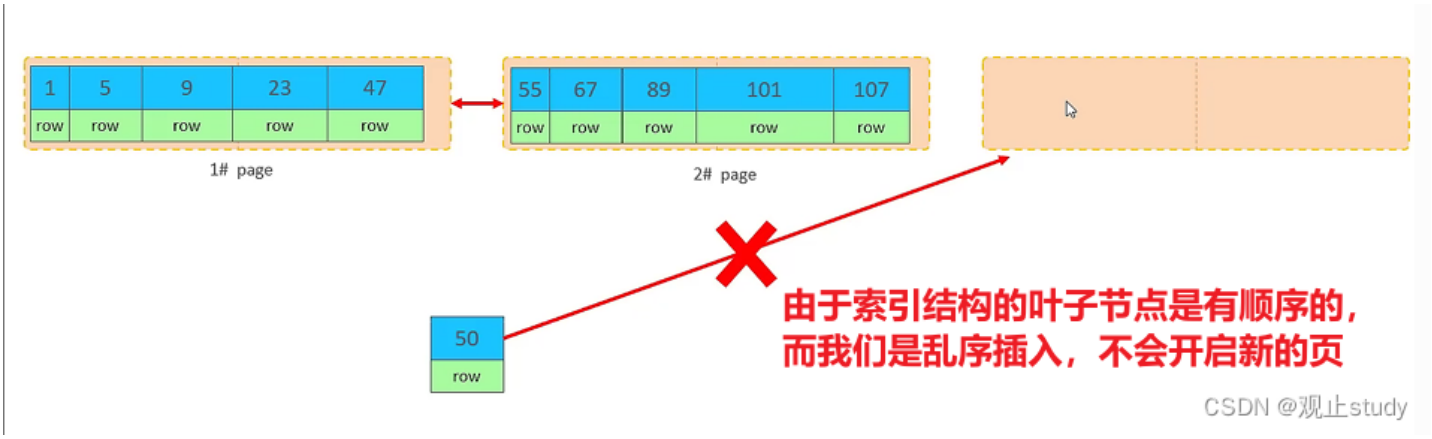
如此往复，没有任何额外损耗性能的情况。

- **主键乱序插入效果**

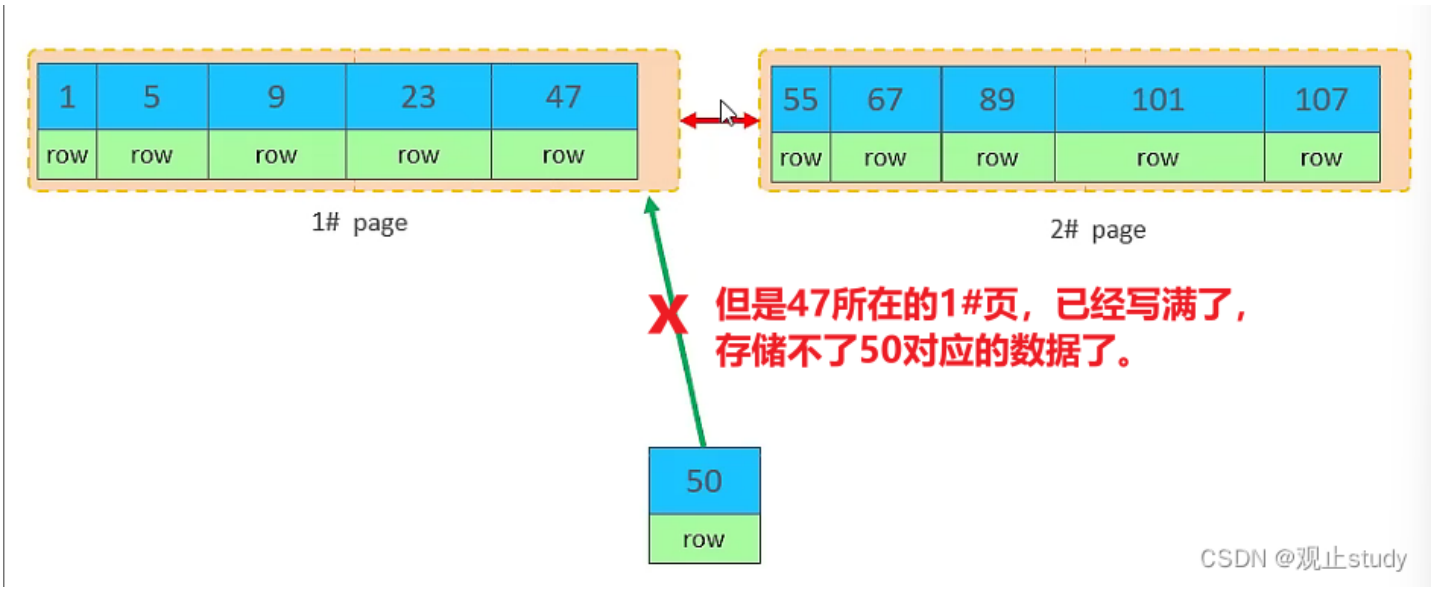
1. 假如1#,2#页都已经写满了



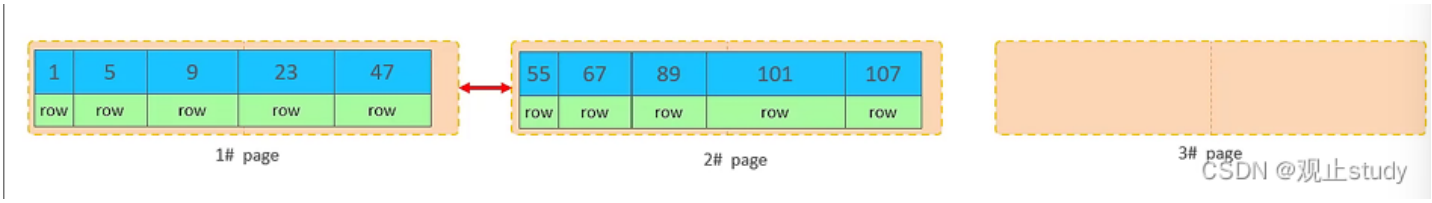
2. 此时再插入id为50的记录



3. 按照顺序，应该存储在47之后

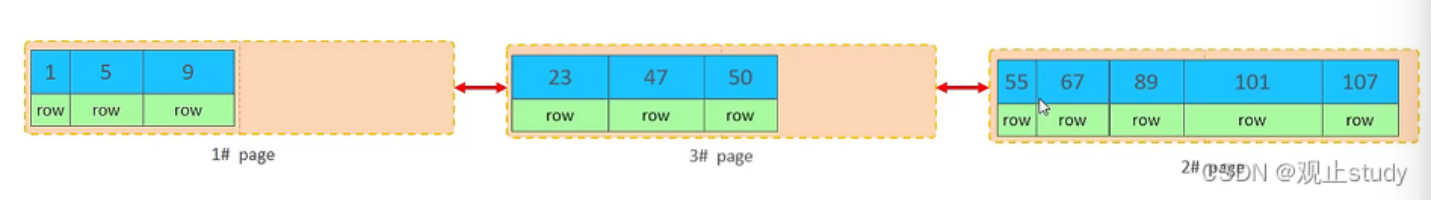


4. 此时会开辟一个新的页 3#



5. 但是并不会直接将50存入3#页，而是会将1#页后一半的数据，移动到3#页，然后在3#页，插入50。

6. 移动数据，并插入id为50的数据之后，那么此时，这三个页之间的数据顺序是有问题的。 1#的下一个 页，应该是3#，3#的下一个页是2#。 所以，此时，需要重新设置链表指针



上述的这种现象，称之为 "**页分裂**"，是**比较耗费性能的操作**。

页分裂指的是：页可以为空，也可以填充一半，也可以填充100%。每个页包含了2-N行数据(如果一行数据过大，会行 溢出)，根据主键排列

(3) 避免对主键的修改

尽量不要使用有意义的值作为主键，如身份证号，避免在进行业务操作对主键产生修改操作。这是因为插入修改删除操作都会导致数据库重新组织索引结构。

三.order by优化

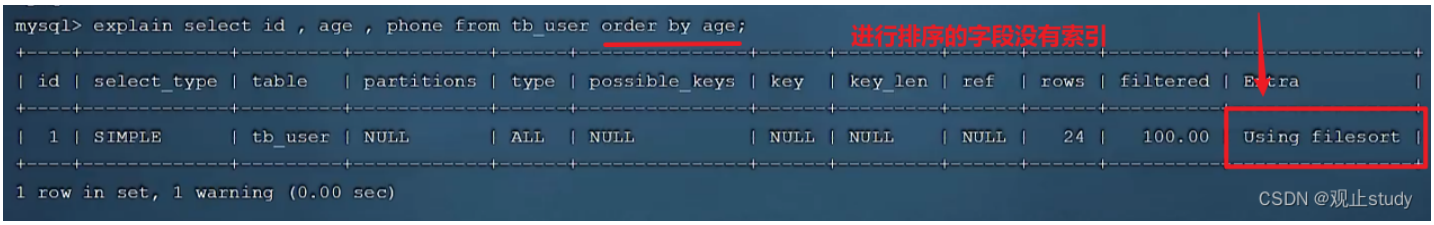
(1) 排序说明

MySQL的排序，有两种方式：

- `Using filesort`：通过表的索引或全表扫描，读取满足条件的数据行，然后在排序缓冲区sort buffer中完成排序操作，所有不是通过索引直接返回排序结果的排序都叫 FileSort 排序。
- `Using index`：通过有序索引顺序扫描直接返回有序数据，这种情况即为 using index，不需要 额外排序，操作效率高。

对于以上的两种排序方式，Using index的性能高，而Using filesort的性能低，我们在优化排序 操作时，尽量要优化为 Using index。

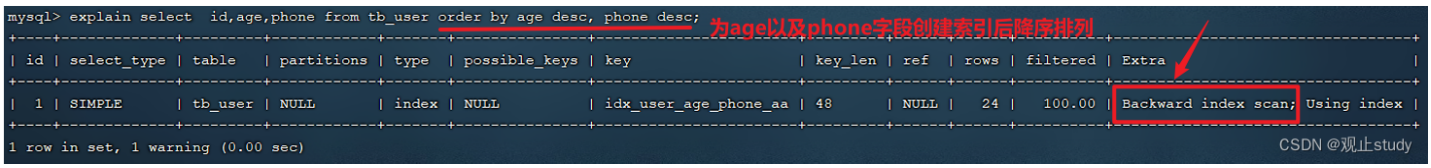
- 排序字段值没有索引，Using filesort



- 为排序字段值创建索引后，Using index

(2) 相关情况

1. 由于我们在MySQL中创建的索引，默认的叶子节点是从小到大排序的。如果我们在查询的时候，`order by`是从大到小即降序`desc`，那么除了出现 Using index， 也会出现了`Backward index scan`，这个代表**反向扫描索引**。 在MySQL8版本中，支持降序索引，我们也可以创建降序索引。




```
-- 语法
create index 索引名 on 表名(字段名 desc);
```

2. 排序时,也需要满足最左前缀法则（与where条件不同的是，此时必须按照创建索引时的顺序进行排序）,否则也会出现filesort。

```
mysql> EXPLAIN SELECT id,age,phone from user ORDER BY phone desc,age desc;
+----+-------------+-------+------------+-------+---------------+-----------------------+---------+------+------+----------+-----------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key                   | key_len | ref  | rows | filtered | Extra                                   |
+----+-------------+-------+------------+-------+---------------+-----------------------+---------+------+------+----------+-----------------------------------------+
|  1 | SIMPLE      | user  | NULL       | index | NULL          | idx_user_age_phone_ad | 48      | NULL |    8 |   100.00 | Using index; Using filesort             |
+----+-------------+-------+------------+-------+---------------+-----------------------+---------+------+------+----------+-----------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

顺序必须与我们创建索引时保持一致

CSDN @观止study

因为在创建索引的时候， age是第一个 字段，phone是第二个字段，所以排序时，也就该按照这个顺序来，否则就会出现Using filesort

3. 在条件允许的情况下，**尽量使用覆盖索引代替***，否则由于回表查询依旧会出现Using filesort

```
mysql> EXPLAIN SELECT id,age,phone from user ORDER BY age desc,phone desc;
+----+-------------+-------+------------+-------+---------------+-----------------------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key                   | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+-----------------------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | user  | NULL       | index | NULL          | idx_user_age_phone_ad | 48      | NULL |    8 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+-----------------------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * from user ORDER BY age desc,phone desc;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
|  1 | SIMPLE      | user  | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    8 |   100.00 | Using filesort |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
1 row in set, 1 warning (0.00 sec)
```

尽量使用覆盖索引

CSDN @观止study

四.group by优化

与order by类似，分组就相当于大范围的排序。我们同样可以通过**使用索引字段进行分组**来提高效率。

```
mysql> explain select profession,count(*) from tb_user group by profession;
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | SIMPLE      | tb_user | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   24 |   100.00 | Using temporary |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)
```

性能相对较低

```
mysql>
mysql>
mysql>
mysql> create index idx_user_pro_age_sta on tb_user(profession,age,status);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

创建索引

```
mysql>
mysql> explain select profession,count(*) from tb_user group by profession;
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-------------+
| id | select_type | table   | partitions | type  | possible_keys         | key                   | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | tb_user | NULL       | index | idx_user_pro_age_sta | idx_user_pro_age_sta | 54      | NULL |   24 |   100.00 | Using index |
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

性能相对较高

CSDN @观止study

此外，对于分组操作，在使用联合索引时，也是符合最左前缀法则的。例如下面：我们发现，如果仅仅根据age分组，就会出现 Using temporary； 而如果是 根据 profession,age两个字段同时分组，则不会出现 Using temporary。

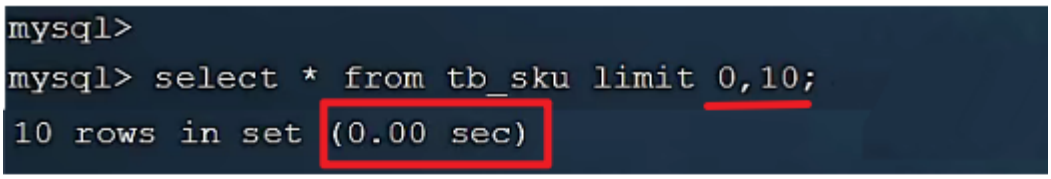
```
mysql> explain  select  profession , count(*)  from  tb_user   group  by  profession , age;
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-------------+
| id | select_type | table   | partitions | type  | possible_keys         | key                   | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | tb_user | NULL       | index | idx_user_pro_age_sta | idx_user_pro_age_sta | 54      | NULL |   24 |   100.00 | Using index |
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

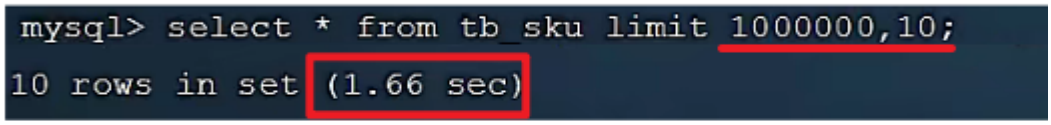
```
mysql> explain  select  age , count(*)  from  tb_user   group  by  age;
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+--------------------------+
| id | select_type | table   | partitions | type  | possible_keys         | key                   | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | tb_user | NULL       | index | idx_user_pro_age_sta | idx_user_pro_age_sta | 54      | NULL |   24 |   100.00 | Using index; Using temporary |
+----+-------------+---------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

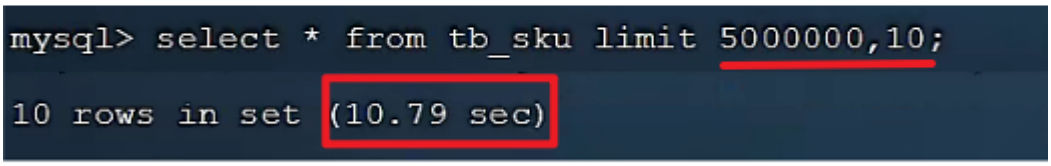
CSDN @观止study

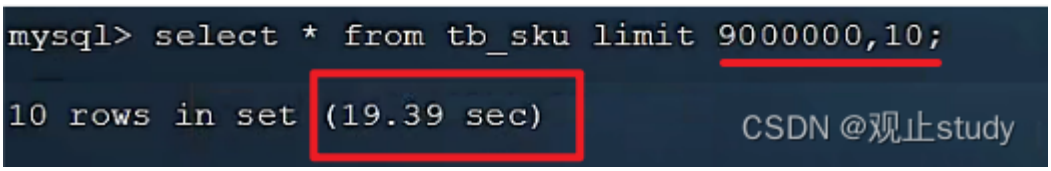
五.limit优化

在数据量比较大时，如果进行limit分页查询，在查询时，**越往后，分页查询效率越低**。这是因为当在进行分页查询时，例如执行 limit 2000000,10 ，此时需要MySQL排序前2000010 记 录，仅仅返回 2000000 - 2000010 的记录，其他记录丢弃，查询排序的代价非常大。





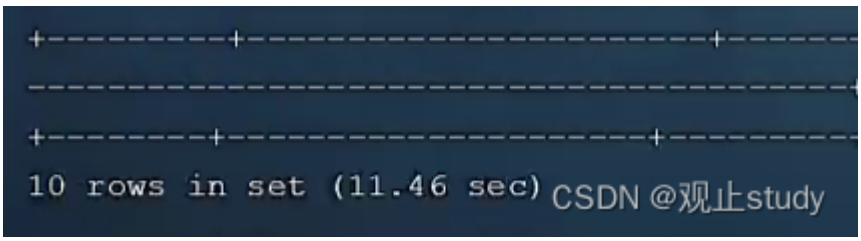




通过测试我们会看到，分页越往后，查询效率越低。

- 一般在进行分页查询时，我们可以通过 **覆盖索引 + 子查询** 的形式进行优化以提高性能。

```
--  select  *  from  tb_sku limit 9000000,10;
-- 例如对于上述测试示例 9000000,10 我们可以进行如下优化
explain select * from tb_sku t, (select id from tb_sku order by id limit 9000000,10) a where t.id
 =  a.id;
```



测试我们可以看到，耗费时间缩短了了近7秒，但是同时也增加了SQL语句复杂度，需要我们根据自身业务情况选择使用~

六.count优化

如果在数据量很大的情况下执行count操作是非常耗时的。

- 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； 但是如果是带条件的count，MyISAM也慢。
- InnoDB 引擎在执行 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，比较耗时。

如果说要提升InnoDB表的count效率，主要的优化思路是：

1. 自己计数，可以借助于redis这样的数据库进行,例如，插入一条数据，进行+1记录，删除一条数据进行-1记录。
2. 通过改进count的用法来提升count的效率。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，他有着如下四种不同的写法：

count用法	含义
`count(主键)`	InnoDB 引擎会遍历整张表，把每一行的 主键id 值都取出来，返回给服务层。 服务层拿到主键后，直接按行进行累加(主键不可能为null)
`count(字段)`	没有not null 约束 : InnoDB 引擎会遍历整张表把每一行的字段值都取出 来，返回给服务层，服务层判断是否为null，不为null，计数累加。 有not null 约束：InnoDB 引擎会遍历整张表把每一行的字段值都取出来，返 回给服务层，直接按行进行累加。
`count(数字)`	InnoDB 引擎遍历整张表，但不取值。服务层对于返回的每一行，放一个数字“1” 进去，直接按行进行累加。
`count(*)`	InnoDB引擎并不会把全部字段取出来，而是专门做了优化，不取值，服务层直接 按行进行累加

按照效率排序：`count(字段) < count(主键 id) < count(1) ≈ count(*)`，所以尽量使用 `count(*)`。

七.update优化

InnoDB默认事务级别使用的是行锁，**但是行锁是针对索引加的锁，不是针对记录加的锁 ,并且该索引不能失效，否则会从行锁 升级为表锁** 。也就是说在开启事务时：

- 我们能同时根据带有主键索引的不同id字段修改行记录

```
update  course  set  name = 'javaEE' where id  =  1 ;
update  course  set  name = 'Vue' where id  =  4 ;
```

- 但是无法同时根据不带索引的name字段修改行记录，因为此时行锁会升级为表锁，无法操作。

```
update course set name = 'SpringBoot' where name = 'PHP' ;
update update course set name = 'SpringBoot' where name = 'JS' ;
```

也就是说为了避免行锁升级为表锁影响执行效率，我们应当根据索引字段来进行更新操作。

八.全文概览