[数据库连接池到底应该设多大？](https://mp.weixin.qq.com/s?__biz=MzUxOTAxODc2Mg==&mid=2247486433&idx=1&sn=40dd9abbcd202d6a18b6960721ea337c&chksm=f9814b84cef6c2927842dba897575b415d7429d54d40206b143cc4f84e1e0f76d6bfb77ab7fb&mpshare=1&scene=1&srcid=&sharer_sharetime=1584342860713&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=bf6e306c5874d3c3ef6d324e96e03904cfa89e6f3dce067ee1581af27dec115dc04c7fa0ea1d90c5a158a8d8c7acc5ad774c8c9a3612cbb42a97202d2c41d9b3472a05a09a23a6a8707341ab230941ec&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AVMltAp8kFmIYN2iLRR5Kwc%3D&pass_ticket=B%2BlG9Q73ij2CS0QWkPAU7hN%2FOrst0lFE3qTly6tnhpu0h5HiSVVUv9bzLxO44OxF)

# Part 1 疫情项目优化案例

三个阶段

## 1.1 第一次

二月疫情项目上线，时间紧，任务重。很快出现性能问题。

现场环境：

Tomcat+mysql，默认配置

第三方直接操作DB，流量无限制

### 调整JVM，mysql buffer

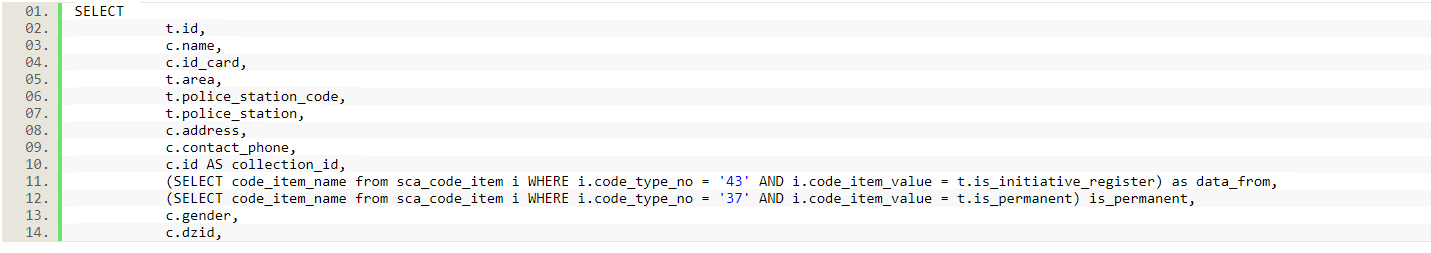
### 子查询

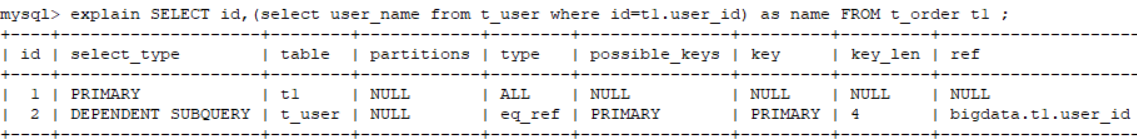
SubQuery

子查询：一个查询嵌套在另一个查询中。增强可读性高。

子查询位置：select中，from后，where中（group by和order by中无实用意义）

select子查询



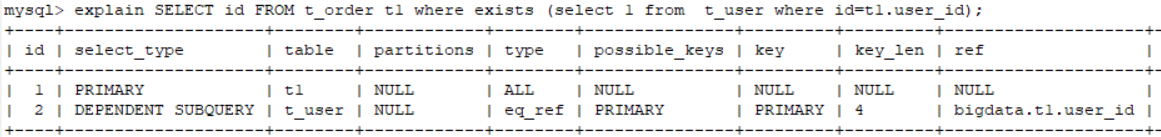


先执行子查询，select中的子查询为dependent subquery

对Outer Query的每一行对相关子查询进行一次评估

mysql嵌套子查询效率比较低，可以将其优化成连接查询。

Exist也是dependent subquery



### 索引遗漏

### 字段计算 索引失效

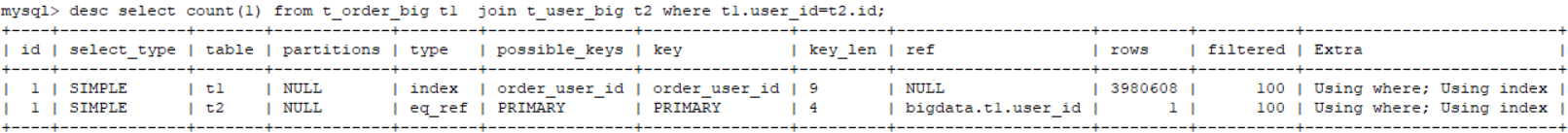
1. **where** substring(t.police\_station\_code,1,6) = :code
2. 优化后
3. **where** t.police\_station\_code like  concat(:code,"%")

### Group by去重

只为去重，代价太大。

### jpa工具类

优化count



时间: 7.696s

Web开发页面初始进入页面常常没有过滤条件

虽然join t2走索引，查找和比较索引次数都是R\*S

* 在短字段建索引，查询时间从3.35s下降到了0.26s，查询速度提升近13倍。
* t2数据冗余到t1，节省连表查找时间
* 连表占用大量查找时间，根据where中搜索条件，将t1,t2字段拆分到一张表

Count

不同引擎的优化

MyISAM：记录表的总行数。Select count直接返回，但不能有where条件

InnoDB：扫表的时候选择最小的索引来降低成本。

优化的前提都是没有where的条件查询。

为什么InnoDB不保存表的行数?

MyISAM是表级锁，不会有并发的数据库行数修改，所以查询得到的行数是准确的。

InnoDB支是行级锁，表的行数可能会被并发修改，select的缓存总行数就不准确了。

优化到6s以内

分页

mysql分页查询，抛弃读前n条，然后读m条，所以n越大，偏移量越大，性能越差。

1. **select** id,**name** **from** table\_name limit 866613, 20

优化

1. **select** id,**name** **from** table\_name **where** id> 866612 limit 20

## 1.2 第二次

背景：四月，武汉解禁，人口流动性增加，并发量突然大增。

Tomcat MaxConnect

## 1.3 第三次

背景：六月，北京人第二波疫情爆发，公安采集了很多数据，TASK表数据量最大单表100w左右

保险起见，调整JVM，mysql buffer/JVM/连接池

现象：时快时慢，mysql负载高

去掉左模糊

ICP

# Part 2 实现原理

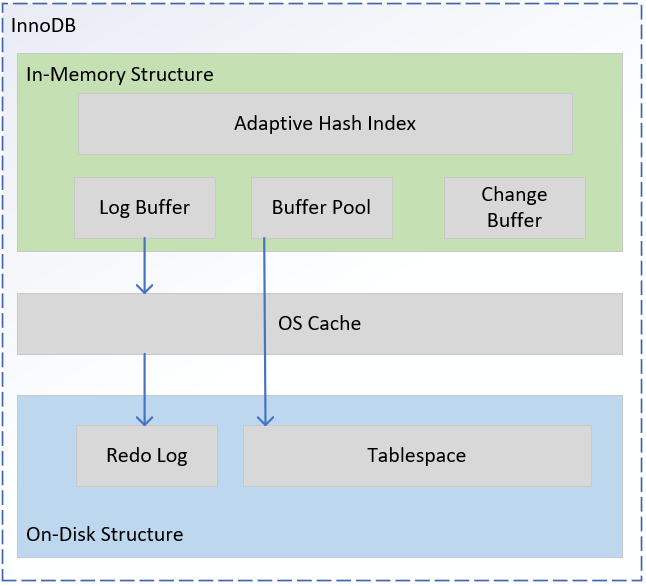
Mysql逻辑架构



* SQL生命周期：客户端🡪服务器🡪解析🡪生成执行计划🡪执行🡪客户端
* Connection/Thread handing：为请求做连接处理，授权认证，安全等。
* 执行：调用storage engine和调用后的数据处理，包括排序、分组等。
* Query Cache:

## 2.1 InnoDB内存结构

2.1逻辑架构



In-Memory Structure

内存结构包含四大核心组件

* Adaptive Hash Index：加速读请求，减少索引查询的寻路路径
* Buffer Pool：加速读请求。
* Change Buffer：加速写请求
* Log Buffer：包括redo log buffer和undo log buffer

### 3.1.1 Log Buffer

Redo log

数据库事务未提交时，记录修改后的新版本数据。磁盘随机写性能较低，如果每次都刷盘，会极大影响数据库的吞吐量。

* 修改行为先写到redo日志（随机优化为顺序写），定期将数据刷到磁盘上，极大提高性能
* 数据库崩溃，未刷盘数据在DB重启后，重做redo日志里已提交的事务内容。

Undo log

数据库事务未提交时，记录修改前的旧版本数据。事务回滚只需撤销未提交事务。

* insert：记录新数据的PK(ROW\_ID)，回滚时直接删除；
* delete/update:undo日志记录旧数据row，回滚时直接恢复；

他们分别存放在不同的buffer里。

回滚段(rollback segment)

存储undo日志的地方，记录事务修改前的数据，用于rollback操作。

undo日志和回滚段和InnoDB的MVCC密切相关，这里举个例子展开说明一下。

t(id PK,name)



此时没有事务未提交，故回滚段是空的。

接着启动了一个事务：

start trx;

delete (1, zhangsan);

update set(2, lisi) to (2, xxx);

insert (4, wangwu);

并且事务处于未提交的状态。



可以看到：

(1)被删除前的(1, shenjian)作为旧版本数据，进入了回滚段；

(2)被修改前的(3, lisi)作为旧版本数据，进入了回滚段；

(3)被插入的数据，PK(4)进入了回滚段；

接下来，假如事务rollback，此时可以通过回滚段里的undo日志回滚。

画外音：假设事务提交，回滚段里的undo日志可以删除。



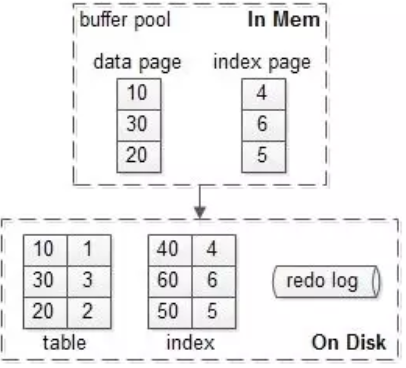
### 3.1.2 Buffer Pool

预读磁盘的数据，缓存表数据与索引数据。减少磁盘IO，加速访问。

什么是预读？

磁盘读写，并不是按需读取，而是按页读取，一次至少读一页数据（一般是4K），如果未来要读取的数据就在页中，就能够省去后续的磁盘IO，提高效率。

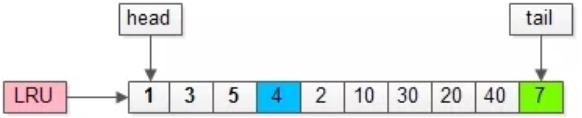
简单看一下：



* MySQL数据存储包含内存与磁盘两个部分；
* 内存缓冲池(buffer pool)以页为单位，缓存最热的数据页与索引页；

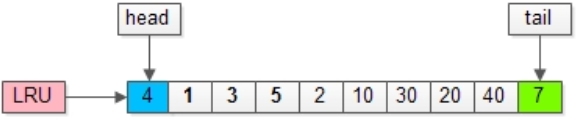
LRU算法

把入缓冲池的页放到LRU的头部，作为最近访问的元素



如上图，假如管理缓冲池的LRU长度为10，缓冲了页号为1，3，5…，40，7的页。

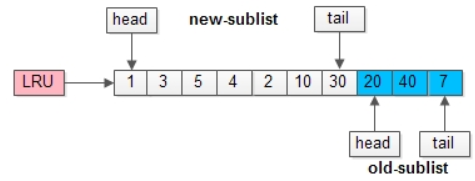
假如，接下来要访问的数据在页号为4的页中：



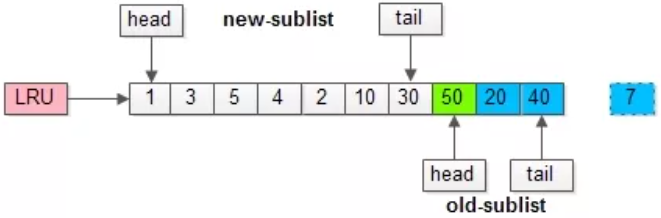
传统LRU算法十分直观，OS，memcache等很多软件都在用，两个问题：

* 预读失效：预读数据没用到。
* 缓冲池污染：批量扫描大量数据时，大量热数据被换出，导致性能急剧下降。

InnoDB以变种LRU算法管理缓冲池,将LRU链分为两个部分：新生代(new sublist)和老生代(old sublist)



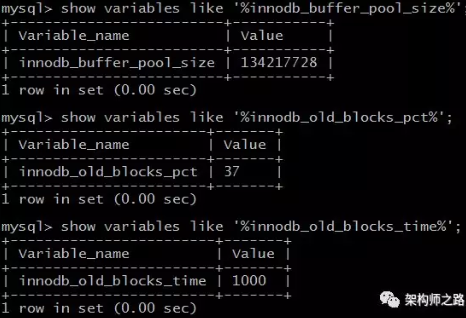
新生代的尾(tail)连接着老生代的头(head)，预读新页从old-sblist head处插入，假如有一个页号为50的新页被预读加入缓冲池



缓冲池污染

MySQL缓冲池加入了一个“老生代停留时间窗口”的机制：只有满足“被访问”并且“在老生代停留时间”大于T，才会被放入新生代头部，短时间内被大量加载的页，并不会立刻插入新生代头部，而是优先淘汰那些，短期内仅仅访问了一次的页

参数配置



* innodb\_buffer\_pool\_size：缓冲池大小，内存允许，建议调大，缓存更多数据。
* innodb\_old\_blocks\_pct：老生代长度比例，默认37，即new:old是63:37。
* innodb\_old\_blocks\_time：老生代停留时间窗口，同时满足被访问与停留时间才插入到新生代。

2.Flush 链表

Flush 链表里面保存的都是脏页，也会存在于LRU 链表

Free 链表 存放的是空闲页面，初始化的时候申请一定数量的页面

LRU 链表和Flush链表的区别？

同一个链表的不同引用？

3.Free 链表

三种Page

Buffer Pool按照Page大小来分配，受innodb\_page\_size控制。

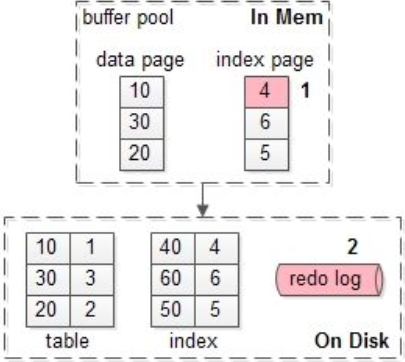
* Free Page：Page未被使用，位于 Free 链表
* Clean Page：已被使用，但是页面未发生修改，位于LRU 链表。
* Dirty Page：已被使用，页面已经被修改，其数据和磁盘上的数据已经不一致。当脏页上的数据写入磁盘后，内存数据和磁盘数据一致，那么该Page 就变成了干净页。脏页同时存在于LRU 链表和Flush 链表。

[Innodb Buffer Pool的三种Page和链表](https://mp.weixin.qq.com/s?__biz=MzAxODcyNjEzNQ==&mid=2247489264&idx=2&sn=167cf8585d8003234d6ad5571ba40974&chksm=9bd0bb68aca7327e94e75dd47270a9d9063e59f00128f916692244506eeac18aa6a0bb3aee52&mpshare=1&scene=1&srcid=&sharer_sharetime=1587720163766&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=2a099776fa69e9e756deb7222e697018a8442c88aaf5c0439eb62691d88bf2c2da61ced558bbf0408a86c85fbe21dbea11a6c833325c31655f9a3df9656deb071c50ca0f6fb80fef8297f6135f17f334&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AWVjcl90rc59jhFwE6mjf80%3D&pass_ticket=zLWDTjdsv9KsBGl19PRdRNKjngb7fO6WPxhRxwczjdSVP0Ore6DN%2BLkZhZm5R%2F2v)

### 3.1.3 Change Buffer

毫无疑问，对于读请求，缓冲池能够减少磁盘IO，提升性能。问题来了，那写请求呢？

情况一



如上图序号1-2：

（1）直接修改缓冲池中的页，一次内存操作；

（2）写入redo log，一次磁盘顺序写操作；

这样的效率是最高的。

画外音：像写日志这种顺序写，每秒几万次没问题。

是否会出现一致性问题呢？

并不会。

（1）读取，会命中缓冲池的页；

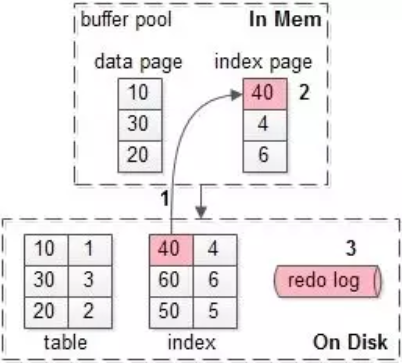
（2）缓冲池LRU数据淘汰，会将“脏页”刷回磁盘；

（3）数据库异常奔溃，能够从redo log中恢复数据；

什么时候缓冲池中的页，会刷到磁盘上呢？

定期刷磁盘，而不是每次刷磁盘，能够降低磁盘IO，提升MySQL的性能。

情况二



此时麻烦一点，如上图需要1-3：

（1）先把需要为40的索引页，从磁盘加载到缓冲池，一次磁盘随机读操作；

（2）修改缓冲池中的页，一次内存操作；

（3）写入redo log，一次磁盘顺序写操作；

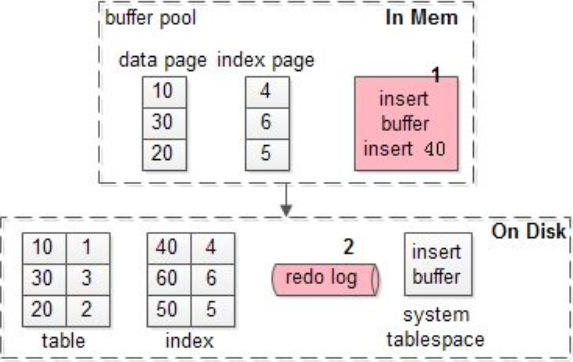
没有命中缓冲池的时候，至少产生一次磁盘IO，对于写多读少的业务场景，是否还有优化的空间呢？

写缓存

MySQL5.5前叫插入缓冲(insert buffer)，只针对insert优化；现在对delete和update也有效，叫做写缓冲(change buffer)。

它是一种应用在非唯一普通索引页，不在缓冲池中，对页进行了写操作，并不会立刻将磁盘页加载到缓冲池，而仅仅记录缓冲变更(buffer changes)，等未来数据被读取时，再将数据合并(merge)恢复到缓冲池中的技术。写缓冲的目的是降低写操作的磁盘IO，提升数据库性能。

假如要修改页号为40的索引页，而这个页正好不在缓冲池内。



加入写缓冲优化后，流程优化为：

（1）在写缓冲中记录这个操作，一次内存操作；

（2）写入redo log，一次磁盘顺序写操作；

其性能与这个索引页在缓冲池中相近。

是否会出现一致性问题呢？

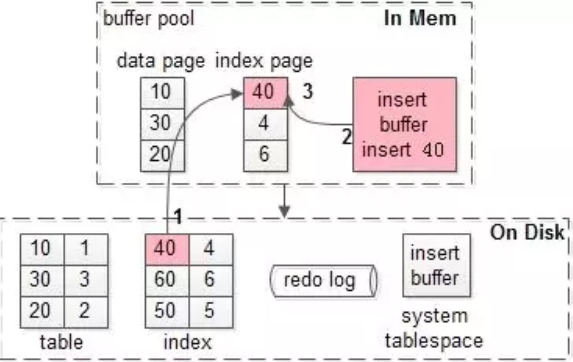
也不会。

（1）数据库异常奔溃，能够从redo log中恢复数据；

（2）写缓冲不只是一个内存结构，它也会被定期刷盘到写缓冲系统表空间；

（3）数据读取时，有另外的流程，将数据合并到缓冲池；

假设稍后的一个时间，有请求查询索引页40的数据。



此时的流程如序号1-3：

（1）载入索引页，缓冲池未命中，这次磁盘IO不可避免；

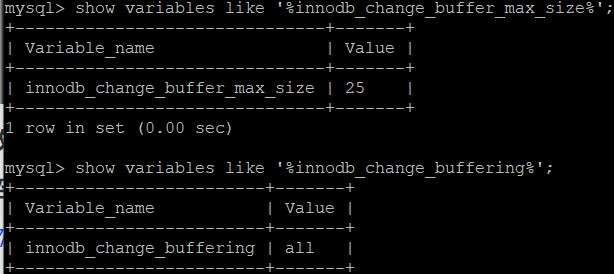
（2）从写缓冲读取相关信息；

（3）恢复索引页，放到缓冲池LRU里；

可见，40这一页，在真正被读取时，才会被加载到缓冲池中。

上述原理，对应InnoDB里哪些参数？

有两个比较重要的参数



innodb\_change\_buffer\_max\_size

配置写缓冲的大小，占整个缓冲池的比例，默认值是25%，最大值是50%。

写多读少的业务，才需要调大这个值，读多写少的业务，25%其实也多了。

innodb\_change\_buffering

配置哪些写操作启用写缓冲，可以设置成all/none/inserts/deletes等。

## 2.2 并发控制

事务

事务是一组操作完成一个任务。特性：

* Atomic（原子性）：要么全部成功，要么全部失败。执行的完整性。
* Consistency（一致性）：数据不因事务的执行而遭受破坏.只有合法的数据可被写入DB，
* Isolation（隔离性）：允许多个用户对同一个数据进行并发访问，而不破坏数据的正确性和完整性。
* Durability（持久性）

事务的并发控制实现手段。

* Lock-Based Concurrency Control。基于锁。
* MVCC：Multi-Versioning Concurrency Control.不加锁实现读写并发。

### 3.2.1 Locking

普通锁：

1)操作数据前，锁住，实施互斥，不允许其他的并发任务操作；

2)操作完成后，释放锁，让其他任务执行；

如此这般，来保证一致性。存在什么问题？

简单的锁住太过粗暴，连“读任务”也无法并行，任务执行过程本质上是串行的。

于是出现了共享锁与排他锁：

Share and eXclusive Locks

* S锁：共享锁之间不互斥，读读可并行。数据加了S锁，不允许其他事务更新或删除这些行，防止数据被修改。
* X锁：排他锁又称写锁，写读，写写不可以并行。T1对数据A加排他锁后，T2不能再对A加任任何类型的锁。

可以看到，一旦写数据的任务没有完成，数据是不能被其他任务读取的，这对并发度有较大的影响。对应到数据库，可以理解为，写事务没有提交，读相关数据的select也会被阻塞。

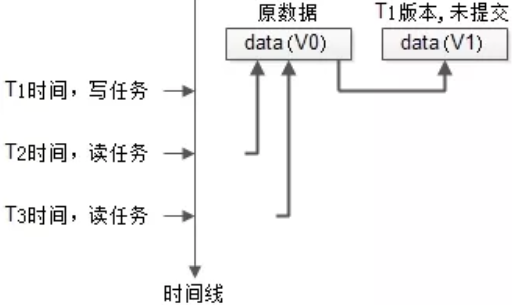
有没有可能，进一步提高并发呢？

即使写任务没有完成，其他读任务也可能并发，这就引出了MVCC。

### 3.2.2 MVCC

多版本并发控制。数据临时保存多个版本。是一种能够进一步提高并发的方法，它的核心原理是：

* 写任务发生时，将数据克隆一份，以版本号区分；
* 写任务操作新克隆的数据，直至提交；
* 并发读任务可以继续读取旧版本的数据，不至于阻塞；



1. 最开始数据的版本是V0；
2. T1时刻发起了写任务，把数据clone了一份，进行修改，版本变为V1，但任务还未完成；
3. T2时刻并发了一个读任务，依然可以读V0版本的数据；
4. T3时刻又并发了一个读任务，依然不会阻塞；

可以看到，数据多版本，通过“读取旧版本数据”，能够极大提高任务的并发度。

提高并发的演进思路，就在如此：

* 普通锁，本质是串行执行
* 读写锁，可以实现读读并发
* 数据多版本，可以实现读写并发

多版本导致read的数据可能过期，会出现丢失更新。

Mysql默认隔离级别Repeate read有lost update 问题。引起其他潜在问题,比如营业额增加，银行转账。

|  |  |  |
| --- | --- | --- |
| **时间** | **trx1** | **trx2** |
| T1 | m:=100 |  |
| T2 |  | m :=100 |
| T3 |  | update m=m+100 |
| T4 |  | commit |
| T5 | update m=m+100 |  |
| T6 | commit |  |

解决思路：

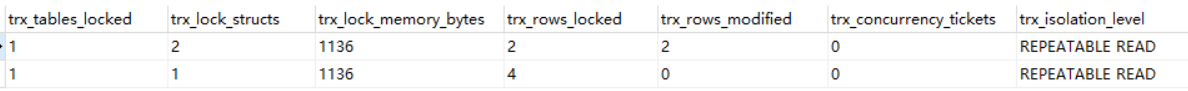
* 加悲观锁: 。trx1和trx2在select都for update
* 设置为Serializable
* 乐观锁：应用层实现，比如Hibernate。

查看事务innodb\_trx

1. **select** \* **FROM** information\_schema.innodb\_trx\G;



* trx\_state：running：事务start，commit前为running。lock wait：更新一个加锁的数据时，事务状态。
* trx\_stared 事务开始时间
* trx\_wait\_stated:block事务wait stated
* trx\_query:
* trx\_id
* trx\_mysql\_thread\_id



* trx\_rows\_locked 锁定row行数
* trx\_tables\_locked 锁定table数量

Connections

show full processlist;



每一条代表一个Thread，一个Tcp连接。

查看lock数据

1. **select** \* **from** information\_schema.innodb\_locks;



* lock\_trx\_id：事务id
* lock\_table+lock\_data：找到具体锁定的数据。lock\_data为数据id，
* Lock\_mode: X/S
* lock\_type： RECORD行锁

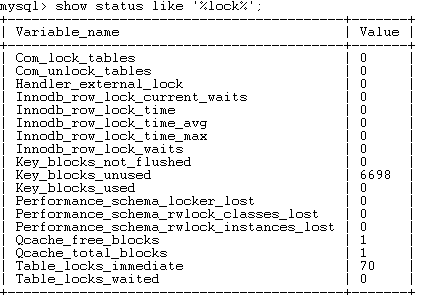
统计lock信息

Syntax: show status like '%Innodb\_row\_lock%';



Innodb\_row\_lock\_current\_waits 当前阻塞线程数

Innodb\_row\_lock\_waits 总共wait次数



## 2.3 InnoDB加锁分析

普通select

不显式加S/X锁的select操作都属于快照读，使用MVCC，不同隔离级别

* RU：不加锁，直接读取最新记录，可能发生脏读、不可重复读和幻读问题。
* RC：不加锁，每次select生成ReadView，解决脏读。
* RR：不加锁，第一次select生成ReadView，解决脏读、不可重复读和幻读问题。
* SERIALIZABLE：自动提交同RR。手动提交，转为select..lock in share mode。

加锁语句

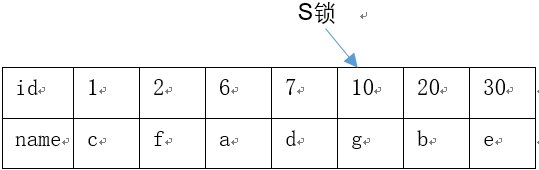
* select...lock in share mode;
* select...lock in share mode for update
* update/delete/delete首先定位到被改动的记录并自动加X锁

1. 主键+RC/RR

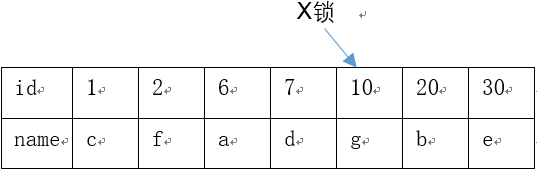
这个组合是分析最简单的，到执行该语句时，只需要将主键的记录加锁。

Table:T1(id primary key,name)

Statement1：select \* from T1 where id=10 lock in share mode;



Statement2：update T1 set name=’e1’ where id=10 for update;

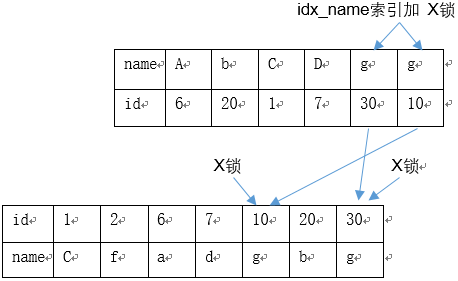


这条update语句没有更新二级索引列。delete主键加锁，同update。

1. 二级索引+RC

Table：T1(id primary key,name key idx\_name);

Statement：update T1 set name=’e1’ where id=10



二级索引上，满足id = 10的记录均加上X锁。同时对应主键索引上的记录也加上X锁。RC下唯一索引同非唯一。

唯一索引+RR相同

无索引+RC

Table:T1(id primary key,name)

update T1 set sex=1 where name=’g’

无索引，name列上没有索引，只能走聚簇索引，进行全表扫描。由于是在MySQl Server层面进行的。因此每条记录无论是否满足条件，都会加上X锁。但在实际操作中，MySQL进行了改进，在进行过滤条件时，发现不满足条件后，会调用 unlock\_row 方法，把不满足条件的记录放锁（违背了2PL原则）。保证最后满足条件的记录加上锁，但是每条记录的加锁操作是不能省略的。

非唯一索引+RR

Table:T1(id primary key,name key),statement：update T1 set sex=1 where name=’g’

idx\_name索引加 X锁

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| name | A | b | C | D | g | g |
| id | 6 | 20 | 1 | 7 | 30 | 10 |

X锁 X锁

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| id | 1 | 2 | 6 | 7 | 10 | 20 | 30 |
| name | C | f | a | d | g | b | g |

死锁(RC)

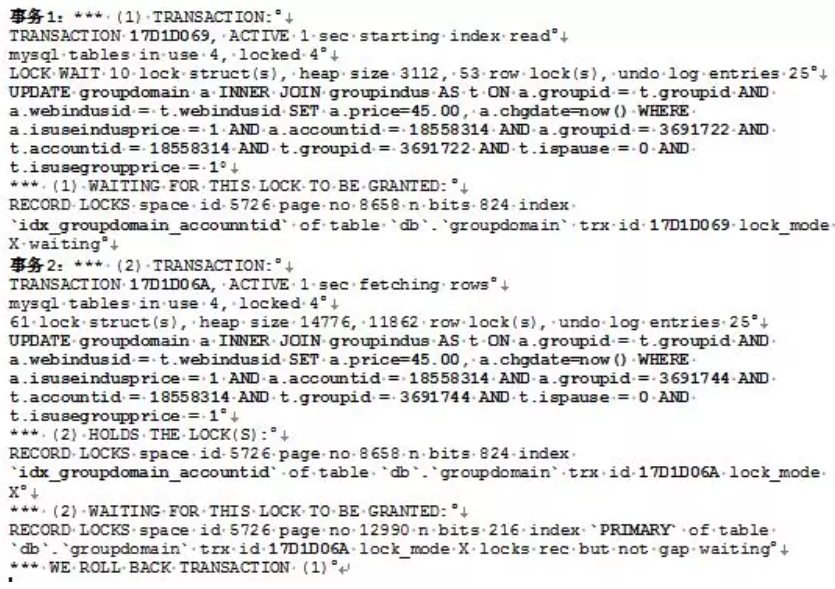
Table:T1(id primary key,name key)

T1：select \* from T1 where name=’e’ lock in share mode;

T2: update T1 set name=’e1’ where id=30;

T2持有id的X锁等待获取二级索引name上的锁，T1持有了name的S锁等待id上的锁。

两个事务都分别持有一个锁，且都在等待对方已经持有的那个锁，这种情况就是所谓的死锁，两个事务都无法运行下去，必须选择一个进行回滚，对性能影响比较大。



## 2.4 B+Tree索引

## 2.5 Join算法

Join算法

驱动表：也叫外表。inner join时mysql会自动找出数据少的表作用驱动表。（A left join B时A表为驱动表）

R join S,R是驱动表

### 3.1.1 Simple Nested-Loop Join

1. **For** each row r in R do
2. Foreach row s in S do
3. If r and s satisfy the join condition
4. **Then** **output** the tuple <r,s>

性能开销：

|  |  |
| --- | --- |
| **开销统计** | **SNLJ** |
| 外表扫描次数 | 1 |
| 内表扫描次数 | R |
| 读取记录数 | R + R\*S |
| 比较次数 | R\*S |
| 回表读取记录次数 | 0 |

读取记录数和比较次数的成本都是S\*R，即笛卡儿积。假设外表内表都是1万条记录，那么读取的记录数和比较次数都上亿，开销较大。

### 3.1.2 Index Nested-Loop Join

在Join的优化时，通常会在内表建立索引，以此降低Nested-Loop Join算法的开销。由于内表上有索引，比较不再需要一条条记录进行比较，可通过索引来减少比较，加速查询。

1. **For** each row r in R do
2. lookup r in S **index**
3. if found s == r **then**
4. **output** the tuple <r,s>

性能开销：

|  |  |  |
| --- | --- | --- |
| **开销统计** | **SNLJ** | **INLJ** |
| 外表扫描次数 | 1 | 1 |
| 内表扫描次数 | R | 0 |
| 读取记录数 | R + R\*S | R + Smatch |
| 比较次数 | R\*S | R \* Indexheight |
| 回表读取记录次数 | 0 | Smatch(if possible) |

* Smatch：通过索引匹配的记录数。
* Indexheight：索引树的高度,每次index lookup的成本就是

通过索引可以大幅降低内表的Join的比较次数，每次比较1条外表的记录，其实就是一次indexlookup（索引查找），而每次index lookup的成本就是树的高度，即Indexheight。

缺陷

主键索引：即使是大数据量的情况下，INLJ的效率亦是相当不错的。因为索引查找的开销非常小，并且访问模式也是顺序的（假设大多数聚集索引的访问都是比较顺序的）。

辅助索引：大部分人诟病MySQL的INLJ慢，主要是Join的时候可能用到的是辅助索引，这时INLJ的过程又多一步Fetch的过程，且这个过程开销会相当的大

* 辅助索引index lookup是随机I/O访问操作。
* 若需回表，根据index lookup再进行回表又是一个随机的I/O操作。

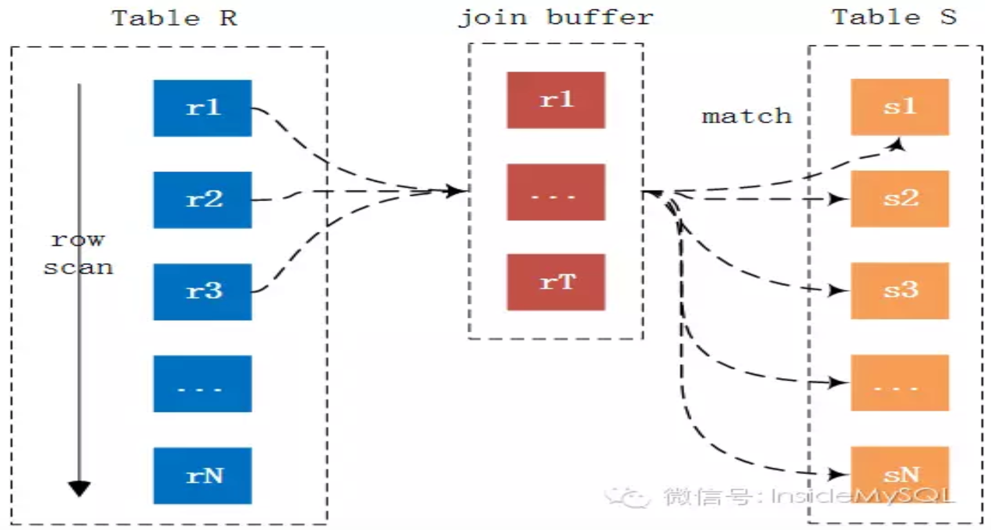
INLJ最大的弊端是其可能需要大量的离散操作。即使SSD的出现大幅提升了随机的访问性能，但对比顺序I/O，依然不在一个数量级上。

### 3.1.3 Block Nested-Loop Join

有些情况，Join列就是没有索引，这时MySQL的选择绝对不会是SNLJ。BNLJ算法较SNLJ改进在于可以减少内表的扫描次数，甚至可以和Hash Join算法一样，仅需扫描内表一次。

1. **For** each tuple r in R do
2. store used columns **as** p **from** R in join buffer
3. **For** each tuple s in S do  //达到buffer size上限，开始扫描S
4. if p and s satisfy the join condition **then**
5. **output** the tuple <p,s>

Join Buffer缓存链接需要的列，然后以Join Buffer批量的形式和内表中的数据进行链接比较。若join buffer可以缓存所有的外表列，那么仅需扫描内外表各一次，从而大幅提升Join的性能。



join\_buffer\_size

控制Join Buffer的大小，当MySQL的Join使用到BNLJ，调大join\_buffer\_size才是有意义的，可避免多次的内表扫描，

Join Buffer缓存的对象

所有参与查询的列”都会保存到Join Buffer，而不是只有Join的列，比如

1. **SELECT** a.col3 **FROM** a,b
2. **WHERE** a.col1 = b.col2 AND a.col2 > …. AND b.col2 = …

假设外表是a，内表是b，存在Join Buffer中的列是所有参与查询的列，在这里就是（a.col1，a.col2，a.col3），所以内表的扫描次数为：

Scaninner\_table = (Rn \* used\_column\_size) / join\_buffer\_size + 1

如果Join Buffer可以缓存外表的数据，那么内表的扫描仅需一次，这和Hash Join非常类似。

总结

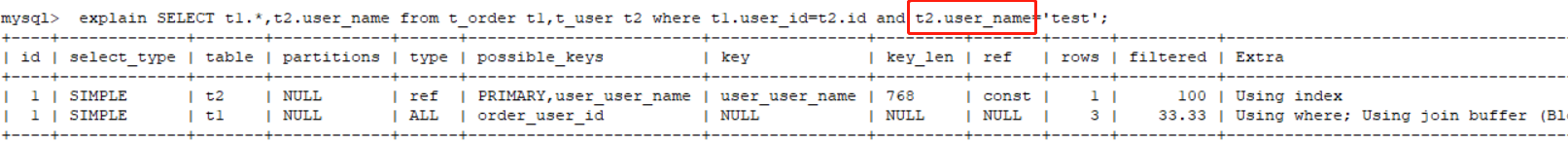
BNLJ极大的避免了内表的扫描次数但没解决Join比较的次数，其仍然通过Join判断式进行比较。综上所述，到目前为止各Join算法的成本比较如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 开销统计 | SNLJ | INLJ | BNLJ |
| 外表扫描次数o | 1 | 1 | 1 |
| 内表扫描次数i | R | 0 | R\*col\_size/join\_buff\_size + 1 |
| 读取记录数R | R + R\*S | R+ Smatch | R+S\*I |
| Join比较次数M | R\*S | R \* Indexheight | S\*R |
| 回表读取记录次数F | 0 | Smatch(if possible) | 0 |

Using join buffer

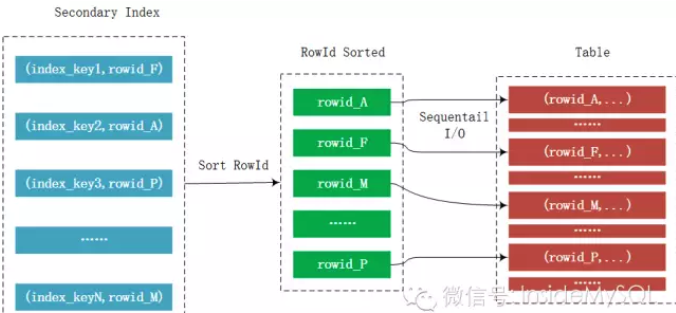
Using join buffer (Block Nested Loop), Using join buffer (Batched Key Access)

Tables from earlier joins are read in portions into the join buffer, and then their rows are used from the buffer to perform the join with the current table. (Block Nested Loop) indicates use of the Block Nested-Loop algorithm and (Batched Key Access) indicates use of the Batched Key Access algorithm. That is, the keys from the table on the preceding line of the EXPLAIN output will be buffered, and the matching rows will be fetched in batches from the table represented by the line in which Using join buffer appears.



### 3.1.4 Batched Key Access Join

INLJ通过辅助索引join后需要回表，产生大量的随机I/O。BKAJ辅助索引读到数据不回表，而是缓存其rowid并排序，再去访问记录。将随机I/O转顺序I/O。如下所示：



MySQL 5.6推出BKAJ，需手动开启。

## 2.6 Order by

### 2.6.1 索引排序

InnoDB默认采用的是B tree索引，B tree索引本身就是有序的，如果有一个查询如下：

select \* from film where actor\_name='苍老师' order by prod\_time;

那么只需要加一个(actor\_name,prod\_time)的索引就能够利用B tree的特性来避免额外排序。

如下图所示：



通过B-tree查找到actor\_name=’苍老师’演员为苍老师的数据以后，只需要按序往右查找就可以了，不需要额外排序操作。

对应的哪些可以利用索引优化排序的列举如下：

复合索引<key\_part1, key\_part2>

* 只Select 索引字段:

1. **SELECT** pk, key\_part1, key\_part2 **FROM** t1
2. **ORDER** **BY** key\_part1[**DESC**], key\_part2[**DESC**];

若Select不在索引的column，扫描索引并回表操作代价可能比表扫描更高，所以Using filesort

* Case 2 key\_part1 is constant

1. **SELECT** \* **FROM** t1
2. **WHERE** key\_part1 = constant
3. **ORDER** **BY** key\_part2 [**DESC**];

若key\_part1区分度比全表扫描cheaper.使用key\_part2排序。

同样key\_part1 >或< constant是否使用索引排序取决于key\_part1的区分度

注意where过滤要有最左索引

不能使用索引排序

* 过滤和排序的索引不同： **SELECT** \* **FROM** t1 **WHERE** key2=constant **ORDER** **BY** key1;
* Order by使用不同的索引： **SELECT** \* **FROM** t1 **ORDER** **BY** key1, key2;
* 索引不连续：**select** \* **from** t1 **WHERE** key2=constant **ORDER** **BY** key1\_part1, key1\_part3;
* 索引未保存row顺序，比如HASH index

group by与order by很类似，其实质是先排序后分组，遵照索引创建顺序的最佳左前缀法则

### 2.6.2 排序算法

不能用索引排序，DB不得不自己实现排序功能以满足用户需求。假设表结构和SQL语句如下：

1. **CREATE** **TABLE** t1(
2. id **int**, col1 **varchar**(64),
3. col2 **varchar**(64),
4. col3 **varchar**(64),
5. **PRIMARY** **KEY**(id),**key**(col1,col2)
6. );
7. **SELECT** col1,col2,col3 **FROM** t1 **WHERE** col1>100 **ORDER** **BY** col2;

* 常规排序

1)从表t1中获取满足where条件的记录

2)对于每条记录，将记录的主键+排序键(id,col2)取出放入sort buffer

3)如果sort\_buffer可存放所有满足条件的(id,col2)对，则进行排序(快速排序)；否则sort\_buffer满后进行排序并固化到临时文件(temporary disk files)。

4)若排序中产生了临时文件，需要利用归并排序算法，保证临时文件中记录是有序的

5)循环执行上述过程，直到所有满足条件的记录全部参与排序

6)扫描排好序的(id,col2)，回表查col3。

7)将获取的结果集返回给用户。

可见，Using filesort并不意味着就是文件排序，也有可能是内存排序，是否用文件排序要看sort\_buffer是否能容下需要排序的(id,col2)对。

此外一次排序需要两次IO，一次是捞(id,col2),第二次是捞(col1,col2,col3)。由于返回的结果集是按col2排序，因此id是乱序的，回表产生大量的随机IO。第二次IO优化：在捞之前首先将id排序，并放入缓冲区，缓存区大小由参数read\_rnd\_buffer\_size控制，然后有序去捞记录，将随机IO转为顺序IO。

优化

* Increase the sort\_buffer\_size variable value
* Increase the read\_rnd\_buffer\_size so that more rows are read at a time
* 优化排序

放入sort buffer (col1,col2,col3)。由于sort buffer中包含了查询需要的所有字段，因此排序完成后可以直接返回，无需回表。减少了第二次IO。

配置参数max\_length\_for\_sort\_data

只有当排序元组小于参数时，才能利用优化排序方式，否则只能用常规排序方式。

* 优先队列排序

针对Order by limit M，N，采用堆排序实现。

虽然仍然需要所有元素参与排序，但是只需要M+N个元组的sort buffer空间即可，对于M，N很小的场景，基本不会因为sort buffer不够而导致需要临时文件进行归并排序的问题。对于升序，采用大顶堆，最终堆中的元素组成了最小的N个元素，对于降序，采用小顶堆，最终堆中的元素组成了最大的N的元素。

Web应用常见的，从一张大表分页展示几条数据:

**select** col1, ..**from** t1 **order** **by** col\_name limit row\_count;

Order by+limit可有效的在内存中排序，找到排序后limit数量的rows，MySQL停止排序,不会排序整个entire result.

上面可以看到MySQL已在尽量优化排序了, 也从侧面说明其不希望排序的出现, 如最开始的SQL, 建立一个(col1, col2)的联合索引, 就可以避免排序了,

#### Using temporay

To resolve the query, MySQL needs to create a temporary table to hold the result(用临时表来存储结果集). This typically happens if the query contains GROUP BY and ORDER BY clauses that list columns differently.

常见于排序和分组查询

MySQL在以下几种情况会创建临时表：

* FROM中的子查询；
* UNION查询
* 用到TEMPTABLE算法或者是UNION查询中的视图；
* ORDER BY和GROUP BY的子句不一样时；
* 表连接中，ORDER BY的列不是驱动表中的；
* DISTINCT查询并且加上ORDER BY时；
* SQL中用到SQL\_SMALL\_RESULT选项时；
* 子查询或者semi-join时创建的表；

如果临时表中需要存储的数据量超过了上限（ tmp-table-size 或max-heap-table-size 中取其大者），这时候就需要生成基于磁盘的临时表了。

# Part 3 SQL优化总结

从开发的角度，更多关注SQL

## 3.1 库表结构优化

MySQL基于Row存储，数据文件和索引文件，以文件形式存在磁盘，而IO时以page（block）读到内存。磁盘IO相对于内存存取，I/O存取的消耗要高几个数量级。减少IO次数手段：

* 读数据：每条记录占用的空间量减小，每页中可存放的数据行数增大。进而减少IO次数。
* 索引查找：查找过程中产生磁盘I/O消耗，IO的存取次数是判断索引性能的关键性指标。

### schema优化

选择数据类型原则

* 更小的长度：占用更少的磁盘/内存/CPU缓存，CUP处理周期。每页存更多数据。
* 简单就好：数值比字符操作代价低，简单类型CPU处理快。

### 数据类型

字符类型

|  |  |  |
| --- | --- | --- |
| **Mysql** | **Oracle** | **说明** |
| char(n) | char | 定长类型，不易产生碎片。浪费空间。 |
| varchar(0-65535) | varchar2(1-4000)  varchar | 比char省空间。处理效率略低于char。经常变更的字段用char避免裂变。 |
| Blob 64k |  | 可存二进制串，不用按字符集进行校验和排序 |
| text |  | 性能要低于char或者是varchar类型 |

char vs varchar

varchar 存长度+内容，处理时先读取长度再读取内容，效率略低于char。宽度越长消耗越多内存(在内存中对字符串使用固定的宽度)。

数值 VS 字符

字符类型可存字符串/数值/日期等，是万能数据类型，但要选择合适的数据类型。

用字符保存数值：

* 增加存储的空间（以utf8为例，一位数字要1byte）。整型可表达的范围广，占用空间少。
* 运算时mysql会将字符串转换为数值，导致索引失效。比如where clol1 > 1;
* 排序混乱。字符串的9是大于22的(长度不同，按最后一位比较)。

Case

* Id:bigint，自动增长
* status字段：tinyint（应用层枚举转换）。但char的可读性好。
* 手机号：用char（11）,+86.方便做字符串的匹配操作。
* 用整型存储IP地址

数值类型(整型+实数)

|  |  |  |
| --- | --- | --- |
| **Mysql** | **Oracle** | **说明** |
| tinyint [1byte] | number(3)代替 | 1 byte表达-128~128,number(3):-999~-999 |
| smallint [2byte] | number (5) |  |
| mediumint [3byte] | number (7) |  |
| int [4byte] | number(10) |  |
| bigint [8byte] | number(20) | 对应java的long。保存日期可处理简单的排序，计算等，最常用。[Reference](https://www.quora.com/What-is-the-best-way-to-store-a-timestamp-in-MySQL) |
| float [4byte] |  |  |
| double [8byte] |  |  |
| decimal |  | 高精确存储。对应java的BigDecimal |

Unsigned：无符号，不允许负值，可使正数的上限提高一倍

Decimal

高精度存储

column\_name DECIMAL(P,D);

* P是表示有效数字数的精度。 范围为1〜65。
* D是表示小数点后的位数。 范围是0~30。MySQL要求D小于或等于(<=)P。

[Decimal float double区别](https://www.cnblogs.com/gulibao/p/5416245.html)

日期

|  |  |  |
| --- | --- | --- |
| **Mysql** | **Oracle** | **说明** |
| date [3byte] 只存日期 |  |  |
| time [3byte] 只存时间 |  |  |
| datetime [8] date+time | date [7] | date+time。可视化，**方便在mysql中做计算,但与db耦合** |
| timesamp: [4byte] | timesamp | 1970~2038存UTC时间，存取时根据db服务器时区自动转换 |

字符 VS timesamp

用字符串，不好计算且占用更多空间，比如20200801123000需要14个字节。

其他

|  |  |  |
| --- | --- | --- |
| **Mysql** | **Oracle** | **说明** |
| 无 | long最大长度2G | 一种较老的数据类型，将来会逐渐被BLOB、CLOB、NCLOB等大的对象数据类型所取代。 |
| Blob | Blob 4G | 二进制数据 |
| Clob | Clob 4G |  |
| set/emum |  | 字符型，限制范围。慎用，开发麻烦，一般在应用层做枚举 |

Fulltext

全文索引以词为基础的，MySQL默认的分词是所有非字母和数字的特殊符号都是分词符。

1. **创建全文索引：**
2. **ALTER** **TABLE** `table\_name` **ADD** FULLTEXT **INDEX** `idx\_user\_name` (`user\_name`);
3. **使用全文索引：**
4. **select** id,fnum,fdst **from** table\_name
5. **where** match(user\_name) against('zhangsan' in boolean mode);

字符长度

长度指字符长度（version 4前指字节），即显示的宽度。



varchar(n)：中文和英文都占一个长度。

int(11)：能存储的最大值和最小值是固定的。长度指显示的宽度。

int，最大数=(2^31)-1=2147483647，范围是-2147483648~2147483647。unsigned int， 最大数=(2^32)-1=4294967295，范围0～4294967295

可见最大数的长度都是10 + 显示正负占一个宽度

Null值

not null：而不是 null 。null使索引更复杂。

* 运算符/函数失效：不能使用=,<,>，对null做算术运算的结果为null
* 占1byte存储空间，用于标识。
* null值到非null的更新无法做到原地更新，更容易发生索引分裂

### Charset&Collation

[Reference](https://www.cnblogs.com/chyingp/p/mysql-character-set-collation.html)

查看 show character set;

|  |  |  |  |
| --- | --- | --- | --- |
| Charset | Default collaction | Maxlen | Description |
| ascii | ascii\_general\_ci | 1 |  |
| gbk | gbk\_chinese\_ci | 2 |  |
| utf8 | utf8\_general\_ci | 3 |  |
| Utf16 | Utf16\_general\_ci | 2 |  |
| utf32 | utf32\_general\_ci | 4 |  |

collation

排序规则。以utf8为例， 查看 show collation where charset = 'utf8';

* utf8\_bin：常用。每个字符用二进制数据存储，区分大小写。
* utf8\_general\_ci：default。ci means case insensitive。不区分大小写，使用时要注意
* utf8\_general\_cs：case sensitive。Mysql用utf8\_bin。

可具体到column级别

`code` varchar(20) character set utf8 collation utf8\_bin

编码规则

* Unicode：通用字符集，两个字节表示一个字符。副作用是原本采用ASCII字符集只需一个字节的，变成2个字节，造成了空间浪费。
* UTF-8：将Unicode编码成 1～4 个字节，ASCII 字符集继续保持了 1 个字节空间，而中文编码成了三个字节

[MySQL 的字符集以及带来的一点存储影响](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247488996&idx=2&sn=b8fc2a67068ae4a08d8424fee2bccec5&chksm=96cd55a8a1badcbed2a7d5fc5d0844a1f53770d19009aadad84ead33282e07df5a4658ca8a58&mpshare=1&scene=1&srcid=&sharer_sharetime=1587104127945&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=a7ced9b5cf2d9fe86362e4d07f8204fd1f2e495d2a23fce370b03825181089ca3870d3c1e6f6ea6ed10b0e2061234da9abb36fc89e8bb67cc6d7e293ce733fb4c486f44f845347331a25a70b0f47d1bf&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AcJLVg9yN3D5fDuXqvEvi2Y%3D&pass_ticket=zLWDTjdsv9KsBGl19PRdRNKjngb7fO6WPxhRxwczjdSVP0Ore6DN%2BLkZhZm5R%2F2v)

## 3.2 索引优化

索引是什么？

满足特定查找算法的数据结构。优点：

1）减少了服务器扫描的数据量

2）帮助服务器避免排序和临时表

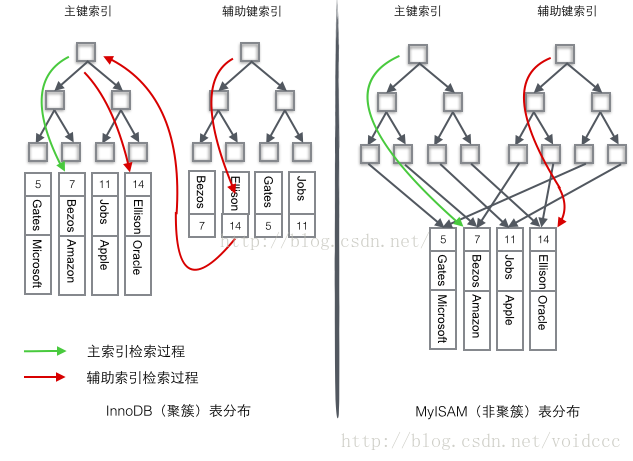
3）将随机I/O变成顺序I/O

### 3.2.1 索引类型

* Primary Index：数据非空值不重复（唯一），表只有一个。
* 普通索引
* Unique Index：保证数据唯一性
* Hash索引
* 全文索引(Full Text)

还有多列索引、短索引（前缀索引）、覆盖索引、聚簇索引和非聚簇索引，这些都不能将单独它归为一个索引类型。

* 覆盖索引：联合索引之上的叶子节点已覆盖要查询的列，不需表数据。组合索引（联合索引、复合索引）?
* 聚簇索引： clustered index。行数据就储存在主键索引树的叶子节点上。
* 非聚簇索引：又称辅助索引/Secondary index 。索引和表数据存储在独立的地方。



InnoDB：数据行和索引存在一起。页节点存主键值（方便通过通过主键索引查找）。

MyISAM 主辅助索引结构上没区别，只是主索引的叶节点是Primary

聚簇索引

InnoDB的页面分为

叶子节点：B树层次为0的页面，存储记录的所有内容

非叶子节点：B树层次大于0的页面，只存储索引键和页面指针。

优点：

* 聚集相关数据，比如userId做主键，相关数据只需读少量数据页，大大减少磁盘I/O。
* 使用覆盖索引扫描，可直接使用叶节点的主键值。

缺点

* 基于聚集索引插入，可能导致页裂变，
* 全表扫描慢，尤其是行比较稀疏，或也分裂导致存储不连续。
* 二级索引包含主键值，可能更大，索引时需要两次查找，而不是一次。

Multiple-Column Indexes

这里指在多个列上分别建索引。建索引的一个误区是在where后面条件每个列建一个索引，这样最好的情况也只能用到一个索引。生效原则：

Or：index merge。耗费性能做数据合并排序。特别是返回数据量较大时

And：

索引顺序:

#### 覆盖索引

* 不用回表 。避免连表时可能产生的大量随机IO操作。
* 索引区分度：sex区分度低， name应在前

多个字段分别建索引，只能用到一个索引

最左前缀匹配原则

覆盖索引<a,b,c,d>的最左前缀 <a>,<a,b>,<a,b,c>，

* 生效：a=’x’， a=’x’ and b=’y’,即包含最左前缀
* 不生效：b=’x’(不包含最左前缀),a like ‘a%’ and b=’b’(a范围查找)
* ICP：a=’x’ and c=’y’, a like ‘a%’ and b=’b’ 都是可以的

索引失效场景

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMTY0NDU3Nw==&mid=2651937946&idx=1&sn=e0797f6334ccd51860f3e6db70e2878f&chksm=8d0f33d4ba78bac2f26f688221235b18a636d43a736b31bf442b1ab53c1b789980d148338ad9&mpshare=1&scene=1&srcid=#rd])

* 负向条件：可优化为in查询。负向条件有：!=、<>、not in、not exists、not like等。
* 对字段计算
* or两边字段，若有一个不是索引字段，会造成查询不走索引。优化1：改成in，in在内存冲处理，数据量大用exisit

### 3.2.2 索引优化

索引增加写数据的额外消耗，频繁写的表不宜建索引。

三星系统 (Three-star index)

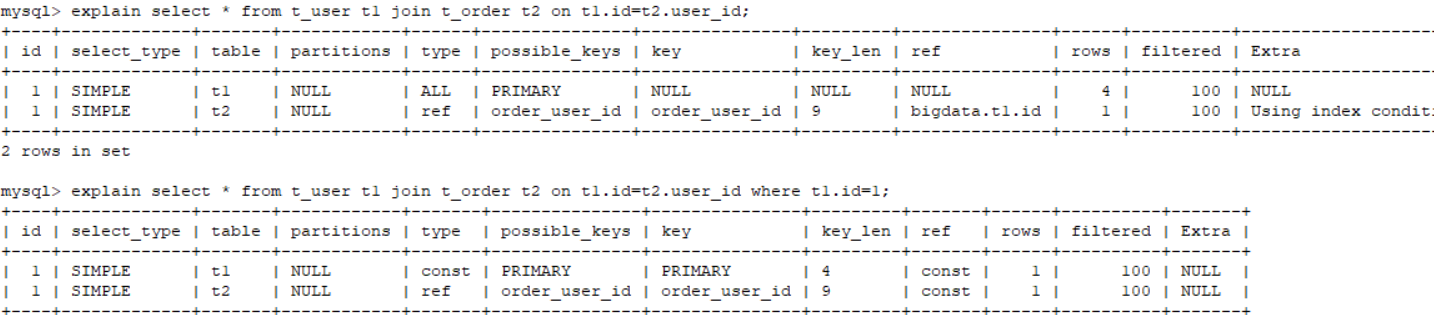
* 在where条件里的列建索引，避免直接扫描DB，避免全表扫描（🟊）
* order by中的排序和索引顺序是否一致,即对排序字段建索引。（🟊🟊）
* 索引行包含所有select的列，建覆盖索引。（🟊🟊🟊）

Q1: MySQL数据库查询记录时是否每次只能使用一个索引?

答：是，数据库查询只能用到一个索引，和全表扫描/只使用一个索引的速度比起来，去分析两个索引二叉树更加耗费时间，所以绝大多数情况下数据库都是是用一个索引。所以索引不是建的越多越好。

举几个开发中场景的案例

连表



语句1 如图和t\_user.id比较的列是t\_order.user\_id

语句2 mysql能对查询部分优化转成变量，比如id在where子句，将查询转成常量，然后就可高效将表从联表执行中移除。

Ref：与索引比较的列。The index actually chosen.

Type ：describes how tables are joined.

all < index < range < ref < eq\_ref < const < system < NULL

* System：The table has only one row (= system table). Const的特殊情况.
* Const：索引是primary key or unique.const tables are very fast because they are read only once.
* eq\_ref：多表join，和ref类似。the index is a primary key or unique NOT NULL index.
* Ref：where字段搜索使用了索引, the index is not a primary key or UNIQUE index
* Fulltext：The join is performed using a FULLTEXT index.
* Range：使用索引范围查找.用于between/like/or.
* index：full index scan，遍历全部的索引树。
* ALL：full table scan，（到存储引擎）遍历全表

1)隐性连接

where子句使用连接语句。

1. **select** \* **from** A,B **where** A.id=B.a\_id;
2. –show warnings
3. **select** \* **from** A join B **where** A.id=B.a\_id;

可见和join没有本质区别，规范写法应写有join的显性连接，有更好的可读性。

2)Join...on <条件> VS where<条件>

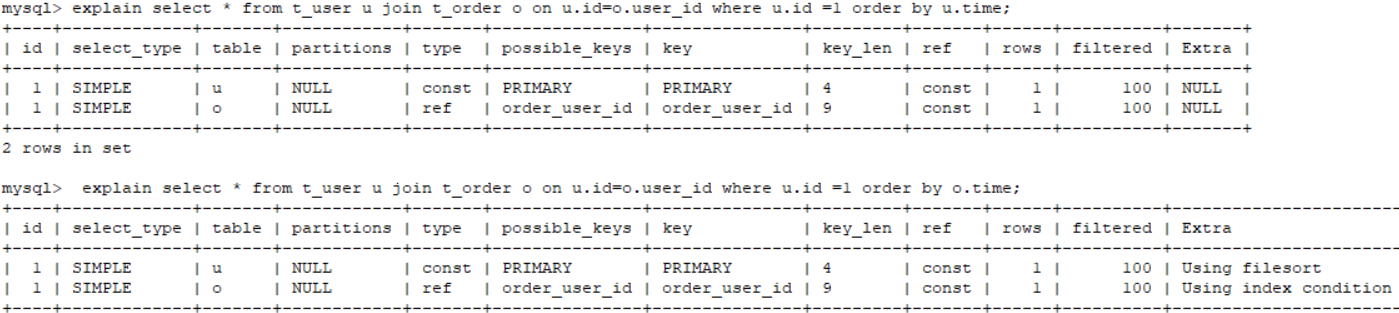
On作用于

A left join B where a.attr=<> 条件对驱动表A不起作用

* Case3 连表

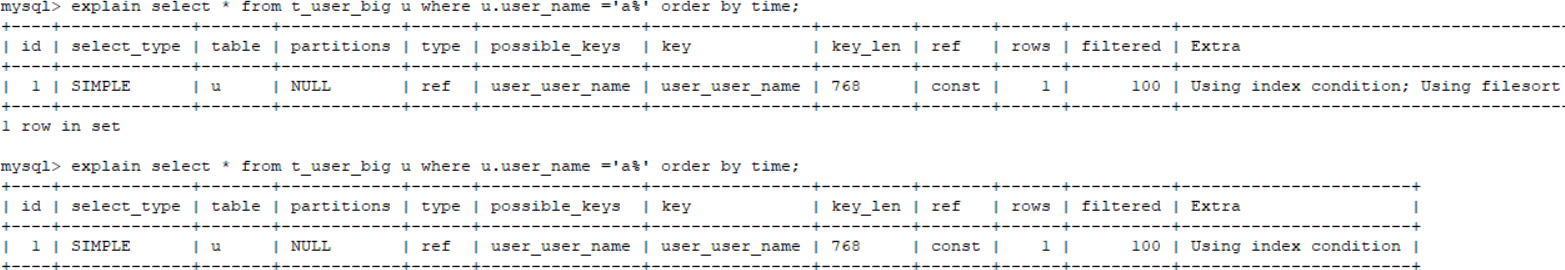
first table的join type是const.

在first table排序？



语句二 连表时order by的字段不全from the first nonconstant table that is used to retrieve rows. (This is the first table in the [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output that does not have a [const](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_const) join type.)

排序用到下推?



#### Case 2 Join优化

* 相关的join字段建索引。

分解关联查询

什么时候需要分解？

复杂查询 or 简单查询？

大的语句会锁住很多数据，占满整个事务日志，耗尽系统资源，阻塞很多小而重要查询。

传统实现总是强调DB完成尽可能的工作，原因在于以前网络通信，而且每个SQL的查询解析和优化代价高。

用单表还是联表，看具体业务场景。在高并发高读环境，遇到性能瓶颈时。好处

* 减少锁竞争 （TODO1? 查询有锁竞争？）
* 方便优化，单表建覆盖索引。
* 缓存更高效，dao层的复用率更好，更好维护，后期好扩展。数据库好拆分。

分解方法

将联表分解成单表冗余字段的形式

建汇总表

[阿里不让多表join？我偏要！](https://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247489683&idx=1&sn=dd98803e12794f0467494715d0d1c660&chksm=e9c5e122deb26834c3a85f6c6b2f17c50c253db2010040918e3e699bc90c3c77922d1b8309ec&mpshare=1&scene=1&srcid=&sharer_sharetime=1567572268193&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=9833ef604dc704e8f0d8174762fb0b878f749a4d78b6c0f85645573fcff05fc3d8092621a751e2441274af35fb6d1ff5ecc45e231fab3dab3352e99d8d891de68b9f371922e7d1e6ab7312b89c11a5e0&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070152&lang=zh_CN&pass_ticket=MCT3fCw7MCE676VxtSEKLj06vJaYj34U2Mb9jdEdZh8vRmoBGD%2FJdxemhK0hUxv2)

优化or

or两边字段，若有一个不是索引字段，会造成查询不走索引。

优化1：改成in，in在内存冲处理，数据量大用exisit。

优化2：用 union all或者union代替“or”。

union VS union all

结果集没有重复数据用union all。union合并结果集需要过滤重复数据，会涉及排序，增加大量的CPU运算，加大资源消耗及延迟，数据量大时延迟更明显。

on vs where

on对连接过滤，where对join后结果集过滤。

* inner join：同where结果集一样。（on中条件会优化到where中）。
* left join： A驱动B表，结果为A的全部记录。on的条件对连接起作用，控制右表输出。

1. **select** \* **from** A left join B **on** A.id = B.a\_id and A.status = 'N' ;

条件A.status并不起作用，left join的B记录为null。

优化数据访问

1.select必要的列和行。

2.explain确保MySQL服务层分析必要的行。

解决方法

* 建索引：type判断是否全表扫描，rows越小越好，理想情况扫描行数=返回行数。
* 汇总表：group by或count一些会引起全表扫描，必要时使用单独的汇总表或者放缓存。
* 适当冗余：通过空间换时间，不用连表，
* 适当拆分：text或varchar大字段：若访问不到，拆分到独立地表。减少数据占用的存储空间，每个数据块中可以存储更多的数据，减少物理IO次数。
* 缓存：缓存重复查询和获取比较慢的数据。

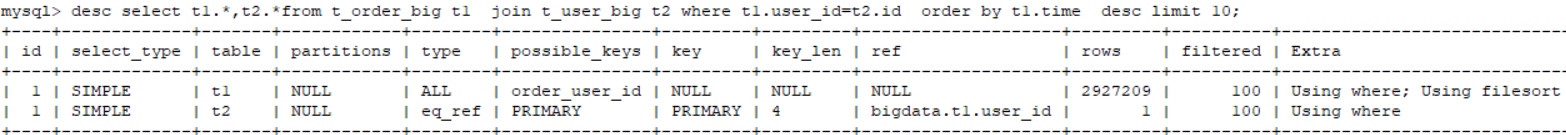
## 3.3 SQL优化实践

SQL的生命周期每个环节都有足够的优化空间，优化的本质是减少SQL对资源的消耗和依赖

### Case0 一次优化实践

1. **select** t1.\*,t2.\*
2. **from** t\_order\_big t1  join t\_user\_big t2
3. **where** t1.user\_id=t2.id
4. and substr(dept\_id,0,8)=:deptId
5. **order** **by** t1.**time**  **desc** limit 10;

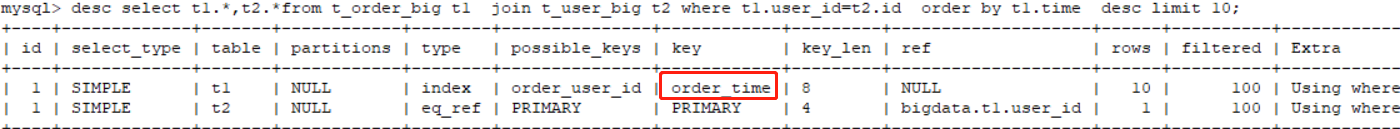
2. 漏加索引



T2根据user\_id查找t2的主键索引，加了limit，查找性能消耗不大。

t1全表扫描，time漏加索引，filesort内存中排序,找到10个最大时间值

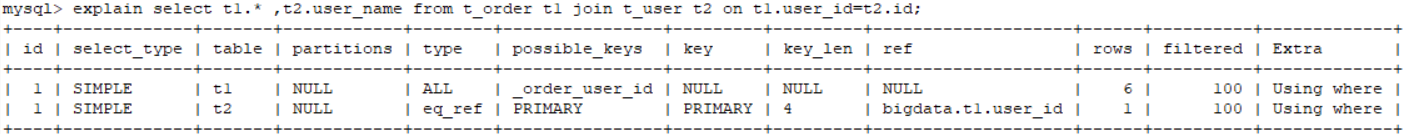
3.time加索引，索引排序



t1 type=all变成type=index，并按time的索引排序，t1有10次回表，但比filesort性能好的多。

Join Case分析

2.select t1.\*,t2.user\_name



此时t2需要回表查user\_name,Using where;Using index过滤索引🡪Using where过滤表

### Where

示例：表t\_user

create table t\_user (

id bigint primary key,

name varchar(10) ,

nick\_name varchar(10) ,

time bigint,

key idx\_user\_name(name)

key idx\_user\_time(time)

)

1) Using index

1. mysql> explain **select** **name** **from** t\_user **where** **name**='x' \G
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. id: 1
4. select\_type: SIMPLE
5. **table**: t\_user
6. partitions: NULL
7. type: ref
8. possible\_keys: idx\_user\_name
9. **key**: idx\_user\_name
10. key\_len: 768
11. ref: const
12. **rows**: 1
13. filtered: 100.00
14. Extra: Using **index**

Using index直接定位到数据无需过滤

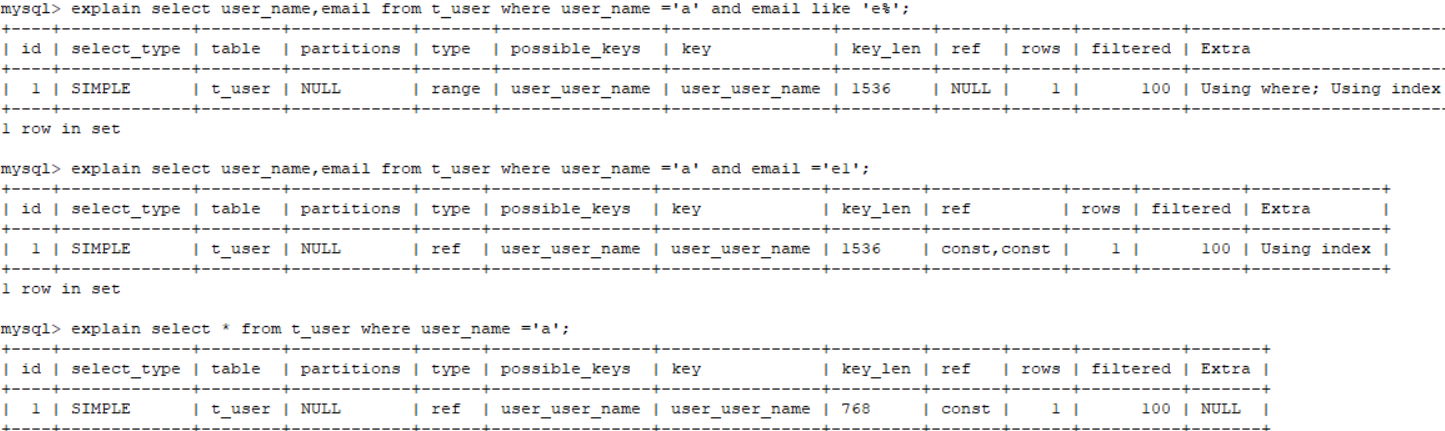
2) Using where;Using index

Using where+Using index，即查到索引数据在server层过滤

1. mysql> explain **select** **name** **from** t\_user **where** **name** like 'a%'  \G
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. id: 1
4. select\_type: SIMPLE
5. **table**: t\_user
6. partitions: NULL
7. type: range
8. possible\_keys: idx\_user\_name
9. **key**: idx\_user\_name
10. key\_len: 768
11. ref: NULL
12. **rows**: 1
13. filtered: 100.00
14. Extra: Using **where**; Using **index**

尽量不要select \*,若索引能覆盖索引数据只select必要的字段。

索引完全覆盖，无需回表。



语句1 是Using where;Using index

语句2 直接定位到数据无需过滤

语句3 索引不覆盖回表了所有没有Using index

3) Using Where

警惕全表扫描，可能未建索引或索引失效。

读出所有数据(index或data)在mysql Server层过滤，

1. mysql> explain **select** \* **from** t\_user **where** nick\_name='x' \G
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. id: 1
4. select\_type: SIMPLE
5. **table**: t\_user
6. partitions: NULL
7. type: ALL
8. possible\_keys: NULL
9. **key**: NULL
10. key\_len: NULL
11. ref: NULL
12. **rows**: 4
13. filtered: 25.00
14. Extra: Using **where**
15. 1 row in **set**, 1 warning (0.00 sec)

Type ALL/index

Using index condition

1.过滤多个字段

1. 覆盖索引<a,b,c,d>
2. **where** a=’xx’ and c like ’%d%’

a,c不符合最左前缀,withICP可以下推

示例：常见页面<idCard,code,name>,据where idcard=‘320xx’ and name like ‘%xx%’。

继续补充like ‘%xx%’的优化还可以用fullText索引。db全文搜索能力相对较弱，有的需求无法实现，比如权重排序，且在db资源宝贵。可在es等搜索引擎实现。

索引条件下推。减少number of full table row reads,进而减少Disk IO.

注：read full table row：读表一行的所有column

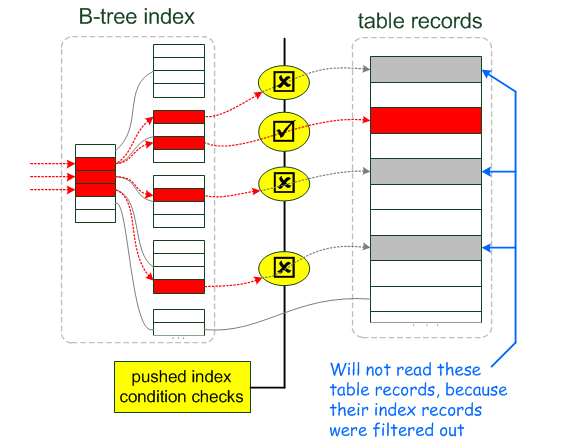
* 可用于range, ref, eq\_ref, and ref\_or\_null when need access full-rows.
* 只能用于二级索引。聚簇索引完整record已read into the InnoDB buffer.不会减少IO。
* 用于select字段过滤的索引不覆盖情况(否则不用回表读数据)

示例

people表,key（zipcode，name）

1. **select** \* **from** people **where** zipcode=’123′ and **name** LIKE ‘%n%’;

withOut ICP,在store engine，查询二级索引zipcode，根据id去回表，查到数据在Server基于name条件过滤。 with ICP, name条件下推到store engine。减少回表次数(disk IO)。



5.6引入,默认开启

查看 show variables like 'optimizer\_switch';

开启set optimizer\_switch='index\_condition\_pushdown=on';

关闭set optimizer\_switch='index\_condition\_pushdown=off';

### Case3 Count

短索引

InnoDB扫表的时候选择最小的索引来降低成本。

非聚簇索引比聚簇索引小很多，建表时，除了主键索引，创建一个非主键索引还是有必要的。

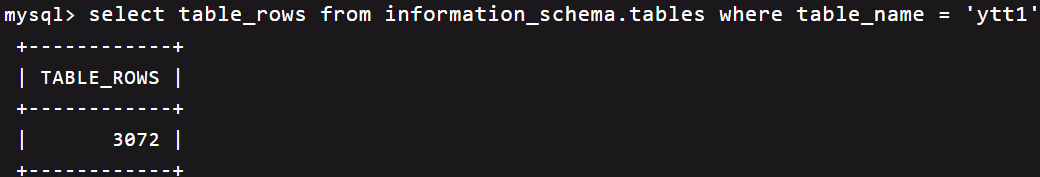
已经建了二级索引用短索引count

标准推荐取法

MySQL 8.0做了优化

下面三种用于sql不加where过滤条件

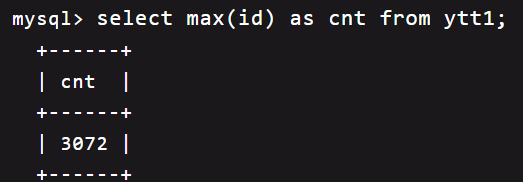
查数据字典



适合新闻展示，比如行数非常多，每页显示几行，一般后面的很多大家也都不怎么去看。缺点是数据不是精确值。

根据表结构特性特殊的取值

这里假设表 ytt1 的主键是连续的，并且没有间隙，那么可以直接



模拟 MyISAM 的计数器

维护表的行数

### Case4 子查询

### Case5 超级大表优化

单表500万行

阿里《Java开发手册》提出单表行数超过500万行或者单表容量超过2GB，才推荐进行分库分表

这个数值是如何评估出来的呢？

事实上数值和实际记录的条数无关，与MySQL的配置以及机器硬件有关。MySQL为提高性能，将数据装载到内存中。InnoDB buffer size足够情况下，能完成全加载进内存，查询不会有问题。但当单表到上限，SQL查询会产生磁盘IO，导致性能下降。最终导致的问题都是内存限制。这里，增加硬件配置，可能带来立竿见影的性能提升。

根据机器的情况综合评估，如果心里没有标准，那么暂时以500万行作为一个统一的标准，相对而言算是一个比较折中的数值。

解决：

* 数据迁移
* 单表优化
* 读写分离
* 缓存
* 表分区
* 垂直拆分
* 水平拆分
* 兼容MySQL且可水平扩展的数据库
* NoSQL
* ES

分段查询

一些场景一条sql不可避免的要查大量数据，造成db扫描行数过多，超过处理能力，延迟较大。比如在一些用户选择页面中，可能一些用户选择的时间范围过大，造成查询缓慢。

优化：通过程序分段查询，多次查询，将结果合并处理进行展示。本质上将压力转移到应用服务器，好处在于方便水平扩展，db服务器的资源相对更宝贵。

1.分区表

2.Hadoop

* 开源解决方案：hadoop家族。hbase/hive。投入和运维成本较高
* 云解决方案：是一种未来趋势，大数据由专业公司提供服务，按量付费，成本极低。

参考

[Reference1](https://mp.weixin.qq.com/s?__biz=MzA4NjExMjE2OQ==&mid=2650119077&idx=1&sn=0e8cd26af022aa508cdb710270e4fac4&chksm=87cc9963b0bb107588d3f5752908604db58f49be6e92660c0321c9e69ae978c7383850a469e2&mpshare=1&scene=1&srcid=#rd)

PLSQL

**优势**

1 SQL语言只是访问、操作数据库的语言。PLSQL具备高级语言的特性，比如模块化，流程控制，异常。

2.集成在数据库中，调用更快。减少了网络的交互，从而减少网络传输时间。

[一份非常完整的 MySQL 规范](https://mp.weixin.qq.com/s?__biz=MzA4NjgxMjQ5Mg==&mid=2665762427&idx=1&sn=214d86137cb9dc8dad563c80ba5248c5&chksm=84d21c58b3a5954e0cb907556bbaef1c7f8cf9414560724e215157b5bf16001d5b552aa67a4a&mpshare=1&scene=1&srcid=&sharer_sharetime=1575772869866&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=b2910d07836452c9801672029dc7607723bbd3214fa00f64e78d69ec76531376ebae39b823c878eb841a6f40b90cee614118e87c07e726509aad9671e5c5f8c55dae389ee05c8954226ef68b9a7459fd&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=AWZr8BtBTY3ISwQ49tKGTqI%3D&pass_ticket=MIfYYQfosPGnnbARcGezUaxCRYCxWaZT1QLPdDjdlq4awG7Y%2BvZ0lr3vA2uldvPH)

### Limit

* Order by + limit <row\_count>

数据够row\_count rows停止排序,不用对entire result排序. 索引已排序,速度很快. filesort大部分数据需排序,即limit影响不大

### in/exists

[Reference](https://www.jianshu.com/p/f212527d76ff)

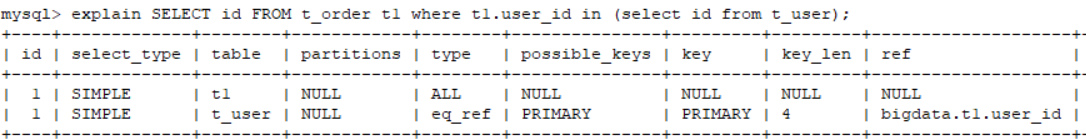
1. **select** \* **from** A **where** id in (**select** id **from** B);
2. **select** \* **from** A **where** exists (**select** 1 **from** B **where** A.id=B.id);

in是内存中遍历比较，exists查DB。B表数据量较大时，exists效率优于in。

exists

1. Array A=(select \* from A);
2. **for**(**int** i=0;i<A.length;i++) {
3. **if**(exists(A[i].id) { //执行select 1 from B where B.id=A.id是否有记录返回
4. resultSet.add(A[i]);
5. }
6. }
7. **return** resultSet;

exists会执行A.length次，不缓存B表的结果。使用场景：B表比A表数据量大。



in原理

1. Array A=(select \* from A);
2. Array B =(select \* from B);
3. **for**(**int** i=0;i<A.length;i++) {
4. **for**(**int** j=0;j<B.length;j++) {
5. **if**(A[i].id==B[j].id) {
6. resultSet.add(A[i]);
7. **break**;
8. }
9. }
10. }

Sql优化器(Mysql 5.7)

Case 1

1. --执行sql
2. **select** h.\* **from** fund\_history h **where** h.fund\_id in (**select** f.id **from** fund f)
3. --优化器处理后
4. **select** h.\* **from** fund f join fund\_history h **where** h.fund\_id=f.id;

case 2

1. **select** f.\* **from** fund f **where** f.id in (**select** h.fund\_id **from** fund\_history h)
2. --优化后
3. **select** f.\* **from**  fund f semi join fund\_history h **where** h.fund\_id = f.id;

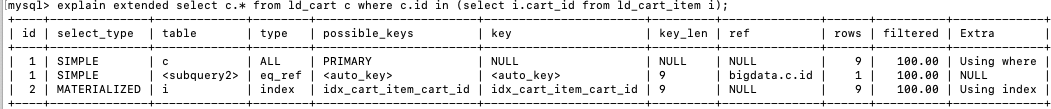
在5.5版本优化结果为

1. **select** f.\* **from**  fund f **where** <in\_optimizer>
2. f.id,<exists> (select from fund\_history  **where** <cache>f.id = h.fund\_id)

有一条满足即返回

case 3

1. **select** i.\* **from** cart\_item i **where** i.cart\_id in (**select** c.id **from** cart c);
2. 优化后
3. **select** c.\* **from** ld\_cart c semi join ld\_cart\_item I
4. **where** (`<subquery2>`.`cart\_id` = c.id)



# 附录 排查工具

mysql提供的指标很多，主要监控4大指标

1. information\_schema
2. Status: show [**global**/session] status like '%xxx%';
3. Variables: show variables like '%max\_connections%';
4. Innodb: show [**global**] status like 'Innodb\_%'

## Show Profile

show profile

[Reference](https://dev.mysql.com/doc/refman/5.7/en/general-thread-states.html)

General Thread States

* checking permissions
* Creating sort index：processing SELECT that is resolved using an internal temporary table
* Sorting result：和Creating sort index类似, but for nontemporary tables.
* Sending data: reading and processing rows, and sending data to the client. Because perform large amounts of disk access (reads), 耗时最长。

## Show Status

[Reference](https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html)

provides server status information

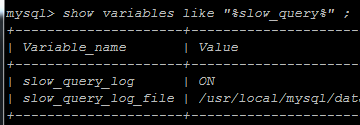
* Bytes\_received：The number of bytes sent from client.
* Bytes\_sent：The number of bytes sent from server.
* Handler\_commit: Number of internal COMMIT statements.
* Handler\_read\_key: Number of requests to read a row based on a key. If this value is high, your tables may be properly indexed for your queries
* Handler\_read\_prev: Number of requests to read the previous row in key order. Mainly used to optimize ORDER BY ... DESC.
* Questions: Number of statements executed by the server.
* Select\_scan: Number of joins that did a full scan of the first table.

[MySQL中的analyze与optimize](https://emacsist.github.io/2016/09/26/mysql%E4%B8%AD%E7%9A%84analyze%E4%B8%8Eoptimize/)

## 慢查询日志

slow\_queries：超过long\_query\_time数量。（show variables like "%long\_query\_time%"）;

show variables like "%slow\_query%" ;



开启日志

set global slow\_query\_log=ON;

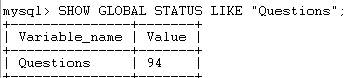
分析

mysqldumpslow /usr/local/mysql/data/mysql/slc11fsp-slow.log

## 查询吞吐量

QPS

Questions ： 查询的总数（统计开始时间点为service start）。



TPS

TPS = (Com\_commit + Com\_rollback)/seconds

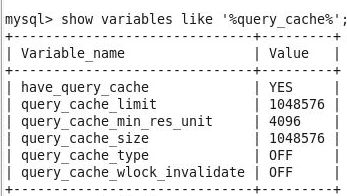
mysql > show status like 'Com\_commit';

mysql > show status like 'Com\_rollback';

## 缓冲池使用情况

Query Cache

show variables like '%query\_cache%'



* query\_cache\_type：0-不启用查询缓存；1-启用，2-启用(SQL+SQL\_CACHE)，默认值为0；
* query\_cache\_size：缓存区总大小
* query\_cache\_limit：缓存单条查询记录最大值，缺省1M；
* query\_cache\_min\_res\_unit缓存块大小，默认4KB，设置值大对大数据查询有好处，但如果查询都是小数据 查询，就容易造成内存碎片和浪费；

临时关闭命令

set global query\_cache\_size=0

set global query\_cache\_type=0

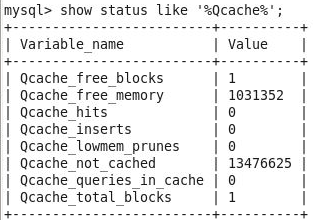
永久的修改配置文件my.cnf ,添加下面的配置即可。

query\_cache\_type=0

query\_cache\_size=0

Cache Status

查看缓存命中率



* Qcache\_hits：缓存命中次数
* Qcache\_free\_blocks：缓存中相邻内存块的个数。数目大说明可能有碎片。FLUSH QUERY CACHE会对缓存中的碎片进行整理，从而得到一个空闲块。
* Qcache\_free\_memory：缓存中的空闲内存。
* Qcache\_inserts：每次插入一个查询时就增大。命中次数除以插入次数就是不中比率。
* Qcache\_lowmem\_prunes：缓存出现内存不足并且必须要进行清理以便为更多查询提供空间的次数。这个数字最好长时间来看;如果这个数字在不断增长，就表示可能碎片非常严重，或者内存很少。(上面的 free\_blocks和free\_memory可以告诉您属于哪种情况)
* Qcache\_not\_cached：不适合进行缓存的查询的数量，通常是由于这些查询不是 SELECT 语句或者用了now()之类的函数。
* Qcache\_queries\_in\_cache：当前缓存的查询(和响应)的数量。
* Qcache\_total\_blocks：缓存中块的数量。

key Buffer 命中率

key\_buffer\_read\_hits = (1-key\_reads / key\_read\_requests) \* 100%

key\_buffer\_write\_hits = (1-key\_writes / key\_write\_requests) \* 100%

mysql> show status like 'Key%';

InnoDB Buffer命中率

innodb\_buffer\_read\_hits = (1 - innodb\_buffer\_pool\_reads / innodb\_buffer\_pool\_read\_requests) \* 100%

mysql> show status like 'innodb\_buffer\_pool\_read%';

Query Cache命中率

Query\_cache\_hits = (Qcahce\_hits / (Qcache\_hits + Qcache\_inserts )) \* 100%;

mysql> show status like 'Qcache%';

Table Cache状态量

mysql> show status like 'open%';

Thread Cache 命中率

Thread\_cache\_hits = (1 - Threads\_created / connections ) \* 100%

mysql> show status like 'Thread%';

mysql> show status like 'Connections';