# 第一章 Mysql

发展历史

MySQL 1.0 (1996）

1996年，MySQL 1.0发布 ，只有CRUD基本操作

1999-2000年，MySQL AB在瑞典成立。 开发出了 Berkeley DB引擎, MySQL从此开始支持事务处理。

2000 年 MySQL 公布了自己的源代码，并采用GPL许可协议，正式进入开源世界。

MySQL 3.23（2001）

实现了SQL 查询系统，MyISAM 代替了老旧且有诸多限制的 ISAM 引擎。InnoDB 引擎也已经可以使用，Mysql真正“诞生”的时刻。

MySQL 4.0（2003）

不支持存储过程、触发程序、服务器端指针或视图，这时候的MySQL还不是一个企业级数据库。

MySQL 5.0（2006）

出现了一些“企业级”特性：视图、触发器、存储过程和存储函数。同时Oracle收购的InnoDB Oy发布了 InnoDB plugin。

MySQL 5.1（2008）

Sun 收购 MySQL AB 以后发布的首个版本，研发时间长达五年。

2008年 MySQL被Sun公司收购。

2009年Oracle收购Sun公司，MySQL 转入Oracle 门下。Oracle将MySQL分成了社区版和企业版。

MySQL 5.5（2010）

默认InnoDB plugin引擎。具有提交、回滚和crash恢复功能、ACID兼容。

行级锁

MySQL 5.6（2012）

MySQL 5.6是MySQL历史上一个里程碑式的版本，这也是目前生产上应用得最广泛的版本。

MySQL 5.7（2015）

MySQL 8.0（2018）

6.1.1 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
* 执行：调用storage engine和调用后的数据处理，包括排序、分组等。
* Connection/Thread handing：为请求做连接处理，授权认证，安全等。
* Query Cache:

## 1.1 Server层

Mysq查询优化器

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Therefore, a view can be thought of as a stored query or a virtual table. You can use views in most places where a table can be used.

A cursor is a handle or name for a private SQL area--an area in memory in which a parsed statement and other information for processing the statement are kept.

## 1.2 存储引擎

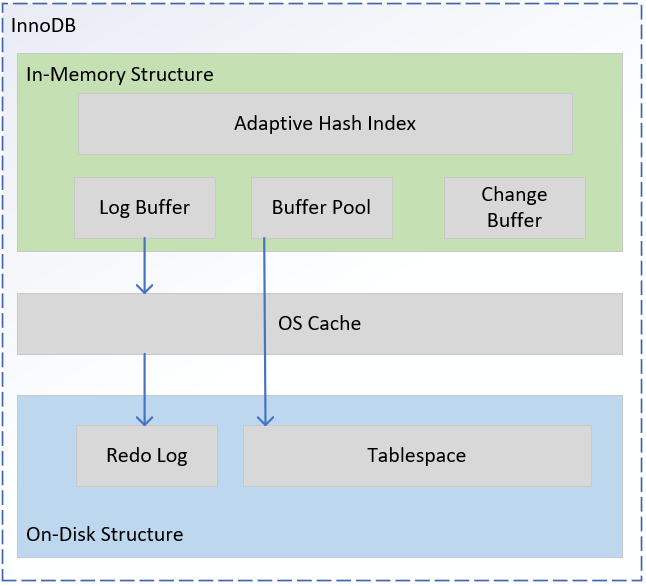
存储引擎。data/index的save和query，事务处理。

* InnoDB：支持事务，采用行级锁，聚簇索引。
* MyISAM：表级锁，不支持事务，
* TokuDB：适合读少写多，提升了在高负载情况下的 InnoDB 的性能。

影响引擎性能的因素：数据大小，I/O请求量，主键还是二级索引。

# 第二章 InnoDB

## 2.1 逻辑架构



* In-Memory Structure：如图，InnoDB内存结构包含四大核心组件
* OS Cache：内核态内存
* On-Disk Structure：文件系统。Redo Log和Tablespace

三层交互：1.通过OS Cache落地数据（短箭头）2.直接O\_Direct落地数据（长箭头）；

内存结构

* 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

[InnoDB架构，一幅图秒懂！](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651962899&idx=1&sn=a7f2458680c8f6d295e8c324c7e86cc7&chksm=bd2d0bcf8a5a82d9ac8cc016d21a34b5e3b3be9e235dae29393c6670244dbd53c57ed3b1df16&mpshare=1&scene=1&srcid=&sharer_sharetime=1575773607966&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=28093a38690020dbff8d96a757f059eb0d987df57acbde812e95d2a0143851956a70b3a34619b04fb3ff99aca757dbfe5511901b0e9f17a8dd9b8f324ea7867cd272c06bc354cfb9a9aa1447e66d09c9&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=Afz4S%2FXREMSr1nRjMBoUZfU%3D&pass_ticket=MIfYYQfosPGnnbARcGezUaxCRYCxWaZT1QLPdDjdlq4awG7Y%2BvZ0lr3vA2uldvPH)

### 2.1.1 Buffer Pool

[Reference](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651962450&idx=1&sn=ce17c4da8d20ce275f75d0f2ef5e40c9&chksm=bd2d098e8a5a809834aaa07da0d7546555385543fb6d687a7cf94d183ab061cd301a76547411&scene=21#wechat_redirect)

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。

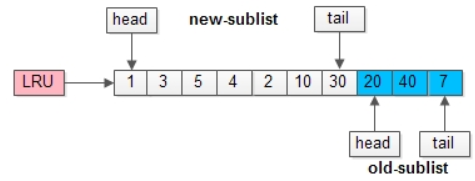
##### 三种链表

LRU链表

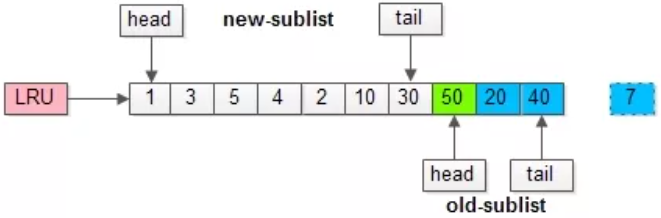
传统LRU算法十分直观，OS，memcache等很多软件都在用，MySQL做了优化，解决两个问题：

* 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。

mysql的LRU链分为两个部分：新生代(new sublist)和老生代(old sublist)



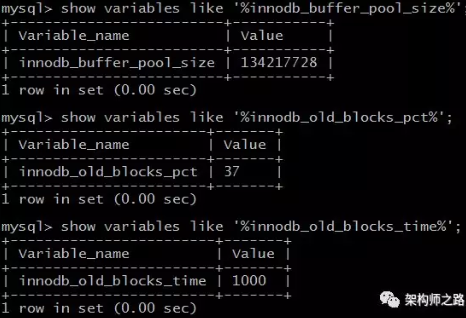
新生代的尾(tail)连接着老生代的头(head)，预读新页从old-sblist head处插入，假如有一个页号为50的新页被预读加入缓冲池



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。

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？

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)

### 2.1.2 Change Buffer

[Reference](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651962467&idx=1&sn=899ea157b0fc6f849ec80a4d055a309b&chksm=bd2d09bf8a5a80a972a2e16a190ed7dffe03f89015ead707bdfcc5aeb8388fb278f397c125f1&scene=21#wechat_redirect)

加速写请求，避免每次写入都进行磁盘IO。

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)。

保证一致性

* 读取，会命中缓冲池的页；
* 缓冲池LRU数据淘汰，会将“脏页”刷回磁盘；
* 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。

### 2.1.3 Adaptive Hash Index

自适应哈希索引.

[Reference](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651962875&idx=1&sn=c6b3e7dc8a41609cfe070026bd27b71d&chksm=bd2d08278a5a813108b1f4116341ff31170574b9098e2708cbc212b008a1fac8dfd1ffeabc6b&scene=21#wechat_redirect)

InnoDB到底支不支持哈希索引?

对于InnoDB的哈希索引，确切的应该这么说：

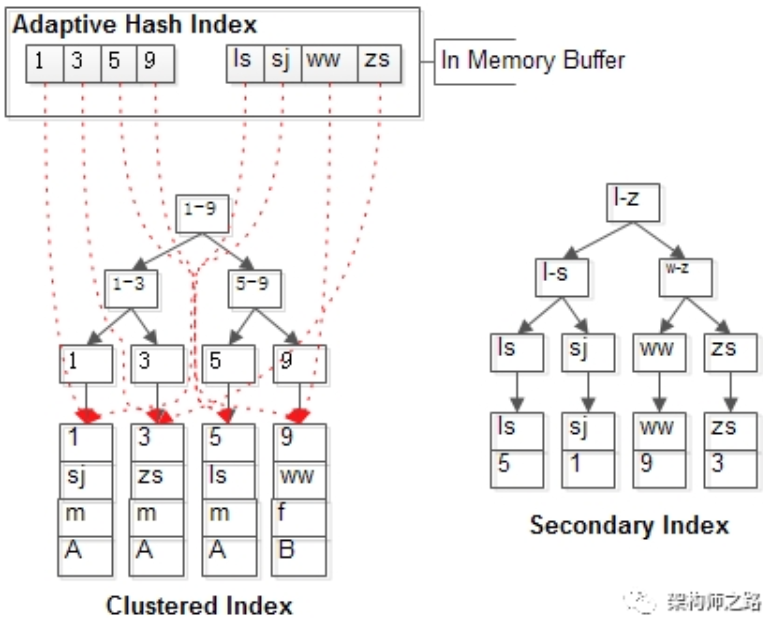
（1）InnoDB用户无法手动创建哈希索引，这一层上说，InnoDB确实不支持哈希索引；

（2）InnoDB会自调优，若判定建立自适应哈希索引，能够提升查询效率，InnoDB会建立相关哈希索引，这一层上说，InnoDB又是支持哈希索引的；

建立AHI过程

不管聚集索引还是普通索引，记录定位的寻路路径(Search Path)都很长

在MySQL运行的过程中，如果InnoDB发现，有很多SQL存在这类很长的寻路，并且有很多SQL会命中相同的页面(page)，InnoDB会在自己的内存缓冲区(Buffer)里，开辟一块区域，建立自适应哈希所有AHI，以加速查询。



从这个层面上来说，InnoDB的自使用哈希索引，更像“索引的索引”，毕竟其目的是为了加速索引寻路。

### 2.1.4 Log Buffer

Redo log

想象成ddl语句，用来恢复数据。written by incomplete transactions

磁盘随机写性能较低，若每次事务提及都刷盘，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。保障已提交事务的持久化特性。

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。

示例：a=1,Update tb where a=2

🡪Begin trans

🡪Update Operate

🡪记录a=2到redo log buffer,记录a=1 to undo log buffer

🡪Undo/undo log写到磁盘

🡪 数据写到磁盘

🡪 Commit

回滚段(rollback segment)

回滚日志，存储undo日志的地方。记录事务修改前的数据，用于rollback操作。

## 2.2 事务

一组操作完成一个任务。在计算机术语中，事务通常就是指数据库事务。

### 事务特性

* 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数据库，否则事务应该将其回滚到最初状态。实现: 完整性约束。这里的一致性偏向于数据的正确性。
* 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。实现：事务的锁来进行控制。
* Durability（持久性）

[Reference](https://tech.meituan.com/innodb_lock.html)

TODO2

应用层加锁：方法串行化执行。等于隔离级别的Serializable。

数据库要保证最大的并发执行…读锁写锁。

Hollis: [Reference](https://www.hollischuang.com/archives/tag/%E4%BA%8B%E5%8A%A1)

### 隔离级别

事务的隔离级别和数据库的性能呈反比。

* Read uncommitted：会出现脏读。
* Read committed：。but其他事务所做的修改或删除，出现不可重复读
* Repeatable read: 其他事务update或delete，可重复读。but插入会发生幻像读。
* Serializable：序列化。表锁。

select是否开启事物的区别？

Select开启事物后可保证repeateable read

### 实现原理

实现手段，这几种的结合。

* mysql锁技术
* MVCC
* redo log/undo log

[MySQL事务的实现原理](https://mp.weixin.qq.com/s?__biz=MzUxOTAxODc2Mg==&mid=2247486306&idx=1&sn=51695f0dcf4aeb3e937efddd7d8978a6&chksm=f9814b07cef6c21174132683a0ef29f0c567fda2c42684b956ed24a576a2b6a5dc15dac5e7b6&mpshare=1&scene=1&srcid=&sharer_sharetime=1584343256499&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=68f6c88a59057261f8127d2e99803a0d54fc1744c9500ff6aaac1aa909280c129ac64e968953f325e12a95dedf994cc3dd42052489cc2eb34870ccc011ffdd6c00cab83020a74818990d31bc80f4d512&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AQg%2FT7IK3krdOsrfVuusoB8%3D&pass_ticket=B%2BlG9Q73ij2CS0QWkPAU7hN%2FOrst0lFE3qTly6tnhpu0h5HiSVVUv9bzLxO44OxF)

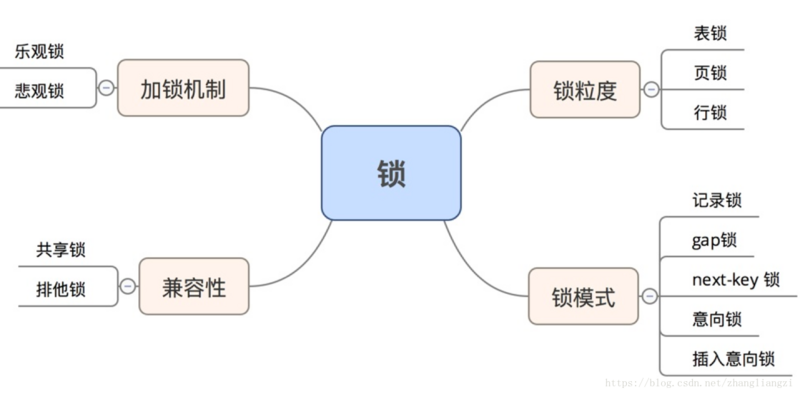
## 2.3 并发控制

[Reference](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651961444&idx=1&sn=830a93eb74ca484cbcedb06e485f611e&chksm=bd2d0db88a5a84ae5865cd05f8c7899153d16ec7e7976f06033f4fbfbecc2fdee6e8b89bb17b&scene=21#wechat_redirect)

保证数据一致性，常见手段：

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

### Locking



锁粒度

锁定的资源包括行、表、页、簇和数据库。尽量锁定比较小的对象，最小化锁成本。

* 行级锁：行级锁分为[共享锁](http://www.hollischuang.com/archives/923)和[排他锁](http://www.hollischuang.com/archives/923)。
* 表级锁：lock tables tablename read|write
* 页级锁

锁模式

* Record Locks：记录锁。锁定索引中的一条记录。
* Gap Locks：间隙锁
* Next-Key Locks：索引记录上的record lock和在记录之前的间隙锁的组合。
* Intention Locks：意向锁
* Insert Intention Locks：插入意向锁
* AUTO-INC Locks：自增锁
* Predicate Locks for Spatial Indexes：空间索引断言锁

意向锁(Intention Locks)

意向锁是表级锁，其设计为了在一个事务中揭示下一行将要被请求锁的类型。InnoDB中的两个表锁：

意向共享锁（IS）：表示事务准备给一个数据行加共享锁前必须先取得该表的IS锁

意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁必须先取得该表的IX锁。

意向锁是InnoDB自动加的，不需要用户干预。

加锁机制

悲观锁&乐观锁非db真正存在的锁，是并发控制的手段。memcache,hibernate都有类似概念。

Shared and Exclusive Locks

* S锁：共享锁之间不互斥，读读可并行。T1对数据A加共享锁后,事务T2能对A再加共享锁，不能加排他锁。用法select...lock in share mode;

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，但是可以读。

1. **SELECT** ... **FOR** **UPDATE**;

|  |  |  |
| --- | --- | --- |
| **时间** | **trx1** | **trx2** |
| T1 | begin; |  |
| T2 | select…for update | begin; |
| T3 |  | Upate：阻塞到Lock wait timeout exceeded. |
| T4 | commit |  |
| T5 |  |  |

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

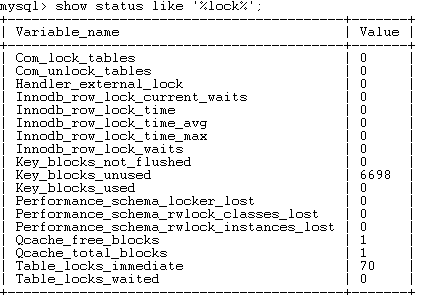
统计lock信息

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



ReadOnly

查看 show global variables like "%read\_only%";

### MVCC

最原生的锁，加锁后禁止其他任何线程访问同一资源，但很多应用都是读多写少的场景。SX锁读加锁，读写冲突，而读读不冲突,大大提高并发度。

MVCC读不加锁，读写不冲突。进一步提高并发度。

快照读&当前读

在MVCC并发控制中，读操作可以分为两类：快照读与当前读。

* 快照读：简单的select操作，读取记录中的可见版本（可能是历史版本），不加锁。
* 当前读：特殊的select操作、insert、delete和update。读取记录的最新版本，且当前读返回的记录都会加上锁，保证其他事务不会再并发修改这条记录。

Problem：多版本导致read的数据可能过期，会出现**丢失更新**(P1)。

Mysql默认隔离级别RR，有lost update 问题。会引起其他潜在问题,比如营业额增加，银行转账。如下，Trx1在更新数据之前，数据已被trx2修改，trx1最终提交数据将trx2的提交覆盖，导致trx2丢失更新。

|  |  |  |
| --- | --- | --- |
| **时间** | **trx1** | **trx2** |
| T1 | begin;select m:=100 |  |
| T2 |  | select m :=100元 |
| 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
* 乐观锁：应用层实现，比如Hibernate。
* 设置为Serializable

(Oracle可以把隔离级别设置成read-only。?)

实现原理

写任务基于undo日志操作，将数据克隆一份，以版本号区分；并发读任务读取旧版本数据，不阻塞；

InnoDB的每个row数据有三个内部属性

* DB\_TRX\_ID：记录最近一次修改它的事务ID；
* DB\_ROLL\_PTR：指向回滚段undo日志的指针；
* DB\_ROW\_ID：单调递增的行ID；

### 场景分析

[一条简单的更新语句，MySQL是如何加锁的？](https://mp.weixin.qq.com/s?__biz=MzUxOTAxODc2Mg==&mid=2247486382&idx=1&sn=5f9c392f354eb044f0a2150e398dac81&chksm=f9814bcbcef6c2dd5050e20547e8d8002631d4a61e42b12689e2ab1b6cfc9de2c6b0dc420d0d&mpshare=1&scene=1&srcid=&sharer_sharetime=1584343016796&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=6aa5fee81a4d4f3deb9ff784f6775c1b9ea8f7a8b9173126b6c58a3567c631aa47674ca2f041ee15a9d5e0d752633be9c581df7b0f1d1b709f3989621083445057eeebc4a34543bbd673c0a95a7221ca&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AUFOt%2FELup%2Fy8w%2FaiUhASJE%3D&pass_ticket=B%2BlG9Q73ij2CS0QWkPAU7hN%2FOrst0lFE3qTly6tnhpu0h5HiSVVUv9bzLxO44OxF)

[超全面的MySQL语句加锁分析](https://mp.weixin.qq.com/s?__biz=MzIxMjE5MTE1Nw==&mid=2653198050&idx=2&sn=68a6594ac35976532ad6a0eec6dc06dd&chksm=8c99e438bbee6d2e8e9bdf80a3312723c92e20afb1918ca6d29cd8ded1d670637c413704ca61&mpshare=1&scene=1&srcid=&key=1873ed4ed1cb893ed110f2754f79b1183d5141799e43f47edf7d9c0e0f45ab26e0a4d4b0a9446eb6c461f63f508e43177ce587fdebbe6ed9a678ddc5a864f9a271eba71ebaa252fc0d283e9befefd059&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62060739&lang=zh_CN&pass_ticket=U1Rycznd0MxcqdQDoQj52v1tXH53Lbffgbif%2FRFGmW%2F1SWy%2F7LEwVLTizUTqU4gb)

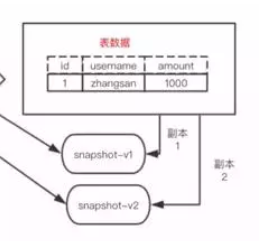
[关于MySQL内核，一定要知道的！](https://mp.weixin.qq.com/s?__biz=MjM5ODYxMDA5OQ==&mid=2651961620&idx=1&sn=d858c302799cad451656129885214767&chksm=bd2d0cc88a5a85de11ed376570f78a22954e88aad06f0138b3fbfb7be1968699421ac0b99889&mpshare=1&scene=1&srcid=0728T4ub8zI10hzbnthQGJvF&sharer_sharetime=1595901780455&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=7674ee21d747fecfb03dada8aa331b6f5aa48abdd0ade0aaa65b8658ac61979df03836695a0357e72d5f48e9b99c8dfdffcac281241063cbde8bb22edeab819bb74347300659e23528f5e6ef65ea31ff&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10+x64&version=62090529&lang=zh_CN&exportkey=AeyHzOVhhEKjNib1wei8%2B5s%3D&pass_ticket=rFecmM8pFM8Xrl6PmMcg1aAnZqDMRuOH49roh8CrtMHGI8h7sh2bvtv8FTh6AVxl)

##### 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。

ReadView



MVCC数据临时保存多个版本，进而实现并发控制。

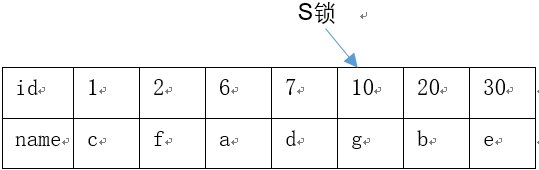
加锁语句

* 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

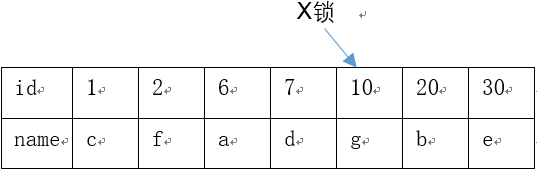
##### 主键+RC/RR

Table:T1(id primary key,name)

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



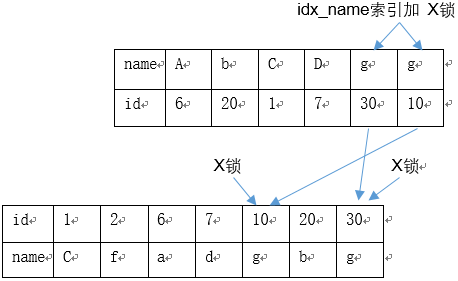
update T1 set name=’e1’ where id=10 for update;



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

##### 二级索引+RC

Table：T1(id primary key,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)

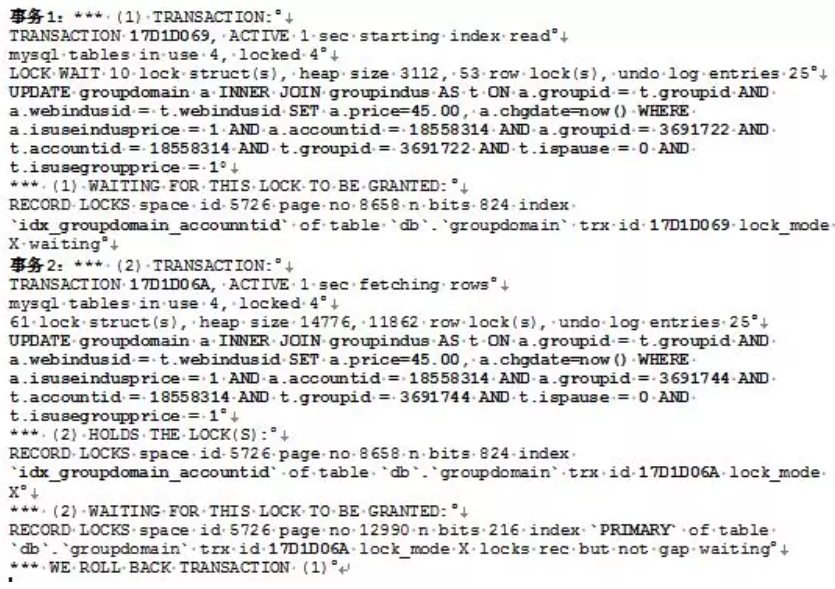
Table:T1(id primary key,name key)

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

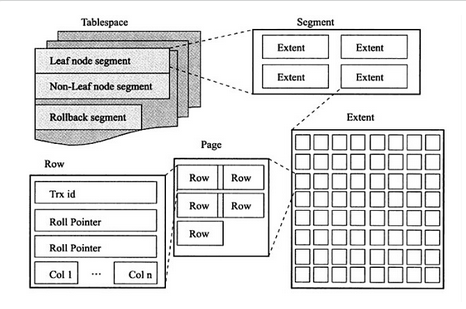
T2: update T1 set name=’e1’ where id=30;

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 存储结构



Tablespace

表空间。innodb\_file\_per\_table默认开启，每张表内数据单独放到一个表空间内

Segment(段)

表空间是由多个段组成的。Innodb每个索引对应两个segment：

* Leaf node segment：B+树的叶子节点，存数据。
* Non-leaf node segment：索引段为B+树的非索引节点。

单次最多分配4个extends给segment。

Rollback segment

Extend(区)

* 区由连续页组成的空间，每个区为64个连续的pages，size为1MB(64 x 16k = 1024k)。
* 为了保证区中页的连续性，InnoDB存储引擎一次从磁盘申请4~5个区

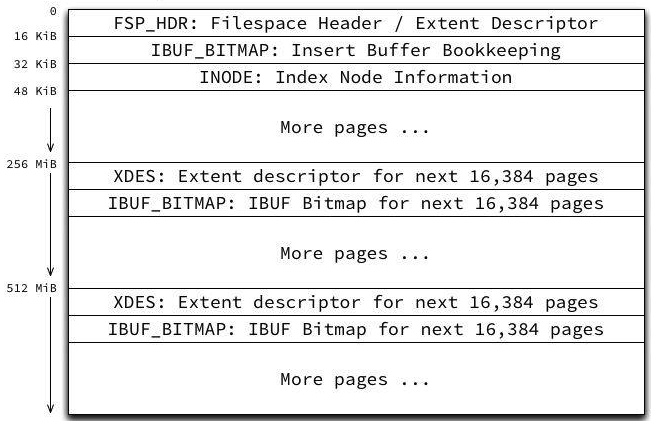
Page

页是InnoDB磁盘管理的最小存储单位。

Row

对应数据表里一条条记录，由page header、page trailer、page body组成

### 2.4.1 Tablespace



* FSP\_HDR：first page。维护256个extents的使用情况。
* IBUF\_BITMAP：跟踪随后的每个page的change buffer信息
* INODE：维护segement。Inode Page的每个Inode Entry描述了一个段(FSEG)。

|  |  |  |
| --- | --- | --- |
| Macro | offset | Desc |
| FSEG\_ID | 0 | Segment id(0表示未使用) |
| FSEG\_NOT\_FULL\_N\_USED | 8 | 链表上已被使用的Page数量 |
| FSEG\_FREE | 12 | 分配给该Segment完全且未使用的extent链表 |
| FSEG\_NOT\_FULL | 28 | 部分可用的extent链表 |
| … |  |  |
| FSEG\_FRAG\_ARR[0] | 64 | 碎页数组首页地址 |
| … |  |  |

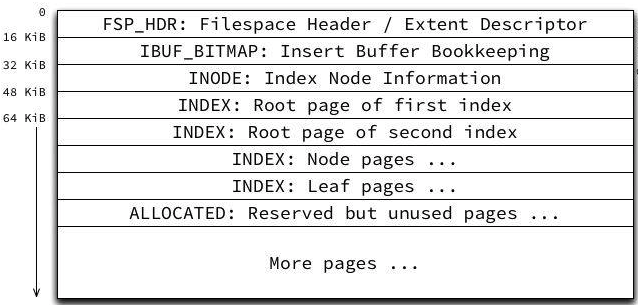
每个inode页默认存储为85个记录,所以一个INODE页最多可以保存42个索引信息(一个索引使用两个段)。如果表空间有超过42个索引,则必须再分配一个INODE页。

todo出现的两个关于INODE的链表的FSP\_HDR结构。

* XDES：XDES和FSP\_HDR页的结构是相同的

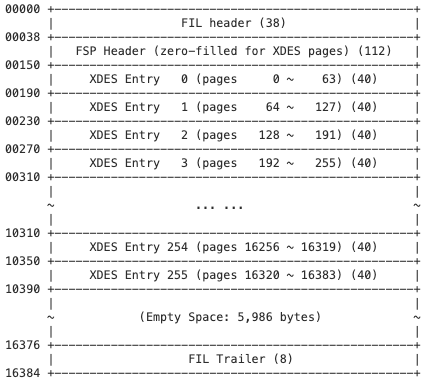
Per-table space files

默认情况下，Innodb有一个共享表空间ibdata1,所有数据都存放在这个space内。如果启用了innodb\_file\_per\_table参数，则每张表内的数据可以单独放到一个表空间内（.ibd文件）。文件结构：

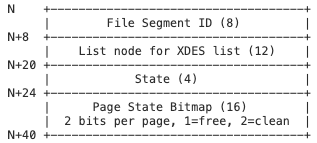


FSP\_HDR

Page Body 中保存了256个 extent descriptors 。



XDES Entry,格式如下



关于Page数

tablespace:则db数据上限为2^32\*16kb=64TB。

Page：default size=16K=16384Byte.

数据行数

Rows：每页存放的rows有硬性定义，最多允许存放16KB/2-200=7992行记录。

|  |  |  |
| --- | --- | --- |
|  | Page数量 | 数据行数 |
| XDES Entry | 64 | 50万 |
| XDES | 256\*64=2^14=16384 | 13亿 |
| tablespace | 2^32。（offset为4Byte,32位） |  |

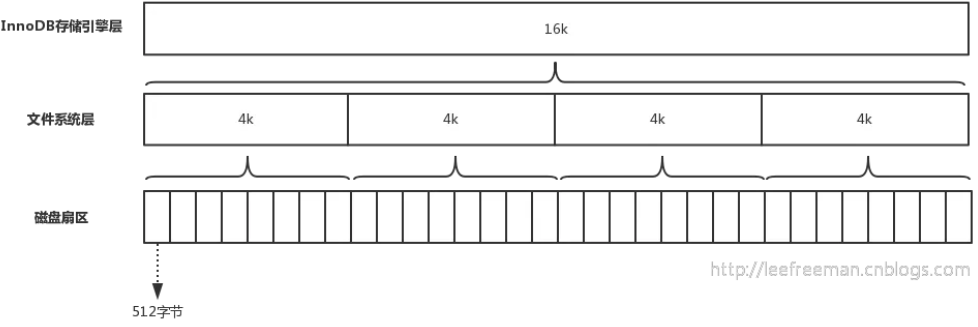
### Page(页)

页是InnoDB磁盘管理的最小存储单位。

存储单元

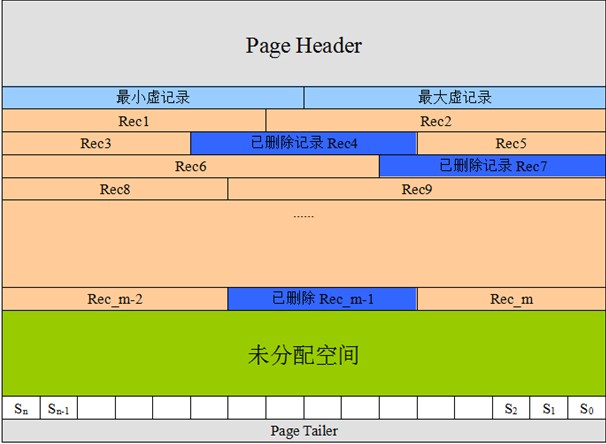
innodb\_page\_size 默认每个页的大小为16KB。

磁盘存储数据最小单元是扇区，一个扇区的大小是512字节，文件系统（如EXT4）最小单元是块，大小是4k，磁盘扇区、文件系统、InnoDB各自的最小存储单元。



#### Page格式

[Reference](https://www.cnblogs.com/vinchen/archive/2012/09/10/2679478.html)



* File Header：page Type, offset(Page Number)，checksum等
* Page Header：page的元信息，Previous/Next Page、页面空间使用情况等。
* Record Heap：图中已删记录连接成自由空间链表。
* 未分配空间：自由空间链表没有合适位置，回在未分配空间中申请。
* slot：和每个record对应。（也可能是稀疏的？）

页内的有序性

按序存储：插入导致记录移动

链表：查找需要逐个比较

Mysql做法：slot和每个record对应，只需保证slot按序存储，record链表存储即可。

#### 数据结构

1. **struct** Page {
2. fil\_header \* header; //38byte
3. page\_header \* phead; //56byte
4. Infimum+supermum\_records;  //主键值的最大/小值
5. user\_record;
6. free\_space;
7. page\_directory ;
8. fil\_trailer \* trailer;  //8byte
9. };

一条Sql查找过程：

B+树索引本身并不能找到具体的一条记录，能找到只是改记录所在的Page。数据库把页载入到内存，然后通过Page Directory再进行二叉查找。(二叉查找的时间复杂度很低，同时在内存中的查找很快，因此通常忽略这部分查找所用的时间)

fil\_header(头部)

<https://s3.51cto.com/wyfs02/M02/9E/28/wKiom1mMDb2Sjp1wAAG0Fgrpk8g954.jpg>

|  |  |  |  |
| --- | --- | --- | --- |
| Macro | Offset | Size | description |
| FIL\_PAGE\_SPACE\_OR\_CHKSUM | 0 |  | 校验和 |
| FIL\_PAGE\_OFFSET | 4 | 4byte | Page no，页的偏移量 |
| FIL\_PAGE\_PREV | 8 |  | 前驱节点 |
| FIL\_PAGE\_NEXT | 12 |  | 后驱节点 |
| FIL\_PAGE\_LSN | 16 |  | 最近一次修改该page的LSN |
| FIL\_PAGE\_TYPE | 24 |  | 页面类型 |
| FIL\_PAGE\_FILE\_FLUSH\_LSN | 26 |  | Checkpoint for system tablespace |
| FIL\_PAGE\_SPACE\_ID | 34 | 4byte | space id |

FIL\_PAGE\_TYPE(页面类型)

Innodb在磁盘实际存储的文件，包括日志文件、用户表空间、系统表空间文件（ibdata1）、undo tablespace文件、临时表空间文件。这些文件具有统一的结构，用space id标示。

* FIL\_PAGE\_INDEX ：B-tree索引页
* FIL\_PAGE\_TYPE\_ALLOCATED ：磁盘新分配未使用的页
* FIL\_PAGE\_UNDO\_LOG ：Undo log页
* FIL\_PAGE\_INODE ：INODE页
* FIL\_PAGE\_TYPE\_SYS ：系统页，
* FIL\_PAGE\_TYPE\_FSP\_HDR ：file space header page
* FIL\_PAGE\_TYPE\_XDES ：extent desc page
* FIL\_PAGE\_TYPE\_BLOB ：未压缩的BLOB页

用户表空间：用户create table产生的ibd数据文件是一个独立的tablespace。

1. bash-4.1$ ls -l
2. -rw-rw---- 1 mysql mysql      13046 Sep 14  2018 ld\_cart.frm
3. -rw-rw---- 1 mysql mysql     147456 Oct 26 21:13 ld\_cart.ibd

FIL\_PAGE\_SPACE\_ID

FIL Trailer(尾)

page trailer是在文件末尾的最后8个字节

|  |  |  |
| --- | --- | --- |
| **Name** | **Size** | **Remarks** |
| FIL\_PAGE\_END\_LSN | 8 | low 4 bytes = checksum of page,  last 4 bytes = same as FIL\_PAGE\_LSN |

# 第三章 SQL执行过程

## 3.1 B+Tree索引

页（Page）

InnoDB的最小储存单元页（Page），大小是16K

* 叶子节点存放数据
* 非叶子节点存放键值+指针

InnoDB 中一棵 B+ 树能存多少行数据？

单个叶子节点（页）中的记录数=16K/1K=16。（假设一行记录的大小为1k，实际上很多业务数据记录大小通常就是1K左右）

非叶子节点能存放多少指针？

假设主键ID为bigint，长度为8字节，而指针大小在InnoDB源码中设置为6字节，共14字节，

* 一页存放指针数：16384/14=1170 。，
* 一棵高度为2的B+树，能存放：1170\*16=18720条。
* 一个高度为3的B+树，能存放 ：1170\*1170\*16=21902400条。

所以在InnoDB中B+树高度一般为1-3层，它就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次IO操作即可查找到数据。

[面试题：InnoDB 中一棵 B+ 树能存多少行数据？](https://mp.weixin.qq.com/s?__biz=MzUxOTAxODc2Mg==&mid=2247486576&idx=1&sn=2d98942a3cc5b47461b8f33d053a6c40&chksm=f9814c15cef6c5035c30b8a94ceeb473beb039351fb6c55c1218a44208e6742ffa56d65f3b83&mpshare=1&scene=1&srcid=&sharer_sharetime=1593600733324&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=fa39e04f3e1bcf44e31f87dba5b2be40331bb14d91b06f6d0b871a3c57d4a4f991ae3a42e7e45383c21470a93357f3d1dc73c81071da36036a1ee2d25edd98e17de54ac0d3467194042b5ca747aeaf16&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=AaJAyWOnP%2F7mF13qwb300J0%3D&pass_ticket=DtOMze%2BKKn12XUbKEoEGU3IDP31hVGcvP5J6U%2FsAKMj6tmsWp1FAUaI4ydzUyB2Z)

为什么MySQL的索引要使用B+树而不是其它树形结构？比如B树？

因为B树不管叶子节点还是非叶子节点，都会保存数据，导致非叶子节点中能保存的指针数量变少，指针少的情况下要保存大量数据，只能增加树的高度，导致IO操作变多，查询性能变低；

## 3.2 Join算法

[MySQL Join算法与调优白皮书](https://mp.weixin.qq.com/s?__biz=MjM5MjIxNDA4NA==&mid=206832837&idx=1&sn=d216a6070ccac14f3d140a7653f58854&mpshare=1&scene=1&srcid=0416pYMH8WoX1rywsb3fZLpz&pass_ticket=mEnwhQ80zh6NextfkpwDeZoMaE0nxwpLMPgh5bZxdeljaVnVtKi3Pp8T5hDgWuor#rd)

驱动表：也叫外表。行数少的表。inner join时mysql会自动找出数据少的表作用驱动表，从而降低查询成本。

外连接消除

t1 left join t2时t1表，t1为驱动表，但下面场景外连接和内连接可以相互转换，优化器可通过评估表的不同连接顺序的成本，选出成本最低的那种连接顺序来执行查询。

1. **select** \* **from** t1 left join t2 **ON** t1.m1 = t2.m2 **where** t2.n2 **IS** NOT NULL;
2. **select** \* **from** t1 left join t2 **ON** t1.m1 = t2.m2 **where** t2.m2 = 2;

这种外连接查询，指定where子句包含被驱动表中的列不为NULL的条件称之为空值拒绝（reject-NULL）。在被驱动表的WHERE子句符合空值拒绝的条件后，外连接和内连接可以相互转换。

### 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。

### 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。

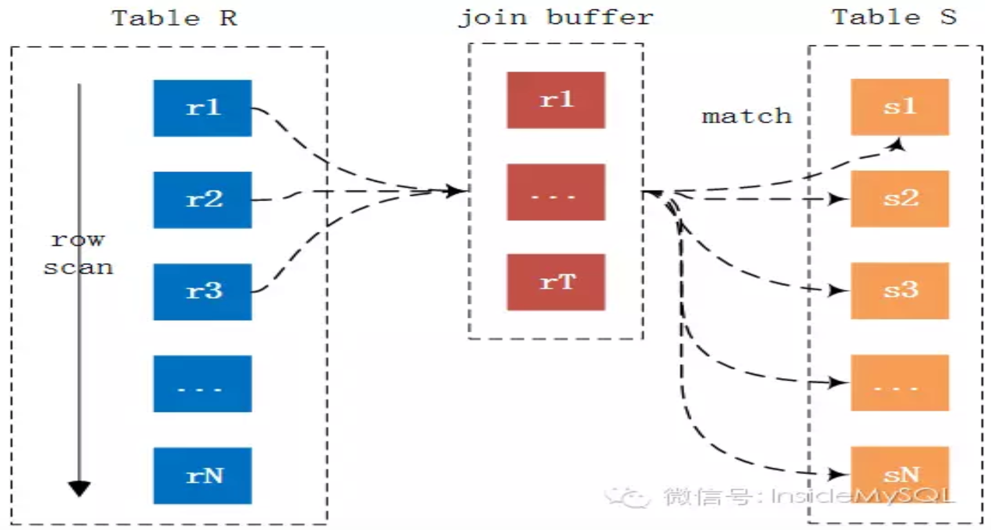
INLJ最大的弊端是其可能需要大量的离散操作。即使SSD的出现大幅提升了随机的访问性能，但对比顺序I/O，依然不在一个数量级上。

### 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。Block Nested-Loop Join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 = …

假设上述SQL语句的外表是a，内表是b，那么存放在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。

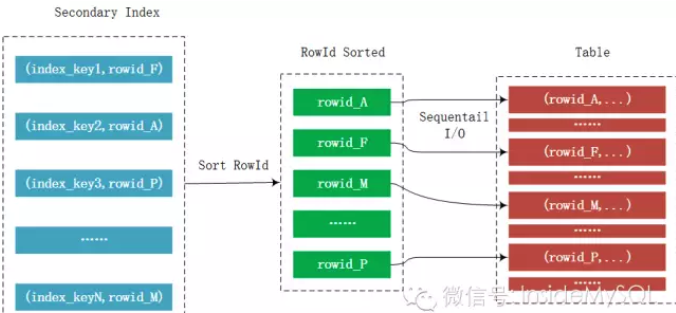
总结

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 |

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。

## in/exists

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

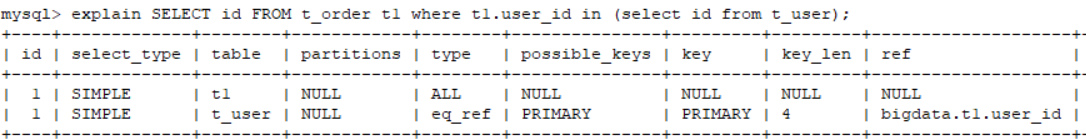
1. **select** \* **from** A **where** id in (**select** id **from** B);
2. **select** \* **from** A **where** exists (**select** 1 **from** B **where** A.id=B.id);

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;

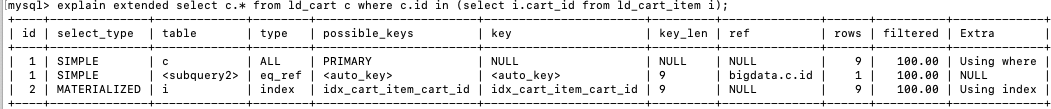
在5.5版本优化结果为

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)



## SubQuery

[Reference](https://juejin.im/book/6844733769996304392/section/6844733770059218951)

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）

别名：内查询（Inner Query）/嵌套查询（Nested Query）

Outer Query：相对于内部查询，包含内部查询。

注意：子查询不能使用order by

SubQuery优化

mysql嵌套子查询效率确实比较低，可以将其优化成连接查询？

子查询的效率

在MySQL 5.6及更高版本，有了更多的子查询优化策略，具体如下：

对于in或者=any子查询：

* semi-Join（半连接）
* Materialization（物化）
* EXISTS策略

对于not in或者<>any子查询：

* Materialization（物化）
* EXISTS策略

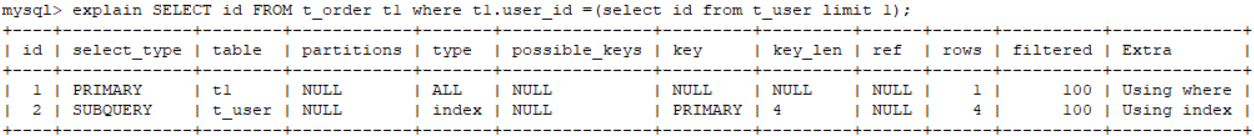
派生表：

* 把派生表结果与外部查询块合并
* 把派生表物化为临时表

派生表：from子句中的子查询。

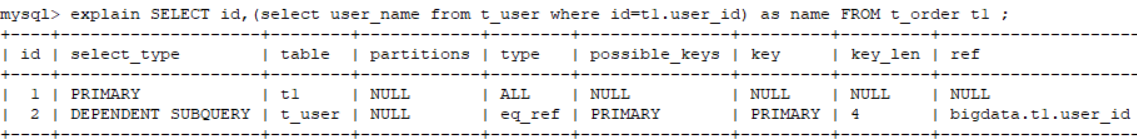
临时表：

Where后

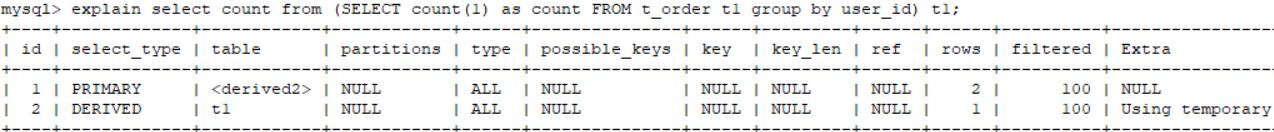


Where后的运算符可以是><=后面只能返回一行数据。

select子查询



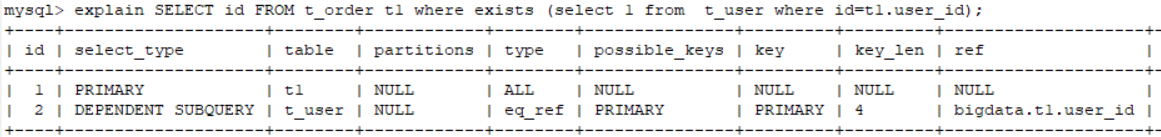
From中子查询

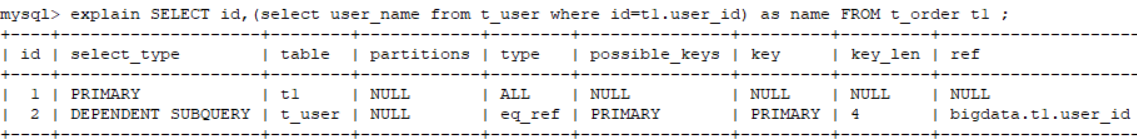


Explain的结果为derived，但这里是子查询

dependent subquery

对Outer Query的每一行对相关子查询进行一次评估





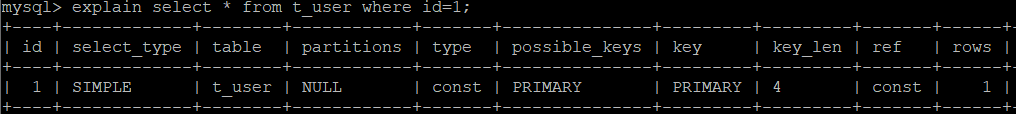
mysql嵌套子查询效率比较低，可以将其优化成连接查询。(Mysql5.6之后已自动优化?)。

TODO exists

可以将其优化成连接查询

## Constant Tables

索引是primary key or unique等值匹配作为搜索条件查表。type为const。这种查询花费的时间特别少，少到可以忽略，通过这种方式查询的表称之为常量表。



优化器在分析一个查询语句时，先首先执行常量表查询，然后把查询中涉及到该表的条件全部替换成常数，最后再分析其余表的查询成本，比方说这个查询语句：

1. **select** \* **from** table1 t1 join table2 t2 **ON** t1.col1 = table2.col2
2. **where** t1.primary\_key = 1;

这个查询中table1表相当于常量表，在分析table2表的查询成本前，就会执行对table1表的查询，并把查询中涉及table1表的条件都替换掉，也就是上边的语句会被转换成这样：

1. **select** table1表记录的各个字段的常量值, table2.\*
2. **from** table1 t1 JOIN t2 **ON** t1.col1的常量值 = t2.col2;

## SQL基本语法

### SQL执行顺序

标准的sql执行顺序

1. (8) **select** (9) **distinct** <select\_list>
2. (1) **from** <left\_table>
3. (3) <join\_type> join <right\_table>
4. (2)                 **on** <join\_condition>
5. (4) **where** <where\_condition>
6. (5) **group** **by** <group\_by\_list>
7. (6) **with** {**cube**|**rollup**}
8. (7) **having** <having\_condition>
9. (10) **order** **by** <order\_by\_list>
10. (11) limit <limit\_number>

### Count

用法

* count(\*)：同count(1)性能无差别。前者为SQL92的标准语法
* count(字段)：较count(\*)，多一个判空步骤，只统计非空的值。

不同引擎的优化

MyISAM：记录表的总行数。Select count直接返回，但不能又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。

[实现细节](https://mp.weixin.qq.com/s?__biz=MzAxMjEwMzQ5MA==&mid=2448887252&idx=2&sn=9aef0fe2a8ca66a5d6190e886212fc6e&chksm=8fb557f9b8c2deefcd7bd65322e9faa501f1410245f0d8fd8c7b112de19b789409c2e44e6b7e&mpshare=1&scene=1&srcid=&key=20a680e825f03f1e323846c1c9cbd180df53987f1929580267d88061712cda22d4d6315d461766f687efad0c666355614c26f88c06d51db0c5e840aa469ff9307b13661449faaef1cd684722aada5526&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62060739&lang=zh_CN&pass_ticket=U1Rycznd0MxcqdQDoQj52v1tXH53Lbffgbif%2FRFGmW%2F1SWy%2F7LEwVLTizUTqU4gb)

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。

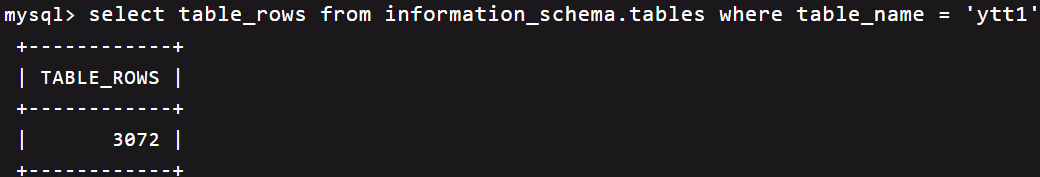
已经减了二级索引用短索引count

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

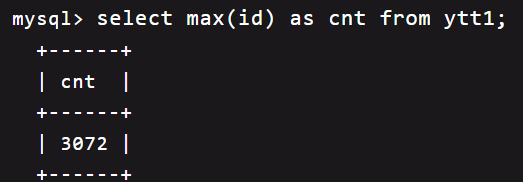
查数据字典



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

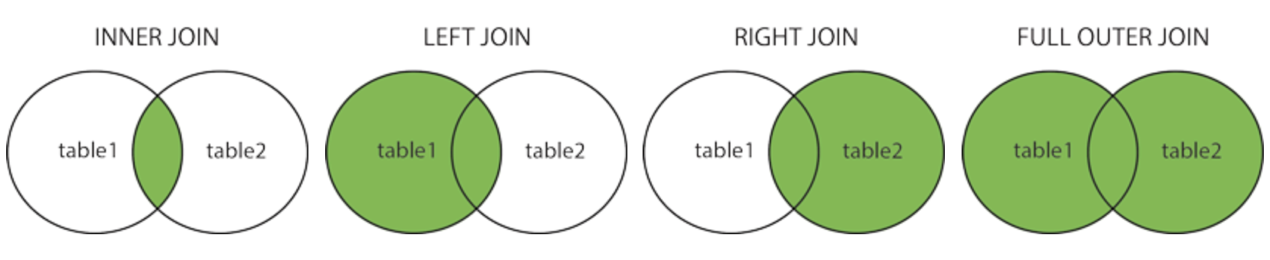
维护表的行数

SQL\_CALC\_FOUND\_ROWS

1. **SELECT** SQL\_CALC\_FOUND\_ROWS \* **FROM** **table** **WHERE** id > 100 LIMIT 10;
2. **SELECT** FOUND\_ROWS();

### Join基本语法

* [inner] join：等同join。
* Left [outer] join：
* full [outer] join：left join + right join。
* cross join：笛卡尔积。不允许有连接条件。返回m\*n条记录。
* union/union all：



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。

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

### 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

游标分页

2.数据缺失

[MySQL 游标分页与传统分页](https://github.com/x1ah/Blog/issues/15)

+---------------------------+

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

+---------------------------+

查询参数：limit=3&offset=0，得到123，删除一条数据（2）后得到

+-----------------------+

| 1 | 3 | 4 | 5 | 6 | 7 |

+-----------------------+

期望：在上次取出 1，2，3 的基础上，这次取出4，5，6

实际：取出的是 5，6，7，导致4这条记录没有被取到

3.数据重复

[分页场景（limit,offset）为什么会慢？](https://mp.weixin.qq.com/s?__biz=MzA4NjgxMjQ5Mg==&mid=2665763135&idx=1&sn=c34876d998ed8ee421ad619404692fc1&chksm=84d2031cb3a58a0a342a7775fd76359cfed7b3a436c090e7c10bf5b4ac466bc5c7b76e5faec7&mpshare=1&scene=1&srcid=&sharer_sharetime=1575869874282&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=1ed37e8b659a146561030d3fa9c216d5ef95903b1b14bea36dd2a35e30b0f266504650ceb8fbdc76b85ca14cc30801e9e724bdba1154e58285020fb21c52b7d619e136cc685c8560679c028b5ad9bb0d&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=ARA88UuVsOHhGDQwZmP6vS0%3D&pass_ticket=MIfYYQfosPGnnbARcGezUaxCRYCxWaZT1QLPdDjdlq4awG7Y%2BvZ0lr3vA2uldvPH)

### Union

Union all如何排序去重

# 第四章 性能优化（架构）

information\_schema/performance\_schema

## 4.1 优化策略

不同业务场景有不同的侧重，比如、读写比例、并发量、数据量和数据增长数据。

* 读写比例：根据具体业务读写比例，降低主库压力
* 高并发：集群
* 数据是否需要强一致🡺NOSQL
* 数据量：数据增长快。

1.针对数量大的表进行历史表分离（如交易流水表）。

2.做分表分库，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。说明：如果预计三年后的数据量根本达不到这个级别，不要在创建表时就分库分表

如何定位问题?

Top –c 查看负载/IO/内存使用情况

### 6.4.1 主从分离架构



多级复制架构

[MySql复制机制](https://mp.weixin.qq.com/s?__biz=MzA4NjgxMjQ5Mg==&mid=2665762450&idx=1&sn=f05aceebfca0ae9bf5f6a6ecaaaa0aa0&chksm=84d21cb1b3a595a70b098e2505dec54f3fedfb6be998c9a7c25dcd5a70c9f1246d8db8f9110b&mpshare=1&scene=1&srcid=&sharer_sharetime=1575772807647&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=1ce68c86504a20883d520b55ae8adaf568eb79dd90b418cba8a8a73c06f6c17e90bb865c2610cc3550ee6fe279a70bbea4c00aa27185d544c12b431699b3b38ffd4b672d524fa19293b1cd8315de5969&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=ARI%2BPC8zrlI3eC%2Beuz%2BaPFE%3D&pass_ticket=MIfYYQfosPGnnbARcGezUaxCRYCxWaZT1QLPdDjdlq4awG7Y%2BvZ0lr3vA2uldvPH)

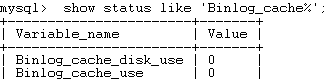
[主从复制配置](https://mp.weixin.qq.com/s?__biz=MzU0MzQ5MDA0Mw==&mid=2247487133&idx=2&sn=88f03b4fcb86cb479cb15416c443ab4b&chksm=fb0be409cc7c6d1ffa653227648473a1222366cfd75963229bc86d73379c3940754409dc071a&mpshare=1&scene=1&srcid=&sharer_sharetime=1576400466686&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=6653ae102befc15ed6f0854471d1b246a392a6edd9cf61c987295615be676afd5e823f182d53fbd1ec9a989c0f838344a46e84414d830eeb8e09cf0108d5ed27149bc533a5ca4c2dd2fdbc320f22f3b3&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=AcO8Lad2OcAbKu8wG5QE88E%3D&pass_ticket=cdl%2FHxApRPu1qQJWHKETt75mdG1tGUA9olgw9Jlydg4g4DKNEiPQBkHDeeRHqJY7)

#### 主从备份

Binlog

[Reference](https://www.cnblogs.com/martinzhang/p/3454358.html)

记录了所有的DDL和DML(除了数据查询语句)语句



使用场景

1.主从复制：MySQL Replication在Master端开启binlog，Master把它的二进制日志传递给slaves来达到master-slave数据一致的目的。

2.数据恢复：使用mysqlbinlog工具来恢复数据。

读写分离

缓存

### 6.4.2 分表分库

## 4.2 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。

### 6.5.1 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。
* not null：而不是 null 。null使索引更复杂。

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247491153&idx=1&sn=bf153add3e3c30010b649c36965bdcb4&chksm=9b943f49ace3b65fba13a530f395c218f6879396135e7be5327f48d8fc141b3bb04f694acc09&mpshare=1&scene=1&srcid=&key=02afbb533fbcad0a917048e2cbe0acc7b7c396158d4342ffeae2a21585ab3a3b9e77703f71df2e1ae0dfcc23dbedcf5d95b2adc944294f2003cc75edd47256b8622cb1515cd803257dd80f346e9238c0&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62060739&lang=zh_CN&pass_ticket=U1Rycznd0MxcqdQDoQj52v1tXH53Lbffgbif%2FRFGmW%2F1SWy%2F7LEwVLTizUTqU4gb)

* 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

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

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

### 6.5.2 数据类型

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 |

Oracle

Number(precision,scale) [Reference](https://www.cnblogs.com/oumyye/p/4448656.html)

precision：数字有效位数，整数+小数点部分，默认38

smallint/int/integer == number (38,0)

Unsigned：不允许负值，可使正数的上限提高一倍

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都都内容，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)。

日期

用字符串，不好计算且占用更多空间，比如2019-04-21 00:56需要16个字节。

|  |  |  |
| --- | --- | --- |
| **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 |

时区：bigint和datetime存储与时区无关,国际化时可存储UTC时间，在view层转成对应时区时间。

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);

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)

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

### 6.5.3 范式

[Reference](http://blog.csdn.net/dongnan591172113/article/details/7910988)

范式和反范式

范式需要联表，代价昂贵。还可能使一些索引策略无效，比如覆盖索引。

### 6.5.4 Integrity Constraints

完整性约束。约束保证数据的正确性，可靠性和一致性。包括域约束，基本表约束等。

[Reference](https://www.indiastudychannel.com/resources/150128-Integrity-Constraints-DBMS.aspx)

Domain Integrity

域完整性

* CHECK(检查)约束：用于限制列中值得范围
* DEFAULT(默认值)约束
* NOT NULL(非空)约束

 Entity integrity constraints

实体约束，表约束。保证Table的记录都是唯一的 。

* PRIMARY KEY(主键)约束：唯一识别每一条记录的标志，可以有多列共同组成
* IDENTITY(自增)约束：列值自增，一般使用此属性设置的列作为主键
* UNIQUE(唯一)约束

Referential Integrity

参照完整性。保证关联table的数据是一致的。 T1(id,name),T2(id,t1\_id),T2的外键t1\_id,在T1里必须是可用的。

* FOREIGN KEY(外键)

### 4.4.4 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)

## 4.3 索引优化

### 4.4.1 索引是什么？

在数据之外，DB维护着满足特定查找算法的数据结构，这样就可以在这些数据结构上实现高级查找算法。索引优化了查询。本质是空间换时间。

索引的优点

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

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。

### 4.4.2 查询算法

Hash

适合等值，即where index=’xx’，1.范围查找引起全表扫描 2.不能排序。

B-Tree

[Reference](https://blog.csdn.net/Roy_70/article/details/76581028)

即Balanced Tree, 有别于二叉查找树（Binary Search Tree）。B+Tree实现其索引结构。

二叉搜索树的查询时间复杂度是O(log(n))，从算法逻辑上来讲，无论查找速度还是比较次数都是最小的，但在数据库的索引通常十分庞大，需要以文件形式存储，而磁盘IO的存取次数就是评价一个数据库索引优劣的关键性指标。

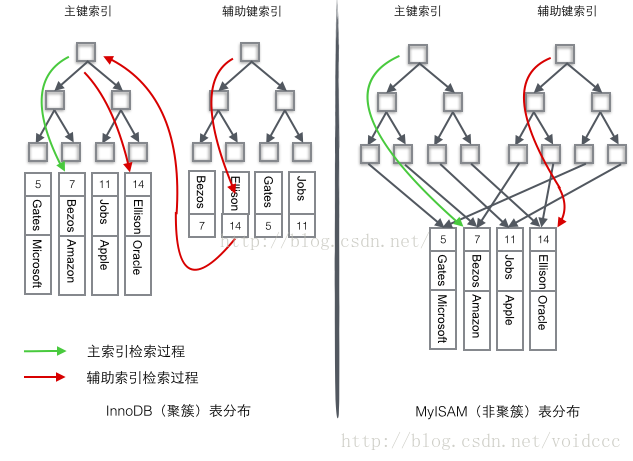
B-Tree优化了磁盘IO

### 4.4.3 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

### 4.4.5 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)

[Reference](https://yq.aliyun.com/articles/5556)

* 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。（🟊🟊🟊）

举几个开发中场景的案例

#### 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)

force index

有时MySQL优化器采取它认为合适的索引来检索sql语句，但可能不是我们想要的。可用force index强制优化器使用我们制定的索引。

straight\_join

作用：强制连接顺序，straight\_join左边的表名就是驱动表，右边则是被驱动表。

有些场景需要选择另个表作为驱动表，比如有group by、order by等「Using filesort」、「Using temporary」时。

注意：使用straight\_join的前提是该查询是内连接。其他join不推荐使用，否则可能造成查询结果不准确。

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。

## 4.4 SQL优化

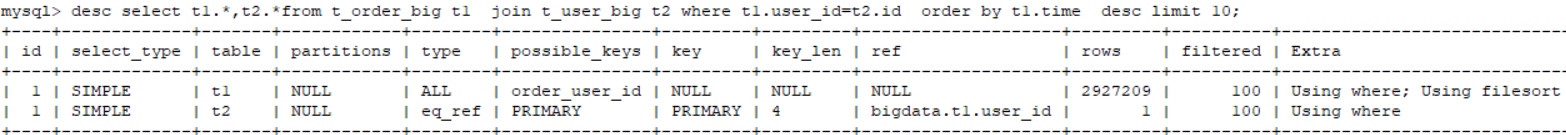
SQL的生命周期每个环节都有足够的优化空间，优化的本质是减少SQL对资源的消耗和依赖

### 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;

1.substr对字段计算导致索引失效

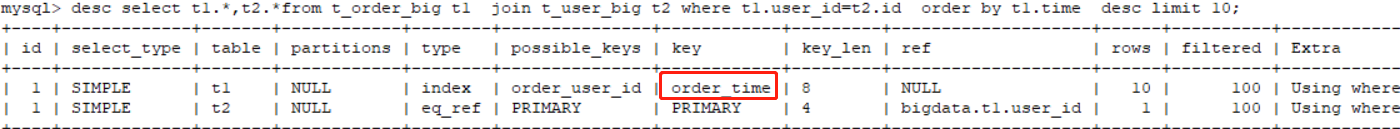
2. 漏加索引



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

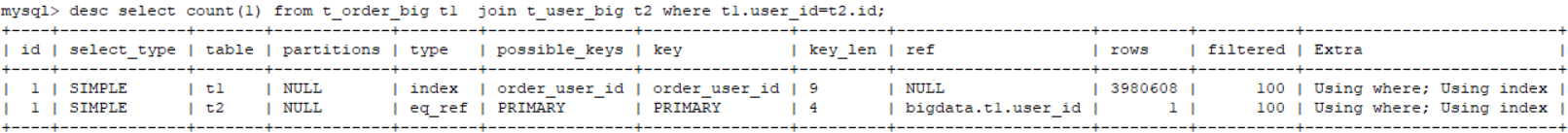


t1 type=all变成type=index，并按time的索引排序，t1有10次回表，但比filesort性能好的多。

4.优化子查询，字典表为dependent subQuery

5.个别搜索条件由完全模糊改成右模糊

优化count



时间: 7.696s

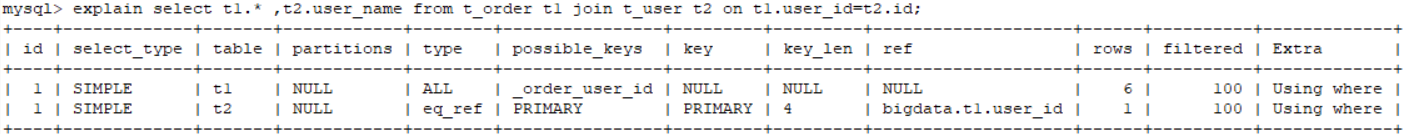
Web开发页面初始进入页面常常没有过滤条件

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字段合并到一张表

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

### Case1 Where过滤

* Using Where

警惕全表扫描，可能未建索引或索引失效。

* 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。

* 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。

### Case2 Order by

假设表结构和SQL语句如下：

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

#### 3.3.1 索引排序

B tree索引，本身就是有序的，无需额外排序操作。

Case1 主键索引

1. **Select** \* **from** **table** **order** **by** id limit M,N;

id列为主键索引，索引本生就是排序好的，使用索引结构即可。

limit是在结果集中返回需要的数据，它尽快的返回需要的行而不用管剩下的行，

Case2 辅助索引

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

索引<actor\_name,prod\_time>



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

可用索引的排序

复合索引<key\_part1, key\_part2>

* Case1 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

* Case2 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

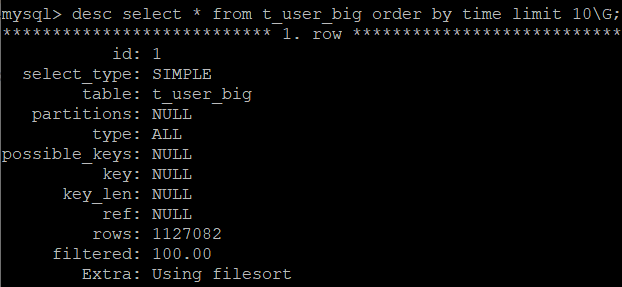
不能使用索引排序

* 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

#### 3.3.2 文件排序

无法用索引排序成为文件排序。Filesort先在内存排序, optimizer预先分配固定内存(sort\_buffer\_size). 结果集过大时使用temporary disk files.



time无索引。Explain的extra可见Filesort。filesort并不就是文件排序，也可能是内存排序，是否用文件排序要看sort\_buffer是否能容下需要排序的键值对(id,col2)。

优化排序

一次排序需要两次IO，一次是捞<id,col2>,第二次是捞字段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。

优化排序：若排序元组小于max\_length\_for\_sort\_data（默认1024Byte），能用优化排序。放入sort buffer (col1,col2,col3)。由于buffer中包含查询的所有字段，排序后可直接返回，无需回表。减少了第二次IO。

此外，对第二次IO优化：捞之前先将id排序，并放入缓冲区（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。。

Trace可见排序模式，选择哪种排序模式，与max\_length\_for\_sort\_data属性有关，：

* sort\_key, rowid：若超过，sort\_buffer存主键+排序键
* sort\_key, additional\_fields：若不超过，sort\_buffer存所有列
* sort\_key, packed\_additional\_fields：如果查询的记录太多，对可变列进行压缩。

排序算法

如果sort列没有索引呢？若把sort列全部排序好在返回需要的行，代价过高。 必须用一种合适的算法，这里就引入堆排序和优先队列。

基于参与排序的数据量的不同，可以选择不同的排序算法

* 优先队列 针对order by limit 返回少量行
* 快速排序：sort\_buffer可存放所有满足条件的(id,key)对
* 外部排序（归并排序）

##### 优先队列排序

Priority Queue。如果排序的结果集很小，小于内存，那么会使用优先级队列进行堆排序；

针对Order by+limit，采用堆排序实现。虽然所有元素参与排序，但只要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。

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.

堆排序

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。

##### 快速排序

如果sort\_buffer可存放所有满足条件的(id,col2)对，则进行qsort(快速排序) 在内存中排好序

##### 外部排序

sort\_buffer不能存下所有满足条件的键值对，则进行外部排序

排序过程

1)获取满足where条件的记录

2)将记录的主键+排序键放入sort buffer

3)每次buffer满后，用qsort排好序，作为一个block写到临时文件(temporary disk files)。

4)产生的临时文件，用归并排序算法，保证临时文件中记录有序

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。

排序优化小洁

* 建立索引, 避免排序,
* Increase the sort\_buffer\_size variable value
* Increase the read\_rnd\_buffer\_size so that more rows are read at a time

[Order by Optimization](https://dev.mysql.com/doc/refman/8.0/en/order-by-optimization.html)

[order by的排序算法竟然是这样的！](https://mp.weixin.qq.com/s?__biz=MzAxMjEwMzQ5MA==&mid=2448888388&idx=2&sn=a90512777f6a16169d71257d3cb87983&chksm=8fb54869b8c2c17ffa8c0bbe7d758cc119a27c77e6adc313746204ef7dddd3e1ba51ff2e9c9d&mpshare=1&scene=1&srcid=0420cAxrIzoxgkrNiIZ3ml2R&sharer_sharetime=1587313128222&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=c7269b485e093f7b70ba950e00a616011435cd53ff04e93a71f5c654b63dc98cc650985df3931eacbaf59064e046f753429013e91c3e197e2ca8ea7fd06e43d949654d79ac02af5b40e0ffe9e439ea81&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AfAHuJwLAk4uNUE%2ByN9Nly8%3D&pass_ticket=zLWDTjdsv9KsBGl19PRdRNKjngb7fO6WPxhRxwczjdSVP0Ore6DN%2BLkZhZm5R%2F2v)

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。

Limit

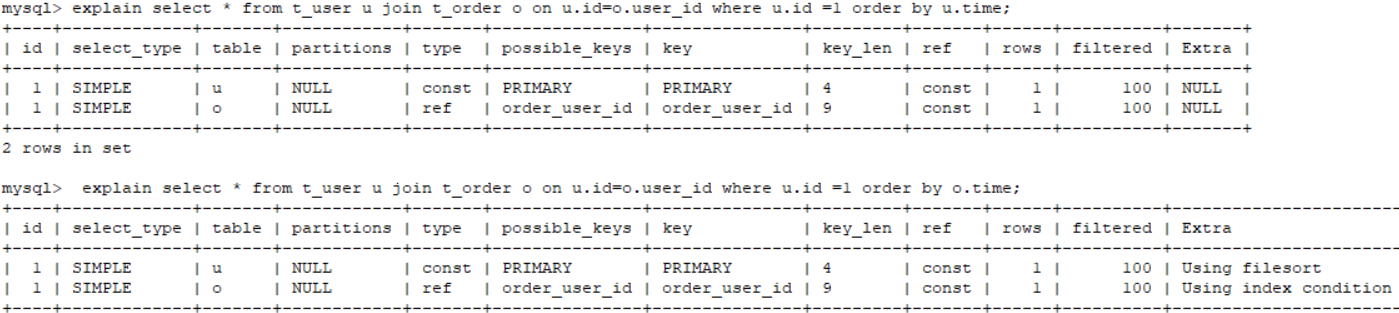
* Order by + limit <row\_count>

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

* Case3 连表

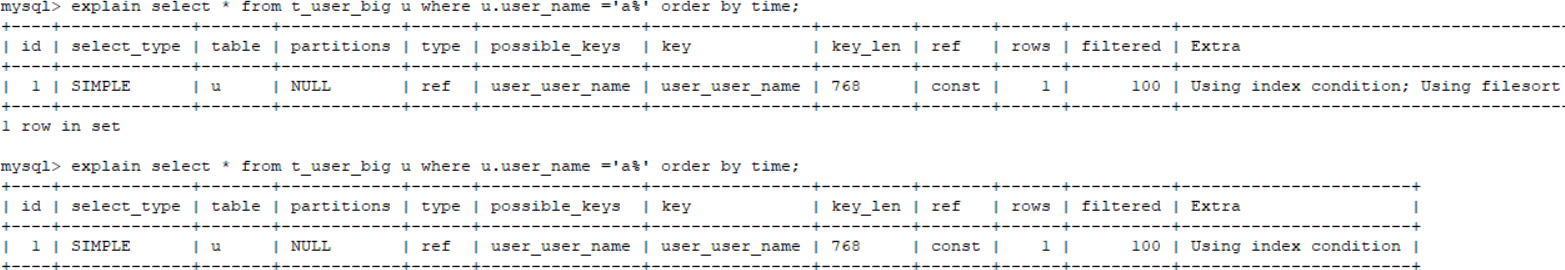
first table的join type是const.

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?



### Case3 超级大表优化

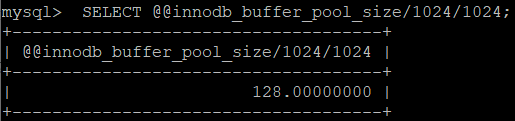
单表500万行：阿里《Java开发手册》提出单表行数超过500万行或者单表容量超过2GB，才推荐进行分库分表，这个数值是如何评估出来的呢？

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\_pool\_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，如果没有标准，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。

innodb\_buffer\_pool\_size

默认128M, set global innodb\_buffer\_pool\_size=xxx;



* 建议设定为服务器内存总大小的50%~70%。同时部署其他应用应适应减小。
* innodb\_buffer\_pool\_instances：内存很大，可以把buffer pool划分成多个instances。chunk\_size=innodb\_buffer\_pool\_size/innodb\_buffer\_pool\_instances

超级大表优化

解决：

* 数据迁移
* 单表优化
* 读写分离
* 缓存
* 表分区
* 拆分：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)

# 附录1 Explain

[Reference](https://dev.mysql.com/doc/refman/5.7/en/explain-output.html#explain-output-columns)

explain执行计划就是Mysql如何执行一条Sql语句,包括Sql查询的顺序、是否使用索引、以及使用的索引信息等内容。



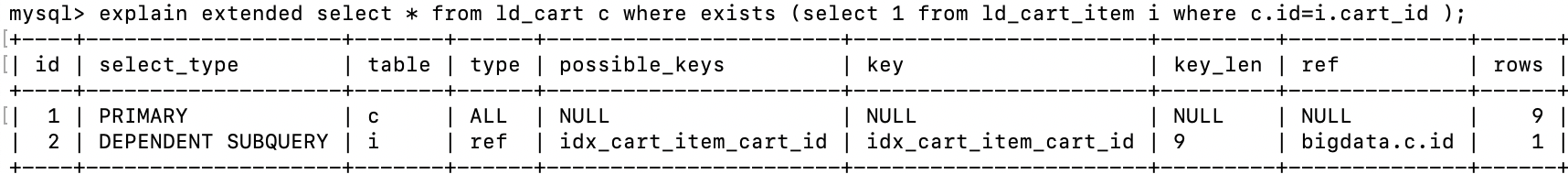
* id：The select identifier。id越大执行优先级越高。若相同，则从上至下。
* select\_type：simple/ primary/derived等。
* type：describe how tables are joined.
* possible\_keys : The possible indexes to choose。
* key：The index actually chosen。
* ref：与索引比较的列。The columns compared to the index.
* rows: the number of rows MySQL believes it must examine to execute the query
* filtered： Percentage of rows filtered by table condition

### 6.8.1 select\_type

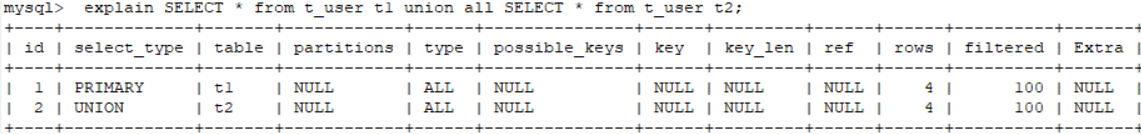
* Simple: Simple SELECT (not using UNION or subqueries)。
* primary：子查询最外层的select。
* derived：派生表，from子查询语句。



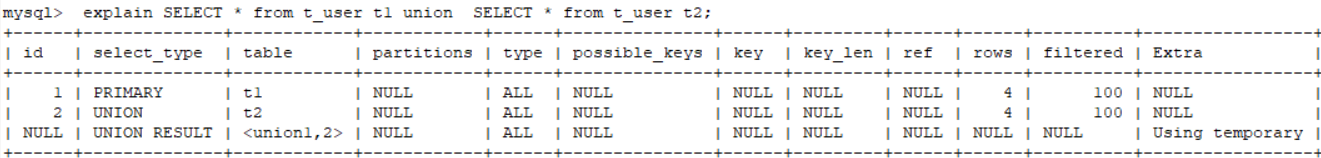
* Subquery：First SELECT in subquery
* dependent subquery：子查询依赖于外层。



* MATERIALIZED
* Union：Second or later SELECT statement in a UNION



* unionresult



### 6.8.2 Type

describes how tables are joined.

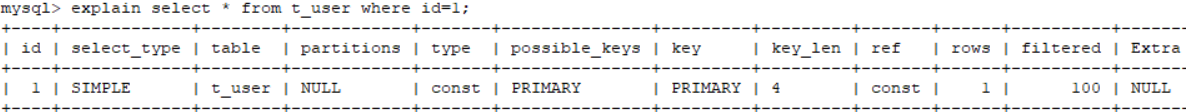
all < index < range < ref < eq\_ref < const < system < NULL

* System：查询的表中一条记录没有，或者只有一条记录。(基于统计数据，no InnoDB)
* Const：索引是primary key or unique.
* eq\_ref：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

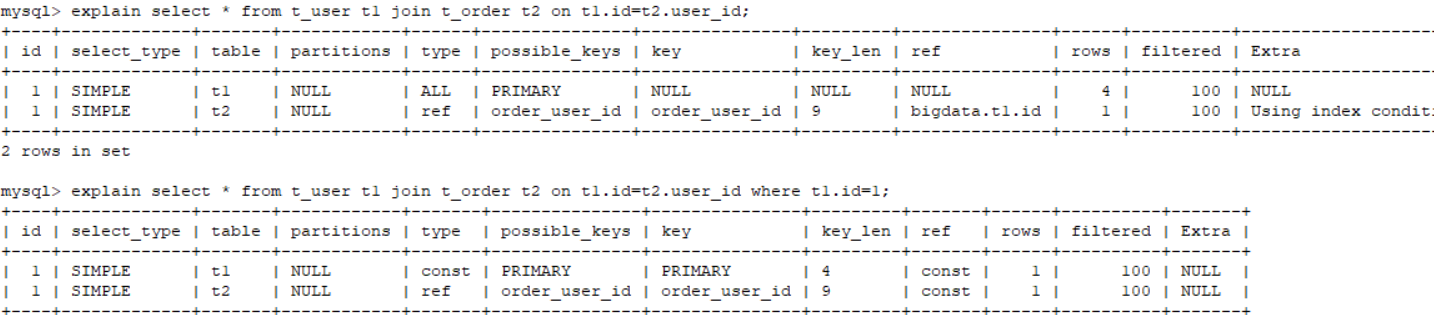
### 6.8.3 Ref

与索引比较的列。The index actually chosen.

1．Constat table



2.连表



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

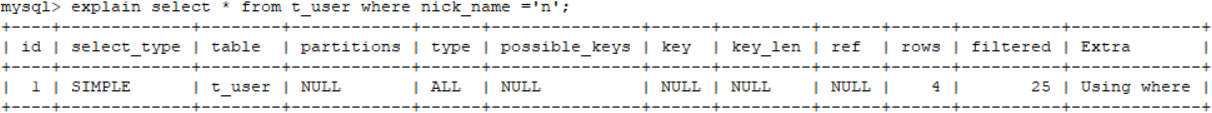
语句2 mysql能对查询部分优化转成变量，比如id在where子句，将查询转成常量，然后就可高效将表从联表执行中移除。

### 6.8.4 Extra

Additional information about how MySQL resolves the query。

#### Using where

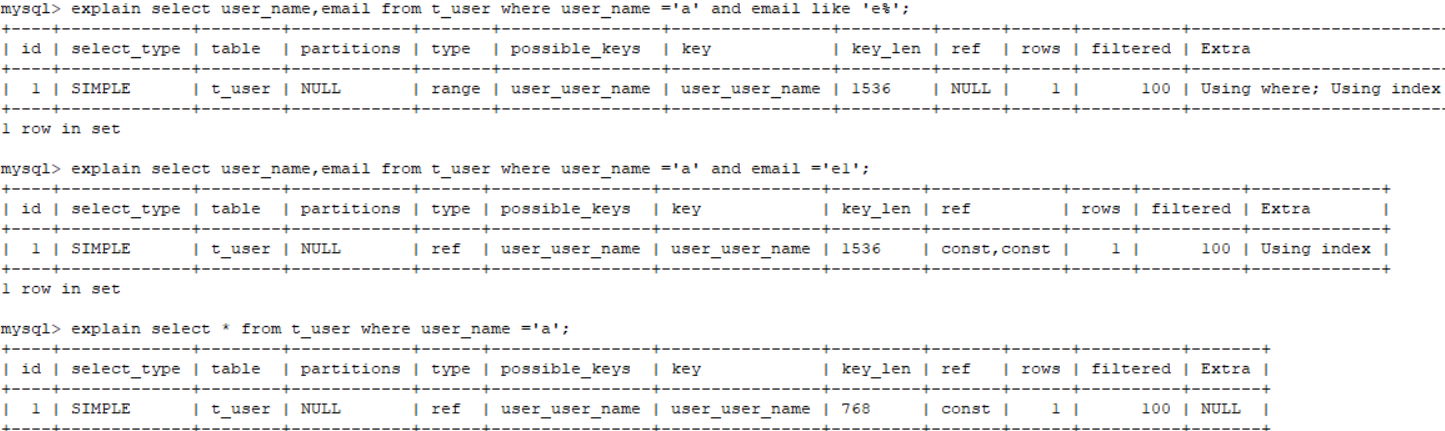
读出所有数据(index或data)在mysql Server层过滤，



Type ALL/index

#### Using index

在mysql server层过滤索引，注意Select的column，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

#### Using where;Using index

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

#### Using index condition

[Index Condition Pushdown Optimization](https://dev.mysql.com/doc/refman/5.6/en/index-condition-pushdown-optimization.html)

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

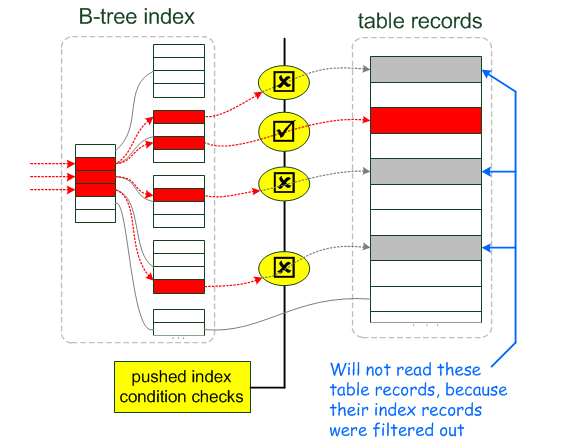
* 可用于range, ref, eq\_ref, and ref\_or\_null when need access full-rows.
* 只能用于二级索引。聚簇索引完整record已read into the InnoDB buffer.不会减少IO。
* 用于select字段过滤的索引不覆盖情况(否则不用回表读数据)

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';

#### Using union

Using intersect(交集)：两个索引同时检索结果再求并集。

ld\_cart(summary key,member\_id key)

1. mysql> explain **select** \* **from** ld\_cart **where** summary='kchC' or member\_id=13\G;
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. id: 1
4. select\_type: SIMPLE
5. **table**: ld\_cart
6. type: index\_merge
7. possible\_keys: idx\_ld\_order\_summary,idx\_ld\_cart\_member\_id
8. **key**: idx\_ld\_order\_summary,idx\_ld\_cart\_member\_id
9. key\_len: 768,9
10. ref: NULL
11. **rows**: 2
12. Extra: Using **union**(idx\_summary,idx\_member\_id); Using **where**

同时扫描summary和member\_id两个列索引，并将结果合并。

Using intersect?

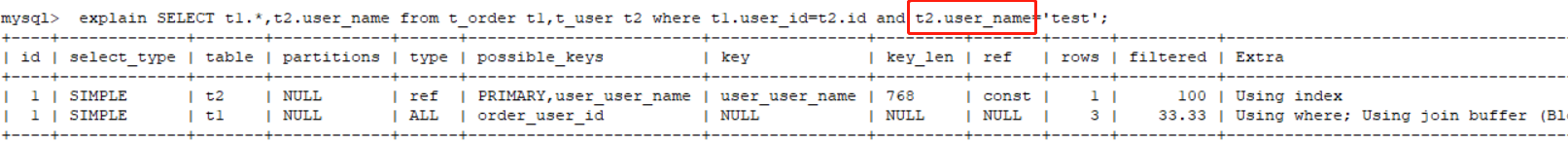
[MySQL数据库查询记录时是否每次只能使用一个索引](https://www.jianshu.com/p/34194ea5a4a3)

[mysql只会使用到一个索引的原因](https://blog.csdn.net/weixin_41926301/article/details/94509637)

#### 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.



#### 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。

#### Using filesort

MySQL must do an extra pass to find out how to retrieve the rows in sorted order. The sort is done by **going through all rows** according to the join type and storing the sort key and pointer to the row for all rows that match the WHERE clause. The keys then are sorted and the rows are retrieved in sorted order.

mysql中无法用索引完成的排序成为文件排序

### 6.8.5 Key\_len

The length of the chosen key. ddl中定义的长度为字符长度，单位为byte。

占用字节和数据类型、字符编码、是否为NULL有关，计算规则

* 字符：varchar存储变长字符的长度需要2字节，char无需额外的字节。
* 整型：占用字节即为索引长度，比如int为4
* Null：占用1个字节

Case1：char(20) utf8+not null。key\_len =20\*3（utf8 3byte）=60byte。

Case2：varchar(20)+utf8+not null。key\_len=20\*4（utf8mb4 4byte）+1 (Null标识) +vlen=83。

。

联合索引

### 6.8.6 Rows& filtered

the number of rows MySQL believes it must examine to execute the query。

比如select \* from t where a = 1 and b = 2;

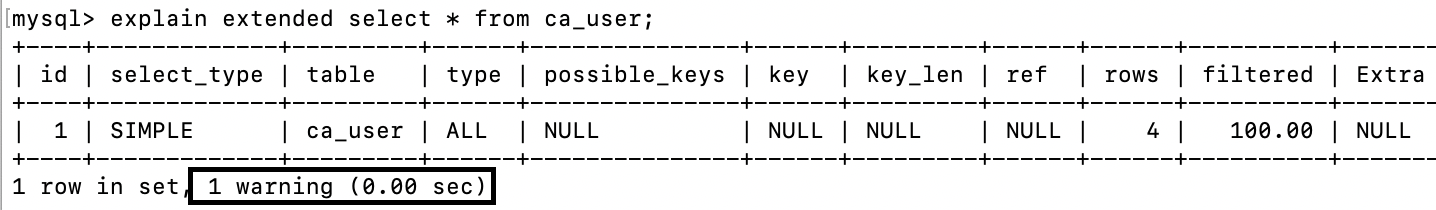
t有100条记录，a字段有索引，b没有索引。a= 1有20条， a =1 and b=2有5条。

explain结果中的rows是20，这20条记录引擎必须逐行检查是否满足where条件。

### explain extended

查看mysql优化器优化后的查询语句

explain extended sql



执行show warnings;

### explain partitions

用于分区表的EXPLAIN

### Format Json

[Reference](http://www.unofficialmysqlguide.com/explain.html)

it provides a query cost

Sql: explain **select**  **name** **from** user\_big limit 3

1. {
2. "query\_block": {
3. "select\_id": 1,
4. "cost\_info": {"query\_cost": "53.80"},  #This query cost 53.80 units
5. "table": {
6. "table\_name": "user\_big",
7. "access\_type": "index",
8. "key": "idx\_user\_name",
9. "used\_key\_parts": ["name"],
10. "key\_length": "768",
11. "rows\_examined\_per\_scan": 239,
12. "rows\_produced\_per\_join": 3923050,
13. "filtered": "100.00",
14. "using\_index": true,
15. "cost\_info": {
16. "read\_cost": "1.00",
17. "eval\_cost": "0.20",
18. "prefix\_cost": "1.20",
19. "data\_read\_per\_join": "3k"
20. },
21. "used\_columns": ["name"]
22. }
23. }
24. }

rows\_examined\_per\_scan

scan时检测行数，加了索引大大减少检测数

read\_cost：IO成本和检测rows × (1 - filter)条记录的CPU成本

eval\_cost：检测 rows × filter条记录的成本。

prefix\_cost：单独查询的成本，等于read\_cost + eval\_cost

data\_read\_per\_join：此次查询中需要读取的数据量

## 附录2 性能分析工具

### optimizer trace

[optimizer trace 表的神奇功效](http://xiaoce.huangqingliang.com/mysql/18.%E7%A5%9E%E5%85%B5%E5%88%A9%E5%99%A8%20%E2%80%94%E2%80%94%20optimizer%20trace%20%E7%9A%84%E7%A5%9E%E5%99%A8%E5%8A%9F%E6%95%88.html)

[手把手教你认识OPTIMIZER\_TRACE](http://www.woqutech.com/docs_info.php?id=456)

什么是成本？

MySQL执行一个查询可以有不同的执行方案，它会选择其中成本最低。

成本组成

I/O成本：查询表记录时，磁盘中数据或者索引数据加载到内存过程花费。

CPU成本：读取以及检测记录是否满足搜索条件、对结果集进行排序等这些操作的花费。

成本常数

InnoDB的页是磁盘和内存之间交互的基本单位，读取一个页面的成本默认是1.0

读取以及检测一条记录是否符合搜索条件的成本默认是0.2。

打开optimizer\_trace

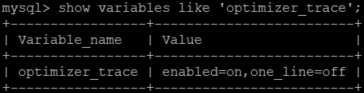
1. session级别：set session optimizer\_trace='enabled=on';

2.<执行SQL>

3.查询trace表 select \* from information\_schema.optimizer\_trace;

1. QUERY: SQL
2. TRACE: {
3. "steps": [
4. {"join\_preparation": {SQL的准备阶段，sql被格式化 }},
5. {"join\_optimization": { SQL优化阶段}},
6. {"join\_explain": { SQL执行阶段}}
7. ]
8. }

查看



one\_line的值是控制输出格式的，如果为on那么所有输出都将在一行中展示

#### join\_optimization

基于成本的优化主要集中在optimize阶段。优化步骤

1. 根据搜索条件，找出所有可能使用的索引

2. 计算全表扫描的代价

3. 计算使用不同索引执行查询的代价

4. 对比各种执行方案的代价，找出成本最低的那一个

1. "join\_optimization": {
2. "select#": 1,
3. "steps": [
4. { "condition\_processing":{}} #优化where条件
5. { "substitute\_generated\_columns":{}} #替换虚拟生成列
6. {"table\_dependencies": []},  #梳理表的依赖关系
7. {"rows\_estimation": []},  #估算表行数和扫描的代价
8. {"considered\_execution\_plans": []},
9. #对DISTINCT、GROUP BY、ORDER BY等语句进行优化
10. {"clause\_processing": {}}
11. {"attaching\_conditions\_to\_tables": {}}
12. {"refine\_plan": {}} #优化后的执行计划
13. ]
14. }

Select \* from t\_user where name like ‘test%’;

1. first\_read = **true**;  //是否是第一次读取
2. **while** (**true**) {
3. **if** (first\_read) {
4. first\_read = **false**;
5. err = index\_read(...); //调用存储引擎接口，定位到第一条符合条件的记录;
6. } **else** {
7. err = index\_next(...); //调用存储引擎接口，读取下一条记录
8. }
9. **if** (err = 存储引擎的查询完毕信息) {
10. **break**;  //结束查询
11. }
12. **if** (WHERE条件) {
13. //回表组装完整记录
14. send\_data();    //将该记录发送给客户端;
15. } **else** {
16. //跳过本记录
17. }
18. }

server层和存储引擎层的交互是以记录为单位。

每次从二级索引中取出一条符合条件的记录就进行回表一次,并非一次性把所有符合条件的二级索引都取出来之后再统一进行回表操作

##### rows\_estimation

预估全表扫描的代价。

1. "rows\_estimation": [
2. {
3. "table": "`order\_big` `t1`",
4. "table\_scan": {
5. "rows": 3880795,
6. "cost": 53056
7. },
8. "range\_analysis": {  #加了where过滤
9. "potential\_range\_indexes":{}
10. "analyzing\_range\_alternatives": {}
11. "chosen\_range\_access\_summary": {}
12. }
13. }
14. ]

* table\_scan：全表扫描的花销，将聚簇索引页面加载到内存中
* 红色{}内potential\_range\_indexes，analyzing\_range\_alternatives指单表的访问方法

如何计算全表扫描的花销？

查询成本=I/O成本+CPU成本，计算全表扫描的代价需要两个信息：

* 聚簇索引占用的页面数
* 该表中的记录数

potential\_range\_indexes

过滤可能使用的索引

1. [
2. {
3. "index": "PRIMARY",
4. "usable": true,
5. "key\_parts": [ "id"]
6. },
7. {
8. "index": "idx\_user\_name",
9. "usable": false,
10. "cause": "not\_applicable"
11. }
12. ]

访问表t\_user可能使用索引primary key id，或者二级索引idx\_user\_name

analyzing\_range\_alternatives

分析各种可能使用的索引的成本

1. "analyzing\_range\_alternatives": {
2. "range\_scan\_alternatives": [
3. {
4. "index": "PRIMARY",
5. "ranges": ["1 **<** **id**"],
6. "index\_dives\_for\_eq\_ranges": true, # 是否使用index dive
7. "rowid\_ordered": true,  # 使用该索引获取的记录是否按照主键排序
8. "using\_mrr": false,   # 是否使用mrr
9. "index\_only": false,  # 是否是索引覆盖访问
10. "rows": 2,            # 使用该索引获取的记录条数
11. "cost": 2.41,          # 使用该索引的成本
12. "chosen": true         # 是否选择该索引
13. },
14. {
15. "index": "idx\_user\_email",
16. "ranges": ["a **<** **email**"],
17. "index\_dives\_for\_eq\_ranges": true,
18. "rowid\_ordered": false,
19. "using\_mrr": false,
20. "index\_only": false,
21. "rows": 1,
22. "cost": 2.21,
23. "chosen": true
24. }
25. ]
26. }

##### considered\_execution\_plans

分析各种可能的执行计划。单表查询同rows\_estimation，对多表查询，这个过程里会写明各种不同的连接方式所对应的成本

1. "considered\_execution\_plans": [
2. {  # 执行计划1  user 连 order
3. "table": "`t\_user` `u`",
4. "best\_access\_path": {
5. "considered\_access\_paths": [
6. {
7. "rows\_to\_scan": 1,
8. "access\_type": "range",
9. "range\_details": {
10. "used\_index": "idx\_user\_email"
11. },
12. "resulting\_rows": 0.6667,
13. "cost": 2.41,
14. "chosen": true
15. }
16. ]
17. },
18. "condition\_filtering\_pct": 100,
19. "rows\_for\_plan": 0.6667,
20. "cost\_for\_plan": 2.41,
21. "rest\_of\_plan": [
22. {
23. "plan\_prefix": ["`t\_user` `u`"],
24. "table": "`t\_order\_big` `o`",
25. "best\_access\_path": {
26. "considered\_access\_paths": [
27. {
28. "access\_type": "ref",
29. "index": "order\_user\_id",
30. "rows": 2.0071,
31. "cost": 1.6057,
32. "chosen": true
33. },
34. {
35. "access\_type": "scan",
36. "cost": 829215,
37. "rows": 3880795,
38. "chosen": false,
39. "cause": "cost"
40. }
41. ]
42. },
43. "condition\_filtering\_pct": 100,
44. "rows\_for\_plan": 1.3381,
45. "cost\_for\_plan": 4.0157,
46. "chosen": true
47. }
48. ]
49. },
50. { # 执行计划2  order 连 user
51. "table": "`t\_order\_big` `o`",
52. ……
53. }
54. ]

access\_type

condition\_filtering\_pct：类似于explain中的filtered列，这是一个估算值,100指条件过滤率100%，指的是 #这里与上一个表进行行过滤的行数

access method

同一个查询语句可能可以使用多种不同的访问方法来执行：

全表扫描

即把表的每一行记录都扫一遍，把符合搜索条件的记录加入到结果集就完了。

使用索引查询

* 针对primary key或unique key的等值查询
* 针对普通二级索引的等值查询
* 针对索引列的范围查询
* 直接扫描整个索引

连接查询的成本

对于两表连接查询来说，它的查询成本由下边两个部分构成

* 单次查询驱动表的成本
* 多次查询被驱动表的成本（次数取决于对驱动表查询的结果集中有多少条记录）

在连接查询中的过滤条件可以分成两种：

涉及单表的条件

涉及两表的条件：关联条件。

### 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

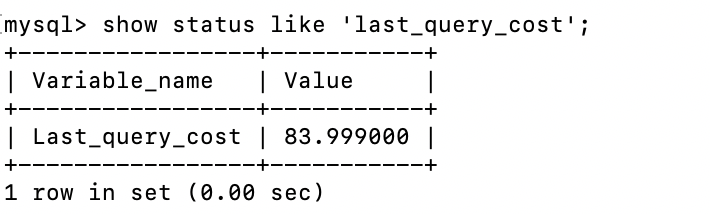
[Reference](https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html)

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/)

### last\_query\_cost



83是数据页个数

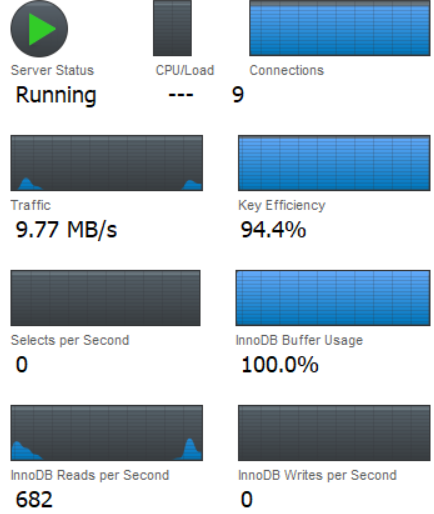
### TimeOut

等待TimeOut： show variables like 'innodb\_lock\_wait\_timeout';

### innodb status

show engine innodb status

MySQL Workbench



Key Efficiency

代表有多少指从the index caches held within MySQL's memory获取.

## 附录3 performance\_schema

### 数据大小

select data\_length,index\_length from information\_schema.tables where table\_schema='bigdata' and table\_name = 'ld\_cart';

默认单位是byte，优化显示

select table\_name,concat(round(sum(data\_length/1024/1024),2),'MB') as data\_length\_MB,concat(round(sum(index\_length/1024/1024),2),'MB') as index\_length\_MB from information\_schema.tables where table\_schema='bigdata' and table\_name = 't\_user\_big';

### 统计

DDL统计

Innodb\_rows\_read/inserted/deleted/updated

Innodb\_data\_read/writes

show status like 'innodb\_log\_waits';

### 事务

查看InnoDB存储引擎 系统级的隔离级别和会话级的隔离级别

1. **select** @@**global**.tx\_isolation,@@tx\_isolation;

语法

1. start **transaction** | **begin**；
2. DML...
3. **commit**;

begin会挂起autocommit，commit后恢复原来的值。set autocommit = 0也可开启。

提交即commit

Autocommit

自动提交事务，对当前connect生效，相当于开启事务。

1. --查看
2. show variables like 'autocommit' / **select** @@autocommit

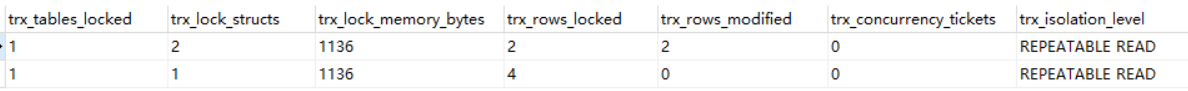
set autocommit = 0 关闭自动提交。

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

## 附录4 监控

mysql提供的指标很多，主要监控4大指标

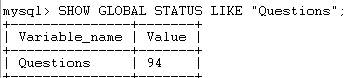
[Reference](http://blog.oneapm.com/apm-tech/754.html)

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\_%'

### 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';

mysql workBench可查看

### 查询执行性能

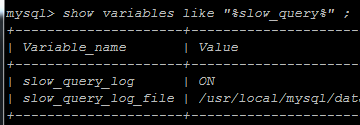
### 慢查询日志

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

show variables like 'log%';

show query

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

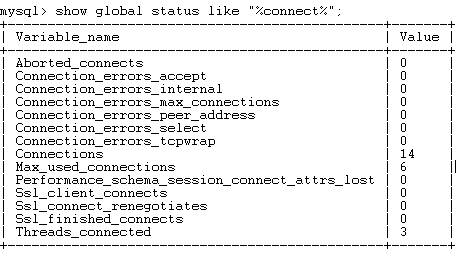
### 连接情况

Connections

show full processlist;



每一条代表一个Thread，一个Tcp连接。



Aborted\_connects 尝试已经失败的MySQL服务器的连接的次数。

Threads\_connected：当前连接数。只有数量值;

Threads\_running：正在运行线程数。

connections：试图连接mySQL服务器的次数

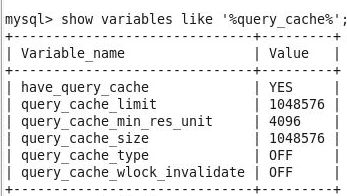
Max\_used\_connections

Uptime 服务器工作了多少秒。

### 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

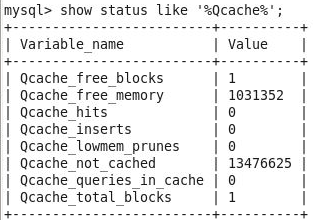
永久的修改配置文件my.cnf ,添加下面的配置即可。

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';

技巧

[细聊一下我的3条面试标准](https://mp.weixin.qq.com/s?__biz=MzU0MzQ5MDA0Mw==&mid=2247486086&idx=1&sn=f61eb2f290eef74376af41e8fd18ab40&chksm=fb0be012cc7c6904a4e8d42a83d364fdb1dff34a2209907c79c52858962da41ff4e40946a085&mpshare=1&scene=1&srcid=&sharer_sharetime=1576401194035&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=b2910d07836452c9accced79424e8c730a134251f5fdf6a55273e34cd55afc454a08b102d317616b5d2b5218c3d4e206401d863e1686dc59bac8f1a1938ac9bb7c87f996cecbe805c565484eff352ba0&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=Aau2y72gi09yc1cA55ZJ4xw%3D&pass_ticket=cdl%2FHxApRPu1qQJWHKETt75mdG1tGUA9olgw9Jlydg4g4DKNEiPQBkHDeeRHqJY7)

[如何在面试中介绍自己的项目经验](https://mp.weixin.qq.com/s?__biz=MzU0MzQ5MDA0Mw==&mid=2247486199&idx=1&sn=6fb927a8b857a6ac273c3a90836f2f49&chksm=fb0be063cc7c697543cbc54e8645b59c7dce393e8c18045100d903915fee7e94195ccf4ac9c1&mpshare=1&scene=1&srcid=&sharer_sharetime=1576401095140&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=b2910d07836452c90146cc63a4127c03490381ecec75f5b2caf4e9c087646a9e60113b73cd427f518c1dbd6504690916c2bd691db4a6a0d73a0fce292f65fec33c97fb56dd02356d4f15bc6fab7a7ddf&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=ASqju8hn5tpvjLwa1f8F2l0%3D&pass_ticket=cdl%2FHxApRPu1qQJWHKETt75mdG1tGUA9olgw9Jlydg4g4DKNEiPQBkHDeeRHqJY7)