MySQL高级内容

本文是Java开发工程师写出高性能SQL语句的知识。

随着淘宝去IOE(去除IBM小型机、Oracle数据库及EMC存储设备)化的推进，

MySQL 数据库在当前的互联网应用中变得越来越重要，使得Oracle在互联网企业中越来越少，而Mysql越来越多。如何写出性能参数调优，高性能SQL数据查询语句。

性能高效可靠稳定简洁

面试常问：

1、对SQL调优谈谈你的看法和理解？

2、什么是索引，什么情况下该建索引，什么情况下不该建索引？

3、索引建立完成之后是如何保证不失效的？

4、mysql优化你是怎么干的？

有四种方法，ecplain是一种

定个小目标，往数据库里面插入一个亿。

# MySQL架构介绍

## 1.1、Mysql简介

### 1.1.1、概述

MySQL是一个关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于Oracle公司。

MySQL是一种关联数据库管理系统，将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。

Mysql是开源的，所以你不需要支付额外的费用。

Mysql是可以定制的，采用了GPL协议，你可以修改源码来开发自己的Mysql系统。

Mysql支持大型的数据库。可以处理拥有上千万条记录的大型数据库。

MySQL使用标准的SQL数据语言形式。

Mysql可以允许于多个系统上，并且支持多种语言。这些编程语言包括C、C++、Python、Java、Perl、PHP、Eiffel、Ruby和Tcl等。

MySQL支持大型数据库，支持5000万条记录的数据仓库，32位系统表文件最大可支持4GB，64位系统支持最大的表文件为8TB。

### 1.1.2、Mysql高手是怎样炼成的

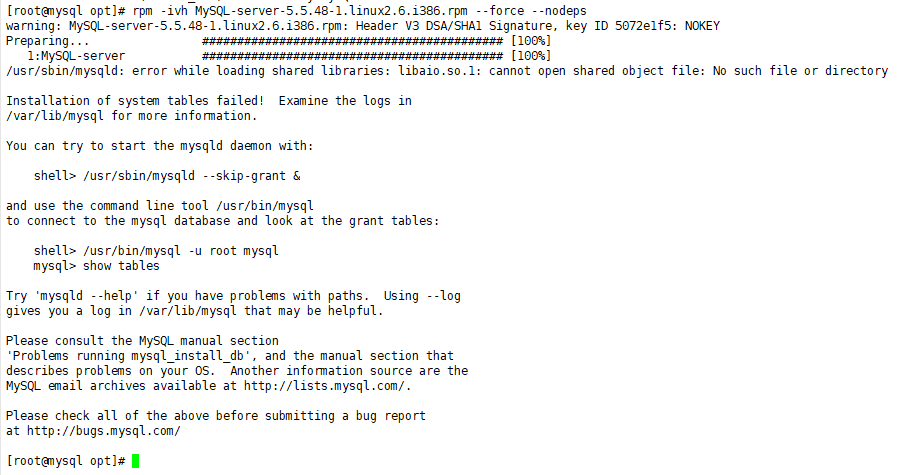
数据库内部结构和原理、数据库建模优化、数据库索引建立、SQL语句优化、mysql服务器的安装位置、数据库的性能监控分析与系统优化、各种参数常量设定、主从复制、分布式架构搭建、垂直切割和水平切割、数据迁移、容灾备份和恢复、shell或python等脚本语言开发、对开源数据库进行二次开发。

## 1.2、Mysql的Linux版本安装

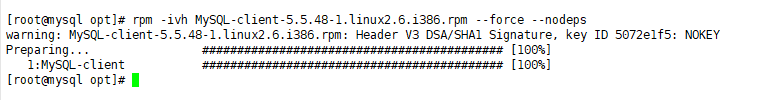
二进制源码安装 rpm包安装

rpm包安装命令：

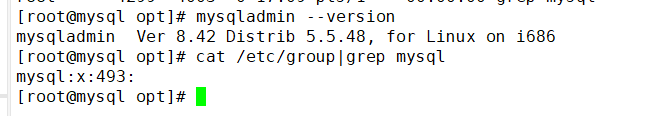
[root@mysql opt]# rpm -ivh MySQL-server-5.5.48-1.linux2.6.i386.rpm --force --nodeps



[root@mysql opt]# rpm -ivh MySQL-client-5.5.48-1.linux2.6.i386.rpm --force --nodeps



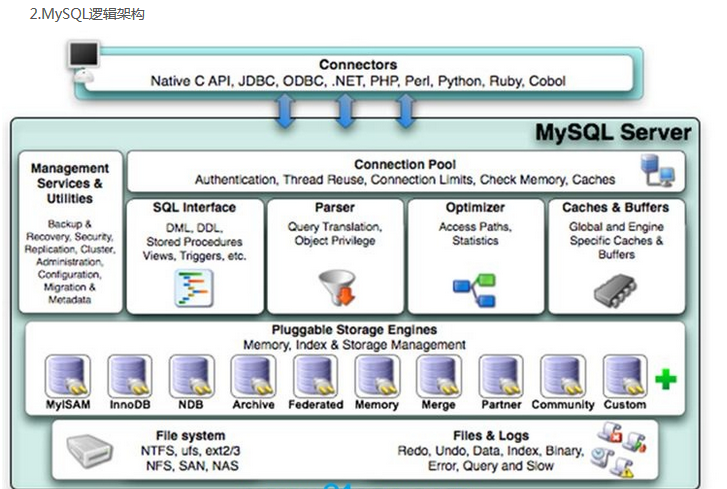
看见如下命令代表安装成功：



## 1.6、Mysql逻辑架构

### 1.6.1、总体概述

和其它数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎的架构上，插件式的存储引擎架构将查询处理和其它的系统任务以及数据的存储提取相分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。



1.连接层

 最上层是一些客户端和连接服务，包含本地sock通信和大多数基于客户端/服务端工具实现的类似于tcp/ip的通信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

2.服务层

2.1 Management Serveices & Utilities： 系统管理和控制工具

2.2  SQL Interface: SQL接口

      接受用户的SQL命令，并且返回用户需要查询的结果。比如select from就是调用SQL Interface

2.3 Parser: 解析器

       SQL命令传递到解析器的时候会被解析器验证和解析。

2.4 Optimizer: 查询优化器。

     SQL语句在查询之前会使用查询优化器对查询进行优化。

     用一个例子就可以理解： select uid,name from user where  gender= 1;

     优化器来决定先投影还是先过滤。

2.5 Cache和Buffer： 查询缓存。

      如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。

      这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等

       缓存是负责读，缓冲负责写。

3.引擎层

  存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取。后面介绍MyISAM和InnoDB

4.存储层

  数据存储层，主要是将数据存储在运行于裸设备的文件系统之上，并完成与存储引擎的交互。

和其它数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎的架构上，插件式的存储引擎架构将查询处理和其它的系统任务以及数据的存储提取相分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。

1.连接层

最上层是一些客户端和连接服务，包含本地sock通信和大多数基于客户端/服务端工具实现的类似于tcp/ip的通信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

2.服务层

2.1 Management Serveices & Utilities： 系统管理和控制工具

2.2 SQL Interface: SQL接口

接受用户的SQL命令，并且返回用户需要查询的结果。比如select from就是调用SQL Interface

2.3 Parser: 解析器

SQL命令传递到解析器的时候会被解析器验证和解析。

2.4 Optimizer: 查询优化器。

SQL语句在查询之前会使用查询优化器对查询进行优化。

用一个例子就可以理解： select uid,name from user where gender= 1;

优化器来决定先投影还是先过滤。

2.5 Cache和Buffer： 查询缓存。

如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。

这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等

缓存是负责读，缓冲负责写。

3.引擎层

存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取。后面介绍MyISAM和InnoDB

4.存储层

数据存储层，主要是将数据存储在运行于裸设备的文件系统之上，并完成与存储引擎的交互。

### 1.6.2、Mysql查询

mysql的查询流程大致是：mysql客户端通过协议与mysql服务器建连接，发送查询语句，先检查查询缓存，如果命中(一模一样的sql才能命中)，直接返回结果，否则进行语句解析,也就是说，在解析查询之前，服务器会先访问查询缓存(query cache)——它存储SELECT语句以及相应的查询结果集。如果某个查询结果已经位于缓存中，服务器就不会再对查询进行解析、优化、以及执行。它仅仅将缓存中的结果返回给用户即可，这将大大提高系统的性能。

语法解析器和预处理：首先mysql通过关键字将SQL语句进行解析，并生成一颗对应的“解析树”。mysql解析器将使用mysql语法规则验证和解析查询；预处理器则根据一些mysql规则进一步检查解析数是否合法。

查询优化器当解析树被认为是合法的了，并且由优化器将其转化成执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。然后，mysql默认使用的BTREE索引，并且一个大致方向是:无论怎么折腾sql，至少在目前来说，mysql最多只用到表中的一个索引。

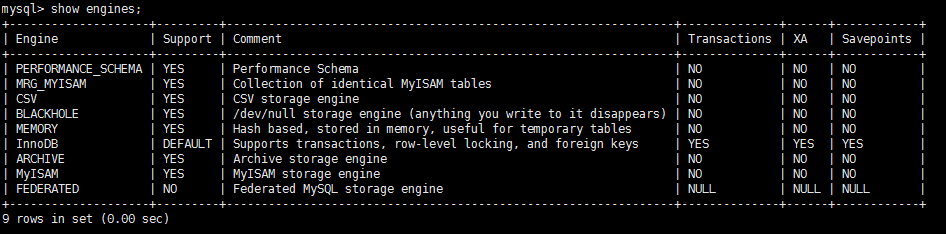
## 1.7、Mysql存储引擎

### 1.7.1、查看现在使用什么引擎

 1 如何用命令查看

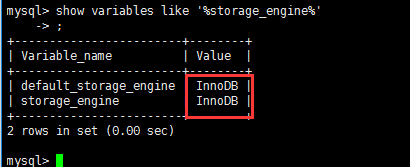
  #看你的mysql现在已提供什么存储引擎:

  mysql> show engines;



  #看你的mysql当前默认的存储引擎:

  mysql> show variables like '%storage\_engine%';



### 1.7.2、各种引擎介绍

1、InnoDB存储引擎

InnoDB是MySQL的默认事务型引擎，它被设计用来处理大量的短期(short-lived)事务。除非有非常特别的原因需要使用其他的存储引擎，否则应该优先考虑InnoDB引擎。行级锁，适合高并发情况

2、MyISAM存储引擎

MyISAM提供了大量的特性，包括全文索引、压缩、空间函数(GIS)等，但MyISAM不支持事务和行级锁(myisam改表时会将整个表全锁住)，有一个毫无疑问的缺陷就是崩溃后无法安全恢复。

3、Archive引擎

Archive存储引擎只支持INSERT和SELECT操作，在MySQL5.1之前不支持索引。

Archive表适合日志和数据采集类应用。适合低访问量大数据等情况。

根据英文的测试结论来看，Archive表比MyISAM表要小大约75%，比支持事务处理的InnoDB表小大约83%。

4、Blackhole引擎

Blackhole引擎没有实现任何存储机制，它会丢弃所有插入的数据，不做任何保存。但服务器会记录Blackhole表的日志，所以可以用于复制数据到备库，或者简单地记录到日志。但这种应用方式会碰到很多问题，因此并不推荐。

5、CSV引擎

CSV引擎可以将普通的CSV文件作为MySQL的表来处理，但不支持索引。

CSV引擎可以作为一种数据交换的机制，非常有用。

CSV存储的数据直接可以在操作系统里，用文本编辑器，或者excel读取。

6、Memory引擎

如果需要快速地访问数据，并且这些数据不会被修改，重启以后丢失也没有关系，那么使用Memory表是非常有用。Memory表至少比MyISAM表要快一个数量级。(使用专业的内存数据库更快，如redis)

7、Federated引擎

Federated引擎是访问其他MySQL服务器的一个代理，尽管该引擎看起来提供了一种很好的跨服务器的灵活性，但也经常带来问题，因此默认是禁用的。

### 1.7.3、MyISAM和InnoDB存储引擎的对比

|  |  |  |
| --- | --- | --- |
| 对比项 | MyISAM | InnoDB |
| 主外键 | 不支持 | 支持 |
| 事务 | 不支持 | 支持 |
| 行表锁 | 表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作 | 行锁,操作时只锁某一行，不对其它行有影响，适合高并发的操作 |
| 缓存 | 只缓存索引，不缓存真实数据 | 不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响 |
| 表空间 | 小 | 大 |
| 关注点 | 性能 | 事务 |
| 默认安装 | Y | Y |
| 用户表默认使用 | N | Y |
| 自带系统表使用 | Y | N |

innodb 索引 使用 B+TREE myisam 索引使用 b-tree

innodb 主键为聚簇索引，基于聚簇索引的增删改查效率非常高。

Percona 为 MySQL 数据库服务器进行了改进，在功能和性能上较 MySQL 有着很显著的提升。该版本提升了在高负载情况下的 InnoDB 的性能、为 DBA 提供一些非常有用的性能诊断工具；另外有更多的参数和命令来控制服务器行为。该公司新建了一款存储引擎叫xtradb完全可以替代innodb,并且在性能和并发上做得更好,

阿里巴巴大部分mysql数据库其实使用的percona的原型加以修改。AliSql+AliRedis

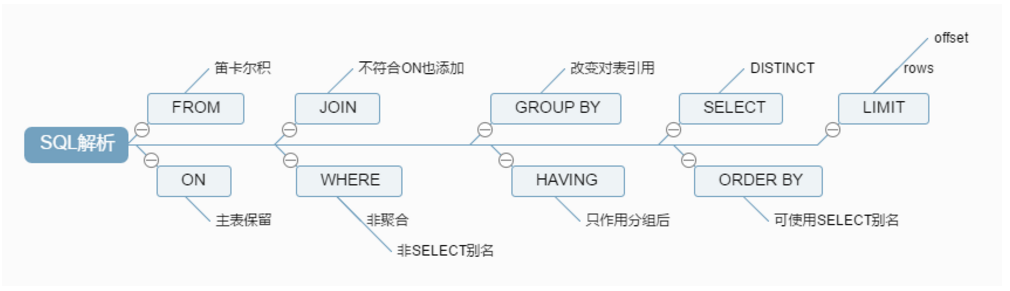
# 索引优化分析

## 2.1、性能下降，SQL慢执行时间长，等待时间长

查询的数据过多、关联了太多的表，写了太多的join、没有利用索引、服务器调优及各个参数设置（缓存，线程数量等）

## 2.2、常见通用的Join查询

### 2.2.1、SQL执行顺序



### 2.2.2、Join

|  |  |  |
| --- | --- | --- |
| 内连接 | select \*  from TableA a  inner join TableB b  on a.id = b.id |  |
| 左外连接 | select \*  from TableA a  left join TableB b  on a.key = b.key |  |
| 右外连接 | select \*  from TableA a  right join TableB b  on a.key = b.key |  |
|  | select \*  from TableA a  left join TableB b  on a.key = b.key  where b.key is null |  |
|  | select \*  from TableA a  right join TableB b  on a,key = b.key  where a.key = null |  |
| 全连接 | select \*  from TableA a  full outer join TableB b  on a.key = b.key |  |
|  | select \*  from TableA a  full outer join TableB b  on a.key = b.key  where a.key is null  or b.key is null |  |

## 2.3、索引简介

### 2.3.1、索引是什么？

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。可以得到索引的本质：索引是数据结构。**排好序的快速查找数据结构就是索引。**一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。

我们平常所说的索引，如果没有特别指明，都是指B树(多路搜索树，并不一定是二叉的)结构组织的索引。其中聚集索引，次要索引，覆盖索引，复合索引，前缀索引，唯一索引默认都是使用B+树索引，统称索引。当然，除了B+树这种类型的索引之外，还有哈稀索引(hash index)等。

索引会影响到order by 排序和where后面的查找。

数据本身之外，数据库还维护着一个满足特定查找算法的数据结构，这些数据结构以某种方式指向数据，这样就可以在这些数据结构的基础上实现高级查找算法，这种数据结构就是索引。

### 2.3.2、索引的优势

类似大学图书馆建书目索引，提高数据检索的效率，降低数据库的IO成本。

通过索引列对数据进行排序，降低数据排序的成本，降低了CPU的消耗。

### 2.3.3、索引的劣势

1、实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也是要占用空间的。

2、虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

**3、索引只是提高效率的一个因素，如果你的MySQL有大数据量的表，就需要花时间研究建立最优秀的索引，或优化查询语句**

### 2.3.4、Mysql索引分类

**主键索引：**设定为主键后数据库会自动建立索引，innodb为聚簇索引

随表一起建索引：

CREATE TABLE customer (

id INT(10) UNSIGNED AUTO\_INCREMENT ,

customer\_no VARCHAR(200),

customer\_name VARCHAR(200),

PRIMARY KEY(id)

);

unsigned (无符号的)，使用AUTO\_INCREMENT 关键字的列必须有索引(只要有索引就行)。

单独建主键索引：ALTER TABLE customer add PRIMARY KEY customer(customer\_no);

删除建主键索引：ALTER TABLE customer drop PRIMARY KEY ;

修改建主键索引：必须先删除掉(drop)原索引，再新建(add)索引

**单值索引：**即一个索引只包含单个列，一个表可以有多个单列索引

随表一起建索引：

CREATE TABLE customer (

id INT(10) UNSIGNED AUTO\_INCREMENT ,

customer\_no VARCHAR(200),

customer\_name VARCHAR(200),

PRIMARY KEY(id),

KEY (customer\_name)

);

随表一起建立的索引 索引名同 列名(customer\_name)

单独建单值索引：

CREATE INDEX idx\_customer\_name ON customer(customer\_name);

删除索引：

DROP INDEX idx\_customer\_name ;

**唯一索引：**索引列的值必须唯一，但允许有空值

随表一起建索引：

CREATE TABLE customer (

id INT(10) UNSIGNED AUTO\_INCREMENT ,

customer\_no VARCHAR(200),

customer\_name VARCHAR(200),

PRIMARY KEY(id),

KEY (customer\_name),

UNIQUE (customer\_no)

);

建立 唯一索引时必须保证所有的值是唯一的（除了null），若有重复数据，会报错。

单独建唯一索引：

CREATE UNIQUE INDEX idx\_customer\_no ON customer(customer\_no);

删除索引：

DROP INDEX idx\_customer\_no on customer ;

**复合索引：**即一个索引包含多个列，在数据库操作期间，复合索引比单值索引所需要的开销更小(对于相同的多个列建索引) 当表的行数远大于索引列的数目时可以使用复合索引。

随表一起建索引：

CREATE TABLE customer (

id INT(10) UNSIGNED AUTO\_INCREMENT,

customer\_no VARCHAR(200),customer\_name VARCHAR(200),

PRIMARY KEY(id),

KEY (customer\_name),

UNIQUE (customer\_name),

KEY (customer\_no,customer\_name)

);

单独建索引：

CREATE INDEX idx\_no\_name ON customer(customer\_no,customer\_name);

删除索引：

DROP INDEX idx\_no\_name on customer ;

**基本语法**

**创建：**ALTER mytable ADD [UNIQUE ] INDEX [indexName] ON (columnname(length))

**删除**：DROP INDEX [indexName] ON mytable;

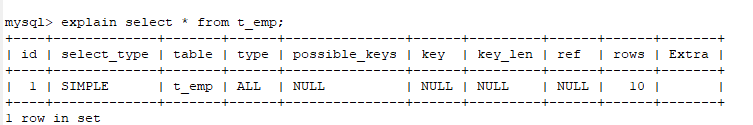
## 2.4、那些情况该建立索引，那些情况不该建

|  |  |
| --- | --- |
| 应该建索引 | 不应该建索引 |
| 主键自动建立唯一索引  频繁作为查询条件的字段应该创建索引where字句  查询中与其它表关联的字段，外键关系建立索引  单键/组合索引的选择问题，在高并发的情况下倾向于创建组合索引  查询中排序字段，排序字段如果通过索引去访问将大大提高排序速度  查询中统计或者分组字段 | 表的记录太少  经常会增删改的表  where条件里用不到的字段不创建索引  Why:提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。  因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件  数据重复且分布平均的表字段，因此应该只为最经常查询和最经常排序的数据列建立索引。  注意，如果某个数据列包含许多重复的内容，为它建立索引就没有太大的实际效果。 |

## 2.5、性能分析explain

explain + sql语句

使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，



表头

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| id | select\_type | table | key | type | possible\_keys | key\_len | ref | rows | Extra |

**最重要的五个: id,type,key,rows,extra这五个最重要**

**id:** select查询的序列号,包含一组数字，表示**查询中执行select子句或操作表的顺序**

id相同，执行顺序由上至下

id不同，如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行

id相同不同，同时存在

小表驱动大表，需要逐步的微调和优化加载顺序。

**select\_type**: 查询的类型，主要是用于区别普通查询、联合查询、子查询等的复杂查询

|  |  |
| --- | --- |
| simple | 简单的 select 查询,查询中不包含子查询或者UNION |
| primiary | 查询中若包含任何复杂的子部分，最外层查询则被标记为Primary |
| derived | 在FROM列表中包含的子查询被标记为DERIVED(衍生)  MySQL会递归执行这些子查询, 把结果放在临时表里。 |
| subquery | 在SELECT或WHERE列表中包含了子查询 |
| dependent subquery | 在SELECT或WHERE列表中包含了子查询,子查询基于外层 |
| uncacheable subquery | 无法被缓存的子查询 |
| union | 若第二个SELECT出现在UNION之后，则被标记为UNION；  若UNION包含在FROM子句的子查询中,外层SELECT将被标记为：DERIVED |
| union result | 从UNION表获取结果的SELECT |

**table :**显示这一行的数据是关于那张表的

**type:** 显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：

system>const>eq\_ref>ref>range>index>ALL 一般来说，得保证查询至少达到range级别，最好能达到ref。

**possible\_key** :显示可能应用在这张表上的索引，一个或者多个，查询涉及到的字段上若存在索引，则该索引将被抛出，但不一定被查询实际应用

**key**:实际使用的查询，如果为null，表示没有使用索引，查询中若使用了覆盖索引，则索引和查询的select字段重叠。

**key\_len:** 表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。key\_len字段能够帮你检查是否充分的利用上了索引

**ref:** 显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列上的值

**rows:** rows列显示MySQL认为它执行查询时必须检查的行数。

**extra**: 包含不适合在其他列中显示但十分重要的额外信息

1、Using filesort文件内排序 九死一生，这种时候比较危险，就是建立的索引没有全部使用，只是使用了一部分。

2、Using temporary 新建了临时表 十死无生 使了用临时表保存中间结果,MySQL在对查询结果排序时使用临时表。常见于排序 order by 和分组查询 group by。

3、Using Index表示相应的select操作中使用了覆盖索引(Covering Index)，避免访问了表的数据行，效率不错！

如果同时出现using where，表明索引被用来执行索引键值的查找;如果没有同时出现using where，表明索引只是用来读取数据而非利用索引执行查找。

覆盖索引：建立的索引是col1\_col2，查询的列表刚好就是col1 col2或者部分满足。

4、 Using where 使用了where进行过滤

5、Using join buffer 使用了连接缓存

6、impossible where where字句的值总是false

## 2.6、单表查询优化案例分析

单表建表脚本

DROP TABLE IF EXISTS `article`;

CREATE TABLE if not exists `article` (

`id` int(10) unsigned NOT NULL AUTO\_INCREMENT,

`author\_id` int(10) unsigned NOT NULL,

`category\_id` int(10) unsigned NOT NULL,

`views` int(10) unsigned NOT NULL,

`commments` int(10) unsigned NOT NULL,

`tiele` varchar(255) NOT NULL,

`content` text NOT NULL,

PRIMARY KEY (`id`)

);

INSERT INTO `article` VALUES ('1', '1', '1', '1', '1', '1', '1');

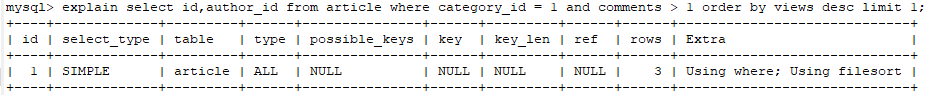
INSERT INTO `article` VALUES ('2', '2', '2', '2', '2', '2', '2');

INSERT INTO `article` VALUES ('3', '1', '1', '3', '3', '3', '3');

先功能再性能

查询category\_id 为1 并且 comments 大于1的情况下，views最多的article\_id

explain select id,author\_id from article where category\_id = 1 and comments > 1 order by views desc limit 1;



很显然,type 是 ALL,即最坏的情况。Extra 里还出现了 Using filesort,也是最坏的情况。优化是必须的。

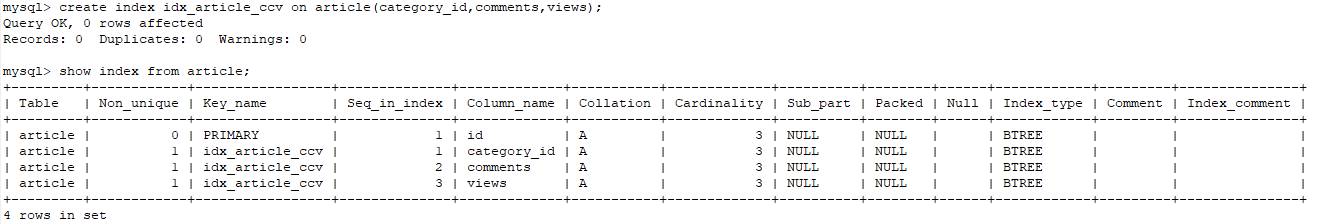
#开始优化：

# 1.1 新建索引+删除索引 有三种方法，选择第二种：

#ALTER TABLE `article` ADD INDEX idx\_article\_ccv ( `category\_id` , `comments`, `views` );

create index idx\_article\_ccv on article(category\_id,comments,views);

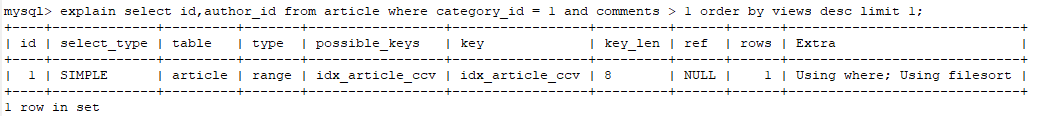
DROP INDEX idx\_article\_ccv ON article



# 1.2 第2次EXPLAIN

这个时候再去输入刚刚的那句相同的sql语句

EXPLAIN SELECT id,author\_id FROM `article` WHERE category\_id = 1 AND comments >1 ORDER BY views DESC LIMIT 1;



#结论：

#type 变成了 range,这是可以忍受的。但是 extra 里使用 Using filesort 仍是无法接受的。

#但是我们已经建立了索引,为啥没用呢?

#这是因为按照 BTree 索引的工作原理,

# 先排序 category\_id,

# 如果遇到相同的 category\_id 则再排序 comments,如果遇到相同的 comments 则再排序 views。

#当 comments 字段在联合索引里处于中间位置时,

#因comments > 1 条件是一个范围值(所谓 range),

#MySQL 无法利用索引再对后面的 views 部分进行检索,即 range 类型查询字段后面的索引无效。

# 1.3 删除第一次建立的索引

DROP INDEX idx\_article\_ccv ON article;

# 1.4 第2次新建索引

#ALTER TABLE `article` ADD INDEX idx\_article\_cv ( `category\_id` , `views` ) ;

create index idx\_article\_cv on article(category\_id,views);

# 1.5 第3次EXPLAIN

EXPLAIN SELECT id,author\_id FROM article WHERE category\_id = 1 AND comments > 1 ORDER BY views DESC LIMIT 1;

#结论：可以看到,type 变为了 ref,Extra 中的 Using filesort 也消失了,结果非常理想。

DROP INDEX idx\_article\_cv ON article;

## 2.7、两表查询优化案例分析

结论就是相反加。左连接加在右边表上

CREATE TABLE IF NOT EXISTS `class` (

`id` INT(10) UNSIGNED NOT NULL AUTO\_INCREMENT,

`card` INT(10) UNSIGNED NOT NULL,

PRIMARY KEY (`id`)

);

CREATE TABLE IF NOT EXISTS `book` (

`bookid` INT(10) UNSIGNED NOT NULL AUTO\_INCREMENT,

`card` INT(10) UNSIGNED NOT NULL,

PRIMARY KEY (`bookid`)

);

INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));

INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));

INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() \* 20)));

INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));

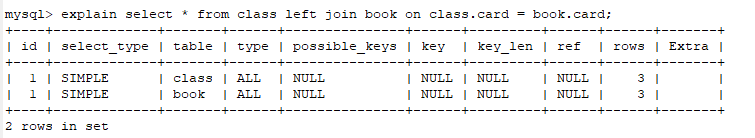
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));

INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() \* 20)));

# 下面开始explain分析

左连接

EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;



#结论：type 有All

# 添加索引优化

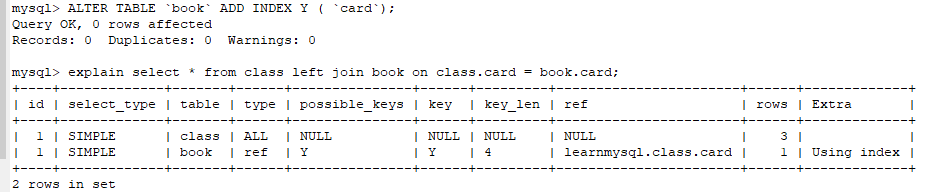
左连接把索引加在右边

ALTER TABLE `book` ADD INDEX Y ( `card`);

# 第2次explain

EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;

#可以看到第二行的 type 变为了 ref,rows 也变成了优化比较明显。



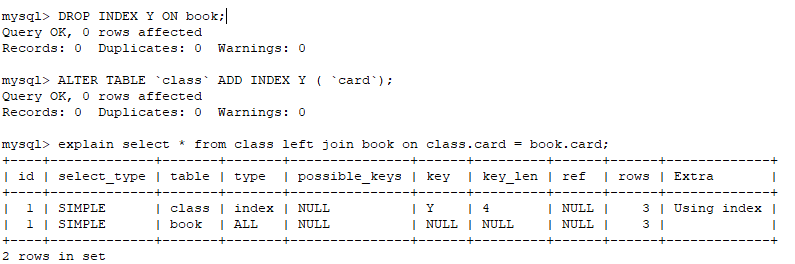
继续优化索引：

左连接左表全部都有

#这是由左连接特性决定的。LEFT JOIN 条件用于确定如何从右表搜索行,左边一定都有,

#所以右边是我们的关键点,一定需要建立索引。

# 删除旧索引 + 新建 + 第3次explain



DROP INDEX Y ON book;

ALTER TABLE class ADD INDEX X (card);

EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;

## 2.8、三表查询优化案例分析

尽可能减少join语句中的NestedLoop的循环次数，永远使用小的结果集驱动大的结果集；

优先优化嵌套循环NestLoop里面的内层循环；

保证join语句中被驱动表上join条件字段已经被索引。

当无法保证被驱动表的Join条件字段被索引且内存资源充足的前提下，不要太吝惜JoinBuffer的设置。

create table if not exists phone(

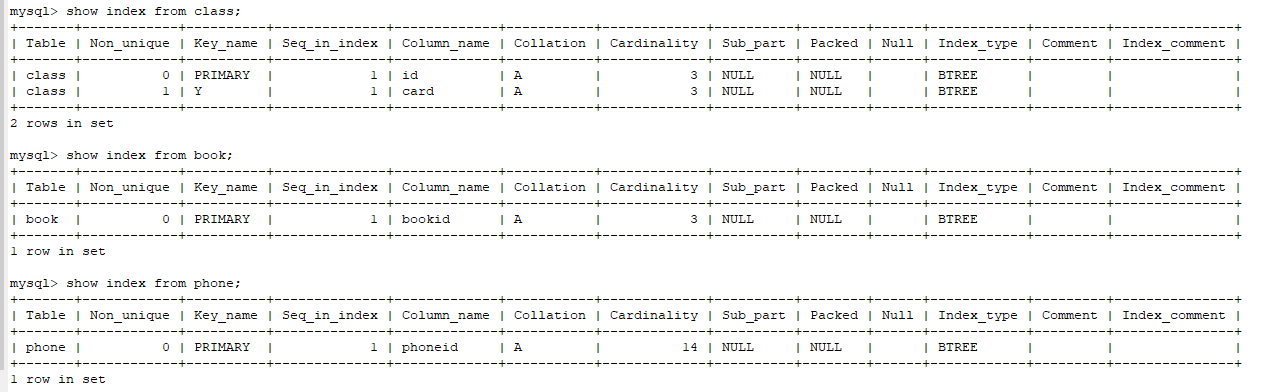
phoneid int(10) UNSIGNED NOT NULL AUTO\_INCREMENT,

card int(10) UNSIGNED NOT NULL,

PRIMARY KEY(phoneid)

);

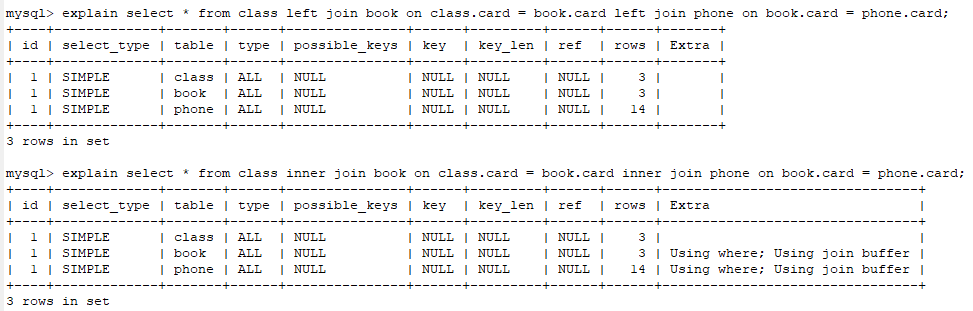
INSERT INTO phone(card) VALUES(FLOOR(1+(RAND()\*20)));



将所有表的索引删除

drop index Y on class;

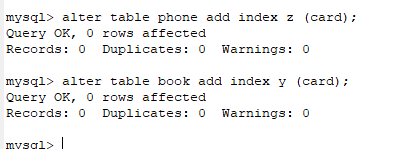
多表连接的时候回产生Join buffer

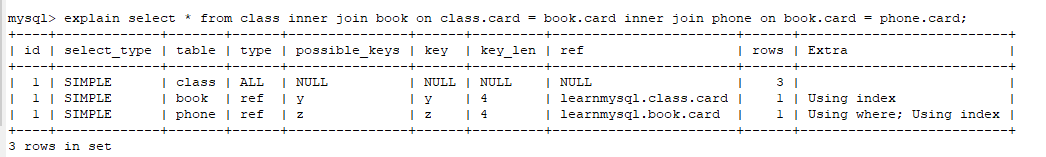


三表连接的SQL

select \* from class left join book on class.card = book.card left join phone on book.card = phone.card;

建立索引之后再去explain这个sql语句：





## 2.9、索引失效

建立索引的意思就是给数据表的某个字段排序，索引失效就是指建立好的索引用不上了

避免索引失效：

1. 全值匹配我最爱
2. 最佳左前缀法则 如果索引了多列，要遵守最佳左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列。

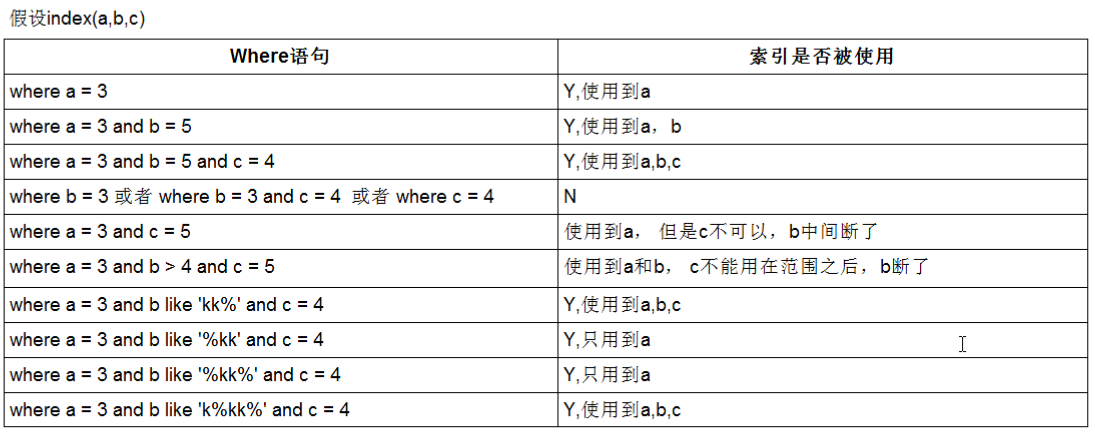
带头大哥不能死，中间兄弟不能断，索引列上不计算，like百分加右边，范围之后全失效，字符串里有引号

建了三个列的符合索引abc，用的时候，没有a去用bc会导致索引失效。

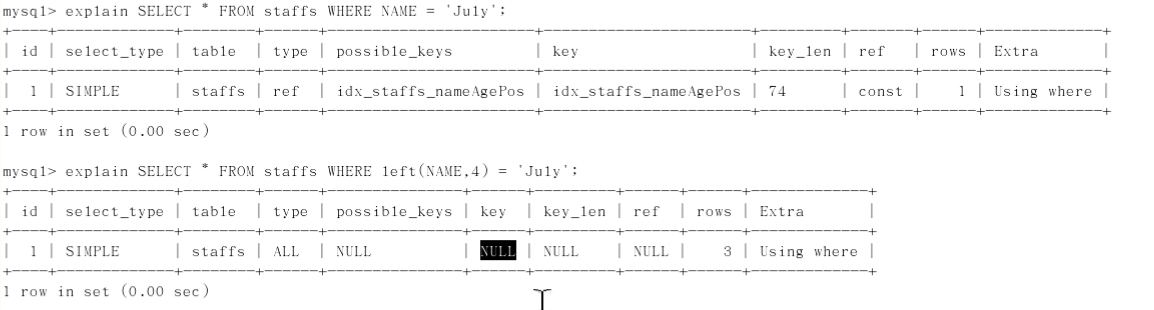
1. 不再索引列上做任何操作（计算、函数、类型转换）
2. 存储引擎不能使用索引中范围条件右边的列
3. 尽量使用覆盖索引（只访问索引的查询，索引列和查询列一致），减少select \*
4. mysql 在使用不等于（!= <>）的时候无法使用索引会导致全表扫描
5. is null,is not null 也无法使用索引
6. like以通配符开头（ like ‘%abc…’）mysql索引失效变成全表扫描
7. 字符串不加单引号索引会失效
8. 少用or，用它来连接时会失效

假设index(a,b,c)

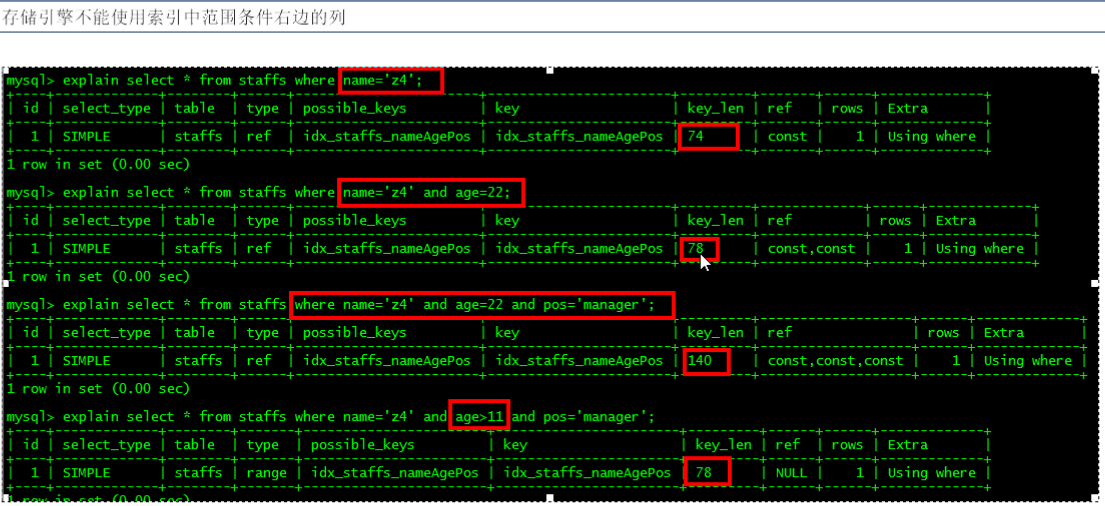
|  |  |
| --- | --- |
| Where a=3 | Y a |
| Where a = 3 and b = 5 | Y a,b |
| Where a = 3 and b = 5 and c = 4 | Y a,b,c |
| Where b = 3 或者 where b = 3 and c = 4或者 where c = 4 | N 大哥掉了 |
| Where a = 3 ande c = 5 | Y a c不可用 |
| Where a = 3 and b > 4 and c = 5 | Y ab c不能用在范围之后，b断了 |
| Where a = 3 and b like ‘kk%’ and c = 4 | Y a b能用 c不能用 |



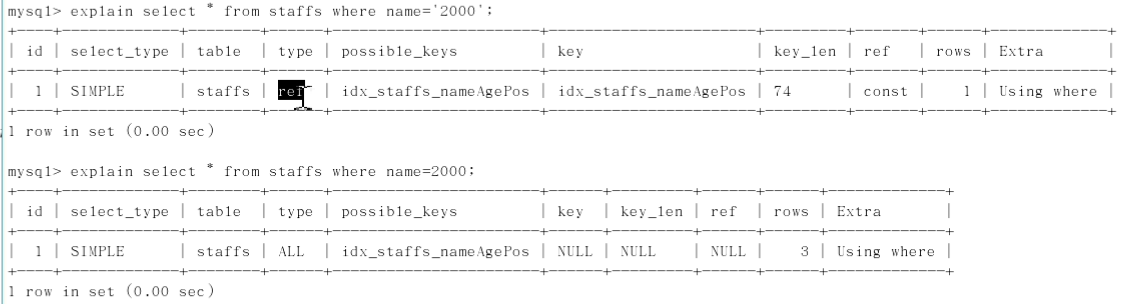
3案例



4案例



类型转换



例题分析

explain select \* from test where c1=’c2’ and c2 =’c2’ and c3>’a4’ and c4 = ‘c4’;

用到了3个

explain select \* from test where c1=’c2’ and c2 =’c2’ and c4>’a4’ and c3 > ‘a3’;

用到了4个

explain select \* from test where c1=’c2’ and c2 =’c2’ and c4>’a4’ and order by a3;

用到了2个 c3用于排序

explain select \* from test where c1=’c2’ and c2 =’c2’ order by a3;

用到了2个 c3用于排序

explain select \* from test where c1=’c2’ and c2 =’c2’ order by a4;

出现了filesort

explain select \* from test where c1 =’al’ and c5=’c5’ order by c2,c3;

用到了1个 只用到了c1一个字段索引，但是c2,c3用于排序，没有filesort

explain select \* from test where c1 =’al’ and c5=’c5’ order by c3,c2;

出现了filesort 建立的索引是1234，它没有按照顺序来，颠倒了

explain select \* from test where c1 =’al’ and c2=’c2’ order by c2,c3;

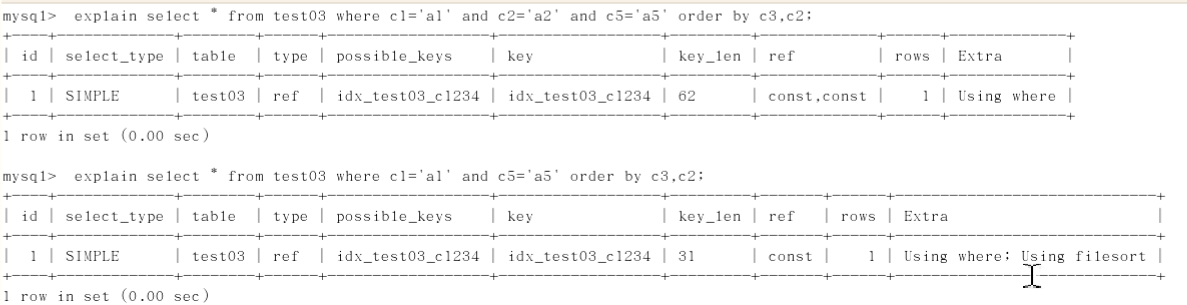
用到了两个

explain select \* from test where c1 =’al’ and c2=’c2’ and c5 = ‘a5’ order by c2,c3;

用到了两个 加条件没关系 c1 c2用于索引 c2 c3用于排序

explain select \* from test where c1 =’al’ and c2=’c2’ and c5 = ‘a5’ order by c3,c2;

这个没有出现filesort 因为是需要排序的字段在前面已经变成一个常量了 固定的唯一值



explain select \* from test where c1 =’al’ and c2=’c2’ and c4 = ‘a4’ order by c2,c3;

只用到一个索引

explain select \* from test where c1 =’al’ and c2=’c2’ and c4 = ‘a4’ order by c3,c2;

索引没有失效但是出现了 using temporary 和 using filesort 毁天灭地。

结论：

对于单键索引，尽量选择针对当前query过滤性更好的索引

在选择组合索引的时候，当前Query中过滤最好的字段在索引字段顺序中，位置越靠左越好

在选择组合索引的时候，尽量选择可以能够包含当前query中的where字句更多字段的索引

尽可能通过分析统计信息和调整query的写法来达到选择合适索引的目的

## SQL索引优化总结口诀

全值匹配我最爱，最左前缀要遵守

带头大哥不能死，中间兄弟不能掉

索引列上少计算，范围之后全失效

Like百分写最有，覆盖索引不写星

不等空值还有or, 索引失效要少用

字符引号不能丢，SQL高级也easy

# 查询截取分析

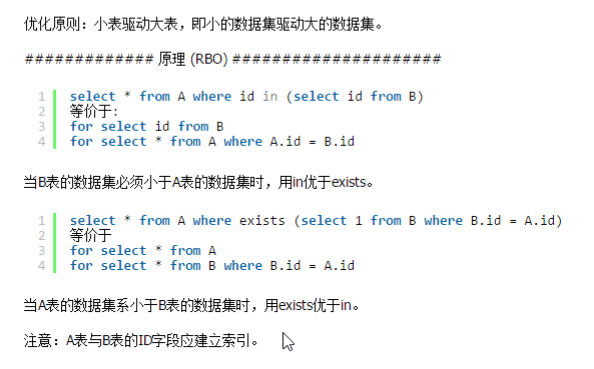
分析步骤

1. 观察，看看成产环境的慢SQL情况
2. 开启慢查询日志，设置阀值，比如：超过5秒钟的就是慢SQL
3. explain + 慢SQL分许 这一过程能解决80%的SQL跑的慢的问题
4. 使用show prifile 查询SQL在MySQL服务器里面的执行细节和生命周期情况，到这个过程大概能够解决99%的问题
5. 最后才是 运维工程师或者DBA进行SQL数据库服务器

## 3.1、查询优化

### 3.1.1、子查询优化

子查询优化，是使用in 还是使用exists?



### 3.1.2、order by 子句优化

create table if not exists tblA(

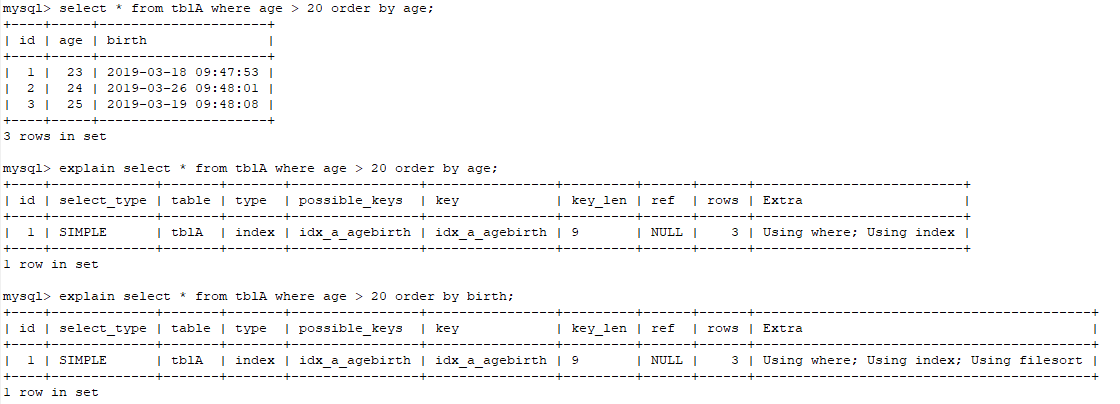
id int primary key not null auto\_increment,

age int,

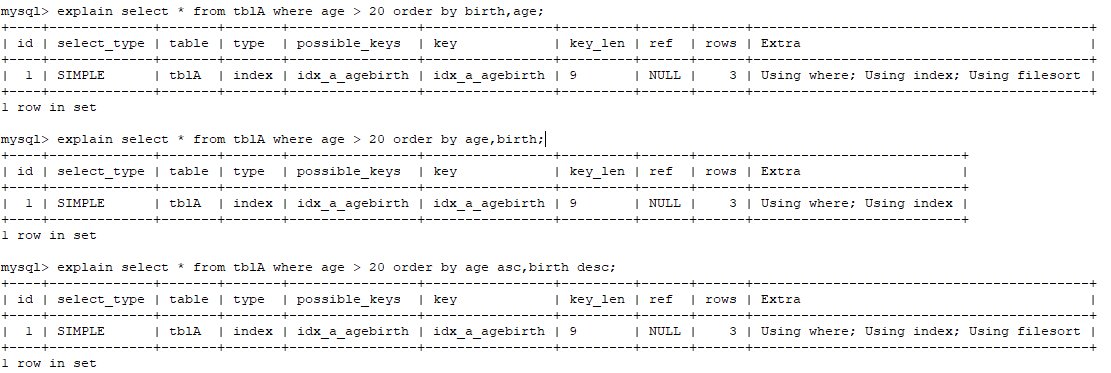
birth timestamp not null

);

create index idx\_a\_agebirth on tblA(age,birth);



索引：排好序的快速查找数据结构



ORDER BY子句，尽量使用Index方式排序,避免使用FileSort方式排序

尽可能在索引列上完成排序，遵照索引的最佳左前缀原则，如果不再索引列上，filesort有两种算法，mysql就要启动双路排序和单路排序。

双路排序：

Mysql4.1之前是使用双路排序，字面意思就是两次扫描磁盘，最终得到数据，读取行指针和orderby列，对他们排序，然后扫描已经排好序的列表，按照列表中的值重新从列表中的值重新从列表中读取对应的数据输出。

单路排序：

从磁盘读取查询需要的所有列，按照orderby列在buffer对它们进行排序，然后扫描排序后的列表进行输出，它的效率更快一些，避免了第二次读取数据。并且把随机IO变成了顺序IO,到那时它会使用更多的空间，因为它把每一行都保存在内存中了。

但是单路排序的使用是会产生问题的，就是能不能一次抓完。有可能取出的数据的总大小超过了sort\_buffer的容量，导致每次只能娶sort\_buffer容量大小的数据进行排序，排序完成之后再次读取，导致磁盘多次IO,反而得不偿失。

优化策略是：

增大sort\_buffer\_size参数的设置、增大max\_length\_for\_sort\_data参数的设置

结论：提高Order By的速度：

1. Order by时 select \* 是一个大忌，只写需要query的字段就可以，这里会产生的影响是：当query的字段大小总和小鱼max\_length\_for\_sort\_data 而且排序字段不是text或者blob类型时候，会使用单路排序算法，否则就会使用双路排序算法。两种算法都有可能超出sort\_buffer的容量，超出之后，会创建tmp文件进行合并排序，导致多次IO，但是使用单路排序算法的风险会更大一些，所以要提高sort\_buffer\_size。select \* 会很容易把缓冲区用满。
2. 尝试提高sort\_buffer\_size，不管使用哪种排序算法，提高这个参数都会提高效率，当然需要根据系统的性能去调整。
3. 尝试提高max\_length\_for\_sort\_data,会增加用改进算法的概率，但是设置的太高，数据总量超出sort\_buffer\_size的概率也就增大。

小结：

为排序使用索引

Mysql两种排序方式：文件排序 filesort 或扫描有序索引排序表 using index

Mysql能为排序与查询使用相同的索引

**key a\_b\_c(a,b,c)**

|  |  |
| --- | --- |
| Order by 能使用索引最左前缀 | 如果where使用索引的最左前缀定义为常量，则order by 能使用索引 |
| order by a  order by a,b  order by a,b,c  order by a desc,b desc,c desc | Where a = const order by b,c  Where a = const and b = const ordrt by c  Where a = const and b > const order by b,c |

不能使用索引进行排序的情况

order by a asc,b desc,c desc 排序不一致

where g = const order by b,c 索引a丢失

where a = const order by c 索引b丢失

where a = const order by a,d d不是索引的一部分

where a in (…) order by b,c 对于排序来说，多个相等的条件也是范围查询

### 3.1.3、group by 字句优化

group by 的实质是先排序后进行分组，遵照索引建的最佳左前缀原则

当无法使用索引列，增大max\_length\_for\_sort\_data参数的设置，增大sort\_buffer\_size参数的设置

where高于having，能写在where限定的条件就不要去having 限定了

## 3.2、慢查询日志

### 3.2.1、什么是慢查询日志

MySQL的慢查询日志是MySQL提供的一种日志记录，它用来记录在MySQL中响应时间超过阀值的语句，具体指运行时间超过long\_query\_time值的SQL，则会被记录到慢查询日志中。

具体指运行时间超过long\_query\_time值的SQL，则会被记录到慢查询日志中。long\_query\_time的默认值为10，意思是运行10秒以上的语句。

由他来查看哪些SQL超出了我们的最大忍耐时间值，比如一条sql执行超过5秒钟，我们就算慢SQL，希望能收集超过5秒的sql，结合之前explain进行全面分析。

默认情况下，MySQL数据库没有开启慢查询日志，需要我们手动来设置这个参数。当然，如果不是调优需要的话，一般不建议启动该参数，因为开启慢查询日志会或多或少带来一定的性能影响。慢查询日志支持将日志记录写入文件。

Mysql大部分的牛X功能默认都是没有开启的。

### 3.2.2、怎么玩？

查看是否开启及如何开启：

默认情况下slow\_query\_log的值为OFF，表示慢查询日志是禁用的，可以通过设置slow\_query\_log的值来开启。

SHOW VARIABLES LIKE '%slow\_query\_log%';

使用set global slow\_query\_log=1开启了慢查询日志只对当前数据库生效，如果MySQL重启后则会失效。

如果要永久生效，就必须修改配置文件my.cnf（其它系统变量也是如此）

修改my.cnf文件，[mysqld]下增加或修改参数

slow\_query\_log 和slow\_query\_log\_file后，然后重启MySQL服务器。也即将如下两行配置进my.cnf文件

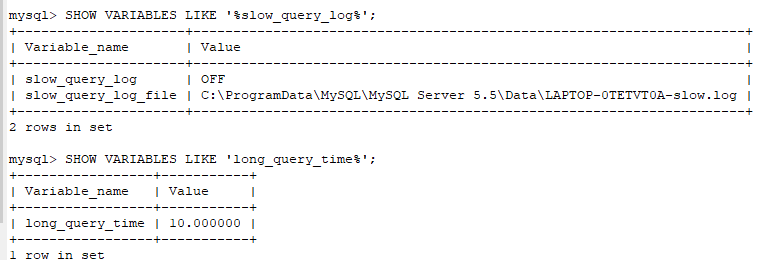
slow\_query\_log =1

slow\_query\_log\_file=/var/lib/mysql/atguigu-slow.log

关于慢查询的参数slow\_query\_log\_file ，它指定慢查询日志文件的存放路径，系统默认会给一个缺省的文件host\_name-slow.log（如果没有指定参数slow\_query\_log\_file的话）

由参数long\_query\_time控制，默认情况下long\_query\_time的值为10秒，

命令：SHOW VARIABLES LIKE 'long\_query\_time%';



### 3.2.3、日志分析工具 mysqldumpshow

得到返回记录集最多的10个SQL

mysqldumpslow -s r -t 10 /var/lib/mysql/atguigu-slow.log

得到访问次数最多的10个SQL

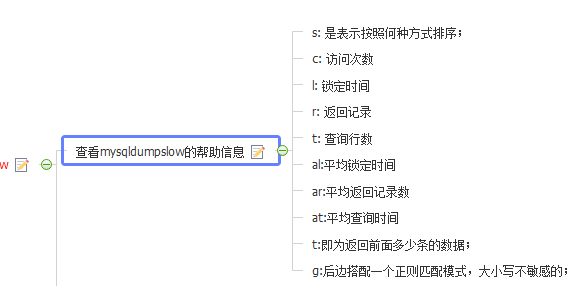
mysqldumpslow -s c -t 10 /var/lib/mysql/atguigu-slow.log

得到按照时间排序的前10条里面含有左连接的查询语句

mysqldumpslow -s t -t 10 -g "left join" /var/lib/mysql/atguigu-slow.log

另外建议在使用这些命令时结合 | 和more 使用 ，否则有可能出现爆屏情况

mysqldumpslow -s r -t 10 /var/lib/mysql/atguigu-slow.log | more



## 3.3、批量数据脚本

### 3.3.1、随机字符和数字

用于随机产生字符串

DELIMITER $$

CREATE FUNCTION rand\_string(n INT) RETURNS VARCHAR(255)

BEGIN

DECLARE chars\_str VARCHAR(100) DEFAULT 'abcdefghijklmnopqrstuvwxyzABCDEFJHIJKLMNOPQRSTUVWXYZ';

DECLARE return\_str VARCHAR(255) DEFAULT '';

DECLARE i INT DEFAULT 0;

WHILE i < n DO

SET return\_str =CONCAT(return\_str,SUBSTRING(chars\_str,FLOOR(1+RAND()\*52),1));

SET i = i + 1;

END WHILE;

RETURN return\_str;

END $$

用于随机产生部门编号：

DELIMITER $$

CREATE FUNCTION rand\_num( )

RETURNS INT(5)

BEGIN

DECLARE i INT DEFAULT 0;

SET i = FLOOR(100+RAND()\*10);

RETURN i;

END $$

### 3.3.2、创建函数

创建往emp表中插入数据的存储过程:

DELIMITER $$

CREATE PROCEDURE insert\_emp(IN START INT(10),IN max\_num INT(10))

BEGIN

DECLARE i INT DEFAULT 0;

SET autocommit = 0;

REPEAT

SET i = i + 1;

INSERT INTO emp (empno, ename ,job ,mgr ,hiredate ,sal ,comm ,deptno ) VALUES ((START+i) ,rand\_string(6),'SALESMAN',0001,CURDATE(),FLOOR(1+RAND()\*20000),FLOOR(1+RAND()\*1000),rand\_num());

UNTIL i = max\_num

END REPEAT;

COMMIT;

END $$

往dept表添加随机数据的存储过程

DELIMITER $$

CREATE PROCEDURE insert\_dept(IN START INT(10),IN max\_num INT(10))

BEGIN

DECLARE i INT DEFAULT 0;

SET autocommit = 0;

REPEAT

SET i = i + 1;

INSERT INTO dept (deptno ,dname,loc ) VALUES (START +i ,rand\_string(10),rand\_string(8));

UNTIL i = max\_num

END REPEAT;

COMMIT;

END $$

### 3.3.3、调用存储过程

调用存储过程

DELIMITER ;

CALL insert\_dept(100,10);

执行存储过程，往emp表添加50万条数据

DELIMITER ; #将 结束标志换回 ;

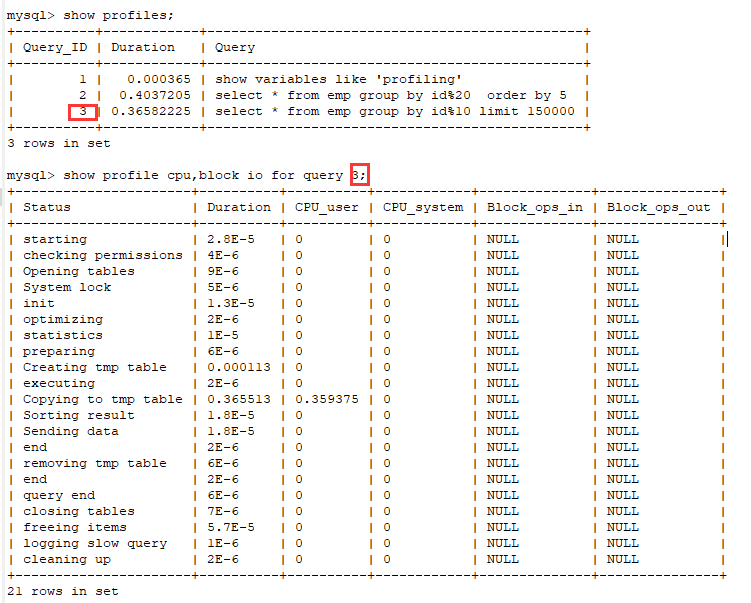
CALL insert\_emp(100001,500000);

CALL insert\_emp10000(100001,10000);

## 3.4、show profile

直接show profile是查看进10条SQL语句的运行时间

诊断SQL，show profile cpu,block io for query n (n为上一步前面的问题SQL数字号码);



type:

| ALL --显示所有的开销信息

| BLOCK IO --显示块IO相关开销

| CONTEXT SWITCHES --上下文切换相关开销

| CPU --显示CPU相关开销信息

| IPC --显示发送和接收相关开销信息

| MEMORY --显示内存相关开销信息

| PAGE FAULTS --显示页面错误相关开销信息

| SOURCE --显示和Source\_function，Source\_file，Source\_line相关的开销信息

| SWAPS --显示交换次数相关开销的信息



## 3.5、全局查询日志

尽量不要在生产环境开启这个功能。

配置启用：

在mysql的my.cnf中，设置如下：

#开启

general\_log=1

# 记录日志文件的路径

general\_log\_file=/path/logfile

#输出格式

log\_output=FILE

编码启用：

命令

set global general\_log=1;

#全局日志可以存放到日志文件中，也可以存放到Mysql系统表中。存放到日志中性能更好一些，存储到表中

set global log\_output='TABLE';

此后 ，你所编写的sql语句，将会记录到mysql库里的general\_log表，可以用下面的命令查看

select \* from mysql.general\_log;

# Mysql锁机制

## 4.1、概述

锁是计算机协调多个进程或线程并发访问某一资源的机制。在数据库中，除传统的计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

## 4.2、分类

从对数据操作的类型（读\写）分：读锁、写锁

读锁(共享锁)：针对同一份数据，多个读操作可以同时进行而不会互相影响。

写锁（排它锁）：当前写操作没有完成前，它会阻断其他写锁和读锁。

从对数据操作的粒度分：表锁，行锁

为了尽可能提高数据库的并发度，每次锁定的数据范围越小越好，理论上每次只锁定当前操作的数据的方案会得到最大的并发度，但是管理锁是很耗资源的事情（涉及获取，检查，释放锁等动作），因此数据库系统需要在高并发响应和系统性能两方面进行平衡，这样就产生了“锁粒度（Lock granularity）”的概念。

一种提高共享资源并发发性的方式是让锁定对象更有选择性。尽量只锁定需要修改的部分数据，而不是所有的资源。更理想的方式是，只对会修改的数据片进行精确的锁定。任何时候，在给定的资源上，锁定的数据量越少，则系统的并发程度越高，只要相互之间不发生冲突即可。

## 4.3、三锁

偏向MyISAM存储引擎，开销小，加锁快；无死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。

### 4.3.1、表锁

#### 加读锁

|  |  |
| --- | --- |
| Session\_1 | Session\_2 |
| 获得mylock的read锁定  lock table mylock read; | 连接终端 |
| 当前session可以查询锁定表的记录  Select \* from mylock | 其他的session可以查询该锁定表的记录  Select \* from mylock |
| 当前session不能查询其他没有锁定的表  Select \* from book; | 其他session可以查询或者更新没有锁定的表  Select \* from book;  Update book… |
| 当前session插入或者更新锁定的表都会提示错误 | 其他session插入或者更新锁定表的时候会一直等待 |
| 释放锁 unlock tables; | Session2完成等待的操作 |

#### 加写锁

|  |  |
| --- | --- |
| Session\_1 | Session\_2 |
| Lock table mylock write; | 待session1开启了写锁之后，再连接终端 |
| 当前session对锁定表的查询、更新、插入操作都是可以执行的 | 其他的session对表的查询被阻塞，需要等待锁的释放 |
| 释放锁 unlock tables; | Session获得锁，查询返回 |

MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行增删改操作前，会自动给涉及的表加写锁。

MySQL的表级锁有两种模式：

表共享读锁（Table Read Lock）

表独占写锁（Table Write Lock）

|  |  |  |  |
| --- | --- | --- | --- |
| 锁类型 | 可否兼容 | 读锁 | 写锁 |
| 读锁 | 是 | 是 | 否 |
| 写锁 | 是 | 否 | 否 |

结论：

结合上表，所以对MyISAM表进行操作，会有以下情况：

1、对MyISAM表的读操作（加读锁），不会阻塞其他进程对同一表的读请求，但会阻塞对同一表的写请求。只有当读锁释放后，才会执行其它进程的写操作。

2、对MyISAM表的写操作（加写锁），会阻塞其他进程对同一表的读和写操作，只有当写锁释放后，才会执行其它进程的读写操作。

简而言之，就是读锁会阻塞写，但是不会堵塞读。而写锁则会把读和写都堵塞

### 4.3.2、行锁

偏向InnoDB存储引擎，开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

InnoDB与MyISAM的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁

varchar类型转换可能导致行锁边表锁

间隙锁：

间隙锁带来的插入问题

Session\_1

Session\_2

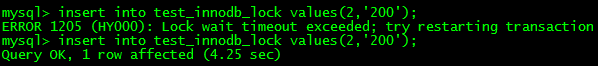
C:\Users\ASUS\AppData\Local\Temp\mindmaster\92965166\001\D5B92B88-DDC0-4CE7-9CC8-7FA5AA5A12DE.png

阻塞产生，暂时不能插入

C:\Users\ASUS\AppData\Local\Temp\mindmaster\92965166\001\270012F4-11B6-4A22-A723-1857619F6B4E.png

commit;

阻塞解除，完成插入



【什么是间隙锁】

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，

InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（GAP Lock）。

【危害】

因为Query执行过程中通过过范围查找的话，他会锁定整个范围内所有的索引键值，即使这个键值并不存在。

间隙锁有一个比较致命的弱点，就是当锁定一个范围键值之后，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定键值范围内的任何数据。在某些场景下这可能会对性能造成很大的危害

如何锁定一行？ select \* from test\_innodb\_lock where a=8 for update;

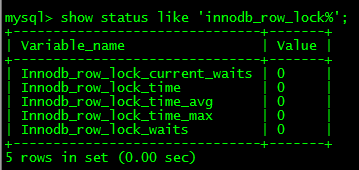
Innodb存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会要更高一些，但是在整体并发处理能力方面要远远优于MyISAM的表级锁定的。当系统并发量较高的时候，Innodb的整体性能和MyISAM相比就会有比较明显的优势了。

但是，Innodb的行级锁定同样也有其脆弱的一面，当我们使用不当的时候，可能会让Innodb的整体性能表现不仅不能比MyISAM高，甚至可能会更差。

【如何分析行锁定】

通过检查InnoDB\_row\_lock状态变量来分析系统上的行锁的争夺情况

mysql>show status like 'innodb\_row\_lock%';



对各个状态量的说明如下：

Innodb\_row\_lock\_current\_waits：当前正在等待锁定的数量；

Innodb\_row\_lock\_time：从系统启动到现在锁定总时间长度；

Innodb\_row\_lock\_time\_avg：每次等待所花平均时间；

Innodb\_row\_lock\_time\_max：从系统启动到现在等待最常的一次所花的时间；

Innodb\_row\_lock\_waits：系统启动后到现在总共等待的次数；

对于这5个状态变量，比较重要的主要是

Innodb\_row\_lock\_time\_avg（等待平均时长），

Innodb\_row\_lock\_waits（等待总次数）

Innodb\_row\_lock\_time（等待总时长）这三项。

尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

最后可以通过

SELECT \* FROM information\_schema.INNODB\_TRX\G;

来查询正在被锁阻塞的sql语句。

