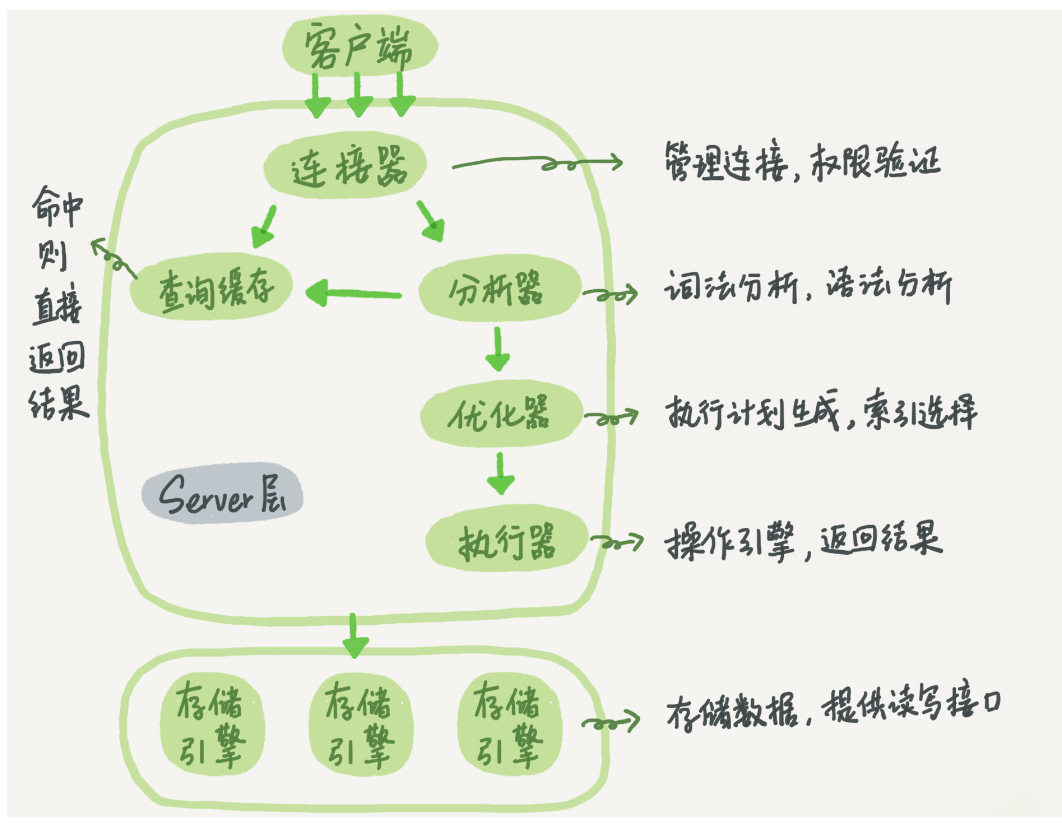
# MySQL执行流程



## MySQL可以分为Server层和存储引擎层两部分

Server层包括连接器、查询缓存、分析器、优化器、执行器等，涵盖MySQL的大多数核心服务功能，以及所有的内置函数（如日期、时间、数学和加密函数等），所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图等。

## 重要的日志模块：redo log

其实就是MySQL里经常说到的WAL技术，WAL的全称是Write-Ahead Logging，它的关键点就是先写日志，再写磁盘，也就是先写粉板，等不忙的时候再写账本。

具体来说，当有一条记录需要更新的时候，InnoDB引擎就会先把记录写到redo log（粉板）里面，并更新内存，这个时候更新就算完成了。同时，InnoDB引擎会在适当的时候，将这个操作记录更新到磁盘里面，而这个更新往往是在系统比较空闲的时候做，这就像打烊以后掌柜做的事。

有了redo log，InnoDB就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力称为crash-safe。

## 重要的日志模块：binlog

这两种日志有以下三点不同。

1.redo log是InnoDB引擎特有的；binlog是MySQL的Server层实现的，所有引擎都可以使用。

2.redo log是物理日志，记录的是“在某个数据页上做了什么修改”；binlog是逻辑日志，记录的是这个语句的原始逻辑，比如“给ID=2这一行的c字段加1 ”。

3.redo log是循环写的，空间固定会用完；binlog是可以追加写入的。“追加写”是指binlog文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

有了对这两个日志的概念性理解，我们再来看执行器和InnoDB引擎在执行这个简单的update语句时的内部流程。

1.执行器先找引擎取ID=2这一行。ID是主键，引擎直接用树搜索找到这一行。如果ID=2这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。

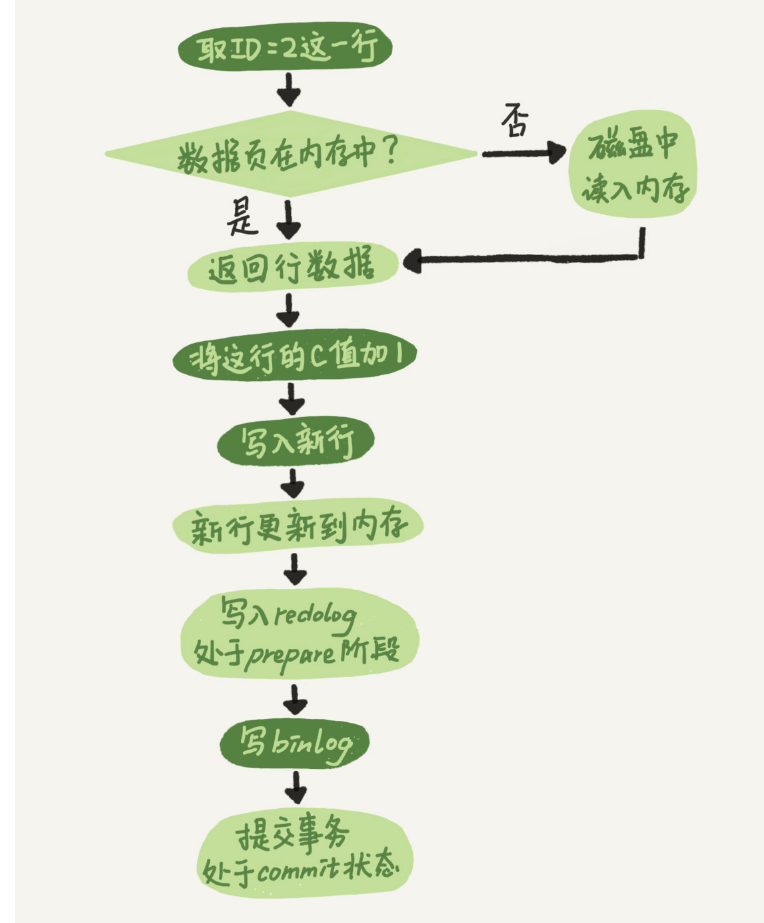
2.执行器拿到引擎给的行数据，把这个值加上1，比如原来是N，现在就是N+1，得到新的一行数据，再调用引擎接口写入这行新数据。

3.引擎将这行新数据更新到内存中，同时将这个更新操作记录到redo log里面，此时redo log处于prepare状态。然后告知执行器执行完成了，随时可以提交事务。

4.执行器生成这个操作的binlog，并把binlog写入磁盘。

5.执行器调用引擎的提交事务接口，引擎把刚刚写入的redo log改成提交（commit）状态，更新完成。

这里我给出这个update语句的执行流程图，图中浅色框表示是在InnoDB内部执行的，深色框表示是在执行器中执行的。



binlog会记录所有的逻辑操作，并且是采用“追加写”的形式。如果你的DBA承诺说半个月内可以恢复，那么备份系统中一定会保存最近半个月的所有binlog，同时系统会定期做整库备份。这里的“定期”取决于系统的重要性，可以是一天一备，也可以是一周一备。

## 两阶段提交

仍然用前面的update语句来做例子。假设当前ID=2的行，字段c的值是0，再假设执行update语句过程中在写完第一个日志后，第二个日志还没有写完期间发生了crash，会出现什么情况呢？

1.先写redo log后写binlog。假设在redo log写完，binlog还没有写完的时候，MySQL进程异常重启。由于我们前面说过的，redo log写完之后，系统即使崩溃，仍然能够把数据恢复回来，所以恢复后这一行c的值是1。

但是由于binlog没写完就crash了，这时候binlog里面就没有记录这个语句。因此，之后备份日志的时候，存起来的binlog里面就没有这条语句。

然后你会发现，如果需要用这个binlog来恢复临时库的话，由于这个语句的binlog丢失，这个临时库就会少了这一次更新，恢复出来的这一行c的值就是0，与原库的值不同。

2.先写binlog后写redo log。如果在binlog写完之后crash，由于redo log还没写，崩溃恢复以后这个事务无效，所以这一行c的值是0。但是binlog里面已经记录了“把c从0改成1”这个日志。所以，在之后用binlog来恢复的时候就多了一个事务出来，恢复出来的这一行c的值就是1，与原库的值不同。

可以看到，如果不使用“两阶段提交”，那么数据库的状态就有可能和用它的日志恢复出来的库的状态不一致。

# 隔离性与隔离级别

## 隔离级别

* 读未提交是指，一个事务还没提交时，它做的变更就能被别的事务看到。
* 读提交是指，一个事务提交之后，它做的变更才会被其他事务看到。
* 可重复读是指，一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的。
* 串行化，顾名思义是对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

## 事务隔离的实现

### 多版本并发控制MVVC

InnoDB的行数据有多个版本，每个数据版本有自己的row trx\_id，每个事务或者语句有自己的一致性视图。普通查询语句是一致性读，一致性读会根据row trx\_id和一致性视图确定数据版本的可见性。

对于可重复读，查询只承认在事务启动前就已经提交完成的数据；

对于读提交，查询只承认在语句启动前就已经提交完成的数据；

### 应该避免长事务

# 深入浅出索引

## 索引的常见模型哈希表、有序数组和搜索树

## InnoDB 的索引模型

InnoDB使用了B+树索引模型

索引类型分为主键索引和非主键索引。

主键索引的叶子节点存的是整行数据。在InnoDB里，主键索引也被称为聚簇索引（clustered index）。

非主键索引的叶子节点内容是主键的值。在InnoDB里，非主键索引也被称为二级索引（secondary index）。

### 回表

## 索引维护

### 数据页

### 主键长度越小，普通索引的叶子节点就越小，普通索引占用的空间也就越小

### 覆盖索引

覆盖索引是指，索引上的信息足够满足查询请求，不需要再回到主键索引上去取数据。

由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段，避免回表

Using index：查询的列被索引覆盖，并且where筛选条件是索引的是前导列

Using where Using index：查询的列被索引覆盖，并且where筛选条件是索引列之一但是不是索引的不是前导列，意味着无法直接通过索引查找来查询到符合条件的数据

Using where：查询的列未被索引覆盖，where筛选条件非索引的前导列

Using index condition：查询列不完全被索引覆盖，查询条件完全可以使用到索引

### 最左前缀原则

### 索引下推

### 唯一索引和普通索引

### 索引优化

### 怎么给字符串加索引

1直接创建完整索引，这样可能比较占用空间；

2创建前缀索引，节省空间，但会增加查询扫描次数，并且不能使用覆盖索引；

3倒序存储，再创建前缀索引，用于绕过字符串本身前缀的区分度不够的问题；

4创建hash字段索引，查询性能稳定，有额外的存储和计算消耗，跟第三种方式一样，都不支持范围扫描。

# 锁

## 全局锁

全局锁就是对整个数据库实例加锁。MySQL提供了一个加全局读锁的方法，命令是 Flush tables with read lock (FTWRL)。当你需要让整个库处于只读状态的时候，可以使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。

全局锁的典型使用场景是，做全库逻辑备份

危险：

如果你在主库上备份，那么在备份期间都不能执行更新，业务基本上就得停摆；

如果你在从库上备份，那么备份期间从库不能执行主库同步过来的binlog，会导致主从延迟。

一致性读是好，但前提是引擎要支持这个隔离级别。比如，对于MyISAM这种不支持事务的引擎，如果备份过程中有更新，总是只能取到最新的数据，那么就破坏了备份的一致性。这时，我们就需要使用FTWRL命令了

既然要全库只读，为什么不使用set global readonly=true的方式呢

用FTWRL方式，主要有两个原因：

一是，在有些系统中，readonly的值会被用来做其他逻辑，比如用来判断一个库是主库还是备库。因此，修改global变量的方式影响面更大，我不建议你使用。

二是，在异常处理机制上有差异。如果执行FTWRL命令之后由于客户端发生异常断开，那么MySQL会自动释放这个全局锁，整个库回到可以正常更新的状态。而将整个库设置为readonly之后，如果客户端发生异常，则数据库就会一直保持readonly状态，这样会导致整个库长时间处于不可写状态，风险较高。

## 表级锁

### 表锁

MySQL里面表级别的锁有两种：一种是表锁，一种是元数据锁

表锁的语法是 lock tables … read/write。与FTWRL类似，可以用unlock tables主动释放锁，也可以在客户端断开的时候自动释放。需要注意，lock tables语法除了会限制别的线程的读写外，也限定了本线程接下来的操作对象。

举个例子, 如果在某个线程A中执行lock tables t1 read, t2 write; 这个语句，则其他线程写t1、读写t2的语句都会被阻塞。同时，线程A在执行unlock tables之前，也只能执行读t1、读写t2的操作。连写t1都不允许，自然也不能访问其他表。

### 元数据锁（meta data lock，MDL)

MDL不需要显式使用，在访问一个表的时候会被自动加上。MDL的作用是，保证读写的正确性。你可以想象一下，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，删了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，在MySQL 5.5版本中引入了MDL，当对一个表做增删改查操作的时候，加MDL读锁；当要对表做结构变更操作的时候，加MDL写锁。

读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。

读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性。因此，如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。

### 如何安全地给小表加字段

ALTER TABLE tbl\_name NOWAIT add column ...

ALTER TABLE tbl\_name WAIT N add column ...

## 行锁

MySQL的行锁是在引擎层由各个引擎自己实现的。但并不是所有的引擎都支持行锁，比如MyISAM引擎就不支持行锁。不支持行锁意味着并发控制只能使用表锁，对于这种引擎的表，同一张表上任何时刻只能有一个更新在执行，这就会影响到业务并发度。InnoDB是支持行锁的，这也是MyISAM被InnoDB替代的重要原因之一。

### 两阶段锁协议

在InnoDB事务中，行锁是在需要的时候才加上的，但并不是不需要了就立刻释放，而是要等到事务结束时才释放。这个就是两阶段锁协议。

### 死锁和死锁检测

当并发系统中不同线程出现循环资源依赖，涉及的线程都在等待别的线程释放资源时，就会导致这几个线程都进入无限等待的状态，称为死锁。

当出现死锁以后，有两种策略：

一种策略是，直接进入等待，直到超时。这个超时时间可以通过参数innodb\_lock\_wait\_timeout来设置。

另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务，让其他事务得以继续执行。将参数innodb\_deadlock\_detect设置为on，表示开启这个逻辑。

# 常用语句原理

## count(\*)

MyISAM表虽然count(\*)很快，但是不支持事务；

show table status命令虽然返回很快，但是不准确；

InnoDB表直接count(\*)会遍历全表，虽然结果准确，但会导致性能问题。

### 用缓存系统保存计数

### 在数据库保存计数

按照效率排序的话，count(字段)<count(主键id)<count(1)≈count(\*)，所以我建议你，尽量使用count(\*)。

## Orderby

### 全字段排序

select city,name,age from t where city='杭州' order by name limit 1000 ;执行流程

1初始化sort\_buffer，确定放入name、city、age这三个字段；

2从索引city找到第一个满足city='杭州’条件的主键id，也就是图中的ID\_X；

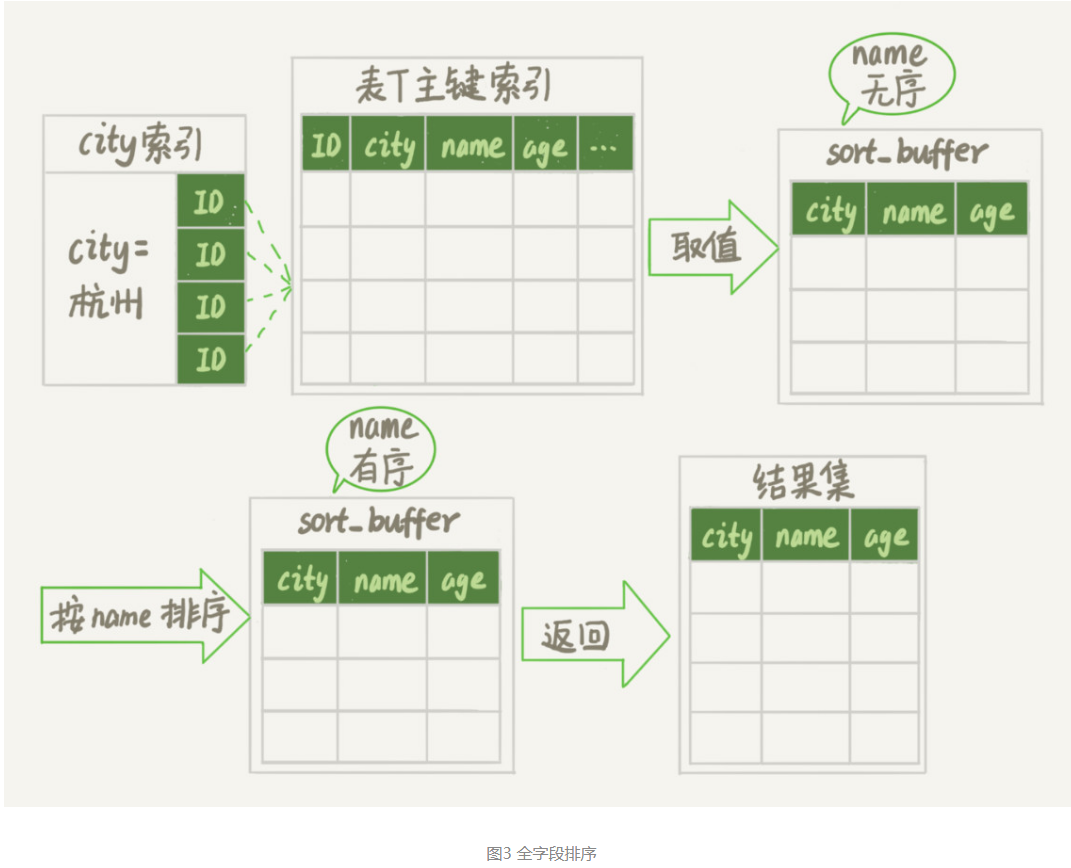
3到主键id索引取出整行，取name、city、age三个字段的值，存入sort\_buffer中；

4从索引city取下一个记录的主键id；

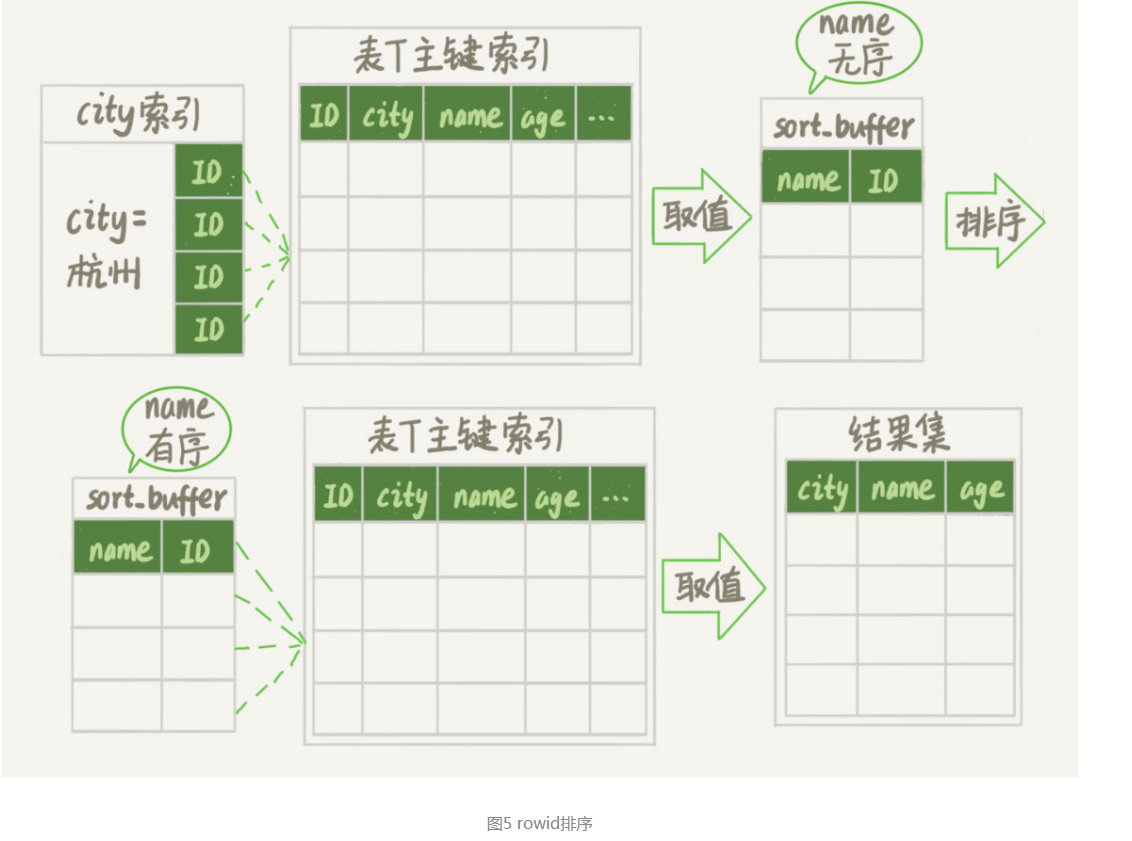
5重复步骤3、4直到city的值不满足查询条件为止，对应的主键id也就是图中的ID\_Y；

6对sort\_buffer中的数据按照字段name做快速排序；

7按照排序结果取前1000行返回给客户端。



### rowid排序



### 内存临时表

order by rand()使用了内存临时表，内存临时表排序的时候使用了rowid排序方法

### 磁盘临时表

那么，是不是所有的临时表都是内存表呢？其实不是的。tmp\_table\_size这个配置限制了内存临时表的大小，默认值是16M。如果临时表大小超过了tmp\_table\_size，那么内存临时表就会转成磁盘临时表。

磁盘临时表使用的引擎默认是InnoDB，是由参数internal\_tmp\_disk\_storage\_engine控制的。

当使用磁盘临时表的时候，对应的就是一个没有显式索引的InnoDB表的排序过程。

### 优先队列排序

### 随机排序方法

## Join

### Index Nested-Loop Join

select \* from t1 straight\_join t2 on (t1.a=t2.a);

可以看到，在这条语句里，被驱动表t2的字段a上有索引，join过程用上了这个索引，因此这个语句的执行流程是这样的：

1从表t1中读入一行数据 R；

2从数据行R中，取出a字段到表t2里去查找；

3取出表t2中满足条件的行，跟R组成一行，作为结果集的一部分；

4重复执行步骤1到3，直到表t1的末尾循环结束。

这个过程是先遍历表t1，然后根据从表t1中取出的每行数据中的a值，去表t2中查找满足条件的记录。在形式上，这个过程就跟我们写程序时的嵌套查询类似，并且可以用上被驱动表的索引，所以我们称之为“Index Nested-Loop Join”，简称NLJ。

通过上面的分析我们得到了两个结论：

1使用join语句，性能比强行拆成多个单表执行SQL语句的性能要好；

2如果使用join语句的话，需要让小表做驱动表。

### Simple Nested-Loop Join

select \* from t1 straight\_join t2 on (t1.a=t2.b);

由于表t2的字段b上没有索引，因此再用图2的执行流程时，每次到t2去匹配的时候，就要做一次全表扫描。

### Block Nested-Loop Join

这时候，被驱动表上没有可用的索引，算法的流程是这样的：

1把表t1的数据读入线程内存join\_buffer中，由于我们这个语句中写的是select \*，因此是把整个表t1放入了内存；

2扫描表t2，把表t2中的每一行取出来，跟join\_buffer中的数据做对比，满足join条件的，作为结果集的一部分返回。

在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，过滤完成之后，计算参与join的各个字段的总数据量，数据量小的那个表，就是“小表”，应该作为驱动表

## Join优化

### Multi-Range Read优化(顺序读盘)

就是MRR优化的设计思路。此时，语句的执行流程变成了这样：

1根据索引a，定位到满足条件的记录，将id值放入read\_rnd\_buffer中;

2将read\_rnd\_buffer中的id进行递增排序；

3排序后的id数组，依次到主键id索引中查记录，并作为结果返回。

### Batched Key Access

## 查询缓慢的原因

### 如果对字段做了函数计算，就用不上索引了，这是MySQL的规定

select count(\*) from tradelog where month(t\_modified)=7;

### 隐式类型转换

在MySQL中，字符串和数字做比较的话，是将字符串转换成数字。

### 隐式字符编码转换

字符编码子集转换为父级再比较

# 临时表

## 临时表的特性

临时表在使用上有以下几个特点：

1建表语法是create temporary table …。

2一个临时表只能被创建它的session访问，对其他线程不可见。所以，图中session A创建的临时表t，对于session B就是不可见的。

3临时表可以与普通表同名。

4session A内有同名的临时表和普通表的时候，show create语句，以及增删改查语句访问的是临时表。

5show tables命令不显示临时表。

由于临时表只能被创建它的session访问，所以在这个session结束的时候，会自动删除临时表。也正是由于这个特性，临时表就特别适合我们文章开头的join优化这种场景。

## union 执行流程

union all

## group by 执行流程

### group by 优化方法 --索引

alter table t1 add column z int generated always as(id % 100), add index(z);

select z, count(\*) as c from t1 group by z;

### group by优化方法 --直接排序

select SQL\_BIG\_RESULT id%100 as m, count(\*) as c from t1 group by m;

这个语句涉及的数据量很大，请直接用磁盘临时表