

11 MySQL之锁、事务、优化、OLAP、OLTP

MySQL之锁、事务、优化、OLAP、OLTP

本节目录

- [一 锁的分类及特性](#)
- [二 表级锁定\(MyISAM举例\)](#)
- [三 行级锁定](#)
- [四 查看死锁、解除锁](#)
- [五 事务](#)
- [六 慢日志、执行计划、sql优化](#)
- [七 OLTP与OLAP的介绍和对比](#)
- [八 关于autocommit的测试](#)
-

一 锁的分类及特性

数据库锁定机制简单来说，就是数据库为了保证数据的一致性，而使各种共享资源在被并发访问变得有序所设计的一种规则。对于任何一种数据库来说都需要有相应的锁定机制，所以MySQL自然也不能例外。MySQL数据库由于其自身架构的特点，存在多种数据存储引擎，每种存储引擎所针对的应用场景特点都不太一样，为了满足各自特定应用场景的需求，每种存储引擎的锁定机制都是为各自所面对的特定场景而优化设计，所以各存储引擎的锁定机制也有较大区别。MySQL各存储引擎使用了三种类型（级别）的锁定机制：表级锁定，行级锁定和页级锁定。

1.表级锁定 (table-level)

表级别的锁定是MySQL各存储引擎中最大颗粒度的锁定机制。该锁定机制最大的特点是实现逻辑非常简单，带来的系统负面影响最小。所以获取锁和释放锁的速度很快。由于表级锁一次会将整个表锁定，所以可以很好的避免困扰我们的死锁问题。

当然，锁定颗粒度大所带来最大的负面影响就是出现锁定资源争用的概率也会最高，致使并度大打折扣。

使用表级锁定的主要是MyISAM，MEMORY，CSV等一些非事务性存储引擎。

2.行级锁定 (row-level)

行级锁定最大的特点就是锁定对象的颗粒度很小，也是目前各大数据库管理软件所实现的锁定颗粒度最小的。由于锁定颗粒度很小，所以发生锁定资源争用的概率也最小，能够给予应用程序尽可能大的并发处理能力而提高一些需要高并发应用系统的整体性能。

虽然能够在并发处理能力上面有较大的优势，但是行级锁定也因此带来了不少弊端。由于锁定资源的颗粒度很小，所以每次获取锁和释放锁需要做的事情也更多，带来的消耗自然也就更大了。此外，行级锁定也最容易发生死锁。

使用行级锁定的主要是InnoDB存储引擎。

3.页级锁定 (page-level)

页级锁定是MySQL中比较独特的一种锁定级别，在其他数据库管理软件中也并不是太常见。页级锁定的特点是锁定颗粒度介于行级锁定与表级锁之间，所以获取锁定所需要的资源开销，以及所能提供的并发处理能力也同样介于上面二者之间。另外，页级锁定和行级锁定一样，会发生死锁。

在数据库实现资源锁定的过程中，随着锁定资源颗粒度的减小，锁定相同数据量的数据所需要消耗的内存数量是越来越多的，实现算法也会越来越复杂。不过，随着锁定资源颗粒度的减小，应用程序的访问请求遇到锁等待的可能性也会随之降低，系统整体并发度也随之提升。

使用页级锁定的主要是BerkeleyDB存储引擎。

总的来说，MySQL这3种锁的特性可大致归纳如下：

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低；

行级锁：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；

页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

适用：从锁的角度来说，表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web应用；而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLTP）系统。

二 表级锁定(MyISAM举例)

由于MyISAM存储引擎使用的锁定机制完全是由MySQL提供的表级锁定实现，所以下面我们将以MyISAM存储引擎作为示例存储引擎。

1. MySQL表级锁的锁模式

MySQL的表级锁有两种模式：表共享读锁（Table Read Lock）和表独占写锁（Table Write Lock）。锁模式的兼容性：

对MyISAM表的读操作，不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求；

对MyISAM表的写操作，则会阻塞其他用户对同一表的读和写操作；

MyISAM表的读操作与写操作之间，以及写操作之间是串行的。当一个线程获得对一个表的写锁后，只有持有锁的线程可以对表进行更新操作。其他线程的读、写操作都会等待，直到锁被释放为止。

总结：表锁，读锁会阻塞写，不会阻塞读。而写锁则会把读写都阻塞。

2. 如何加表锁

MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT等）前，会自动给涉及表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用LOCK TABLE命令给MyISAM表显式加锁。

显示加锁：

共享读锁：lock table tableName read;

独占写锁：lock table tableName write;

同时加多锁：lock table t1 write,t2 read;

批量解锁：unlock tables;

3. MyISAM表锁优化建议

对于MyISAM存储引擎，虽然使用表级锁定在锁定实现的过程中比实现行级锁定或者页级锁所带来的附加成本都要小，锁定本身所消耗的资源也是最少。但是由于锁定的颗粒度比较粗，所以造成锁定资源的争用情况也会比其他的锁定级别都要多，从而在较大程度上会降低并发处理能力。所以，在优化MyISAM存储引擎锁定时，最关键的就是如何让其提高并发度。由于锁定级别是不可能改变的了，所以我们首先需要尽可能让锁定的时间变短，然后就是让可能并发进行的操作尽可能的并发。

(1) 查询表级锁争用情况

MySQL内部有两组专门的状态变量记录系统内部锁资源争用情况：



```
mysql> show status like 'table%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Table_locks_immediate | 100 |
| Table_locks_waited    | 11 |
+-----+-----+
```



这里有两个状态变量记录MySQL内部表级锁定的情况，两个变量说明如下：

Table_locks_immediate：产生表级锁定的次数；

Table_locks_waited：出现表级锁定争用而发生等待的次数；此值越高则说明存在着越严重的表级锁争用情况。此外，MyISAM的读写锁调度是写优先，这也是MyISAM不适合做写为主表的存储引擎。因为写锁后，其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永久阻塞。

两个状态值都是从系统启动后开始记录，出现一次对应的事件则数量加1。如果这里的Table_locks_waited状态值比较高，那么说明系统中表级锁定争用现象比较严重，就需要进一步分析为什么会有较多的锁定资源争用了。

(2) 缩短锁定时间

如何让锁定时间尽可能的短呢？唯一的办法就是让我们的Query执行时间尽可能的短。

a) 尽两减少大的复杂Query，将复杂Query分拆成几个小的Query分布进行；

b) 尽可能的建立足够高效的索引，让数据检索更迅速；

c) 尽量让MyISAM存储引擎的表只存放必要的信息，控制字段类型；

d) 利用合适的机会优化MyISAM表数据文件。

(3) 分离能并行的操作

说到MyISAM的表锁，而且是读写互相阻塞的表锁，可能有些人会认为在MyISAM存储引擎的表上就只能是完全的串行化，没办法再并行了。大家不要忘记了，MyISAM的存储引擎还有一个非常有用的特性，那就是ConcurrentInsert（并发插入）的特性。

MyISAM存储引擎有一个控制是否打开Concurrent Insert功能的参数选项：concurrent_insert，可以设置为0，1或者2。三个值的具体说明如下：

concurrent_insert=2，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录；

concurrent_insert=1，如果MyISAM表中没有空洞（即表的中间没有被删除的行），MyISAM允许在一个进程读表的同时，另一个进程从表尾插入记录。这也是MySQL的默认设置；

concurrent_insert=0，不允许并发插入。

可以利用MyISAM存储引擎的并发插入特性，来解决应用中对同一表查询和插入的锁争用。例如，将concurrent_insert系统变量设为2，总是允许并发插入；同时，通过定期在系统空闲时段执行OPTIMIZE TABLE语句来整理空间碎片，收回因删除记录而产生的中间空洞。

（4）合理利用读写优先级

MyISAM存储引擎的是读写互相阻塞的，那么，一个进程请求某个MyISAM表的读锁，同时另一个进程也请求同一表的写锁，MySQL如何处理呢？

答案是写进程先获得锁。不仅如此，即使读请求先到锁等待队列，写请求后到，写锁也会插到读锁请求之前。

这是因为MySQL的表级锁定对于读和写是有不同优先级设定的，默认情况下是写优先级要大于读优先级。

所以，如果我们可以根据各自系统环境的差异决定读与写的优先级：

通过执行命令SET LOW_PRIORITY_UPDATES=1，使该连接读比写的优先级高。如果我们的系统是一个以读为主，可以设置此参数，如果以写为主，则不用设置；

通过指定INSERT、UPDATE、DELETE语句的LOW_PRIORITY属性，降低该语句的优先级。

虽然上面方法都是要么更新优先，要么查询优先的方法，但还是可以用其来解决查询相对重要的应用（如用户登录系统）中，读锁等待严重的问题。

另外，MySQL也提供了一种折中的办法来调节读写冲突，即给系统参数max_write_lock_count设置一个合适的值，当一个表的读锁达到这个值后，MySQL就暂时将写请求的优先级降低，给读进程一定获得锁的机会。

这里还要强调一点：一些需要长时间运行的查询操作，也会使写进程“饿死”，因此，应用中应尽量避免出现长时间运行的查询操作，不要总想用一条SELECT语句来解决问题，因为这种看似巧妙的SQL语句，往往比较复杂，执行时间较长，在可能的情况下可以通过使用中间表等措施对SQL语句做一定的“分解”，使每一步查询都能在较短时间完成，从而减少锁冲突。如果复杂查询不可避免，应尽量安排在数据库空闲时段执行，比如一些定期统计可以安排在夜间执行。

InnoDB默认采用行锁，在未使用索引字段查询时升级为表锁。MySQL这样设计并不是给你挖坑。它有自己的设计目的。

即便你在条件中使用了索引字段，MySQL会根据自身的执行计划，考虑是否使用索引（所以explain命令中会有possible_key 和 key）。如果MySQL认为全表扫描效率更高，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。**关于执行计划**

第一种情况：**全表更新**。事务需要更新大部分或全部数据，且表又比较大。若使用行锁，会导致事务执行效率低，从而可能造成其他事务长时间锁等待和更多的锁冲突。

第二种情况：**多表级联**。事务涉及多个表，比较复杂的关联查询，很可能引起死锁，造成大量事务回滚。这种情况若能一次性锁定事务涉及的表，从而可以避免死锁、减少数据库因事务回滚带来的开销。

三 行级锁定

行级锁定不是MySQL自己实现的锁定方式，而是由其他存储引擎自己所实现的，如广为大家所知的InnoDB存储引擎，以及MySQL的分布式存储引擎NDBCluster等都是实现了行级锁定。考虑到行级锁定由各个存储引擎自行实现，而且具体实现也各有差别，而InnoDB是目前事务型存储引擎中使用最为广泛的存储引擎，所以这里我们就主要分析一下InnoDB的锁定特性。

1. InnoDB锁定模式及实现机制

考虑到行级锁定由各个存储引擎自行实现，而且具体实现也各有差别，而InnoDB是目前事务型存储引擎中使用最为广泛的存储引擎，所以这里我们就主要分析一下InnoDB的锁定特性。

总的来说，InnoDB的锁定机制和Oracle数据库有不少相似之处。InnoDB的行级锁定同样分为两种类型，共享锁和排他锁，而在锁定机制的实现过程中为了让行级锁定和表级锁定共存，InnoDB也同样使用了意向锁（表级锁定）的概念，也就有了意向共享锁和意向排他锁这两种。

当一个事务需要给自己需要的某个资源加锁的时候，如果遇到一个共享锁正锁定着自己需要的资源的时候，自己可以再加一个共享锁，不过不能加排他锁。但是，如果遇到自己需要锁定的资源已经被一个排他锁占有之后，则只能等待

该锁定释放资源之后自己才能获取锁定资源并添加自己的锁定。而意向锁的作用就是当一个事务在需要获取资源锁定的时候，如果遇到自己需要的资源已经被排他锁占用的时候，该事务可以需要锁定行的表上面添加一个合适的意向锁。如果自己需要一个共享锁，那么就在表上面添加一个意向共享锁。而如果自己需要的是某行（或者某些行）上面添加一个排他锁的话，则先在表上面添加一个意向排他锁。意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在。所以，可以说InnoDB的锁定模式实际上可以分为四种：共享锁（S），排他锁（X），意向共享锁（IS）和意向排他锁（IX），我们可以通过以下表格来总结上面这四种锁的共存逻辑关系：

	共享锁（S）	排他锁（X）	意向共享锁（IS）	意向排他锁（IX）
共享锁（S）	兼容	冲突	兼容	冲突
排他锁（X）	冲突	冲突	冲突	冲突
意向共享锁（IS）	兼容	冲突	兼容	兼容
意向排他锁（IX）	冲突	冲突	兼容	兼容

如果一个事务请求的锁模式与当前的锁兼容，InnoDB就将请求的锁授予该事务；反之，如果两者不兼容，该事务就要等待锁释放。

意向锁是InnoDB自动加的，不需用户干预。对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X）；对于普通SELECT语句，InnoDB不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。

共享锁（S）：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE
排他锁（X）：SELECT * FROM table_name WHERE ... FOR UPDATE

用SELECT ... IN SHARE MODE获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行UPDATE或者DELETE操作。

但是如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用SELECT... FOR UPDATE方式获得排他锁。

2. InnoDB行锁实现方式

InnoDB行锁是通过给索引上的索引项加锁来实现的，只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁

在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。下面通过一些实际例子来加以说明。

- （1）在不通过索引条件查询的时候，InnoDB确实使用的是表锁，而不是行锁。
- （2）由于MySQL的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。
- （3）当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行，另外，不论是使用主键索引、唯一索引或普通索引，InnoDB都会使用行锁来对数据加锁。
- （4）即便在条件中使用了索引字段，但是否使用索引来检索数据是由MySQL通过判断不同执行计划的代价来决定的，如果MySQL认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。

3. 间隙锁（Next-Key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（Next-Key锁）。

例：

假如emp表中只有101条记录，其empid的值分别是 1,2,...,100,101，下面的SQL：

```
mysql> select * from emp where empid > 100 for update;
```

是一个范围条件的检索，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁。

InnoDB使用间隙锁的目的：

（1）防止幻读，以满足相关隔离级别的要求（关于事务的隔离级别）。对于上面的例子，要是不使用间隙锁，如果其他事务插入了empid大于100的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；

（2）为了满足其恢复和复制的需要。

很显然，在使用范围条件检索并锁定记录时，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。

除了间隙锁给InnoDB带来性能的负面影响之外，通过索引实现锁定的方式还存在其他几个较大的性能隐患：

（1）当Query无法利用索引的时候，InnoDB会放弃使用行级别锁定而改用表级别的锁定，造成并发性能的降低；

(2) 当Query使用的索引并不包含所有过滤条件的时候, 数据检索使用到的索引键所只想要的部分并不属于该Query的结果集的行列, 但是也会被锁定, 因为间隙锁锁定的是一个范围, 而不是具体的索引键;

(3) 当Query在使用索引定位数据的时候, 如果使用的索引键一样但访问的数据行不同的时候 (索引只是过滤条件的一部分), 一样会被锁定。

因此, 在实际应用开发中, 尤其是并发插入比较多的应用, 我们要尽量优化业务逻辑, 尽量使用相等条件来访问更新数据, 避免使用范围条件。

还要特别说明的是, InnoDB除了通过范围条件加锁时使用间隙锁外, 如果使用相等条件请求给一个不存在的记录加锁, InnoDB也会使用间隙锁。

4. 死锁

上文讲过, MyISAM表锁是deadlock free的, 这是因为MyISAM总是一次获得所需的全部锁, 要么全部满足, 要么等待, 因此不会出现死锁。但在InnoDB中, 除单个SQL组成的事务外, 锁是逐步获得的, 当两个事务都需要获得对方持有的排他锁才能继续完成事务, 这种循环锁等待就是典型的死锁。

在InnoDB的事务管理和锁定机制中, 有专门检测死锁的机制, 会在系统中产生死锁之后的很短时间就检测到该死锁的存在。当InnoDB检测到系统中产生了死锁之后, InnoDB会通过相应的判断来选这产生死锁的两个事务中较小的事务来回滚, 而让另外一个较大的事务成功完成。

那InnoDB是以什么来为标准判定事务的大小的呢? MySQL官方手册中也提到了这个问题, 实际上在InnoDB发现死锁之后, 会计算出两个事务各自插入、更新或者删除的数据量来判定两个事务的大小。也就是说哪个事务所改变的记录条数越多, 在死锁中就越不会被回滚掉。

但是有一点需要注意的就是, 当产生死锁的场景中涉及到不止InnoDB存储引擎的时候, InnoDB是没办法检测到该死锁的, 这时候就只能通过锁定超时限制参数InnoDB_lock_wait_timeout来解决。

需要说明的是, 这个参数并不是只用来解决死锁问题, 在并发访问比较高的情况下, 如果大量事务因无法立即获得所需的锁而挂起, 会占用大量计算机资源, 造成严重性能问题, 甚至拖跨数据库。我们通过设置合适的锁等待超时阈值, 可以避免这种情况发生。

通常来说, 死锁都是应用设计的问题, 通过调整业务流程、数据库对象设计、事务大小, 以及访问数据库的SQL语句, 绝大部分死锁都可以避免。下面就通过实例来介绍几种避免死锁的常用方法:

(1) 在应用中, 如果不同的程序会并发存取多个表, 应尽量约定以相同的顺序来访问表, 这样可以大大降低产生死锁的机会。

(2) 在程序以批量方式处理数据的时候, 如果事先对数据排序, 保证每个线程按固定的顺序来处理记录, 也可以大大降低出现死锁的可能。

(3) 在事务中, 如果要更新记录, 应该直接申请足够级别的锁, 即排他锁, 而不应先申请共享锁, 更新后再申请排他锁, 因为当用户申请排他锁时, 其他事务可能又已经获得了相同记录的共享锁, 从而造成锁冲突, 甚至死锁。

(4) 在REPEATABLE-READ隔离级别下, 如果两个线程同时对相同条件记录用SELECT...FOR UPDATE加排他锁, 在没有符合该条件记录情况下, 两个线程都会加锁成功。程序发现记录尚不存在, 就试图插入一条新记录, 如果两个线程都这么做, 就会出现死锁。这种情况下, 将隔离级别改成READ COMMITTED, 就可避免问题。

(5) 当隔离级别为READ COMMITTED时, 如果两个线程都先执行SELECT...FOR UPDATE, 判断是否存在符合条件的记录, 如果没有, 就插入记录。此时, 只有一个线程能插入成功, 另一个线程会出现锁等待, 当第1个线程提交后, 第2个线程会因主键重出错, 但虽然这个线程出错了, 却会获得一个排他锁。这时如果有第3个线程又来申请排他锁, 也会出现死锁。对于这种情况, 可以直接做插入操作, 然后再捕获主键重异常, 或者在遇到主键重错误时, 总是执行ROLLBACK释放获得的排他锁。

5. 什么时候使用表锁

对于InnoDB表, 在绝大部分情况下都应该使用行级锁, 因为事务和行锁往往是我们之所以选择InnoDB表的理由。但在个别特殊事务中, 也可以考虑使用表级锁:

(1) 事务需要更新大部分或全部数据, 表又比较大, 如果使用默认的行锁, 不仅这个事务执行效率低, 而且可能造成其他事务长时间锁等待和锁冲突, 这种情况下可以考虑使用表锁来提高该事务的执行速度。

(2) 事务涉及多个表, 比较复杂, 很可能引起死锁, 造成大量事务回滚。这种情况也可以考虑一次性锁定事务涉及的表, 从而避免死锁、减少数据库因事务回滚带来的开销。

当然, 应用中这两种事务不能太多, 否则, 就应该考虑使用MyISAM表了。

在InnoDB下, 使用表锁要注意以下两点。

(1) 使用LOCK TABLES虽然可以给InnoDB加表级锁, 但必须说明的是, 表锁不是由InnoDB存储引擎层管理的, 而是由其上一层——MySQL Server负责的, 仅当autocommit=0 (不自动提交, 默认是自动提交的)、InnoDB_table_locks=1 (默认设置) 时, InnoDB层才能知道MySQL加的表锁, MySQL Server也才能感知InnoDB加的行锁, 这种情况下, InnoDB才能自动识别涉及表级锁的死锁, 否则, InnoDB将无法自动检测并处理这种死锁。

(2) 在用 LOCK TABLES对InnoDB表加锁时要注意, 要将AUTOCOMMIT设为0, 否则MySQL不会给表加锁; 事务结束前, 不要用UNLOCK TABLES释放表锁, 因为UNLOCK TABLES会隐含地提交事务; COMMIT或ROLLBACK并不能释放用LOCK TABLES加的表级锁, 必须用UNLOCK TABLES释放表锁。正确的方式见如下语句:

例如, 如果需要写表t1并从表t2读, 可以按如下做:

```
SET AUTOCOMMIT=0;
LOCK TABLES t1 WRITE, t2 READ, ...;
[do something with tables t1 and t2 here];
COMMIT;
UNLOCK TABLES;
```


6.InnoDB行锁优化建议

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。

(1) 要想合理利用InnoDB的行级锁定，做到扬长避短，我们必须做好以下工作：

- a)尽可能让所有的数据检索都通过索引来完成，从而避免InnoDB因为无法通过索引键加锁而升级为表级锁定；
- b)合理设计索引，让InnoDB在索引键上加锁的时候尽可能准确，尽可能的缩小锁定范围，避免造成不必要的锁定而影响其他Query的执行；
- c)尽可能减少基于范围的数据检索过滤条件，避免因为间隙锁带来的负面影响而锁定了不该锁定的记录；
- d)尽量控制事务的大小，减少锁定的资源量和锁定时间长度；
- e)在业务环境允许的情况下，尽量使用较低级别的事务隔离，以减少MySQL因为实现事务隔离级别所带来的附加成本。

(2) 由于InnoDB的行级锁定和事务性，所以肯定会产生死锁，下面是一些比较常用的减少死锁产生概率的小建议：

- a)类似业务模块中，尽可能按照相同的访问顺序来访问，防止产生死锁；
- b)在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；
- c)对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率。

(3) 可以通过检查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%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| InnoDB_row_lock_current_waits | 0      |
| InnoDB_row_lock_time          | 0      |
| InnoDB_row_lock_time_avg     | 0      |
| InnoDB_row_lock_time_max     | 0      |
| InnoDB_row_lock_waits        | 0      |
+-----+-----+
```



InnoDB 的行级锁定状态变量不仅记录了锁定等待次数，还记录了锁定总时长，每次平均时长，以及最大时长，此外还有一个非累积状态量显示了当前正在等待锁定的等待数量。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）以及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。

如果发现锁争用比较严重，如InnoDB_row_lock_waits和InnoDB_row_lock_time_avg的值比较高，还可以通过设置InnoDB Monitors 来进一步观察发生锁冲突的表、数据行等，并分析锁争用的原因。

锁冲突的表、数据行等，并分析锁争用的原因。具体方法如下：

```
mysql> create table InnoDB_monitor(a INT) engine=InnoDB;
```

然后就可以用下面的语句来进行查看：

```
mysql> show engine InnoDB status;
```

监视器可以通过发出下列语句来停止查看：

```
mysql> drop table InnoDB_monitor;
```

设置监视器后，会有详细的当前锁等待的信息，包括表名、锁类型、锁定记录的情况等，便于进行进一步的分析和问题的确定。可能会有读者朋友问为什么要先创建一个叫InnoDB_monitor的表呢？因为创建该表实际上就是告诉InnoDB我们开始要监控他的细节状态了，然后InnoDB就会将比较详细的事务以及锁定信息记录进入MySQL的errorlog中，以便我们后面做进一步分析使用。打开监视器以后，默认情况下每15秒会向日志中记录监控的内容，如果长时间打开会导致.err文件变得非常的巨大，所以用户在确认问题原因之后，要记得删除监控表以关闭监视器，或者通过使用“--console”选项来启动服务器以关闭写日志文件。

四 查看死锁、解除锁

结合上面对表锁和行锁的分析情况，解除正在死锁的状态有两种方法：

第一种：

1. 查询是否锁表

```
show OPEN TABLES where In_use > 0;
```

2. 查询进程（如果您有SUPER权限，您可以看到所有线程。否则，您只能看到您自己的线程）

```
show processlist
```

3. 杀死进程id（就是上面命令的id列）

```
kill id
```

第二种：

1. 查看下在锁的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;
```

2. 杀死进程id（就是上面命令的trx_mysql_thread_id列）

```
kill 线程ID
```

例子：

```
查出死锁进程：SHOW PROCESSLIST
```

```
杀掉进程      KILL 420821;
```

其它关于查看死锁的命令：

1：查看当前的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;
```

2：查看当前锁定的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
```

3：查看当前等锁的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;
```

五 事务

1. MySQL 事务属性

事务是由一组SQL语句组成的逻辑处理单元，事务具有ACID属性。

原子性（Atomicity）：事务是一个原子操作单元。在当时原子是不可分割的最小元素，其对数据的修改，要么全部成功，要么全部都不成功。

- 一致性** (Consistent)：事务开始到结束的时间段内，数据都必须保持一致状态。
- 隔离性** (Isolation)：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的"独立"环境执行。
- 持久性** (Durable)：事务完成后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

2.事务常见问题

更新丢失 (Lost Update)

原因：当多个事务选择同一行操作，并且都是基于最初选定的值，由于每个事务都不知道其他事务的存在，就会发生更新覆盖的问题。类比github提交冲突。

脏读 (Dirty Reads)

原因：事务A读取了事务B已经修改但尚未提交的数据。若事务B回滚数据，事务A的数据存在不一致性的问题。

不可重复读 (Non-Repeatable Reads)

原因：事务A第一次读取最初数据，第二次读取事务B已经提交的修改或删除数据。导致两次读取数据不一致。不符合事务的隔离性。

幻读 (Phantom Reads)

原因：事务A根据相同条件第二次查询到事务B提交的新增数据，两次数据结果集不一致。不符合事务的隔离性。

幻读和脏读有点类似
脏读是事务B里面修改了数据，
幻读是事务B里面新增了数据。

3.事务的隔离级别

数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大。这是因为事务隔离实质上是事务在一定程度上"串行"进行，这显然与"并发"是矛盾的。根据自己的业务逻辑，权衡能接受的最大副作用。从而平衡了"隔离"和"并发"的问题。MySQL默认隔离级别是可重复读。

脏读，不可重复读，幻读，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。



隔离级别	读数据一致性	脏读	不可重复读	幻读
未提交读(Read uncommitted)	最低级别	是	是	是
已提交读(Read committed)	语句级	否	是	是
可重复读(Repeatable read)	事务级	否	否	是
可序列化(Serializable)	最高级别，事务级	否	否	否



查看当前数据库的事务隔离级别：show variables like 'tx_isolation';

```
mysql> show variables like 'tx_isolation';
+-----+-----+
| Variable_name | Value          |
+-----+-----+
| tx_isolation  | REPEATABLE-READ |
+-----+-----+
```

4.事务级别的设置



1.未提交读 (READ UNCOMMITTED) 解决的障碍：无；引入的问题：脏读
set SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

- 2.已提交读 (READ COMMITTED) 解决的障碍: 脏读; 引入的问题: 不可重复读
set SESSION TRANSACTION ISOLATION LEVEL read committed;
- 3.可重复读 (REPEATABLE READ) 解决的障碍: 不可重复读; 引入的问题:
set SESSION TRANSACTION ISOLATION LEVEL repeatable read;
- 4.可串行化 (SERIALIZABLE) 解决的障碍: 可重复读; 引入的问题: 锁全表, 性能低下
set SESSION TRANSACTION ISOLATION LEVEL repeatable read;



总结:

事务隔离级别为可重复读时, 如果有索引 (包括主键索引) 的时候, 以索引列为条件更新数据, 会存在间隙锁间、行锁、页锁的问题, 从而锁住一些行; 如果没有索引, 更新数据时会锁住整张表

事务隔离级别为串行化时, 读写数据都会锁住整张表

隔离级别越高, 越能保证数据的完整性和一致性, 但是对并发性能的影响也越大, 对于多数应用程序, 可以优先考虑把数据库系统的隔离级别设为Read Committed, 它能够避免脏读取, 而且具有较好的并发性能。

5.事务保存点, 实现部分回滚

我们可以在[mysql事务处理](#)过程中定义保存点(SAVEPOINT), 然后回滚到指定的保存点前的状态。

定义保存点, 以及回滚到指定保存点前状态的语法如下。

- 1.定义保存点---SAVEPOINT 保存点名;
- 2.回滚到指定保存点---ROLLBACK TO SAVEPOINT 保存点名;



1、查看user表中的数据

```
mysql> select * from user;
+-----+-----+-----+-----+
| mid | name | scx | word |
+-----+-----+-----+-----+
| 1 | zhangsan | 0 | NULL |
| 2 | wangwu | 1 | NULL |
+-----+-----+-----+-----+
2 rows in set (0.05 sec)
```

2、mysql事务开始

```
mysql> BEGIN; -- 或者start transaction;
Query OK, 0 rows affected (0.00 sec)
```

3、向表user中插入2条数据

```
mysql> INSERT INTO user VALUES ('3','one','0','');
Query OK, 1 row affected (0.08 sec)
mysql> INSERT INTO user VALUES ('4','two','0','');
Query OK, 1 row affected (0.00 sec)
mysql> select * from user;
+-----+-----+-----+-----+
| mid | name | scx | word |
+-----+-----+-----+-----+
| 1 | zhangsan | 0 | NULL |
| 2 | wangwu | 1 | NULL |
| 3 | one | 0 | |
| 4 | two | 0 | |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

4、指定保存点, 保存点名为test

```
mysql> SAVEPOINT test;
Query OK, 0 rows affected (0.00 sec)
5、向表user中插入第3条数据

mysql> INSERT INTO user VALUES ('5','three','0','');
Query OK, 1 row affected (0.00 sec)
mysql> select * from user;
+-----+-----+-----+-----+
| mid | name | scx | word |
+-----+-----+-----+-----+
| 1 | zhangsan | 0 | NULL |
| 2 | wangwu | 1 | NULL |
| 3 | one | 0 | |
| 4 | two | 0 | |
| 5 | three | 0 | |
+-----+-----+-----+-----+
5 rows in set (0.02 sec)
6、回滚到保存点test
```

```
mysql> ROLLBACK TO SAVEPOINT test;
Query OK, 0 rows affected (0.31 sec)
mysql> select * from user;
+-----+-----+-----+-----+
| mid | name | scx | word |
+-----+-----+-----+-----+
| 1 | zhangsan | 0 | NULL |
| 2 | wangwu | 1 | NULL |
| 3 | one | 0 | |
| 4 | two | 0 | |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```



我们可以看到保存点test以后插入的记录没有显示了，即成功回滚到了定义保存点test前的状态。利用保存点可以实现只提交事务中部分处理的功能。

6 事务控制语句



BEGIN或START TRANSACTION；显式地开启一个事务；
 COMMIT；也可以使用COMMIT WORK，不过二者是等价的。COMMIT会提交事务，并使已对数据库进行的所有修改成为永久；
 ROLLBACK；有可以使用ROLLBACK WORK，不过二者是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改；
 SAVEPOINT identifier；SAVEPOINT允许在事务中创建一个保存点，一个事务中可以有多多个SAVEPOINT；
 RELEASE SAVEPOINT identifier；删除一个事务的保存点，当没有指定的保存点时，执行该语句会抛出一个异常；
 ROLLBACK TO identifier；把事务回滚到标记点；
 SET TRANSACTION；用来设置事务的隔离级别。InnoDB存储引擎提供事务的隔离级别有READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ、SERIALIZABLE。

用 BEGIN, ROLLBACK, COMMIT来实现

BEGIN 开始一个事务

ROLLBACK 事务回滚

COMMIT 事务确认

直接用 SET 来改变 MySQL 的自动提交模式:

SET AUTOCOMMIT=0或者off 禁止自动提交

SET AUTOCOMMIT=1或者on 开启自动提交



六 慢查询、执行计划、sql优化

什么是慢查询

慢查询日志，顾名思义，就是查询慢的日志，是指mysql记录所有执行超过long_query_time参数设定的时间阈值的SQL语句的日志。该日志能为SQL语句的优化带来很好的帮助。默认情况下，慢查询日志是关闭的，要使用慢查询日志功能，首先要开启慢查询日志功能。

慢查询基本配置

slow_query_log 启动停止技术慢查询日志

slow_query_log_file 指定慢查询日志得存储路径及文件（默认和数据文件放一起）

long_query_time 指定记录慢查询日志SQL执行时间得伐值（单位：秒，默认10秒）

log_queries_not_using_indexes 是否记录未使用索引的SQL

log_output 日志存放的地方【TABLE】【FILE】【FILE, TABLE】

配置了慢查询后，它会记录符合条件的SQL

包括：

查询语句

数据修改语句

已经回滚得SQL

实操：

通过下面命令查看下上面的配置：

```
show VARIABLES like '%slow_query_log%'
```

```
show VARIABLES like '%slow_query_log_file%'
```

```
show VARIABLES like '%long_query_time%'
```

```
show VARIABLES like '%log_queries_not_using_indexes%'
```

```
show VARIABLES like 'log_output'
```

```
set global long_query_time=0; ---默认10秒，这里为了演示方便设置为0
```

```
set GLOBAL slow_query_log = 1; --开启慢查询日志
```

```
set global log_output='FILE, TABLE' --项目开发中日志只能记录在日志文件中，不能记表中
```

设置完成后，查询一些列表可以发现慢查询的日志文件里面有数据了。

DESKTOP-2EKGEE5.err	2018/9/4 22:49	ERR 文件	181 KB
DESKTOP-2EKGEE5.pid	2018/9/4 22:49	PID 文件	1 KB
DESKTOP-2EKGEE5-slow.log	2018/9/4 22:48	文本文档	4,014 KB
ib_logfile0	2018/9/5 10:44	文件	49,152 KB
ib_logfile1	2018/9/2 22:30	文件	49,152 KB
ibdata1	2018/9/5 10:44	文件	667,648 KB

慢查询解读

从慢查询日志里面挑选一条慢查询日志，数据组成如下

```

1 # User@Host: root[root] @ localhost [127.0.0.1] Id: 10
2 # Query_time: 0.001042
3 # Lock_time: 0.000000
4 # Rows_sent: 2
5 # Rows_examined: 2
6 SET timestamp=1535462721;
7 SELECT * FROM `myarchive` LIMIT 0, 1000;

```

第一行：用户名、用户的IP信息、线程ID号

第二行：执行花费的时间【单位：毫秒】

第三行：执行获得锁的时间

第四行：获得的结果行数

第五行：扫描的数据行数

第六行：这SQL执行的具体时间

第七行：具体的SQL语句

执行计划 (explain select..., explain extended select...)

使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈。

执行计划作用

表的读取顺序

数据读取操作的操作类型

哪些索引可以使用

哪些索引被实际使用

表之间的引用

每张表有多少行被优化器查询

执行计划的语法

执行计划的语法其实非常简单：在SQL查询的前面加上EXPLAIN关键字就行。

比如：EXPLAIN select * from table1

重点的就是EXPLAIN后面你要分析的SQL语句

通过 EXPLAIN 关键分析的结果由以下列组成，接下来挨个分析每一个列

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
----	-------------	-------	------	---------------	-----	---------	-----	------	-------

ID列

ID列：描述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

根据ID的数值结果可以分成一下三种情况

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

分别举例来看

```
mysql> explain select t2.*
-> from t1, t2, t3
-> where t1.id = t2.id and t1.id = t3.id
-> and t1.other_column = '';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ref | PRIMARY, idx_t1 | idx_t1 | 92 | const | 1 | Using where |
| 1 | SIMPLE | t2 | eq_ref | PRIMARY | PRIMARY | 4 | test.t1.ID | 1 | Using index |
| 1 | SIMPLE | t3 | eq_ref | PRIMARY | PRIMARY | 4 | test.t1.ID | 1 | |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

如上图所示，ID列的值全为1，代表执行的允许从t1开始加载，依次为t3与t2

EXPLAIN select t2.* from t1,t2,t3 where t1.id = t2.id and t1.id = t3.id and t1.other_column = '';

Id不同

```
mysql> explain SELECT t2.*
-> FROM t2
-> WHERE id = (SELECT id
-> FROM t1
-> WHERE id = (SELECT t3.id
-> FROM t3
-> WHERE t3.other_column = ''));
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | t2 | const | PRIMARY | PRIMARY | 4 | const | 1 | |
| 2 | SUBQUERY | t1 | const | PRIMARY | PRIMARY | 4 | | 1 | Using index |
| 3 | SUBQUERY | t3 | ALL | NULL | NULL | NULL | NULL | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> explain SELECT t2.*
-> FROM t2
-> WHERE id = (SELECT id
-> FROM t1
-> WHERE id = (SELECT t3.id
-> FROM t3
-> WHERE t3.other_column = ''));
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | t2 | const | PRIMARY | PRIMARY | 4 | const | 1 | |
| 2 | SUBQUERY | t1 | const | PRIMARY | PRIMARY | 4 | | 1 | Using index |
| 3 | SUBQUERY | t3 | ALL | NULL | NULL | NULL | NULL | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

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

EXPLAIN select t2.* from t2 where id = (

select id from t1 where id = (select t3.id from t3 where t3.other_column='')

);

Id相同又不同

```
mysql> explain select t2.* from (
-> select t3.id
-> from t3
-> where t3.other_column = '') s1, t2
-> where s1.id = t2.id;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | PRIMARY | <derived2> | system | NULL | NULL | NULL | NULL | 1 | |
| 1 | PRIMARY | t2 | const | PRIMARY | PRIMARY | 4 | const | 1 | |
| 2 | DERIVED | t3 | ALL | NULL | NULL | NULL | NULL | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

id如果相同，可以认为是一组，从上往下顺序执行；

在所有组中，id值越大，优先级越高，越先执行

EXPLAIN select t2.* from (select t3.id from t3 where t3.other_column = '') s1 ,t2 where s1.id =

t2.id;

select_type列

Select_type:查询的类型，要是用于区别:普通查询、联合查询、子查询等的复杂查询

类型如下

类型	描述
SIMPLE	简单的 select 查询,查询中不包含子查询或者UNION
PRIMARY	查询中若包含任何复杂的子部分,最外层查询则被标记为
SUBQUERY	在SELECT或WHERE列表中包含子查询
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。
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

SIMPLE

EXPLAIN select * from t1

简单的 select 查询,查询中不包含子查询或者UNION

```
1 EXPLAIN select * from t1
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)

PRIMARY与SUBQUERY

PRIMARY: 查询中若包含任何复杂的子部分, 最外层查询则被标记为

SUBQUERY: 在SELECT或WHERE列表中包含了子查询

EXPLAIN select t1.*, (select t2.id from t2 where t2.id = 1) from t1

```
3 select t1.*, (select t2.id from t2 where t2.id = 1) from t1 |
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)
2	SUBQUERY	t2	const	PRIMARY	PRIMAR	4	const	1	Using indi

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。

```
1 EXPLAIN select t1.* from t1, (select t2.* from t2 where t2.id = 1) s2 where t1.id = s2.id
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	(Null)	(Null)	(Null)	(Null)	1	(Null)
1	PRIMARY	t1	const	PRIMARY	PRIMAR	4	const	1	(Null)
2	DERIVED	t2	const	PRIMARY	PRIMAR	4	const	1	(Null)

UNION RESULT 与 UNION

UNION: 若第二个SELECT出现在UNION之后, 则被标记为UNION;

UNION RESULT: 从UNION表获取结果的SELECT

#UNION RESULT , UNION

EXPLAIN select * from t1 UNION select * from t2

```
2 select * from t1
3 UNION
4 select * from t2
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)
2	UNION	t2	ALL	(Null)	(Null)	(Null)	(Null)	1	(Null)
(Null)	UNION RESULT	<union1,2>	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	Using temporary

table列

显示这一行的数据是关于哪张表的

```

47
16 #UNION RESULT , UNION
17 EXPLAIN
18 select * from t1
19 UNION
20 select * from t2

```

信息	结果1	概况	状态
id	select_type	table	
1	PRIMARY	t1	
2	UNION	t2	
(Null)	UNION RESULT	<union1,2>	

type	possible_keys	key	key_len
ALL	(Null)	(Null)	(Null)
ALL	(Null)	(Null)	(Null)
ALL	(Null)	(Null)	(Null)

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 > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL
```

system>const>eq_ref>ref>range>index>ALL

System与const

略不计

const用于比较primary key或者unique索引。因为只匹配一行数据，所以很快

如将主键置于where列表中，MySQL就能将该查询转换为一个常量

```
1 EXPLAIN
2 SELECT * from (select * from t2 where id = 1) d1;
3
```

信息	结果1	概況	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	PRIMARY	<derived>	system	(Null)	(Null)	(Null)	(Null)	1	(Null)	
2	DERIVED	t2	const	PRIMARY	PRIMARY		const	1	(Null)	

唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描

```
2 SELECT * from t1,t2 where t1.id = t2.id
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	PRIMARY	(Null)	(Null)	(Null)	1	(Null)
1	SIMPLE	t2	eq_ref	PRIMARY	PRIMARY	4	mysqld.1	(Null)	(Null)

非唯一性索引扫描，返回匹配某个单独值的所有行。

本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而，它可能会找到多个符合条件的行。

查询创建工具

查询编辑器

```

1 EXPLAIN
2 select col1 from t1 where col1 = 'ac'

```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	idx_col1_col2	idx_col1_63		const	1	Using where

只检索给定范围的行,使用一个索引来选择行。key 列显示使用了哪个索引

一般就是在你的where语句中出现了between、<、>、in等的查询

这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一点，不用扫

```
mysql> explain select * from t1 where id in (1, 2, 6);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	range	PRIMARY	PRIMARY	4	NULL	3	Using where

1 row in set (0.00 sec)

Index

当查询的结果全为索引列的时候，虽然也是全部扫描，但是只查询的索引库，而没有去查询数据。

```
3
4 EXPLAIN
5 select c2 from testdemo
6
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	testdemo	index	(Null)	idx_c2	5	(Null)	5	Using index

All

Full Table Scan, 将遍历全表以找到匹配的行

```
mysql> explain select * from t1 where column_without_index = '';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | ALL | NULL | NULL | NULL | NULL | 516 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

possible_keys 与Key

possible_keys:可能使用的key

Key:实际使用的索引。如果为NULL, 则没有使用索引

查询中若使用了**覆盖索引**，则该索引和查询的select字段重叠

```
mysql> explain select col1, col2 from t1;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	index	NULL	idx_col1_col2	390	NULL	682	Using index

1 row in set (0.00 sec)

key_len列

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。在不损失精确性的情况下，长度越短越好

key_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的

注意

根据底层使用的不同存储引擎,受影响的行数这个指标可能是一个估计值,也可能是一个精确值。即使受影响的行数是一个估计值(例如当使用 InnoDB 存储引擎管理表存储时),通常情况下这个估计值也足以使优化器做出一个有充分依据的决定。

key_len表示索引使用的字节数,

根据这个值，就可以判断索引使用情况，特别是在组合索引的时候，判断所有的索引字段是否都被查询用到。

char和varchar跟字符编码也有密切的联系,

latin1 占用1个字节, gbk 占用2个字节, utf8 占用3个字节。(不同字符编码占用的存储空间不同)

```
mysql> explain select * from t1 where col1 = 'ab';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	idx_col1_col2	idx_col1_col2	13	const	143	

```
1 row in set (0.00 sec)
```



```
mysql> explain select * from t1 where col1 = 'ab' and col2 = 'ac';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	idx_col1_col2	idx_col1_col2	26	const,const	1	

```
1 row in set (0.01 sec)
```

字符类型

字符串类型

字符串类型CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM和SET。该节描述了这些类型如何工作以及如何在查询中使用这些类型。

类型	大小	用途
CHAR	0-255字节	定长字符串
VARCHAR	0-65535 字节	变长字符串
TINYBLOB	0-255字节	不超过 255 个字节的二进制字符串
TINYTEXT	0-255字节	短文本字符串
BLOB	0-65 535字节	二进制形式的长文本数据
TEXT	0-65 535字节	长文本数据
MEDIUMBLOB	0-16 777 215字节	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215字节	中等长度文本数据
LONGBLOB	0-4 294 967 295字节	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295字节	极大文本数据

CHAR和VARCHAR类型类似，但它们保存和检索的方式不同。它们的最大长度和是否尾部空格被保留等方面也不同。在存储或检索过程中不进行大小写转换。

字符类型-索引字段为char类型+不可为Null时

```
1 CREATE TABLE `s1` (  
2   `id` int(11) NOT NULL AUTO INCREMENT,  
3   `name` char(10) NOT NULL,  
4   `addr` varchar(20) DEFAULT NULL,  
5   PRIMARY KEY (`id`),  
6   KEY `name` (`name`)  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
8  
9  
10 explain select * from s1 where name='enjoy';  
11
```

信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	s1	ref	name	name	30	const	1	Using index	

name这一列为char(10),字符集为utf-8占用3个字节Keylen=10*3

字符类型-索引字段为char类型+允许为Null时

```
1 CREATE TABLE `s2` (  
2   `id` int(11) NOT NULL AUTO INCREMENT,  
3   `name` char(10) DEFAULT NULL,  
4   `addr` varchar(20) DEFAULT NULL,  
5   PRIMARY KEY (`id`),  
6   KEY `name` (`name`)  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
8  
9  
10 explain select * from s2 where name='enjoyedu';
```

信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	s2	ref	name	name	31	const	1	Using index	

name这一列为char(10),字符集为utf-8占用3个字节,外加需要存入一个null值

Keylen=10*3+1(null) 结果为31

索引字段为varchar类型+不可为Null时

```
1 CREATE TABLE `s3` (  
2   `id` int(11) NOT NULL AUTO INCREMENT,  
3   `name` varchar(10) NOT NULL,  
4   `addr` varchar(20) DEFAULT NULL,  
5   PRIMARY KEY (`id`),  
6   KEY `name` (`name`)  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
8  
9  
10 explain select * from s3 where name='enjoyeud';
```

信息	结果1	概况	状态							
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	
1	SIMPLE	s3	ref	name	name	32	const	1	Using index	

Keylen=varchar(n)变长字段+不允许Null=n*(utf8=3,gbk=2,latin1=1)+2

索引字段为varchar类型+允许为Null时

```

1 CREATE TABLE `s4` (
2   `id` int(11) NOT NULL AUTO INCREMENT,
3   `name` varchar(10) DEFAULT NULL,
4   `addr` varchar(20) DEFAULT NULL,
5   PRIMARY KEY (`id`),
6   KEY `name` (`name`)
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8
8
9 explain select * from s4 where name='enjoyedu';
10

```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	s4	ref	name	name	33	const	1	Using index

Keylen=varchar(n)变长字段+允许Null=n*(utf8=3,gbk=2,latin1=1)+1(NULL)+2

总结

字符类型

变长字段需要额外的2个字节（VARCHAR值保存时只保存需要的字符数，另加一个字节来记录长度(如果列声明的长度超过255，则使用两个字节)，所以VARCAHR索引长度计算时候要加2），固定长度字段不需要额外的字节。

而NULL都需要1个字节的额外空间,所以索引字段最好不要为NULL，因为NULL让统计更加复杂并且需要额外的存储空间。

复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之和，这也可以用来判定复合索引是否部分使用，还是全部使用。

整数/浮点数/时间类型的索引长度

NOT NULL=字段本身的字段长度

NULL=字段本身的字段长度+1(因为需要有是否为空的标记，这个标记需要占用1个字节)

datetime类型在5.6中字段长度是5个字节，datetime类型在5.5中字段长度是8个字节

Ref列

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

```
mysql> explain select * from t1, t2 where t1.col1 = t2.col1 and t1.col2 = 'ac';
```

id	table	type	possible_keys	key	key_len	ref	rows
1	t2	ALL	NULL	NULL	NULL	NULL	640
1	t1	ref	idx_col1_col2	idx_col1_col2	26	shared.t2.col1,const	82

2 rows in set (0.01 sec)

由key_len可知t1表的idx_col1_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac'

其中【shared.t2.col1】为【数据库.表.列】

Rows列

根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数

```
mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ALL	PRIMARY	NULL	NULL	NULL	640	Using where
1	SIMPLE	t1	eq_ref	PRIMARY	PRIMARY	4	shared.t2.ID	1	

2 rows in set (0.00 sec)

```
mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001 Duplicates: 0 Warnings: 0
```

```
mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t2	ref	PRIMARY,idx_col1_col2	idx_col1_col2	195	const	142	
1	SIMPLE	t1	eq_ref	PRIMARY	PRIMARY	4	shared.t2.ID	1	

2 rows in set (0.00 sec)

Extra列

包含不适合在其他列中显示但十分重要的额外信息。

值	描述
Using filesort	说明mysql会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。 MySQL中无法利用索引完成的排序操作称为“文件排序”
Using temporary	使用了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。
USING index	是否用了覆盖索引
Using where	表明使用了where过滤
Using join buffer	使用了连接缓存:
Impossible where	where子句的值总是false，不能用来获取任何元组

Using filesort

说明mysql会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。MySQL中无法利用索引完成的排序操作称为“文件排序”

当发现有Using filesort 后，实际上就是发现了可以优化的地方

```
mysql> explain select col1 from t1 where col1 = 'ac' order by col3\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
        type: ref
possible_keys: idx_col1_col2_col3
         key: idx_col1_col2_col3
        key_len: 13
         ref: const
        rows: 142
   Extra: Using where; Using index; Using filesort
1 row in set (0.00 sec)
```

上图其实是一种索引失效的情况，后面会讲，可以看出查询中用到了个联合索引，索引分别为 col1,col2,col3

```
mysql> explain select col1 from t1 where col1 = 'ac' order by col2, col3\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
        type: ref
possible_keys: idx_col1_col2_col3
         key: idx_col1_col2_col3
        key_len: 13
         ref: const
        rows: 142
   Extra: Using where; Using index
1 row in set (0.00 sec)
```

当我排序新增了个col2，发现using filesort 就没有了。

Using temporary

使用了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。

```
mysql> explain select col1 from t1 where col1 in ('ac','ab','aa') group by col2\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
        type: range
possible_keys: idx_col1_col2
         key: idx_col1_col2
        key_len: 13
         ref: NULL
        rows: 569
   Extra: Using where; Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

```
mysql> explain select col1 from t1 where col1 in ('ac', 'ab') group by col1, col2\G
***** 1. row *****
id: 1
select_type: SIMPLE
table: t1
type: range
possible_keys: idx_col1_col2_col3
key: idx_col1_col2_col3
key_len: 26
ref: NULL
rows: 4
Extra: Using where; Using index for group-by
1 row in set (0.00 sec)
```

尤其发现在执行计划里面有using filesort而且还有Using temporary的时候，特别需要注意

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；

```
mysql> explain select col2 from t1 where col1 = 'ab';
+----+-----+-----+-----+-----+-----+-----+-----+
| id | ... | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | ... | idx_col1_col2 | idx_col1_col2 | 13 | const | 143 | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

如果没有同时出现using where，表明索引用来读取数据而非执行查找动作

```
mysql> explain select col1, col2 from t1;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t1 | index | NULL | idx_col1_col2 | 390 | NULL | 682 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

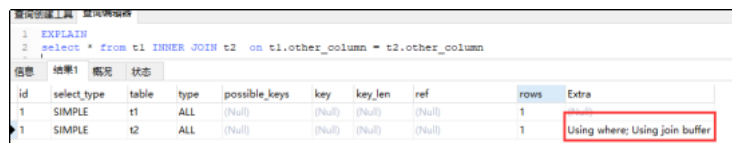
Using where 与 using join buffer

Using where

表明使用了where过滤

using join buffer

使用了连接缓存：



```
1 EXPLAIN
2 select * from t1 INNER JOIN t2 on t1.other_column = t2.other_column
3
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	(Null)	(Null)	(Null)	(Null)	1	Using where
1	SIMPLE	t2	ALL	(Null)	(Null)	(Null)	(Null)	1	Using where; Using join buffer

impossible where

where子句的值总是false，不能用来获取任何元组



```
1 EXPLAIN
2 select * from t1 where t1.other_column = 'enjoy' and t1.other_column = 'edu'
3
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Impossible WHERE

filtered列

使用explain extended时显示，显示针对表里符合某个条件（where子句或者联结条件）的记录数的百分比所做的一个悲观估算，即mysql将要过滤行数的百分比。

sql优化顺口溜

全匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

LIKE百分写最右，覆盖索引不写*；

全职匹配我最爱？

```
mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July' AND age = 25;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 78 | const,const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July' AND age = 25 AND pos = 'dev';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 140 | const,const,const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

当建立了索引列后，能在where条件中使用索引的尽量所用。

最左前缀要遵守，带头大哥不能死，中间兄弟不能断？

```
mysql> EXPLAIN SELECT * FROM staffs WHERE age = 25 AND pos = 'dev';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 2 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE pos = 'dev';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 2 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM staffs WHERE NAME = 'July';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | staffs | ref | idx_staffs_nameAgePos | idx_staffs_nameAgePos | 74 | const | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

如果索引了多列，要遵守最左前缀法则。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。

七 OLTP与OLAP的介绍和对比

OLTP与OLAP的介绍

数据处理大致可以分成两大类：联机事务处理OLTP（on-line transaction processing）、联机分析处理OLAP（On-Line Analytical Processing）。OLTP是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

OLTP 系统强调数据库内存效率，强调内存各种指标的命令率，强调绑定变量，强调并发操作；
OLAP 系统则强调数据分析，强调SQL执行市场，强调磁盘I/O，强调分区等。

OLTP与OLAP之间的比较：



OLTP，也叫联机事务处理（Online Transaction Processing），表示事务性非常高的系统，一般都是高可用的在线系统，以小的事务以及小的查询为主，评估其系统的时候，一般看其每秒执行的Transaction以及Execute SQL的数量。在这样的系统中，单个数据库每秒处理的Transaction往往超过几百个，或者是几千个，Select 语句的执行量每秒几千甚至几万个。典型的OLTP系统有电子商务系统、银行、证券等，如美国eBay的业务数据库，就是很典型的OLTP数据库。

OLTP系统最容易出现瓶颈的地方就是CPU与磁盘子系统。

（1）CPU出现瓶颈常表现在逻辑读总量与计算性函数或者是过程上，逻辑读总量等于单个语句的逻辑读乘以执行次数，如果单个语句执行速度虽然很快，但是执行次数非常多，那么，也可能会导致很大的逻辑读总量。设计的方法与优化的方法就是减少单个语句的逻辑读，或者是减少它们的执行次数。另外，一些计算型的函数，如自定义函数、decode等的频繁使用，也会消耗大量的CPU时间，造成系统的负载升高，正确的设计方法或者是优化方法，需要尽量避

免计算过程，如保存计算结果到统计表就是一个好的方法。

(2) 磁盘子系统在OLTP环境中，它的承载能力一般取决于它的IOPS处理能力。因为在OLTP环境中，磁盘物理读一般都是db file sequential read，也就是单块读，但是这个读的次数非常频繁。如果频繁到磁盘子系统都不能承载其IOPS的时候，就会出现大的性能问题。

OLTP比较常用的设计与优化方式为Cache技术与B-tree索引技术，Cache决定了很多语句不需要从磁盘子系统获得数据，所以，Web cache与Oracle data buffer对OLTP系统是很重要的。另外，在索引使用方面，语句越简单越好，这样执行计划也稳定，而且一定要使用绑定变量，减少语句解析，尽量减少表关联，尽量减少分布式事务，基本不使用分区技术、MV技术、并行技术及位图索引。因为并发量很高，批量更新时要分批快速提交，以避免阻塞的发生。

OLTP系统是一个数据块变化非常频繁，SQL语句提交非常频繁的系统。对于数据块来说，应尽可能让数据块保存在内存当中，对于SQL来说，尽可能使用变量绑定技术来达到SQL重用，减少物理I/O和重复的SQL解析，从而极大的改善数据库的性能。

这里影响性能除了绑定变量，还有可能是热块（hot block）。当一个块被多个用户同时读取时，Oracle为了维护数据的一致性，需要使用Latch来串行化用户的操作。当一个用户获得了latch后，其他用户就只能等待，获取这个数据块的用户越多，等待就越明显。这就是热块的问题。这种热块可能是数据块，也可能是回滚端块。对于数据块来讲，通常是数据库的数据分布不均匀导致，如果是索引的数据块，可以考虑创建反向索引来达到重新分布数据的目的，对于回滚段数据块，可以适当多增加几个回滚段来避免这种争用。

OLAP，也叫联机分析处理（Online Analytical Processing）系统，有的时候也叫DSS决策支持系统，就是我们说的数据仓库。在这样的系统中，语句的执行量不是考核标准，因为一条语句的执行时间可能会非常长，读取的数据也非常多。所以，在这样的系统中，考核的标准往往是磁盘子系统的吞吐量（带宽），如能达到多少MB/s的流量。

磁盘子系统的吞吐量则往往取决于磁盘的个数，这个时候，Cache基本是没有效果的，数据库的读写类型基本上是db file scattered read与direct path read/write。应尽量采用个数比较多的磁盘以及比较大的带宽，如4Gb的光纤接口。

在OLAP系统中，常使用分区技术、并行技术。

分区技术在OLAP系统中的重要性主要体现在数据库管理上，比如数据库加载，可以通过分区交换的方式实现，备份可以通过备份分区表空间实现，删除数据可以通过分区进行删除，至于分区在性能上的影响，它可以使得一些大表的扫描变得很快（只扫描单个分区）。另外，如果分区结合并行的话，也可以使得整个表的扫描会变得很快。总之，分区主要的功能是管理上的方便性，它并不能绝对保证查询性能的提高，有时候分区会带来性能上的提高，有时候会降低。

并行技术除了与分区技术结合外，在Oracle 10g中，与RAC结合实现多节点的同时扫描，效果也非常不错，可把一个任务，如select的全表扫描，平均地分派到多个RAC的节点上去。

在OLAP系统中，不需要使用绑定（BIND）变量，因为整个系统的执行量很小，分析时间对于执行时间来说，可以忽略，而且可避免出现错误的执行计划。但是OLAP中可以大量使用位图索引，物化视图，对于大的事务，尽量寻求速度上的优化，没有必要像OLTP要求快速提交，甚至要刻意减慢执行的速度。

绑定变量真正的用途是在OLTP系统中，这个系统通常有这样的特点，用户并发数很大，用户的请求十分密集，并且这些请求的SQL大多数是可以重复使用的。

对于OLAP系统来说，绝大多数时候数据库上运行着的是报表作业，执行基本上是聚合类的SQL操作，比如group by，这时候，把优化器模式设置为all_rows是恰当的。而对于一些分页操作比较多的网站类数据库，设置为first_rows会更好一些。但有时候对于OLAP系统，我们又有分页的情况下，我们可以考虑在每条SQL中用hint。如：

```
Select a.* from table a;
```

分开设计与优化

在设计上要特别注意，如在高可用的OLTP环境中，不要盲目地把OLAP的技术拿过来用。

如分区技术，假设不是大范围地使用分区关键字，而采用其它的字段作为where条件，那么，如果是本地索引，将不得不扫描多个索引，而性能变得更为低下。如果是全局索引，又失去分区意义。

并行技术也是如此，一般在完成大型任务时才使用，如在实际生活中，翻译一本书，可以先安排多个人，每个人翻译不同的章节，这样可以提高翻译速度。如果只是翻译一页书，也去分配不同的人翻译不同的行，再组合起来，就没必要了，因为在分配工作的时间里，一个人或许早就翻译完了。

位图索引也是一样，如果用在OLTP环境中，很容易造成阻塞与死锁。但是，在OLAP环境中，可能会因为其特有的特性，提高OLAP的查询速度。MV也是基本一样，包括触发器等，在DML频繁的OLTP系统上，很容易成为瓶颈，甚至是Library Cache等待，而在OLAP环境上，则可能会因为使用恰当而提高查询速度。

对于OLAP系统，在内存上可优化的余地很小，增加CPU处理速度和磁盘I/O速度是最直接的提高数据库性能的方法，当然这也意味着系统成本的增加。

比如我们要对几亿条或者几十亿条数据进行聚合处理，这种海量的数据，全部放在内存中操作是很难的，同时也没有必要，因为这些数据很少重用，缓存起来也没有实际意义，而且还会造成物理I/O相当大。所以这种系统的瓶颈往往是磁盘I/O上面的。

对于OLAP系统，SQL的优化非常重要，因为它的数量很大，做全表扫描和索引对性能上来说差异是非常大的。

其他

Oracle 10g以前的版本建库过程中可供选择的模板有：

- Data Warehouse（数据仓库）
- General Purpose（通用目的、一般用途）
- New Database
- Transaction Processing（事务处理）

Oracle 11g的版本建库过程中可供选择的模板有：

一般用途或事务处理

定制数据库

数据仓库

个人对这些模板的理解为：

联机分析处理（OLAP,On-line Analytical Processing），数据量大，DML少。使用数据仓库模板

联机事务处理（OLTP,On-line Transaction Processing），数据量少，DML频繁，并发性事务处理多，但是一般都很短。使用一般用途或事务处理模板。

决策支持系统（DDS，Decision support system），典型的操作是全表扫描，长查询，长事务，但是一般事务的个数很少，往往是一个事务独占系统。

八 autocommit测试

MySQL是默认提交的，也就是说默认保存到磁盘上的，但是如果我们本次会话设置了set autocommit=0;取消了默认提交的话，看一下效果：

可以通过查看“@@AUTOCOMMIT”变量来查看当前自动提交状态，查看此变量SELECT @@AUTOCOMMIT。



```
mysql> use orm3;
Database changed
mysql> select * from app01_publish;
+----+-----+-----+
| id | name      | addr |
+----+-----+-----+
| 1 | 西瓜出版社 | 北京 |
| 2 | 人民出版社 | 天津 |
| 3 | 清华出版社 | 北京 |
| 4 | 南京出版社 | 南京 |
| 5 | hah       | xxxx |
| 6 | 呵呵      | ssss |
+----+-----+-----+
6 rows in set (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into app01_publish values(7,'第七条','上海');
Query OK, 1 row affected (0.00 sec)

mysql> select * from app01_publish;
+----+-----+-----+
| id | name      | addr |
+----+-----+-----+
| 1 | 西瓜出版社 | 北京 |
| 2 | 人民出版社 | 天津 |
| 3 | 清华出版社 | 北京 |
| 4 | 南京出版社 | 南京 |
| 5 | hah       | xxxx |
| 6 | 呵呵      | ssss |
| 7 | 第七条    | 上海 |
+----+-----+-----+
7 rows in set (0.00 sec)
```



```
mysql> quit
Bye
```

```
C:\Users\zequan>mysql
```

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 3

Server version: 5.6.21 MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

```
mysql> use orm3;
```

Database changed

```
mysql> select * from app01_publish; #再进来发现新插入的数据没有了
```

```
+----+-----+-----+
| id | name      | addr |
+----+-----+-----+
| 1 | 西瓜出版社 | 北京 |
| 2 | 人民出版社 | 天津 |
| 3 | 清华大学出版社 | 北京 |
| 4 | 南京出版社 | 南京 |
| 5 | hah       | xxxx |
| 6 | 呵呵      | ssss |
+----+-----+-----+
```

```
6 rows in set (0.00 sec)
```

