MySQL 中的锁（表锁、行锁）

锁是计算机协调多个进程或纯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（CPU、RAM、I/O）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、有效性是所在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。

概述

相对其他数据库而言，MySQL 的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。

MySQL 大致可归纳为以下 3 种锁：

- 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

\- -

\- -

MySQL 表级锁的锁模式（MyISAM)

MySQL 表级锁有两种模式：表共享锁（Table Read Lock）和表独占写锁（Table Write Lock）。

- 对 MyISAM 的读操作，不会阻塞其他用户对同一表请求，但会阻塞对同一表的写请求；
- 对 MyISAM 的写操作，则会阻塞其他用户对同一表的读和写操作；
- MyISAM 表的读操作和写操作之间，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。

MySQL 表级锁的锁模式

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

MySQL 中的表锁兼容性

当前锁模式/是否兼容/请求锁模式	None	读锁	写锁
读锁	是	是	否
写锁	是	否	否

可见，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请求 ;MyISAM 表的读和写操作之间，以及写和写操作之间是**串行**的！（**当一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。** ）

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，会自动给涉及的表加写锁，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。在本书的示例中，显式加锁基本上都是为了方便而已，并非必须如此。

给 MyISAM 表显示加锁，一般是为了一定程度模拟事务操作，实现对某一时间点多个表的一致性读取。例如，有一个订单表 orders，其中记录有订单的总金额 total，同时还有一

个订单明细表 order_detail，其中记录有订单每一产品的金额小计 subtotal，假设我们需要检查这两个表的金额合计是否相等，可能就需要执行如下两条 SQL：

```
SELECT SUM(total) FROM orders;
SELECT SUM(subtotal) FROM order_detail;
```

这时，如果不先给这两个表加锁，就可能产生错误的结果，因为第一条语句执行过程中，order_detail 表可能已经发生了改变。因此，正确的方法应该是：

```
LOCK tables orders read local,order_detail read local;
SELECT SUM(total) FROM orders;
SELECT SUM(subtotal) FROM order_detail;
Unlock tables;
```

要特别说明以下两点内容。

- 上面的例子在 LOCK TABLES 时加了‘local’选项，其作用就是在满足 MyISAM 表并发插入条件的情况下，允许其他用户在表尾插入记录
- 在用 LOCKTABLES 给表显式加表锁是时，必须同时取得所有涉及表的锁，并且 MySQL 支持锁升级。也就是说，在执行 LOCK TABLES 后，只能访问显式加锁的这些表，不能访问未加锁的表；同时，如果加的是读锁，那么只能执行查询操作，而不能执行更新操作。其实，在自动加锁的情况下也基本如此，MySQL 问题一次获得 SQL 语句所需要的全部锁。这也正是 MyISAM 表不会出现死锁（Deadlock Free）的原因

一个 session 使用 LOCK TABLE 命令给表 film_text 加了读锁，这个 session 可以查询锁定表中的记录，但更新或访问其他表都会提示错误；同时，另外一个 session 可以查询表中的记录，但更新就会出现锁等待。

当使用 LOCK TABLE 时，不仅需要一次锁定用到的所有表，而且，同一个表在 SQL 语句中出现多少次，就要通过与 SQL 语句中相同的别名锁多少次，否则也会出错！

并发锁

在一定条件下，MyISAM 也支持查询和操作的并发进行。

MyISAM 存储引擎有一个系统变量 concurrent_insert，专门用以控制其并发插入的行为，其值分别可以为 0、1 或 2。

- 当 concurrent_insert 设置为 0 时，不允许并发插入。
- 当 concurrent_insert 设置为 1 时，如果 MyISAM 允许在一个读表的同时，另一个进程从表尾插入记录。这也是 MySQL 的默认设置。
- 当 concurrent_insert 设置为 2 时，无论 MyISAM 表中有没有空洞，都允许在表尾插入记录，都允许在表尾并发插入记录。

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锁争用。例如，将 concurrent_insert 系统变量为 2，总是允许并发插入；同时，通过定期在系统空闲时段执行 OPTIONMIZE TABLE 语句来整理空间碎片，收到因删除记录而产生的中间空洞。

MyISAM 的锁调度

前面讲过，MyISAM 存储引擎的读和写锁是互斥，读操作是串行的。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，即使读进程先请求先到锁等待队列，写请求后到，写锁也会插

到读请求之前！这是因为 MySQL 认为写请求一般比读请求重要。这也正是 MyISAM 表不太适合于有大量更新操作和查询操作应用的原因，因为，大量的更新操作会造成查询操作很难获得读锁，从而可能永远阻塞。这种情况有时可能会变得非常糟糕！幸好我们可以通过一些设置来调节 MyISAM 的调度行为。

- 通过指定启动参数 low-priority-updates，使 MyISAM 引擎默认给予读请求以优先的权利。
- 通过执行命令 SET LOW_PRIORITY_UPDATES=1，使该连接发出的更新请求优先级降低。
- 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 3 种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 变暂时将写请求的优先级降低，给读进程一定获得锁的机会。

上面已经讨论了写优先调度机制和解决办法。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 锁问题

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。

行级锁和表级锁本来就有许多不同之处，另外，事务的引入也带来了一些新问题。

1.事务（Transaction）及其 ACID 属性

事务是由一组 SQL 语句组成的逻辑处理单元，事务具有 4 属性，通常称为事务的 ACID 属性。

- 原性性（Actomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
- 一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以操持完整性；事务结束时，所有的内部数据结构（如 B 树索引或双向链表）也都必须是正确的。
- 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。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。
- 持久性（Durable）：事务完成之后，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.并发事务带来的问题

相对于串行处理来说，并发事务处理能大大增加数据库资源的利用率，提高数据库系统的事务吞吐量，从而可以支持可以支持更多的用户。但并发事务处理也会带来一些问题，主要包括以下几种情况。

- **更新丢失**（Lost Update）：当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题——最后的更新覆盖了其他事务所做的更新。例如，两个编辑人员制作了同一文档的电子副本。每个编辑人员独立地更改其副本，然后保存更改后的副本，这样就覆盖了原始文档。最后保存其更改保存其更改副本的编辑人员覆盖另一个编辑人员所做的修改。如果在一个编辑人员完成并提交事务之前，另一个编辑人员不能访问同一文件，则可避免此问题
- **脏读**（Dirty Reads）：一个事务正在对一条记录做修改，在这个事务并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”的数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做“脏读”。
- **不可重复读**（Non-Repeatable Reads）：一个事务在读取某些数据已经发生了改变、或某些记录已经被删除了！这种现象叫做“不可重复读”。
- **幻读**（Phantom Reads）：一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读”。

3.事务隔离级别

在并发事务处理带来的问题中，“更新丢失”通常应该是完全避免的。但防止更新丢失，并不能单靠数据库事务控制器来解决，需要应用程序对要更新的数据**加必要的锁**来解决，因此，防止更新丢失应该是应用的责任。

“脏读”、“不可重复读”和“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。数据库实现事务隔离的方式，基本可以分为以下两种。

一种是在读取数据前，对其加锁，阻止其他事务对数据进行修改。
另一种是不用加任何锁，通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot），并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度，好像是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据多版本并发控制（MultiVersion Concurrency Control，简称 MVCC 或 MCC），也经常称为多版本数据库。

数据库的事务隔离级别越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上“串行化”进行，这显然与“并发”是矛盾的，同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。

为了解决“隔离”与“并发”的矛盾，ISO/ANSI SQL92 定义了 4 个事务隔离级别，每个级别的隔离程度不同，允许出现的副作用也不同，应用可以根据自己业务逻辑要求，通过选择不同的隔离级别来平衡 " 隔离 " 与 " 并发 " 的矛盾

事务 4 种隔离级别比较

隔离级别/读数据一致性及允许的并发副作用	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，只能保证不读取物理上损坏的数据	是	是	是
已提交度（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

最后要说明的是：各具体数据库并不一定完全实现了上述 4 个隔离级别，例如，Oracle 只提供 Read committed 和 Serializable 两个标准级别，另外还自己定义的 Read only 隔离级别：SQL Server 除支持上述 ISO/ANSI SQL92 定义的 4 个级别外，还支持一个叫做 " 快照 " 的隔离级别，但严格来说它是一个用 MVCC 实现的 Serializable 隔离级别。MySQL 支持全部 4 个隔离级别，但在具体实现时，有一些特点，比如在一些隔离级下是采用 MVCC 一致性读，但某些情况又不是。

获取 InonoD 行锁争用情况

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 |
+-------------------------------+-------+
5 rows in set (0.00 sec)
```

如果发现争用比较严重，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。

InnoDB 的行锁模式及加锁方法

InnoDB 实现了以下两种类型的行锁。

- 共享锁（s）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
- 排他锁（X）：允许获取排他锁的事务更新数据，阻止其他事务取得相同的数据集共享读锁和排他写锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

意向共享锁（IS）：事务打算给数据行共享锁，事务在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）：事务打算给数据行加排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。

InnoDB 行锁模式兼容性列表

当前锁模式/是否兼容/请求锁模式	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容

S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

如果一个事务请求的锁模式与当前的锁兼容，InnoDB 就请求的锁授予该事务；反之，如果两者两者不兼容，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 会自动给涉及及数据集加排他锁（X）；对于普通 SELECT 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 方式获取排他锁。

InnoDB 行锁实现方式

InnoDB 行锁是通过索引上的索引项来实现的，这一点MySQL 与 Oracle 不同，后者是通过在数据中对相应数据行加锁来实现的。InnoDB 这种行锁实现特点意味者：只有通过索引条件检索数据，InnoDB 才会使用行级锁，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。

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，这种锁机制不是所谓的间隙锁（Next-Key 锁）。

举例来说，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：

```
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，一方面是为了防止幻读，以满足相关隔离级别的要求，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；另一方面，是为了满足其恢复和复制的需要。有关其恢复和复制对机制的影响，以及不同隔离级别下 InnoDB 使用间隙锁的情况。

很显然，在使用范围条件检索并锁定记录时，InnoDB 这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。因此，在实际开发中，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。

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。但在个另特殊事务中，也可以考虑使用表级锁。

- 第一种情况是：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。
- 第二种情况是：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。

在 InnoDB 下 ，使用表锁要注意以下两点。

（ 1 ）使用 LOCK TALBES 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、innodb_table_lock=1（默认设置）时，InnoDB 层才能知道 MySQL 加的表锁，MySQL Server 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 LOCAK TABLES 对 InnoDB 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 UNLOCAK TABLES 释放表锁，因为 UNLOCK TABLES 会隐含地提交事务；COMMIT 或 ROLLBACK 产不能释放用 LOCAK 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 t 读，可以按如下做：

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

关于死锁

MyISAM 表锁是 deadlock free 的，这是因为MyISAM 总是一次性获得所需的全部锁，要么全部满足，要么等待，因此不会出现死锁。但是在 InnoDB 中，除单个 SQL 组成的事务外，锁是逐步获得的，这就决定了 InnoDB 发生死锁是可能的。

发生死锁后，InnoDB 一般都能自动检测到，并使一个事务释放锁并退回，另一个事务获得锁，继续完成事务。但在涉及外部锁，或涉及锁的情况下，InnoDB 并不能完全自动检测到死锁，这需要通过设置锁等待超时参数 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取所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖垮数据库。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，绝大部分都可以避免。下面就通过实例来介绍几种死锁的常用方法。

（ 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为访问表，这样可以大大降低产生死锁的机会。如果两个 session 访问两个表的顺序不同，发生死锁的机会就非常高！但如果以相同的顺序来访问，死锁就可能避免。

（ 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，而不应该先申请共享锁，更新时再申请排他锁，甚至死锁。

（ 4 ）在 REPEATEABLE-READ 隔离级别下，如果两个线程同时对相同条件记录用 SELECT...ROR UPDATE 加排他锁，在没有符合该记录情况下，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，就可以避免问题。

（ 5 ）当隔离级别为 READ COMMITED 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。

尽管通过上面的设计和优化等措施，可以大减少死锁，但死锁很难完全避免。因此，在程序设计中总是捕获并处理死锁异常是一个很好的编程习惯。

如果出现死锁，可以用 SHOW INNODB STATUS 命令来确定最后一个死锁产生的原因和改进措施。

- -

- -

- -

总结

对于MyISAM 的表锁，主要有以下几点

（1）共享读锁（S）之间是兼容的，但共享读锁（S）和排他写锁（X）之间，以及排他写锁之间（X）是互斥的，也就是说读和写是串行的。

（2）在一定条件下，MyISAM 允许查询和插入并发执行，我们可以利用这一点来解决应用中对同一表和插入的锁争用问题。

（3）MyISAM 默认的锁调度机制是写优先，这并不一定适合所有应用，用户可以通过设置 LOW_PRIPORITY_UPDATES 参数，或在 INSERT、UPDATE、DELETE 语句中指定 LOW_PRIORITY 选项来调节读写锁的争用。

（4）由于表锁的锁定粒度大，读写之间又是串行的，因此，如果更新操作较多，MyISAM 表可能会出现严重的锁等待，可以考虑采用 InnoDB 表来减少锁冲突。

对于 InnoDB 表，主要有以下几点

（1）InnoDB 的行销是基于索引实现的，如果不通过索引访问数据，InnoDB 会使用表锁。

（2）InnoDB 间隙锁机制，以及 InnoDB 使用间隙锁的原因。

（3）在不同的隔离级别下，InnoDB 的锁机制和一致性读策略不同。

（4）MySQL 的恢复和复制对 InnoDB 锁机制和一致性读策略也有较大影响。

（5）锁冲突甚至死锁很难完全避免。

在了解 InnoDB 的锁特性后，用户可以通过设计和 SQL 调整等措施减少锁冲突和死锁，包括：

- 尽量使用较低的隔离级别
- 精心设计索引，并尽量使用索引访问数据，使加锁更精确，从而减少锁冲突的机会。
- 选择合理的事务大小，小事务发生锁冲突的几率也更小。
- 给记录集显示加锁时，最好一次性请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁。
- 不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大减少死锁的机会。
- 尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响。
- 不要申请超过实际需要的锁级别；除非必须，查询时不要显示加锁。
- 对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能。