一个phper的日常

昵称：weblee
园龄：1年9个月
粉丝：19
关注：1
+加关注

< 2019年10月 >

日	一	二	三	四	五	六
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类
PHP(27)
PHP拓展(5)
PYTHON(6)
服务器(2)
个人笔记(5)
其他(7)
前端(1)
数据库(6)

随笔档案
2018年3月(6)
2018年2月(11)
2018年1月(41)

Mysql中的锁机制

原文：http://blog.csdn.net/soonfly/article/details/70238902

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。本章我们着重讨论MySQL锁机制 的特点，常见的锁问题，以及解决MySQL锁问题的一些方法或建议。
Mysql用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。这些锁统称为悲观锁(Pessimistic Lock)。

MySQL锁概述

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。比如，MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；BDB存储引擎采用的是页面锁（page-level locking），但也支持表级锁；InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。
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
从上述特点可见，很难笼统地说哪种锁更好，只能就具体应用的特点来说哪种锁更合适！仅从锁的角度 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。

MyISAM表锁

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）**。
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！根据如表20-2所示的 例子可以知道，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。

MyISAM存储引擎的写锁阻塞读例子：

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。

session_1	session_2
获得表film_text的WRITE锁定： mysql> lock table film_text write; Query OK, 0 rows affected (0.00 sec)	

当前session对锁定表的查询、更新、插入操作都可以执行：
mysql> select film_id,title from film_text where film_id = 1001;
+---------+-------------+
| film_id | title |
+---------+-------------+
| 1001 | Update Test |
+---------+-------------+
1 row in set (0.00 sec)
mysql> insert into film_text (film_id,title) values(1003,'Test');
Query OK, 1 row affected (0.00 sec)
mysql> update film_text set title = 'Test' where film_id = 1001;
	其他session对锁定表的查询被阻塞，需要等待锁被释放：
	mysql> select film_id,title from film_text where film_id = 1001;
	等待

释放锁： mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待
	Session2获得锁，查询返回： mysql> select film_id,title from film_text where film_id = 1001; +---------+-------+ | film_id | title | +---------+-------+ | 1001 | Test | +---------+-------+ 1 row in set (57.59 sec)

MyISAM存储引擎的读锁阻塞写例子：

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。

session_1	session_2
获得表film_text的READ锁定： mysql> lock table film_text read; Query OK, 0 rows affected (0.00 sec)	

当前session可以查询该表记录： mysql> select film_id,title from film_text where film_id = 1001; +---------+-------------------------+ | film_id | title | +---------+-------------------------+ | 1001 | ACADEMY DINOSAUR | +---------+-------------------------+ 1 row in set (0.00 sec)	**其他session也可以查询该表的记录** mysql> select film_id,title from film_text where film_id = 1001; +---------+-------------------------+ | film_id | title | +---------+-------------------------+ | 1001 | ACADEMY DINOSAUR | +---------+-------------------------+ 1 row in set (0.00 sec)
当前session不能查询没有锁定的表 mysql> select film_id,title from film where film_id = 1001; ERROR 1100 (HY000): *Table 'film' was not locked with LOCK TABLES*	**其他session可以查询或者更新未锁定的表** mysql> select film_id,title from film where film_id = 1001; +---------+---------------+ | film_id | title | +---------+---------------+ | 1001 | update record | +---------+---------------+ 1 row in set (0.00 sec) mysql> update film set title = 'Test' where film_id = 1001; Query OK, 1 row affected (0.04 sec) Rows matched: 1 Changed: 1 Warnings: 0

当前session中插入或者更新锁定的表都会提示错误：
mysql> insert into film_text (film_id,title) values(1002,'Test');
ERROR 1099 (HY000): *Table 'film_text' was locked with a READ lock and can't be updated*
mysql> update film_text set title = 'Test' where film_id = 1001;
ERROR 1099 (HY000): *Table 'film_text' was locked with a READ lock and can't be updated*
	其他session更新锁定表会等待获得锁：
	mysql> update film_text set title = 'Test' where film_id = 1001;
	等待

释放锁 mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待
	Session获得锁，更新操作完成： mysql> update film_text set title = 'Test' where film_id = 1001; Query OK, 1 row affected (1 min 0.71 sec) Rows matched: 1 Changed: 1 Warnings: 0

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。在示例中，显式加锁基本上都是为了演示而已，并非必须如此。

给MyISAM表显示加锁，一般是为了在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，其中记录有各订单的总金额total，同时还有一个订单明细表order_detail，其中记录有各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符，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：

1、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并发插入记录，有关MyISAM表的并发插入问题，在后面还会进一步介绍。

2、在用LOCK TABLES给表显式加表锁时，必须同时取得所有涉及到表的锁，并且MySQL不支持锁升级。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，MyISAM总是一次获得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。

当使用LOCK TABLES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定多少次，否则也会出错！举例说明如下。

（1）对actor表获得读锁：

```
mysql> lock table actor read;
Query OK, 0 rows affected (0.00 sec)
```

（2）但是通过别名访问会提示错误：

```
mysql> select a.first_name,a.last_name,b.first_name,b.last_name
from actor a,actor b
where a.first_name = b.first_name and a.first_name = 'Lisa' and a.last_name = 'Tom'
and a.last_name <> b.last_name;
```

```
ERROR 1100 (HY000): Table ‘a’ was not locked with LOCK TABLES
```

（3）需要对别名分别锁定：

```
mysql> lock table actor as a read,actor as b read;
```

Query OK, 0 rows affected (0.00 sec)

（4）按照别名的查询可以正确执行：

```
mysql> select a.first_name,a.last_name,b.first_name,b.last_name
from actor a,actor b where a.first_name = b.first_name
and a.first_name = 'Lisa' and a.last_name = 'Tom'
and a.last_name <> b.last_name;
```

+————+————+————+————+
| first_name | last_name | first_name | last_name |
+————+————+————+————+
| Lisa | Tom | LISA | MONROE |
+————+————+————+————+
1 row in set (0.00 sec)

查询表级锁争用情况

可以通过检查table_locks_waited和table_locks_immediate状态变量来分析系统上的表锁定争夺：

```
mysql> show status like 'table%';
```

Variable_name | Value
Table_locks_immediate | 2979
Table_locks_waited | 0
2 rows in set (0.00 sec))

如果Table_locks_waited的值比较高，则说明存在着较严重的表级锁争用情况。

并发插入（Concurrent Inserts）

上文提到过MyISAM表的读和写是串行的，但这是就总体而言的。在一定条件下，MyISAM表也支持查询和插入操作的并发进行。
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。
- 当concurrent_insert设置为2时，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录。

在下面的例子中，session_1获得了一个表的READ LOCAL锁，该线程可以对表进行查询操作，但不能对表进行更新操作；其他的线程（session_2），虽然不能对表进行删除和更新操作，但却可以对该表进行并发插入操作，这里假设该表中间不存在空洞。

MyISAM存储引擎的读写（INSERT）并发例子：

session_1	session_2
获得表film_text的READ LOCAL锁定： mysql> lock table film_text read local; Query OK, 0 rows affected (0.00 sec)	

当前session不能对锁定表进行更新或者插入操作：
mysql> insert into film_text (film_id,title)
values(1002,'Test');
+---------+-------------------------+
| film_id | title |
+---------+-------------------------+
| 1001 | ACADEMY DINOSAUR |
+---------+-------------------------+
ERROR 1099 (HY000): *Table 'film_text' was locked with a READ lock and can't be updated*
mysql> update film_text set title = 'Test' where film_id = 1001;
ERROR 1099 (HY000): *Table 'film_text' was locked with a READ lock and can't be updated*
	其他session也可以查询该表的记录
	mysql> select film_id,title from film_text where film_id = 1001;
	+---------+-------------------------+
	| film_id | title |
	+---------+-------------------------+
	| 1001 | ACADEMY DINOSAUR |
	+---------+-------------------------+
	1 row in set (0.00 sec)

当前session不能查询没有锁定的表
mysql> select film_id,title from film where film_id = 1001;
ERROR 1100 (HY000): *Table 'film' was not locked with LOCK TABLES*
	其他session可以进行插入操作，但是更新会等待：
	mysql> insert into film_text (film_id,title)
	values(1002,'Test');
	Query OK, 1 row affected (0.00 sec)
	mysql> update film_text set title = 'Update Test' where film_id = 1001;
	等待

当前session不能访问其他session插入的记录： mysql> select film_id,title from film_text where film_id = 1002; Empty set (0.00 sec)	
释放锁 mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待
当前session解锁后可以获得其他session插入的记录： mysql> select film_id,title from film_text where film_id = 1002; +---------+-------+ | film_id | title | +---------+-------+ | 1002 | Test | +---------+-------+ 1 row in set (0.00 sec)	**Session2获得锁，更新操作完成：** mysql> update film_text set title = 'Update Test' where film_id = 1001; Query OK, 1 row affected (1 min 17.75 sec) Rows matched: 1 Changed: 1 Warnings: 0

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。

MyISAM的锁调度

前面讲过，MyISAM存储引擎的读锁和写锁是互斥的，读写操作是串行的。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认为写请求一般比读请求要重要。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。

上面已经讨论了写优先调度机制带来的问题和解决办法。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

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。行级锁与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。

- 原子性（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。

在MVCC并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。
在一个支持MVCC并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？以MySQL InnoDB为例：

- 快照读：简单的select操作，属于快照读，不加锁。(当然，也有例外)

```
select * from table where ?;
```

- 当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。
  下面语句都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加S锁 (共享锁)外，其他的操作，都加的是X锁 (排它锁)。

```
select * from table where ? lock in share mode;
select * from table where ? for update;
insert into table values (…);
update table set ? where ?;
delete from table where ?;
```

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，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，应用可以根据自己的业务逻辑要求，通过选择不同的隔离级别来平衡 “隔离”与“并发”的矛盾。下表很好地概括了这4个隔离级别的特性。

	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	√	√	√
已提交读 Read committed	语句级	×	√	√
可重复读 Repeatable read	事务级	×	×	√
可序列化 Serializable	最高级别，事务级	×	×	×

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 (0.01 sec)
```


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。

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）：又称读锁。**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。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。
- **排他锁（X）：又称写锁。**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。若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。
- 对于共享锁大家可能很好理解，就是多个事务只能读数据不能改数据。
  对于排他锁大家的理解可能就有些差别，我当初就犯了一个错误，以为排他锁锁住一行数据后，其他事务就不能读取和修改该行数据，其实不是这样的。排他锁指的是一个事务在一行数据加上排他锁后，其他事务不能再在其上加其他的锁。mysql InnoDB引擎默认的修改数据语句：**update,delete,insert都会自动给涉及到的数据加上排他锁，select语句默认不会加任何锁类型**，如果加排他锁可以使用select ...for update语句，加共享锁可以使用select ... lock in share mode语句。**所以加过排他锁的数据行在其他事务种是不能修改数据的，也不能通过for update和lock in share mode锁的方式查询数据，但可以直接通过select ...from...查询数据，因为普通查询没有任何锁机制。**

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:

	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式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。**

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，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：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。

```
mysql> create table tab_no_index(id int,name varchar(10)) engine=innodb;
```

```
Query OK, 0 rows affected (0.15 sec)
```

```
mysql> insert into tab_no_index values(1,'1'),(2,'2'),(3,'3'),(4,'4');
Query OK, 4 rows affected (0.00 sec)
```

```
Records: 4 Duplicates: 0 Warnings: 0
```

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_no_index where id = 1 ; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_no_index where id =2 ; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)
mysql> select * from tab_no_index where id = 1 for update; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	
	mysql> select * from tab_no_index where id = 2 for update; 等待

在上面的例子中，看起来session_1只给一行加了排他锁，但session_2在请求其他行的排他锁时，却出现了锁等待！**原因就是在没有索引的情况下，InnoDB只能使用表锁。**当我们给其增加一个索引后，InnoDB就只锁定了符合条件的行，如下例所示：
创建tab_with_index表，id字段有普通索引：

```
mysql> create table tab_with_index(id int,name varchar(10)) engine=innodb;
mysql> alter table tab_with_index add index id(id);
```

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_no_index where id = 1 ; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_no_index where id =2 ; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)
mysql> select * from tab_no_index where id = 1 for update; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	
	mysql> select * from tab_no_index where id = 2 for update; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)

（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。应用设计的时候要注意这一点。

在下面的例子中，表tab_with_index的id字段有索引，name字段没有索引：

```
mysql> alter table tab_with_index drop index name;
```

Query OK, 4 rows affected (0.22 sec) Records: 4 Duplicates: 0
Warnings: 0

```
mysql> insert into tab_with_index  values(1,'4');
```

Query OK, 1 row affected (0.00 sec)

```
mysql> select * from tab_with_index where id = 1;
```

```
+------+------+
| id   | name |
+------+------+
| 1    | 1    |
| 1    | 4    |
+------+------+
2 rows in set (0.00 sec)
```

InnoDB存储引擎使用相同索引键的阻塞例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)
mysql> select * from tab_with_index where id = 1 and name = '1' for update; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	
	虽然session_2访问的是和session_1不同的记录，但是因为使用了相同的索引，所以需要等待锁： mysql> select * from tab_with_index where id = 1 and name = '4' for update;

（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。

在下面的例子中，表tab_with_index的id字段有主键索引，name字段有普通索引：

```
mysql> alter table tab_with_index add index name(name);
```

Query OK, 5 rows affected (0.23 sec) Records: 5 Duplicates: 0
Warnings: 0

InnoDB存储引擎的表使用不同索引的阻塞例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)
mysql> select * from tab_with_index where id = 1 for update; +------+------+ | id | name | +------+------+ | 1 | 1 | | 1 | 4 | +------+------+ 2 row in set (0.00 sec)	
	Session_2使用name的索引访问记录，因为记录没有被锁定，所以可以获得锁： mysql> select * from tab_with_index where name = '2' for update; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)
	由于访问的记录已经被session_1锁定，所以等待获得锁： mysql> select * from tab_with_index where name = '4' for update; 等待

（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。
比如，在tab_with_index表里的name字段有索引，但是name字段是varchar类型的，检索值的数据类型与索引字段不同，虽然MySQL能够进行数据类型转换，但却不会使用索引，从而导致InnoDB使用表锁。通过用explain检查两条SQL的执行计划，我们可以清楚地看到了这一点。

```
mysql> explain select * from tab_with_index where name = 1 \G
mysql> explain select * from tab_with_index where name = '1' \G
```

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）。
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；另外一方面，是为了满足其恢复和复制的需 要。有关其恢复和复制对锁机制的影响，以及不同隔离级别下InnoDB使用间隙锁的情况，在后续的章节中会做进一步介绍。

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！下面这个例子假设emp表中只有101条记录，其empid的值分别是1,2,……,100,101。
InnoDB存储引擎的间隙锁阻塞例子

session_1	session_2
mysql> select @@tx_isolation; +-----------------+ | @@tx_isolation | +-----------------+ | REPEATABLE-READ | +-----------------+ 1 row in set (0.00 sec) mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec)	mysql> select @@tx_isolation; +-----------------+ | @@tx_isolation | +-----------------+ | REPEATABLE-READ | +-----------------+ 1 row in set (0.00 sec) mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec)
当前session对不存在的记录加for update的锁： mysql> select * from emp where empid = 102 for update; Empty set (0.00 sec)	
	这时，如果其他session插入empid为201的记录（注意：这条记录并不存在），也会出现锁等待： mysql>insert into emp(empid,...) values(201,...); 阻塞等待
Session_1 执行rollback： mysql> rollback; Query OK, 0 rows affected (13.04 sec)	
	由于其他session_1回退后释放了Next-Key锁，当前session可以获得锁并成功插入记录： mysql>insert into emp(empid,...) values (201,...); Query OK, 1 row affected (13.35 sec)

小结

本文重点介绍了MySQL中MyISAM表级锁和InnoDB行级锁的实现特点，并讨论了两种存储引擎经常遇到的锁问题和解决办法。

对于MyISAM的表锁，主要讨论了以下几点：
（1）共享读锁（S）之间是兼容的，但共享读锁（S）与排他写锁（X）之间，以及排他写锁（X）之间是互斥的，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查询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_PRIORITY_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，本文主要讨论了以下几项内容：
（1）InnoDB的行锁是基于索引实现的，如果不通过索引访问数据，InnoDB会使用表锁。
（2）介绍了InnoDB间隙锁（Next-key)机制，以及InnoDB使用间隙锁的原因。
在不同的隔离级别下，InnoDB的锁机制和一致性读策略不同。

在了解InnoDB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；
- 给记录集显式加锁时，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。

分类: 数据库

weblee
关注 - 1
粉丝 - 19
+加关注

« 上一篇： MySQL数据类型和常用字段属性总结
» 下一篇： PHP的运行机制与原理(底层)

posted on 2018-01-29 18:23 weblee 阅读(29984) 评论(3) 编辑 收藏

评论:

#1楼 2019-05-19 23:01 | 依旧十八岁的夕阳小子
这是我目前看的关于MySQL锁最好的文章!

支持(2) 反对(0)

#2楼 2019-08-23 16:25 | 萤火爱上夏
博主写的关于mysql锁的文章 读起来通俗易懂 让我对数据库锁有了更进一步的了解 博主的语言组织能力很棒！！！

支持(0) 反对(0)

#3楼 2019-09-22 16:01 | zofun
感谢博主，写的太棒了。

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册， 访问 网站首页。

相关博文:
· MySQL学习笔记(五)：MySQL表级锁和行级锁
· MySQL表级锁和行级锁
· MySQL表级锁和行级锁
· [mysql]锁机制
· mysql--锁
» 更多推荐...

最新 IT 新闻:
· 中国电信正式发布工业互联网开放平台
· AWS公司在阿根廷购买土地计划建设三个数据中心
· 戚发轫院士获国际宇航联合会“名人堂（Hall of Fame）”奖