**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。

在如表20-9所示的例子中，开始tab\_no\_index表没有索引：

|  |
| --- |
| 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 |

表20-9         InnoDB存储引擎的表在不使用索引时使用表锁例子

|  |  |
| --- | --- |
| **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 |

在如表20-9所示的例子中，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，如表20-10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; Query OK, 0 rows affected (0.15 sec) mysql> alter table tab\_with\_index add index id(id); Query OK, 4 rows affected (0.24 sec) Records: 4 Duplicates: 0 Warnings: 0 |

表20-10    InnoDB存储引擎的表在使用索引时使用行锁例子

|  |  |
| --- | --- |
| **session\_1** | **session\_2** |
| mysql> set autocommit=0;  Query OK, 0 rows affected (0.00 sec)  mysql> select \* from tab\_with\_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\_with\_index where id = 2 ;  +------+------+  | id   | name |  +------+------+  | 2    | 2    |  +------+------+  1 row in set (0.00 sec) |
| mysql> select \* from tab\_with\_index where id = 1 for update;  +------+------+  | id   | name |  +------+------+  | 1    | 1    |  +------+------+  1 row in set (0.00 sec) |  |
|  | mysql> select \* from tab\_with\_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。

在如表20-11所示的例子中，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) |

表20-11    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;  等待 |

（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。

在如表20-12所示的例子中，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 |

表20-12    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 rows in set (0.00 sec) |  |
|  | Session\_2使用name的索引访问记录，因为记录没有被索引，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; |

（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。关于MySQL在什么情况下不使用索引的详细讨论，参见本章“索引问题”一节的介绍。

在下面的例子中，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。

例子中tab\_with\_index表的name字段有索引，但是name字段是varchar类型的，如果where条件中不是和varchar类型进行比较，则会对name进行类型转换，而执行的全表扫描。

|  |
| --- |
| mysql> alter table tab\_no\_index add index name(name); Query OK, 4 rows affected (8.06 sec) Records: 4 Duplicates: 0 Warnings: 0  mysql> explain select \* from tab\_with\_index where name = 1 \G \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* id: 1 select\_type: SIMPLE table: tab\_with\_index type: ALL possible\_keys: name key: NULL key\_len: NULL ref: NULL rows: 4 Extra: Using where 1 row in set (0.00 sec) mysql> explain select \* from tab\_with\_index where name = '1' \G \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* id: 1 select\_type: SIMPLE table: tab\_with\_index type: ref possible\_keys: name key: name key\_len: 23 ref: const rows: 1 Extra: Using where 1 row in set (0.00 sec) |