

1. 事务隔离级别

Read Uncommitted (读取未提交内容)

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读 (Dirty Read)。

Read Committed (读取提交内容)

这是大多数数据库系统的默认隔离级别 (但不是 MySQL 默认的)。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读 (Nonrepeatable Read)，因为同一事务的其他实例在该实例处理其间可能会有新的 commit，所以同一 select 可能返回不同结果。

Repeatable Read (可重读)

这是 MySQL 的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。InnoDB 和 Falcon 存储引擎通过多版本并发控制 (MVCC, Multiversion Concurrency Control) 机制消除幻读或者锁机制保证对读取的范围加锁，新的满足查询条件的记录不能够插入。

Serializable (可串行化)

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

这四种隔离级别采取不同的锁类型来实现，若读取的是同一个数据的话，就容易发生问题。例如：

脏读 (Dirty Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个 RollBack 了操作，则后一个事务所读取的数据就会是不正确的。

不可重复读 (Non-repeatable read)：在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读 (Phantom Read)：在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列 (Row) 数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

在 MySQL 中，实现了这四种隔离级别，分别有可能产生问题如下所示：

隔离级别	脏读	不可重复读	幻读
读未提交（Read uncommitted）	V	V	V
读已提交（Read committed）	X	V	V
可重复读（Repeatable read）	X	X	V
可串行化（Serializable）	X	X	X

▪ 2. 锁类型

↵

表锁，行锁，InnoDB 引擎支持行锁和表锁，MyISAM 只支持表锁

InnoDB 表锁方式

Lock tables [tablename] [mode]

lock tables test2 read, 给表 test2 加了读表锁

lock tables test2 write 给表 test2 加了写表锁

▪ 行锁(Shared and Exclusive Locks)

共享锁(又称读锁)，排他锁(又称写锁)，InnoDB 标准的行级锁

共享锁 (S)：允许一个事务去读一行，允许其他事务取得共享锁，阻止其他事务获得相同数据集的排他锁。

排他锁 (X)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

共享锁 (S)：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE。

排他锁 (X)：SELECT * FROM table_name WHERE ... FOR UPDATE。

▪ 意向锁(Intention Locks)

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

意向锁是 InnoDB 隐式添加的，无需用户关注

意向锁的作用

①在 mysql 中有表锁，↵

`LOCK TABLE my_table_name READ;` 用读锁锁表，会阻塞其他事务修改表数据。↵

`LOCK TABLE my_table_name WRITE;` 用写锁锁表，会阻塞其他事务读和写。↵

②InnoDB引擎又支持行锁，行锁分为↵

共享锁，一个事务对一行的共享只读锁。↵

排它锁，一个事务对一行的排他读写锁。↵

③这两中类型的锁共存的问题↵

考虑这个例子：↵

事务 A 锁住了表中的一行，让这一行只能读，不能写。↵

之后，事务 B 申请整个表的写锁。↵

如果事务 B 申请成功，那么理论上它就能修改表中的任意一行，这与 A 持有的行锁是冲突的。↵

数据库需要避免这种冲突，就是说要让 B 的申请被阻塞，直到 A 释放了行锁。↵

数据库要怎么判断这个冲突呢？↵

step1：判断表是否已被其他事务用表锁锁表↓

step2：判断表中的每一行是否已被行锁锁住。↵

— 注意 step2，这样的判断方法效率实在不高，因为需要遍历整个表。↓

于是就有了意向锁。↵

在意向锁存在的情况下，事务 A 必须先申请表的意向共享锁，成功后再申请一行的行锁。↵

在意向锁存在的情况下，上面的判断可以改成↵

step1：不变↓

step2：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务 B 申请表的写锁会被阻塞。↵

以上皆为方便理解意向锁的诞生初衷，但是...

A lock is granted to a requesting transaction if it is compatible with existing locks, but not if it conflicts with existing locks. A transaction waits until the conflicting existing lock is released. If a lock request conflicts with an existing lock and cannot be granted because it would cause [deadlock](#), an error occurs.

Thus, intention locks do not block anything except full table requests (for example, LOCK TABLES ... WRITE). The main purpose of IX and IS locks is to show that someone is locking a row, or going to lock a row in the table.

Transaction data for an intention lock appears similar to the following in [SHOW ENGINE INNODB STATUS](#) and [InnoDB monitor](#) output:

TABLE LOCK table `test`.`t` trx id 10080 lock mode IX

官网这段话意思是意向锁对于非表锁事务是不会阻塞，只是起一个告示作用，只有遇到表锁的情况才会阻塞

共享锁，排他锁相容矩阵

	<i>X</i>	<i>IX</i>	<i>S</i>	<i>IS</i>
<i>X</i>	Conflict	Conflict	Conflict	Conflict
<i>IX</i>	Conflict	Compatible	Conflict	Compatible
<i>S</i>	Conflict	Conflict	Compatible	Compatible
<i>IS</i>	Conflict	Compatible	Compatible	Compatible

索引记录锁(Record Locks)

作用在索引数据上的锁， 当一个表有索引的时候， 事务会先尝试取得索引记录锁

验证

```
mysql> desc test3;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   |     | NULL    |      |
| name  | varchar(100)  | YES  |     | NULL    |      |
| value | varchar(100)  | YES  |     | NULL    |      |
| sidx  | varchar(20)   | YES  | MUL | NULL    |      |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

test3 表没有主键，只有 sidx 索引。

```
mysql>
mysql>
mysql>
mysql>
mysql> select * from test3 where sidx='22' for update;
+----+-----+-----+-----+
| id | name | value | sidx |
+----+-----+-----+-----+
| 2  | 22   | NULL  | 22   |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

```
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 267, OS thread handle 10320, query id 734 localhost ::1 root Sending data
select * from test3 where sidx='22' for update
----- TRX HAS BEEN WAITING 7 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 29 page no 4 n bits 72 index test3_sidx of table 'test`.`test3` trx id 2059 lock_mode X waiting
Record lock, heap no 3 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
 0: len 2; hex 3232; asc 22;;
 1: len 6; hex 000000000211; asc      ;;
-----
--TRANSACTION 2058 ACTIVE 9 sec
```

执行 `show engine innodb status`，可以看到 test3_sidx 索引上 x 锁阻塞。

在有主键索引情况。

```
mysql> desc test2;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   | PRI | NULL    |       |
| name  | varchar(100)  | YES  |     | NULL    |       |
| value | varchar(100)  | YES  |     | NULL    |       |
| sidx  | varchar(20)   | YES  | MUL | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

mysql>
```

```
mysql>
mysql>
mysql>
mysql>
mysql> select * from test2 where id=2 for update;
+----+-----+-----+-----+
| id | name | value | sidx |
+----+-----+-----+-----+
| 2  | 22   | NULL  | 22   |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

```
mysql>
mysql>
mysql>
mysql>
mysql> select * from test2 where sidx='22' for update;
+----+-----+-----+-----+
| id | name | value | sidx |
+----+-----+-----+-----+
| 2  | 22   | NULL  | 22   |
+----+-----+-----+-----+
1 row in set (7.10 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

```
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 1136, 2 row lock(s)
MySQL thread id 267, OS thread handle 10320, query id 740 localhost ::1 root Sending data
select * from test2 where sidx='22' for update
----- TRX HAS BEEN WAITING 4 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 28 page no 3 n bits 80 index PRIMARY of table 'test'. 'test2' trx id 2061 lock_mode X locks rec but not gap waiting
Record lock, heap no 8 PHYSICAL RECORD: n_fields 6; compact format; info bits 0
0: len 4; hex 80000002; asc ;;
1: len 6; hex 0000000007cb; asc ;;
2: len 7; hex 3500000140025a; asc 5 @ Z;;
3: len 2; hex 3232; asc 22;;
4: SQL NULL;
5: len 2; hex 3232; asc 22;;
```

执行 `show engine innodb status`, 可以看到主键索引上 x 锁阻塞, 事务 2 先在 `sidx='22'` 获取二级索引的 x 锁, 再通过二级索引找到 `id=2` 的主键, 尝试获取主键索引 x 锁, 但主键索引 x 锁已经被事务 1 获取, 所以阻塞。

A record lock is a lock on an index record. For example, `SELECT c1 FROM t WHERE c1 = 10 FOR UPDATE`; prevents any other transaction from inserting, updating, or deleting rows where the value of `t.c1` is 10.

Record locks always lock index records, even if a table is defined with no indexes. For such cases, InnoDB creates a hidden clustered index and uses this index for record locking.

当表没有定义索引的时候, 引擎会创建一个隐藏聚合索引。

验证。

```
1 row in set (0.00 sec)

mysql> desc test4;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   |     | NULL    |       |
| name  | varchar(100)  | YES  |     | NULL    |       |
| value | varchar(100)  | YES  |     | NULL    |       |
| sidx  | varchar(20)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

test4 表没有主键, 没有索引。

```
mysql>
mysql>
mysql>
mysql> select * from test4 where id=2 for update;
+----+----+----+----+
| id | name | value | sidx |
+----+----+----+----+
| 2  | 22   | NULL  | 22   |
+----+----+----+----+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> select * from test3 where sidx=3 for update;
Empty set (5.80 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> select * from test4 where sidx=3 for update;
Empty set (10.37 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```



```
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 267, OS thread handle 10320, query id 694 localhost ::1 root Sending data
select * from test4 where sid=3 for update
----- TRX HAS BEEN WAITING 7 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 30 page no 3 n bits 72 index GEN_CLUST_INDEX of table `test`.`test4` trx id 2046 lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 7; compact format; info bits 0
0: len 6; hex 000000000215; asc      ;;
1: len 6; hex 0000000007f8; asc      ;;
2: len 7; hex b90000012d0110; asc  -  ;;
3: len 4; hex 80000001; asc      ;;
4: len 2; hex 3131; asc 11;;
5: SQL NULL;
6: len 2; hex 3131; asc 11;;
```

执行 `show engine innodb status` 可以看到 `GEN_CLUST_INDEX` 这段

间隙锁(Gap lock)

间隙锁, 所谓 gap 是指两个索引之间的 gap。每一个 gap 也有一把锁, 称为 gap lock。在第一条数据之前, 最后一条数据之后, 也各有一个 gap, 所以也有 gap lock。

gap	Index record	gap	Index record	gap	Index record	gap	Index record	gap
-----	--------------	-----	--------------	-----	--------------	-----	--------------	-----

Gap Lock 可以有效的避免幻读发生。例如一个事务 A 在执行 `SELECT c1 FROM t WHERE c1 BETWEEN 10 and 20 FOR UPDATE;` 同时另一个事务 B 要 insert 一个 `c1=15` 的行。此时事务 B 是拿不到 gap lock 的, 因为 10 到 20 直接的 gaps locks 都被事务 A 持有。此时并不会管有没有一条 `c1=15` 的记录存在, 事务 B 都拿不到 Gap

Gap locking can be disabled explicitly. This occurs if you change the transaction isolation level to `READ COMMITTED` or enable the `innodb locks unsafe for binlog` system variable. Under these circumstances, gap locking is disabled for searches and index scans and is used only for foreign-key constraint checking and duplicate-key checking.

只有在 `REPEATABLE-READ` 隔离级别的时候 gap lock 才有效

验证

```
mysql> desc test2;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   | PRI | NULL    |       |
| name  | varchar(100)  | YES  |     | NULL    |       |
| value | varchar(100)  | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

```
mysql>
mysql>
mysql>
mysql>
mysql> select * from test2 where id>=2 and id<=5 lock in share mode;
+-----+-----+-----+-----+
| id | name | value | sid |
+-----+-----+-----+-----+
| 2  | 22   | NULL  | 22  |
| 3  | 33   | NULL  | 33  |
| 5  | 55   | NULL  | 55  |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> insert into test2 values(4,44,null,'44');
Query OK, 1 row affected (12.11 sec)

mysql> commit;
Query OK, 0 rows affected (0.07 sec)

mysql>
```

事务 2 插入数据被阻塞。

```
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 281475154251264, not started
0 lock struct(s), heap size 1136, 0 row lock(s)
---TRANSACTION 2064, ACTIVE 3 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 267, OS thread handle 10320, query id 747 localhost ::1 root update
insert into test2 values(4,44,null,'44')
----- TRX HAS BEEN WAITING 3 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 28 page no 3 n bits 80 index PRIMARY of table 'test`.`test2`
trx id 2064 lock_mode X locks gap before rec insert intention waiting
Record lock, heap no 11 PHYSICAL RECORD: n_fields 6; compact format; info bits 0

0: len 4; hex 80000005; asc      ;;
1: len 6; hex 0000000007cb; asc      ;;
2: len 7; hex 350000014002bd; asc 5  @ ;;
3: len 2; hex 3535; asc 55;;
4: SQL NULL;
5: len 2; hex 3535; asc 55;;
```

执行 `show engine innodb status`, 看到 gap 锁阻塞了事务 2。

Next-Key Lock

一个 next-key lock 是结合了一个 index lock 和它之前的 gap lock。

InnoDB 的默认隔离级别是: REPEATABLE_READ, 这种隔离级别下, InnoDB 使用在 index scan 时, 采用的是 next-key。Next-key 本身不存在, 只代表了 index lock 和它之前的 gap lock。

插入意向锁(Insert Intention Locks)

Gap Lock 中存在一种插入意向锁 (Insert Intention Lock), 在 insert 操作时产生。在多事务同时写入不同数据至同一索引间隙的时候, 并不需要等待其他事务完成, 不会发生锁等待。

假设有一个记录索引包含键值 4 和 7, 不同的事务分别插入 5 和 6, 每个事务都会产生一个加在 4-7 之间的插入意向锁, 获取在插入行上的排它锁, 但是不会被互相锁住, 因为数据行并不冲突。

An insert intention lock is a type of gap lock set by [INSERT](#) operations prior to row insertion. This lock signals the intent to insert in such a way that multiple transactions inserting into the same index gap need not wait for each other if they are not inserting at the same position within the gap. Suppose that there are index records with values of 4 and 7. Separate transactions that attempt to insert values of 5 and 6, respectively, each lock the gap between 4 and 7 with insert intention locks prior to obtaining the exclusive lock on the inserted row, but do not block each other because the rows are [nonconflicting](#).

insert 加锁过程

insert 会在 insert 的行对应的索引记录上加一个排它锁，这是一个 record lock，并没有 gap，所以并不会阻塞其他 session 在 gap 间隙里插入记录。

不过在 insert 操作之前，还会加一种锁，官方文档称它为 insertion intention gap lock，也就是意向的 gap 锁。这个意向 gap 锁的作用就是预示着当多事务并发插入相同的 gap 空隙时，只要插入的记录不是 gap 间隙中的相同位置，则无需等待其他 session 就可完成，这样就使得 insert 操作无须加真正的 gap lock。

假设有一个记录索引包含键值 4 和 7，不同的事务分别插入 5 和 6，每个事务都会产生一个加在 4-7 之间的插入意向锁，获取在插入行上的排它锁，但是不会被互相锁住，因为数据行并不冲突。

假设发生了一个唯一键冲突错误，那么将会在重复的索引记录上加读锁。当有多个 session 同时插入相同的行记录时，如果另外一个 session 已经获得该行的排它锁，那么将会导致死锁。

INSERT sets an exclusive lock on the inserted row. This lock is an index-record lock, not a next-key lock (that is, there is no gap lock) and does not prevent other sessions from inserting into the gap before the inserted row.

Prior to inserting the row, a type of gap lock called an insert intention gap lock is set. This lock signals the intent to insert in such a way that multiple transactions inserting into the same index gap need not wait for each other if they are not inserting at the same position within the gap. Suppose that there are index records with values of 4 and 7. Separate transactions that attempt to insert values of 5 and 6 each lock the gap between 4 and 7 with insert intention locks prior to obtaining the exclusive lock on the inserted row, but do not block each other because the rows are nonconflicting.

If a duplicate-key error occurs, a shared lock on the duplicate index record is set. This use of a shared lock can result in deadlock should there be multiple sessions trying to insert the same row if another session already has an exclusive lock.

自增锁(AUTO-INC Locks)

AUTO-INC 锁是一种特殊的表级锁，发生涉及 AUTO_INCREMENT 列的事务性插入操作时产生。

An AUTO-INC lock is a special table-level lock taken by transactions inserting into tables with AUTO_INCREMENT columns. In the simplest case, if one transaction is inserting values into the table, any other transactions must wait to do their own inserts into that table, so that rows inserted by the first transaction receive consecutive primary key values.

3.事务隔离级别和锁的关系

MySQL 官网链接

<https://dev.mysql.com/doc/refman/5.5/en/innodb-transaction-isolation-levels.html>

<https://dev.mysql.com/doc/refman/5.5/en/innodb-transaction-isolation-levels.html>

这里提及了一个重要概念↵

The second UPDATE blocks as soon as it tries to acquire any locks (because first update has retained locks on all rows), and does not proceed until the first UPDATE commits or rolls back:↵

REPEATABLE-READ 级别的情况下，使用没有索引的字段进行查询，由于所有记录没有索引，扫描所有记录的时候不管是否匹配条件，这些记录都被锁上↵

If READ COMMITTED is used instead, the first UPDATE acquires x-locks and releases those for rows that it does not modify:↵

而在 READ-COMMITTED 级别的情况，同样会扫描所有记录并对所有记录上锁，但会对不匹配的记录解锁↵

举例↵

```
mysql> desc test2;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   | PRI | NULL    |       |
| name  | varchar(100)  | YES  |     | NULL    |       |
| value | varchar(100)  | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

```
mysql> select * from test2;
+----+-----+-----+
| id | name | value |
+----+-----+-----+
| 1  | 11   | NULL  |
| 2  | 22   | NULL  |
| 3  | 33   | NULL  |
| 5  | 55   | NULL  |
+----+-----+-----+
4 rows in set (0.00 sec)
```

test2 表有仅主键索引↵

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

mysql> select * from test2 where name='22' for update;
+----+-----+-----+
| id | name | value |
+----+-----+-----+
| 2  | 22   | NULL  |
+----+-----+-----+
1 row in set (0.01 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>

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

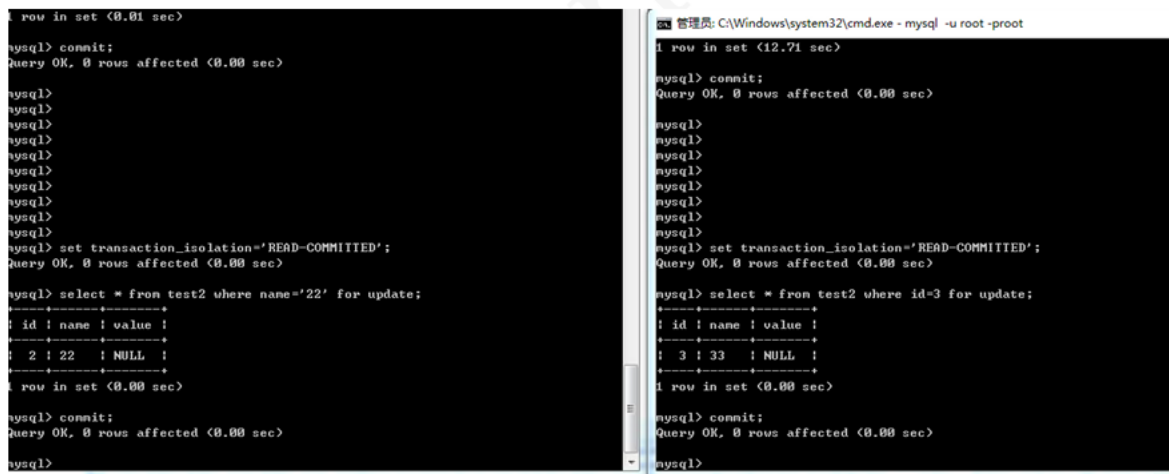
mysql> select * from test2 where id=3 for update;
+----+-----+-----+
| id | name | value |
+----+-----+-----+
| 3  | 33   | NULL  |
+----+-----+-----+
1 row in set (12.71 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

第二个事务执行 select 的时候被阻塞↵

换成 READ-COMMITTED。



```
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> set transaction_isolation='READ-COMMITTED';
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test2 where name='22' for update;
+----+-----+-----+
| id | name | value |
+----+-----+-----+
| 2  | 22   | NULL  |
+----+-----+-----+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>

C:\Windows\system32\cmd.exe - mysql -u root -proot
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> set transaction_isolation='READ-COMMITTED';
Query OK, 0 rows affected (0.00 sec)

mysql> select * from test2 where id=3 for update;
+----+-----+-----+
| id | name | value |
+----+-----+-----+
| 3  | 33   | NULL  |
+----+-----+-----+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

第二个事务执行 select 没有被阻塞。

锁索引公式

- 如果id列是主键索引，MySQL只会为聚簇索引记录加锁。
- 如果id列是唯一二级索引，MySQL会为二级索引叶子节点以及聚簇索引记录加锁。
- 如果id列是非唯一索引，MySQL会为所有满足条件(id = 15)的二级索引叶子节点以及对应的聚簇索引记录加锁。
- 如果id列是无索引的，SQL会走聚簇索引全表扫描，并将扫描结果加载到SQL Server层进行过滤，因此InnoDB会为扫描过的所有记录先加上锁，如果SQL Server层过滤不符合条件，InnoDB会释放该锁。

4.MVCC

MVCC 的目的就是减少锁的竞争，MVCC 是行级锁的一个变种，使大多数的读操作可以不用加锁，写操作也只锁定必要的行。

MVCC 只针对 innodb 的 repeatable read 和 read committed 这两种隔离级别。多版本并发控制的原理就是在每个记录行后面增加三个标示列用来存储该行的状态，分别存储该行的 DB_TRX_ID 和 DB_ROLL_PTR, DB_ROW_ID。

Internally, InnoDB adds three fields to each row stored in the database. A 6-byte DB_TRX_ID field indicates the transaction identifier for the last transaction that inserted or updated the row. Also, a deletion is treated internally as an update where a special bit in the row is set to mark it as deleted. Each row also contains a 7-byte DB_ROLL_PTR field called the roll pointer. The roll pointer points to an undo log record written to the rollback segment. If the row was updated, the undo log record contains the information necessary to rebuild the content of the row before it was updated. A 6-byte DB_ROW_ID field contains a row ID that increases monotonically as new rows are inserted. If InnoDB generates a clustered index automatically, the index contains row ID values. Otherwise, the DB_ROW_ID column does not appear in any index。

- DB_TRX_ID 存储创建，更新，删除的事务 ID 号。
- DB_ROLL_PTR 是回滚指针，指向 UNDO LOG 里的上条日志。
- DB_ROW_ID 隐藏 ID。

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。

如: `select * from table where ?;`

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。

如:

参考文档: <https://www.zhihu.com/question/323723320/answer/678767369>

<https://my.oschina.net/alchemystar?tab=newest&catalogId=5646093>

`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 ?;`

↵

· MVCC 实现的依赖项

MVCC 在 mysql 中的实现依赖的是 undo log 与 read view。

1. undo log: undo log 中记录的是数据表记录行的多个版本，也就是事务执行过程中的回滚段，其实就是 MVCC 中的一行原始数据的多个版本镜像数据。

2. read view: 主要用来判断当前版本数据的可见性，用于一致性读的 snapshot，InnoDB 里称为视图；在需要一致性读时开启一个视图，记录当时的事务状态快照，包括当时活跃的事务 ID 以及事务 ID 的上下水位值，以此用于判断数据的可见性。

· READ_VIEW

innodb 通过 read view 来确定一致性读时的数据库 snapshot，innodb 的 read view 确定一条记录能否看到，有两条法则

↵

1 看不到 read view 创建时刻以后启动的事务

2 看到 read view 创建时活跃的事务

Read view 相关属性如下

```
dulint    low_limit_id;      /* 事务号 >= low_limit_id 的记录，对于当前 Read View 都是不可见的 */  
  
    dulint    up_limit_id;      /* 事务号 < up_limit_id , 对于当前 Read View 都是可见的 */  
  
    ulint    n_trx_ids;      /* Number of cells in the trx_ids array */  
  
    dulint*    trx_ids;      /* Additional trx_ids which the read should  
                                not see: typically, these are the active  
                                transactions at the time when the read is  
                                serialized, except the reading transaction  
                                itself; the trx_ids in this array are in a  
                                descending order */  
  
    dulint    creator_trx_id;      /* trx_id of creating transaction */
```

PS: 活跃事务就是指事务开启后并且有执行 sql 还没有 commit 或者 rollback 的事务, 可以通过 SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX 获取

为什么 repeatable read 和 read committed select 会看到不一样的结果, 那是因为两个隔离级别 read view 生成机制不一样导致

read view 的生成机制:

read-committed:

函数: ha_innobase::external_lock
if (trx->isolation_level <= TRX_ISO_READ_COMMITTED)
 && trx->global_read_view) {
 / At low transaction isolation levels we let
 each consistent read set its own snapshot /
 read_view_close_for_mysql(trx);

即: 在每次语句执行的过程中, 都关闭 read view, 重新在 row_search_for_mysql 函数中创建当前的一份 read view。

这样就可以根据当前的全局事务链表创建 read view 的事务区间, 实现 read committed 隔离级别。

2. repeatable read:

在 repeatable read 的隔离级别下, 创建事务 trx 结构的时候, 就生成了当前的 global read view。

使用 trx_assign_read_view 函数创建, 一直维持到事务结束, 这样就实现了 repeatable read 隔离级别。

5. 幻读

MVCC 可以避免幻读。但是 MVCC 只对 SELECT 语句有效，对于 SELECT ... [LOCK IN SHARE MODE | FOR UPDATE]，UPDATE，DELETE 语句无效。

为了能够通过锁避免幻读，采用了 GAP LOCK 通过锁住记录之间的间隙，来阻止 INSERT 操作。

。

因为 READ COMMITTED 没有 GAP LOCK，导致没有办法锁定查询区间，其他事务可以插入数据到查询区间里，导致幻读。

6. 网上经典例子

查询组合：

- **组合一**：id 列是主键，RC 隔离级别。
- **组合二**：id 列是二级唯一索引，RC 隔离级别。
- **组合三**：id 列是二级非唯一索引，RC 隔离级别。
- **组合四**：id 列上没有索引，RC 隔离级别。
- **组合五**：id 列是主键，RR 隔离级别。
- **组合六**：id 列是二级唯一索引，RR 隔离级别。
- **组合七**：id 列是二级非唯一索引，RR 隔离级别。
- **组合八**：id 列上没有索引，RR 隔离级别。
- **组合九**：Serializable 隔离级别。

排列组合还没有列举完全，但是看起来，已经很多了。真的有必要这么复杂吗？事实上，要分析加锁，就是需要这么复杂。但是从另一个角度来说，只要你选定了一种组合，SQL 需要加哪些锁，其实也就确定了。接下来，就让我们来逐个分析这 9 种组合下的 SQL 加锁策略。

。

注：在前面八种组合下，也就是 RC，RR 隔离级别下，SQL1：select 操作均不加锁，采用的是快照读，因此在下面的讨论中就忽略了，主要讨论 SQL2：delete 操作的加锁。

1. 组合一：id 主键+RC

这个组合，是最简单，最容易分析的组合。id 是主键，Read Committed 隔离级别，给定 SQL: `delete from t1 where id = 10;` 只需要将主键上，id = 10 的记录加上 X 锁即可。如下图所示：

Table: T1(id primary key, name)

Primary Key

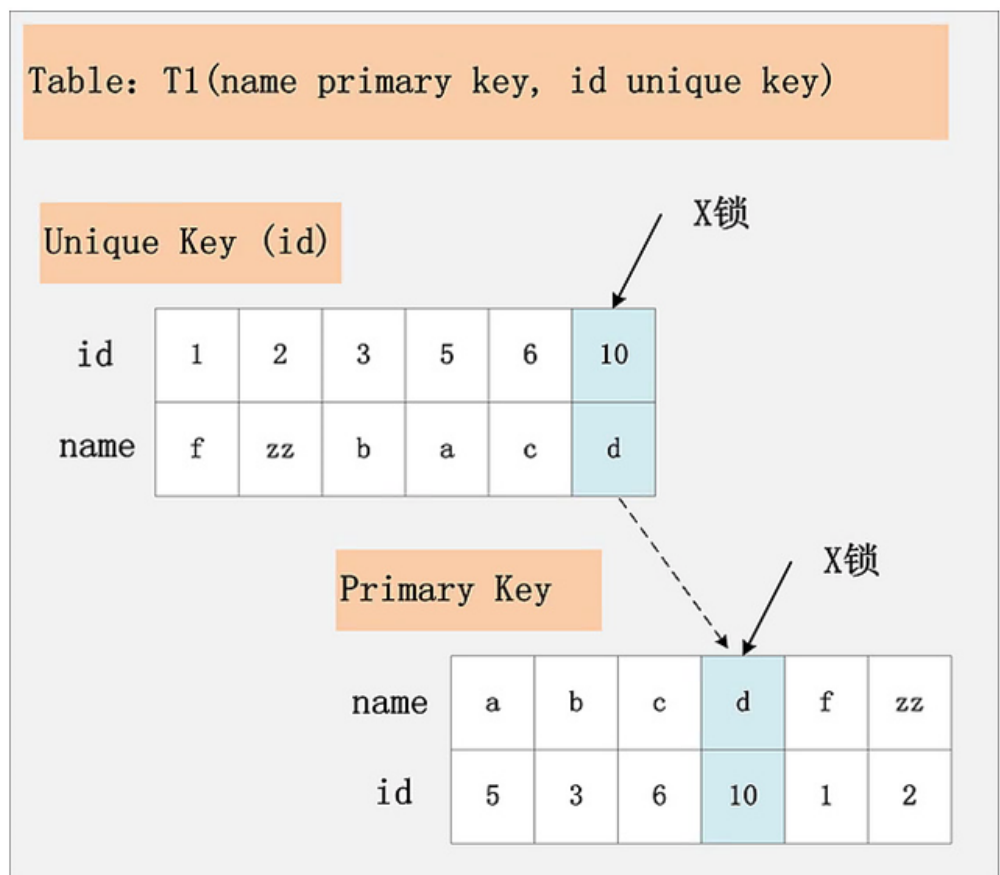
X锁

id	1	4	7	10	20	30
name	a	c	b	a	d	b

结论：id是主键时，此SQL只需要在id=10这条记录上加X锁即可。

2. 组合二：id 唯一索引+RC

这个组合，id 不是主键，而是一个 Unique 的二级索引键值。那么在 RC 隔离级别下，`delete from t1 where id = 10`；需要加什么锁呢？见下图：

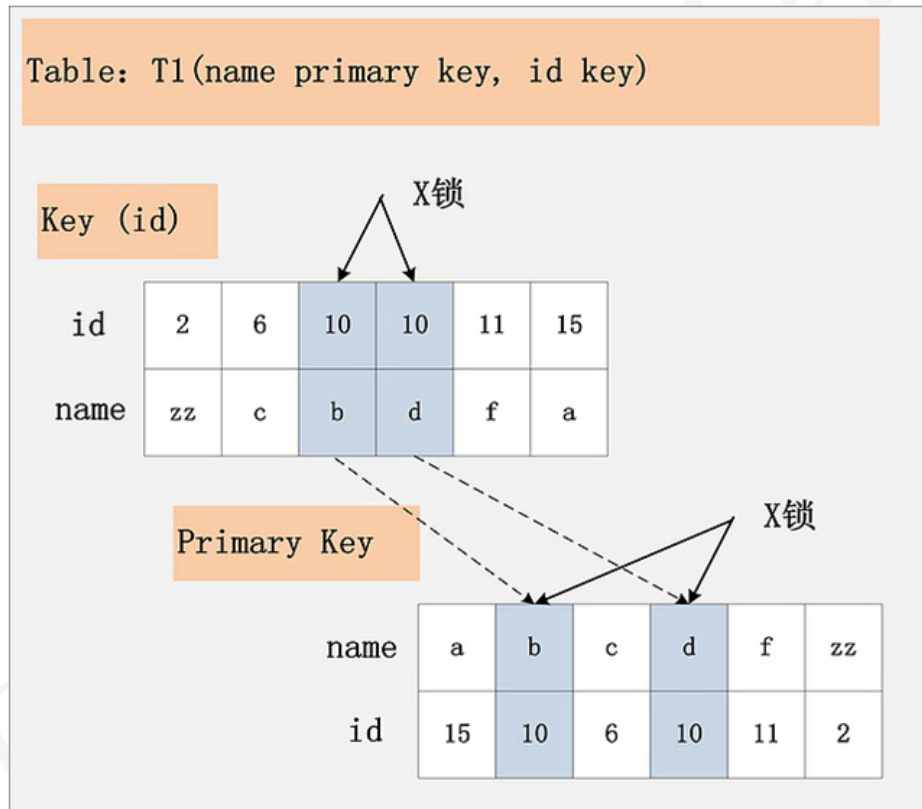


此组合中，id 是 unique 索引，而主键是 name 列。此时，加锁的情况由于组合一有所不同。由于 id 是 unique 索引，因此 delete 语句会选择走 id 列的索引进行 where 条件的过滤，在找到 id=10 的记录后，首先会将 unique 索引上的 id=10 索引记录加上 X 锁，同时，会根据读取到的 name 列，回主键索引（聚簇索引），然后将聚簇索引上的 name = 'd' 对应的主键索引项加 X 锁。为什么聚簇索引上的记录也要加锁？试想一下，如果并发的一个 SQL，是通过主键索引来更新：`update t1 set id = 100 where name = 'd'`；此时，如果 delete 语句没有将主键索引上的记录加锁，那么并发的 update 就会感知不到 delete 语句的存在，违背了同一记录上的更新/删除需要串行执行的约束。

结论：若 id 列是 unique 列，其上有 unique 索引。那么 SQL 需要加两个 X 锁，一个对应于 id unique 索引上的 id = 10 的记录，另一把锁对应于聚簇索引上的 [name='d', id=10] 的记录。

3. 组合三：id 非唯一索引+RC

相对于组合一、二，组合三又发生了变化，隔离级别仍旧是 RC 不变，但是 id 列上的约束又降低了，id 列不再唯一，只有一个普通的索引。假设 `delete from t1 where id = 10;` 语句，仍旧选择 id 列上的索引进行过滤 where 条件，那么此时会持有哪些锁？同样见下图：

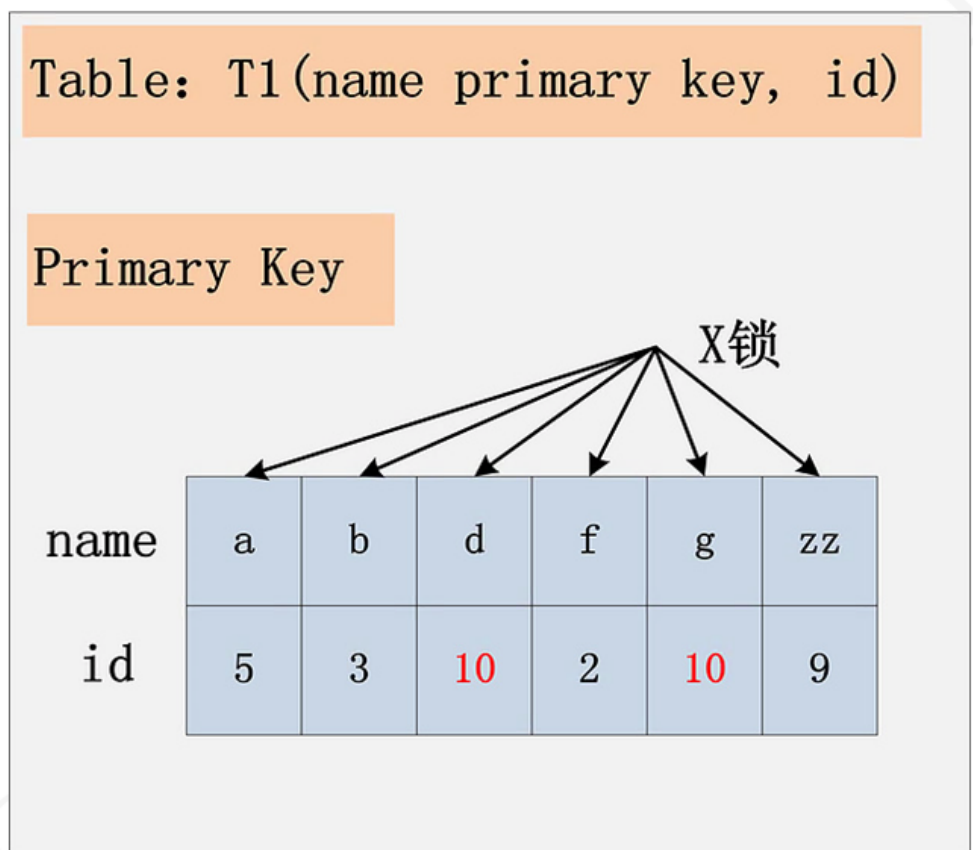


根据此图，可以看到，首先，id 列索引上，满足 `id = 10` 查询条件的记录，均已加锁。同时，这些记录对应的主键索引上的记录也都加上了锁。与组合二唯一的区别在于，组合二最多只有一个满足等值查询的记录，而组合三会将所有满足查询条件的记录都加锁。

结论：若 id 列上有非唯一索引，那么对应的所有满足 SQL 查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。

4. 组合四：id 无索引+RC

相对于前面三个组合，这是一个比较特殊的情况。id 列上没有索引，where id = 10; 这个过滤条件，没法通过索引进行过滤，那么只能走全表扫描做过滤。对应于这个组合，SQL 会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加 X 锁；有人说会将聚簇索引上，选择出来的 id = 10; 的记录加上 X 锁。那么实际情况呢？请看下图：



由于 id 列上没有索引，因此只能走聚簇索引，进行全部扫描。从图中可以看到，满足删除条件的记录有两条，但是，聚簇索引上所有的记录，都被加上了 X 锁。无论记录是否满足条件，全部被加上 X 锁。既不是加表锁，也不是在满足条件的记录上加行锁。

有人可能会问？为什么不是只在满足条件的记录上加锁呢？这是由于 MySQL 的实现决定的。如果一个条件无法通过索引快速过滤，那么存储引擎层面就会将所有记录加锁后返回，然后由 MySQL Server 层进行过滤。因此也就把所有的记录，都锁上了。

注：在实际的实现中，MySQL 有一些改进，在 MySQL Server 过滤条件，发现不满足后，会调用 unlock_row 方法，把不满足条件的记录放锁（违背了 2PL 的约束）。这样做，保证了最后只会持有满足条件记录上的锁，但是每条记录的加锁操作还是不能省略的。

结论：若 id 列上没有索引，SQL 会走聚簇索引的全扫描进行过滤，由于过滤是由 MySQL Server 层面进行的。因此每条记录，无论是否满足条件，都会被加上 X 锁。但是，为了效率考量，MySQL 做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了 2PL 的约束。

5. 组合五: id 主键+RR

上面的四个组合, 都是在 Read Committed 隔离级别下的加锁行为, 接下来的四个组合, 是在 Repeatable Read 隔离级别下的加锁行为。

组合五, id 列是主键列, Repeatable Read 隔离级别, 针对 delete from t1 where id = 10; 这条 SQL, 加锁与组合一: [id 主键, Read Committed] 一致。

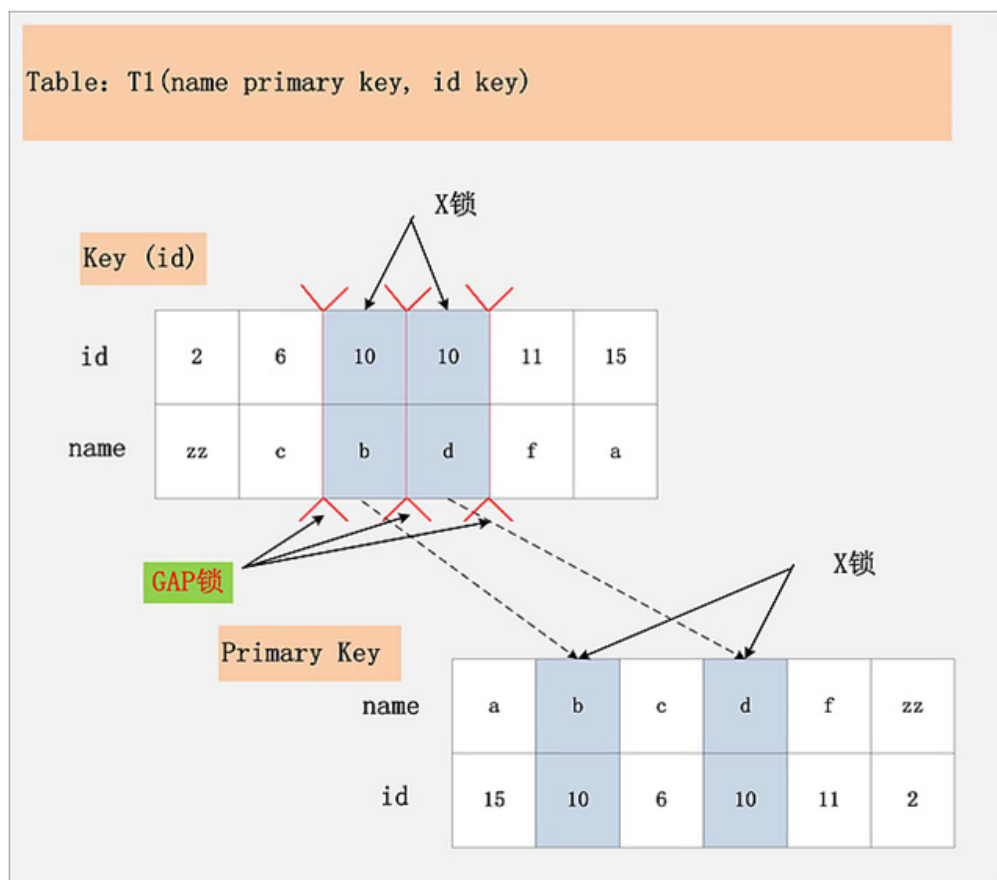
6. 组合六: id 唯一索引+RR

与组合五类似, 组合六的加锁, 与组合二: [id 唯一索引, Read Committed] 一致。两个 X 锁, id 唯一索引满足条件的记录上一个, 对应的聚簇索引上的记录一个。

7. 组合七: id 非唯一索引+RR

还记得前面提到的 MySQL 的四种隔离级别的区别吗? RC 隔离级别允许幻读, 而 RR 隔离级别, 不允许存在幻读。但是在组合五、组合六中, 加锁行为又是与 RC 下的加锁行为完全一致。那么 RR 隔离级别下, 如何防止幻读呢? 问题的答案, 就在组合七中揭晓。

组合七, Repeatable Read 隔离级别, id 上有一个非唯一索引, 执行 delete from t1 where id = 10; 假设选择 id 列上的索引进行条件过滤, 最后的加锁行为, 是怎样的呢? 同样看下面这幅图:



此图, 相对于组合三: [id 列上非唯一锁, Read Committed] 看似相同, 其实却有很大的区别。最大的区别在于, 这幅图中多了一个 GAP 锁, 而且 GAP 锁看起来也不是加在记录上的, 倒像是加载两条记录之间的位置, GAP 锁有何用?

其实这个多出来的 GAP 锁，就是 RR 隔离级别，相对于 RC 隔离级别，不会出现幻读的关键。确实，GAP 锁锁住的位置，也不是记录本身，而是两条记录之间的 GAP。所谓幻读，就是同一个事务，连续做两次当前读（例如：`select * from t1 where id = 10 for update;`），那么这两次当前读返回的是完全相同的记录（记录数量一致，记录本身也一致），第二次的当前读，不会比第一次返回更多的记录（幻象）。↵

如何保证两次当前读返回一致的记录，那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。为了实现这个功能，GAP 锁应运而生。↵

如图中所示，有哪些位置可以插入新的满足条件的项（`id = 10`），考虑到 B+ 树索引的有序性，满足条件的项一定是连续存放的。记录 `[6, c]` 之前，不会插入 `id=10` 的记录；`[6, c]` 与 `[10, b]` 间可以插入 `[10, aa]`；`[10, b]` 与 `[10, d]` 间，可以插入新的 `[10, bb]`，`[10, c]` 等；`[10, d]` 与 `[11, f]` 间可以插入满足条件的 `[10, e]`，`[10, z]` 等；而 `[11, f]` 之后也不会插入满足条件的记录。因此，为了保证 `[6, c]` 与 `[10, b]` 间，`[10, b]` 与 `[10, d]` 间，`[10, d]` 与 `[11, f]` 不会插入新的满足条件的记录，MySQL 选择了用 GAP 锁，将这三个 GAP 给锁起来。↵

Insert 操作，如 `insert [10, aa]`，首先会定位到 `[6, c]` 与 `[10, b]` 间，然后在插入前，会检查这个 GAP 是否已经被锁上，如果被锁上，则 Insert 不能插入记录。因此，通过第一遍的当前读，不仅将满足条件的记录锁上（X 锁），与组合三类似。同时还是增加 3 把 GAP 锁，将可能插入满足条件记录的 3 个 GAP 给锁上，保证后续的 Insert 不能插入新的 `id=10` 的记录，也就杜绝了同一事务的第二次当前读，出现幻象的情况。↵

有心的朋友看到这儿，可能会问：既然防止幻读，需要靠 GAP 锁的保护，为什么组合五、组合六，也是 RR 隔离级别，却不需要加 GAP 锁呢？↵

首先，这是一个好问题。其次，回答这个问题，也很简单。GAP 锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。而组合五，`id` 是主键；组合六，`id` 是 unique 键，都能够保证唯一性。一个等值查询，最多只能返回一条记录，而且新的相同取值的记录，一定不会在新插入进来，因此也就避免了 GAP 锁的使用。其实，针对此问题，还有一个更深入的问题：如果组合五、组合六下，针对 SQL: `select * from t1 where id = 10 for update;` 第一次查询，没有找到满足查询条件的记录，那么 GAP 锁是否还能够省略？此问题留给大家思考。↵

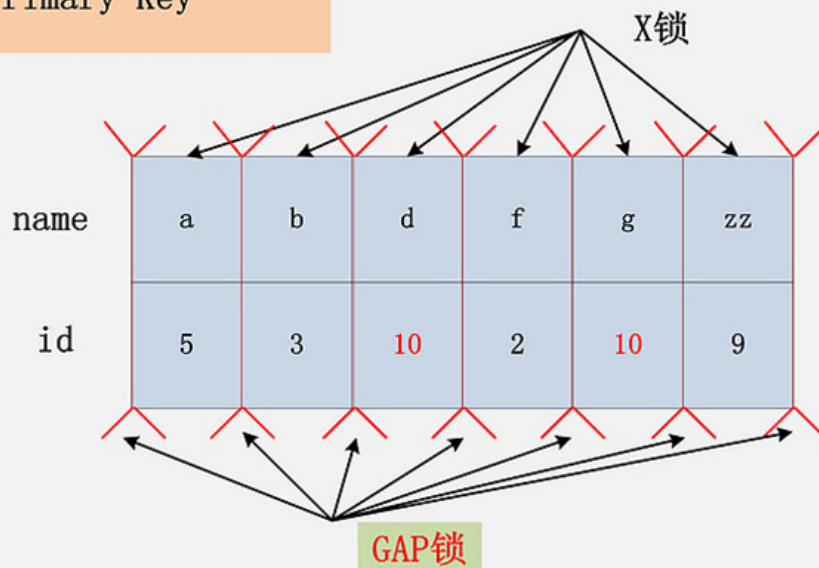
结论：Repeatable Read 隔离级别下，`id` 列上有一个非唯一索引，对应 SQL: `delete from t1 where id = 10;` 首先，通过 `id` 索引定位到第一条满足查询条件的记录，加记录上的 X 锁，加 GAP 上的 GAP 锁，然后加主键聚簇索引上的记录 X 锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录 `[11, f]`，此时，不需要加记录 X 锁，但是仍旧需要加 GAP 锁，最后返回结束。↵

8. 组合八：id 无索引+RR↵

组合八，Repeatable Read 隔离级别下的最后一种情况，`id` 列上没有索引。此时 SQL: `delete from t1 where id = 10;` 没有其他的路径可以选择，只能进行全表扫描。最终的加锁情况，如下图所示：↵

Table: T1(name primary key, id)

Primary Key



如图，这是一个很恐怖的现象。首先，聚簇索引上的所有记录，都被加上了 X 锁。其次，聚簇索引每条记录间的间隙 (GAP)，也同时被加上了 GAP 锁。这个示例表，只有 6 条记录，一共需要 6 个记录锁，7 个 GAP 锁。试想，如果表上有 1000 万条记录呢？

在这种情况下，这个表上，除了不加锁的快照读，其他任何加锁的并发 SQL，均不能执行，不能更新，不能删除，不能插入，全表被锁死。

当然，跟组合四：[\[id 无索引, Read Committed\]](#)类似，这个情况下，MySQL 也做了一些优化，就是所谓的 semi-consistent read。semi-consistent read 开启的情况下，对于不满足查询条件的记录，MySQL 会提前放锁。针对上面的这个用例，就是除了记录 [d, 10], [g, 10] 之外，所有的记录锁都会被释放，同时不加 GAP 锁。semi-consistent read 如何触发：要么是 read committed 隔离级别；要么是 Repeatable Read 隔离级别，同时设置了 [innodb locks unsafe for binlog](#) 参数。更详细的关于 semi-consistent read 的介绍，可参考我之前的一篇博客：[MySQL+InnoDB semi-consistent read 原理及实现分析](#)。

结论：在 Repeatable Read 隔离级别下，如果进行全表扫描的当前读，那么会锁上表中的所有记录，同时会锁上聚簇索引内的所有 GAP，杜绝所有的并发 更新/删除/插入 操作。当然，也可以通过触发 semi-consistent read，来缓解加锁开销与并发影响，但是 semi-consistent read 本身也会带来其他问题，不建议使用。

9. 组合九: Serializable

针对前面提到的简单的 SQL, 最后一个情况: Serializable 隔离级别。对于 SQL2: `delete from t1 where id = 10;` 来说, Serializable 隔离级别与 Repeatable Read 隔离级别完全一致, 因此不做介绍。

Serializable 隔离级别, 影响的是 SQL1: `select * from t1 where id = 10;` 这条 SQL, 在 RC, RR 隔离级别下, 都是快照读, 不加锁。但是在 Serializable 隔离级别, SQL1 会加读锁, 也就是说快照读不复存在, MVCC 并发控制降级为 Lock-Based CC。

结论: 在 MySQL/InnoDB 中, 所谓的读不加锁, 并不适用于所有的情况, 而是隔离级别相关的。Serializable 隔离级别, 读不加锁就不再成立, 所有的读操作, 都是当前读。

10. 一条复杂的 SQL

写到这里, 其实 MySQL 的加锁实现也已经介绍的八八九九。只要将本文上面的分析思路, 大部分的 SQL, 都能分析出其会加哪些锁。而这里, 再来看一个稍微复杂点的 SQL, 用于说明 MySQL 加锁的另外一个逻辑。SQL 用例如下:

Table: t1(id primary key, userid, blogid, pubtime, comment)
Index: idx_t1_pu(pubtime, userid)

idx_t1_pu

pubtime	1	3	5	10	20	100
userid	hdc	yyy	hdc	hdc	bbb	hdc
id	10	4	8	1	100	6

Primary Key

id	1	4	6	8	10	100
userid	hdc	yyy	hdc	hdc	hdc	bbb
blogid	a	b	c	d	e	f
pubtime	10	3	100	5	1	20
comment				good		

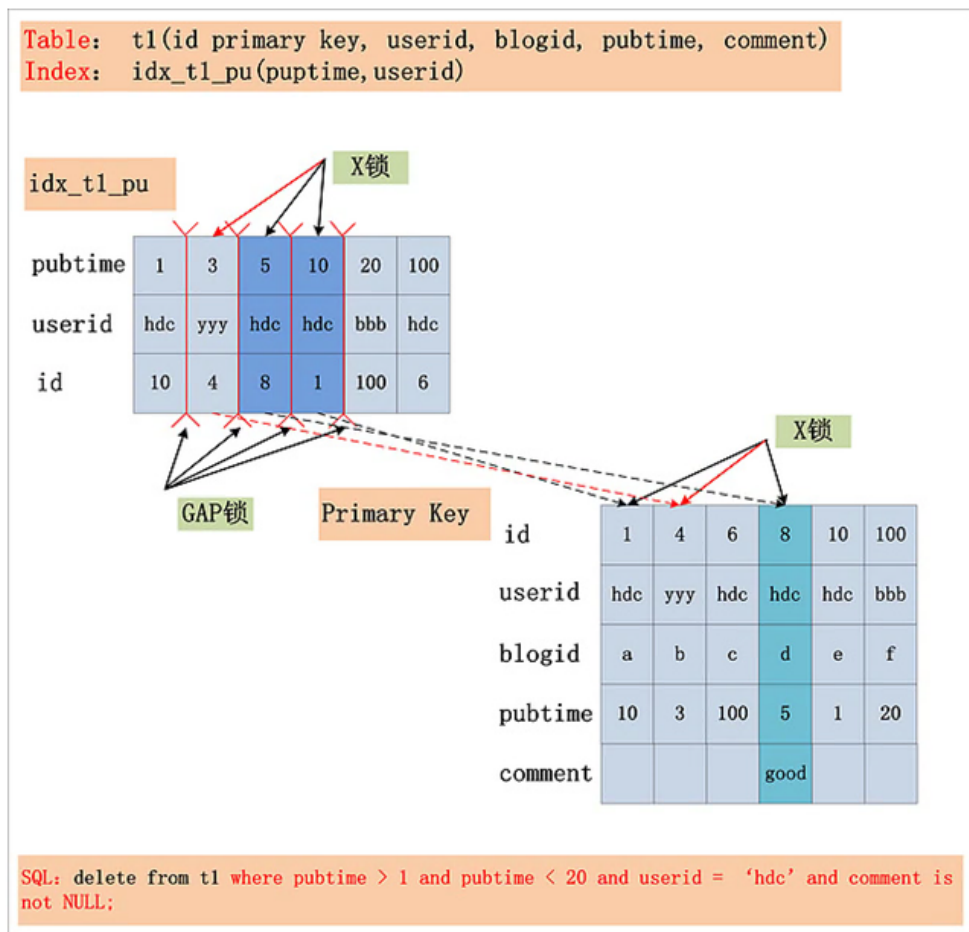
SQL: `delete from t1 where pubtime > 1 and pubtime < 20 and userid = 'hdc' and comment is not NULL;`

如图中的 SQL, 会加什么锁? 假定在 Repeatable Read 隔离级别下 (Read Committed 隔离级别下的加锁情况, 留给读者分析。), 同时, 假设 SQL 走的是 idx_t1_pu 索引。

在详细分析这条 SQL 的加锁情况前，还需要有一个知识储备，那就是一个 SQL 中的 where 条件如何拆分？具体的介绍，建议阅读我之前的一篇文章：[SQL 中的 where 条件，在数据库中提取与应用浅析](#)。在这里，我直接给出分析后的结果：

- **Index key:** `pubtime > 1 and pubtime < 20`。此条件，用于确定 SQL 在 `idx_t1_pu` 索引上的查询范围。
- **Index Filter:** `userid = 'hdc'`。此条件，可以在 `idx_t1_pu` 索引上进行过滤，但不属于 Index Key。
- **Table Filter:** `comment is not NULL`。此条件，在 `idx_t1_pu` 索引上无法过滤，只能在聚簇索引上过滤。

在分析出 SQL where 条件的构成之后，再来看看这条 SQL 的加锁情况（RR 隔离级别），如下图所示：



从图中可以看出，在 Repeatable Read 隔离级别下，由 Index Key 所确定的范围，被加上了 GAP 锁；Index Filter 锁给定的条件 (`userid = 'hdc'`) 何时过滤，视 MySQL 的版本而定，在 MySQL 5.6 版本之前，不支持 [Index Condition Pushdown](#) (ICP)，因此 Index Filter 在 MySQL Server 层过滤，在 5.6 后支持了 Index Condition Pushdown，则在 index 上过滤。若不支持 ICP，不满足 Index Filter 的记录，也需要加上记录 X 锁，若支持 ICP，则不满足 Index Filter 的记录，无需加记录 X 锁（图中，用红色箭头标出的 X 锁，是否要加，视是否支持 ICP 而定）；而 Table Filter 对应的过滤条件，则在聚簇索引中读取后，在 MySQL Server 层面过滤，因此聚簇索引上也需要 X 锁。最后，选出了一条满足条件的记录 [8, hdc, d, 5, good]，但是加锁的数量，要远远大于满足条件的记录数量。

结论：在 Repeatable Read 隔离级别下，针对一个复杂的 SQL，首先需要提取其 where 条件。Index Key 确定的范围，需要加上 GAP 锁；Index Filter 过滤条件，视 MySQL 版本是否支持 ICP，若支持 ICP，则不满足 Index Filter 的记录，不加 X 锁，否则需要 X 锁；Table Filter 过滤条件，无论是否满足，都需要加 X 锁。

11. 死锁原理与分析

本文前面的部分，基本上已经涵盖了 MySQL/InnoDB 所有的加锁规则。深入理解 MySQL 如何加锁，有两个比较重要的作用：

- 可以根据 MySQL 的加锁规则，写出不会发生死锁的 SQL；
- 可以根据 MySQL 的加锁规则，定位出线上产生死锁的原因；

下面，来看看两个死锁的例子（一个是两个 Session 的两条 SQL 产生死锁；另一个是两个 Session 的一条 SQL，产生死锁）：

死锁情况一

Table: t1(id primary key, name)

session 1
begin;
select * from t1 where id = 1 for update;

update t1 set name='qqq' where id = 5;

session 2
begin;
delete from t1 where id = 5;

delete from t1 where id = 1;

死锁发生!!!

id	1	2	3	4	5	6
name	aaa	ccc	aaa	bbb	ccc	zzz

死锁情况二

Table: T2(id primary key, name key, pubtime key, comment)

session 1

update t2 set comment=' abc' where name=' hdc' ;

session 2

select * from t2 where pubtime > 5 for update;

key(name)

name	bbb	hdc	hdc	hdc	hdc	yyy
id	100	1	6	8	10	4

key(pubtime)

pubtime	1	3	5	10	20	100
id	10	4	8	6	100	1

primary key

id	1	4	6	8	10	100
name	hdc	yyy	hdc	hdc	hdc	bbb
pubtime	100	3	10	5	1	20
comment				good		

Deadlock!!

上面的两个死锁用例。第一个非常好理解，也是最常见的死锁，每个事务执行两条 SQL，分别持有了一把锁，然后加另一把锁，产生死锁。

第二个用例，虽然每个 Session 都只有一条语句，仍旧会产生死锁。要分析这个死锁，首先必须用到本文前面提到的 MySQL 加锁的规则。针对 Session 1，从 name 索引出发，读到的 [hdc, 1], [hdc, 6] 均满足条件，不仅会加 name 索引上的记录 X 锁，而且会加聚簇索引上的记录 X 锁，加锁顺序为先 [1, hdc, 100]，后 [6, hdc, 10]。而 Session 2，从 pubtime 索引出发，[10, 6], [100, 1] 均满足过滤条件，同样也会加聚簇索引上的记录 X 锁，加锁顺序为 [6, hdc, 10]，后 [1, hdc, 100]。发现没有，跟 Session 1 的加锁顺序正好相反，如果两个 Session 恰好都持有了第一把锁，请求加第二把锁，死锁就发生了。

结论：死锁的发生与否，并不在于事务中有多少条 SQL 语句，死锁的关键在于：两个（或以上）的 Session **加锁的顺序** 不一致。而使用本文上面提到的，分析 MySQL 每条 SQL 语句的加锁规则，分析出每条语句的加锁顺序，然后检查多个并发 SQL 间是否存在以相反的顺序加锁的情况，就可以分析出各种潜在的死锁情况，也可以分析出线上死锁发生的原因。