数据库锁知识(INNODB)

库锁

库锁主要分为两类：

- FTWRL（Flush tables with read lock），将数据库设置为只读状态，**当客户端异常断开后，该锁自动释放**，官方推荐使用的库锁。
- 设置global全局变量，即`set global readonly=true`，同样是将数据库设置为只读状态，**但无论何时，数据库绝不主动释放锁，即时客户端异常断开连接。**

Flush tables with read lock 具备更安全的异常处理机制，因此建议使用Flush tables with read lock而不是修改全局变量。

表锁

MDL锁

MDL锁，是一种表锁，也称元数据锁(metadata lock)，元数据锁是server层的锁，MYSQL下的所有引擎几乎都提供表级锁定，表级锁定分为**表共享读锁（S锁）与表独占写锁（X锁）。**

在我们执行DDL语句时，诸如创建表、修改表数据等语句，我们都需要申请表锁以防止DDL语句之间出现的并发问题，只有S锁与S锁是共享的，其余锁都是互斥的，这种锁不需

要我们显示的申请，当我们执行DDL语句时会自动申请锁。

当然我们也可以显示的申请表锁：

- `LOCK TABLE table_name READ;` 使用读锁锁表，会阻塞其他事务修改表数据，而对读表共享。
- `LOCK TABLE table_name WRITE;` 使用写锁锁表，会阻塞其他事务读和写。

MDL锁主要**用于解决多条DDL语句之间的并发安全问题**，但除了DDL与DDL之间的问题，DDL与DML语句之间也会出现并发问题，因此INNODB下，还会存在一种隐式的上锁方式。

意向锁

事实上，为解决DDL和DML之间的冲突问题，在INNODB下，数据库还会为每一条DML语句隐式地加上表级锁，这并不需要我们显示的指定。

来看看数据库为什么这么做，我们假设事务A执行一条DDL语句`ALTER TABLE test DROP COLUMN id;`，而事务B正在执行两条DML语句`SELECT * FROM`，如果对数据库比较了解，你应该很快的就会发现其中存在一些并发安全问题：

事务A	事务B
`BEGIN`	`BEGIN`
`SELECT * FROM test;`	
...	`ALTER TABLE test DROP COLUMN id;`
`SELECT * FROM test;`	
`COMMIT`	`COMMIT`

这就产生了冲突现象，事务A执行的两条查询语句不一致，违背了事务的一致性，而为了解决这个问题，MYSQL引入了意向锁，**官方将这种锁分为意向共享锁(IS锁)和意向排他锁(IX锁)，意向锁是由DML操作产生的，请注意区分与上文所说的S锁与X锁，意向共享锁表明当前表上存在某行记录持有行共享锁(区别表S锁)，意向排他锁表明当前表上存在某行记录持有行排他锁(区别表X锁)。**

每执行一条DML语句都要申请意向锁，意向锁的类型是由行锁的类型决定的，例如SELECT语句会申请行共享锁，同时也会申请意向共享锁，UPDATE会申请意向排他锁，**同一个表内的DML语句不会冲突，这意味着意向锁都是兼容的，要注意意向锁是由DML语句产生的，而DDL语句不会申请意向锁。**

如上文说所，DDL语句申请的只是普通的X锁或S锁，但它必须根据规则要等待IX锁或IS锁释放。

表锁兼容性规则如下图所示：

右侧是已加的锁 (+ 代表兼容， - 代表不兼容)	IS	IX	S	X
IS	+	+	+	-
IX	+	+	-	-
S	+	-	+	-
X	-	-	-	-

现在可以正常执行了，事务B必须要等到事务A提交后才可执行：

事务A	事务B
`BEGIN`	`BEGIN`
`SELECT * FROM test;`(申请IX锁)	
…	`ALTER TABLE test DROP COLUMN id;`(申请X锁，需要等待IX释放)
`SELECT * FROM test;(重入)`	
`COMMIT`(释放IX锁)	`COMMIT`(释放X锁)

这种隐式的上锁是在MYSQL5.5之后才引入的，**在之前的版本中，执行DML操作并不会对表上锁，因此执行DDL操作不仅需要申请X锁，还需要遍历表中的每一行记录，判定是否存在行锁，如果的确存在，则放弃操作。因此在以前的版本中，是不支持在线DDL的，要想执行DDL操作就必须停止关于该表的一切活动，除此之外，执行DDL操作需要遍历所有行，这也是非常低效的。**

有了这种隐式MDL锁之后，解决了DML与DDL操作之间的冲突，在线DDL变得可能，同时无须遍历所有行，只需要申请表锁即可。

所以说，**这种隐式的表锁，解决了DML与DDL操作之间的冲突，使得数据库可以支持在线DDL，同时增加了执行DDL的效率。**

注意一个包含关系，IX、IS、X、S锁均属于MDL锁。

在线DDL的效率问题

尽管在MYSQL5.5后提出隐式MDL锁后，在线DDL操作变得可能，但我们不得不来思考它的效率问题，考虑下图：

session 1	session 2	session 3
begin		
select * from user		
	alter table user add age int (block)	
		select * from user (block)

有三个事务，第一个事务执行DQL语句同时申请IS锁；第二个事务执行DDL语句同时申请X锁，X锁是排他的，因此必须等待事务一IS锁的释放，事务二被堵塞；事务三同样执行DQL语句，**但由于写锁的优先级高于读锁，事务三不得不排在事务二的后面**，事务三被堵塞(不只是数据库中，绝大多数场景下都是写锁优先)；如果后面有N个DQL语句，那么这N个语句都会被堵塞，而如果没有事务二，**由于读是共享的，所有事务都不会堵塞**，在线DDL使得整体效率变得异常低下。

这种现象产生的原因主要是使用了隐式MDL锁和写锁优先原则，因此我们很难根治这种现象，只能去缓解，MYSQL5.6版本后提出锁升降级机制。

锁升降级机制

在上述示例中，事务二以后的事务都必须等待事务二执行完毕，而事务二是一种DDL操作，DDL操作涉及到文件读写，会写REDO LOG并发起磁盘IO，这是非常缓慢的，既然无法改变写者优先的原则，MYSQL试图加快DDL操作的执行以减少后续事务的等待，但DDL操作本身已经很难再做改变了，MYSQL想到了一种曲线救国的方式——让它暂时放弃对写锁持有！

具体的流程为：

1. 事务开始，申请表级写锁；
2. 降级为表级读锁，使得后续DQL操作不被堵塞(DML仍被堵塞);
3. 执行具体更改。

4. 升级为写锁；
5. 事务提交，释放锁；

当DDL事务由写锁降级时，后续的DQL操作得以运行，提高了效率。

要理解这一升一降，当事务开始时，DDL操作由写锁降级为读锁时，由于读锁与写锁排斥，可以保证DDL更改表数据时不会有任何其他写表操作，避免了并发问题；当事务提交时，读锁升级为写锁，又可以保证同一时刻没有其他读表操作，即避免了读写不一致问题。

但**假如有过多的读者，使得该锁无法从读锁升级为写锁，就可能存在饿死该DDL操作的问题**，这是为了提高性能而带来的弊端。

行锁

行锁是对针对某一行记录上锁，是更细粒度的一种锁，在MYSQL中，只有INNODB执行行锁，而其他的引擎不支持行锁。

INNODB下实现了两种标准的行级锁(区别表锁中的S锁与X锁，这里的锁是行锁！)：

- 共享锁(S锁)，允许事务读一行数据。
- 排他锁(X锁)，允许事务修改或删除一行数据。

行锁在INNODB下是基于索引实现的，当索引未命中时，任何操作都将全表匹配查询，行锁会退化为表锁，数据库会先锁表，再执行全表检索。

因此要注意所有的行锁都是在索引上的。

四种隔离级别

1. 读未提交(Read uncommitted)。即事务可以读取其他事务还未提交的数据，事务完全是透明的，这种级别下连脏读都无法避免。
2. 读已提交(Read committed)。这种隔离级别下，事务可以通过MVVC读取数据而无须等待X锁的释放，但事务总是读取最新版本的记录，例如事务A正在修改某行数据，事务B读取两次，第一次读取发现A正在修改，数据被上锁，因此读取上一个版本的数据，此时A事务修改完毕并提交，事务B开始第二次读取，它总是会尝试读取最新版本的数据，于是事务B第二次读取了事务A修改后的数据，事务B两次读取不一致，**发生了不可重复读问题。**
3. 可重复读(Repeatable read)。INNDOB下默认的级别，与读已提交类似，唯一的区别是这种隔离级别下，在一个事务内，总是会读取一致的记录版本，一开始读什么，后面就读什么，不会发生不可重复读问题。起初存在幻读问题，后来引入Next Key Lock解决了这个问题。
4. 可串行化(Serializable)。禁止MVVC功能，不会出现问题，但效率低。

隔离级别	脏读	不可重复读	幻读（Phantom Read）
读未提交（Read uncommitted）	可能	可能	可能
读已提交（Read committed）	不可能	可能	可能(引入临建锁解决)
可重复读（Repeatable read）	不可能	不可能	不可能
可串行化（Serializable）	不可能	不可能	不可能

行锁的分类

INNODB下有三种行锁，根据不同的条件使用不同的锁。

为了统一，我们执行如下SQL语句，建立test表：

```
drop table if exists test;
create table test(
        a int,
    b int,
    primary key(a),
    key(b)
)engine=InnoDB charset=utf8;

insert into test select 1, 1;
insert into test select 3, 1;
insert into test select 5, 3;
insert into test select 7, 6;
insert into test select 10, 8;
```

a(主索引)	b(普通索引)
1	1
3	1
5	3
7	6
10	8

行记录锁（Record Locks）

记录锁锁住唯一索引上的行记录，请注意这里是锁住记录而不是锁住索引，这意味着你无法绕开索引去访问记录。

行记录锁何时生效？**仅当查询列是唯一索引等值查询时(等值查询就是where xxx = xxx)，next key lock会降级为行记录锁。**

为什么查询列是唯一索引等值查询时，可以使用行记录锁呢？其实很简单，由于**唯一索引列仅对应唯一的行记录**，当我们执行等值查询时，已经确保了我们只会访问这一条行记录，因此对该记录上锁，使得其他操作无法无法影响该记录，并且插入新记录的操作会由于主键冲突而被拒绝，幻读问题也不会产生，并发安全得以保证。

要注意MYSQL默认条件下是使用next key lock的，而仅仅在条件满足时降级为行记录锁。而使用Record Lock的冲要条件是查询的记录是唯一的。

其他条件下难道不可以使用行记录锁吗？答案是不可以！其他任意条件，我们都无法满足行记录锁的重要条件。

如果不是等值查询，那么必然会出现多个结果，在RR级别下，我们来看一个范围查询，考虑事务A与事务B：

事务A	事务B
BEGIN;	BEGIN;
SELECT * FROM test WHERE a >= 1 FOR UPDATE;(上X锁，禁止读取快照)	…
…	INSERT INTO test SELECT 101, 5;
SELECT * FROM test WHERE a >= 1 FOR UPDATE;	…
COMMIT;	COMMIT;

试想，如果仅锁住一条记录，事务A前后两次将读出不同的结果，第二次读取将多一条记录，即幻读现象！因此，**我们必须要锁住一个范围。**

再考虑非唯一索引下的等值查询，想想为什么这种情况下不能加行记录锁。

其实也很简单，回到我们之前说的，**行记录锁锁的是一条记录而不是索引值**，例如语句`SELECT * FROM test WHERE b = 1 FOR UPDATE;`，该语句对应的是两条记录，行记录锁只能锁住现有记录，但是**不能阻止增加其他记录，产生幻读**！这是冲突的，因此无法使用行记录锁。

所以我们必须要有个范围锁，这就是间隙锁！

> 在源码实现中，行记录锁是通过 哈希+位图 实现的，首先通过 表空间+数据所在页号 定位到哈希表，通过记录的 heap_no 字段查询哈希表中的 bitmap，若 1 则表

> 示锁住。

间隙锁(Gap Lock)

间隙锁锁住的是一个范围，但不会锁住记录本身，**即锁条件值而非锁记录数据**，这是与行记录锁相反的。在非唯一索引的等值查询下，**间隙锁锁住的是前后两端的间隙**；而在范围查询下，间隙锁将锁住一个范围。

例如执行语句 `BEGIN; SELECT * FROM test WHERE b = 3 FOR UPDATE;` 将会导致 b = 3 **前后间隙**被锁住，如图：

a(主索引)	b(普通索引)
1	1
3	1
5	3
7	6
10	8

此时 (3, 1) ~ (5, 3) 和 (5, 3) ~ (7, 6) 之间的间隙被锁住，如果我们在上诉事务未提交下执行如下语句：

```
INSERT INTO test(a, b) SELECT 4, 0;
INSERT INTO test(a, b) SELECT 100, 1;
INSERT INTO test(a, b) SELECT 101, 2;
INSERT INTO test(a, b) SELECT 102, 3;
INSERT INTO test(a, b) SELECT 103, 4;
INSERT INTO test(a, b) SELECT 104, 6;
```

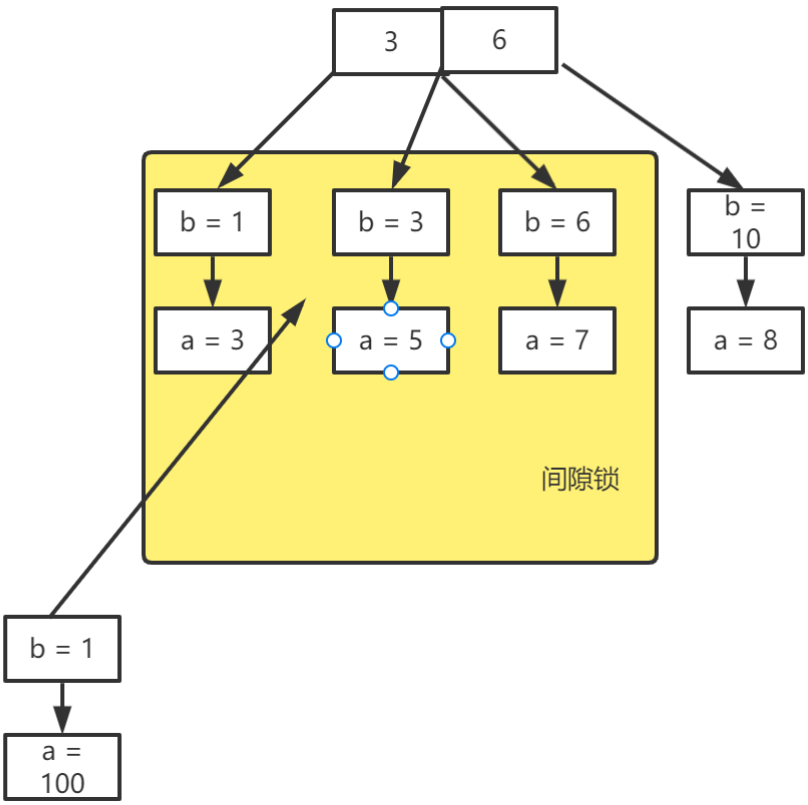
会发现只有第一条和最后一条执行成功，其他语句都被堵塞，这意味着 b = [1, 6) 区间的数据都被堵塞，上锁的规则我们在 Next Key Lock 中讲述，这个例子可以看出间隙锁仅仅只锁了 b，前后位置也是基于 b 的排序顺序来的，而没有锁 a，因为 a = 4 的记录被执行。

结合 MySql 存储结构可知，Gap Lock 应当是在非聚集索引上上锁，形象的来说就是锁住 B+ 树上两片叶子之间的距离，任何妄想破坏二者之间距离的行为都将被堵塞。

而对于范围查询，例如：`SELECT * FROM test WHERE b > 1 FOR UPDATE` 这条语句只会锁住 b = (1, 正无穷) 处的范围，并不会锁住 b = 1 或 前一条数据的间隙，这个应该比较好理解。

现在来看一个严峻的问题，如果大家了解的话，应该都知道间隙锁的范围是左闭右开的，但在我们的实验中，我们发现锁的范围是 b = [1, 6)，刚好和左开右闭相反？难道官网说错了？

其实并没有，6 作为开区间这个是由于一个优化，我们马上就会说到；而 1 是闭区间这个说法其实有问题的，如下图：



上图是关于索引 b 的 B+ 树， 仔细观察，插入的 (100, 1) 其实是在 (3, 1) ~ (5, 3) 之间的，由此可见 **间隙锁锁的不是索引值，而是索引所在的位置，这点非常重要。**

例如，插入一条 (a = -2, b = 1) 的数据将会被成功插入，因为 -a < 3，这条数据将会在 (a = 3, b = 1) 的左边，由此也可以证明上诉结论。

其实我们可以发现间隙锁的实现应该是通过比较行记录实现的，例如上述锁住的区间是 ({1, 3}, {6, 7}]，刚好是一个左开右闭的区间，那这样大于 {1, 3} 和 小于等于 {6, 7} 的记录将被锁住，在这个例子中，{1, 100} 显然大于 {1, 3}，因此被锁，而 {1, -2} 小于 {1, 3}，所以可以正常插入。

临键锁(Next Key Lock)

临键锁与间隙锁仅在RR(可重复读)隔离级别下生效，目的是为了解决幻读问题，而RC级别下连不可重复读都无法解决，更别说幻读问题了。

临建锁是行记录锁和间隙锁的结合，间隙锁锁的是一个范围，而临建锁会对这个范围内存在的记录值加行记录锁。

临建锁的规则：

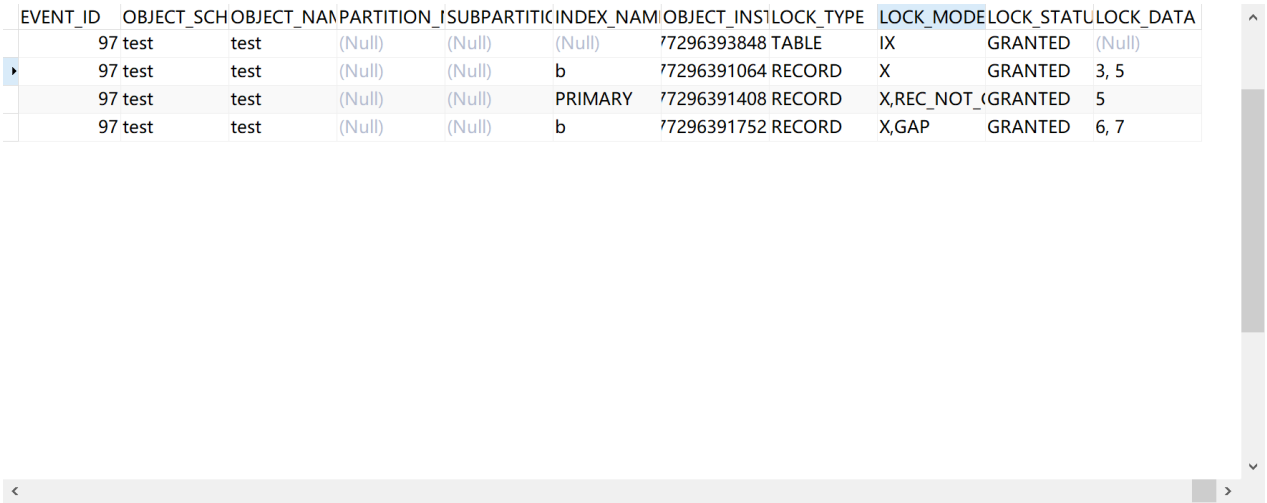
- 先按照间隙锁的规则进行上锁，对查询前后的区间上左开右闭的范围锁。
- 如果闭区间不在查询范围内，那么闭区间退化成开区间，优化性能。

- 扫描这个范围，对范围内存在的记录上行记录锁。

根据第二点规则，可以解释上文为什么 b = 6 为什么会成开区间，这是因为我们的查询是 b = 3，6 不在查询范围内，因此退化。

而在整个范围内，只有 (a = 5, b = 3) 这条记录在范围内，其他都不在范围内，因此 (a = 5, b = 3) 这条记录会被加上行记录锁。

执行`use performance_schema; SELECT * FROM data_locks;`查看锁信息：



EVENT_ID	OBJECT_SCH	OBJECT_NAM	PARTITION_I	SUBPARTITIC	INDEX_NAM	OBJECT_INST	LOCK_TYPE	LOCK_MODE	LOCK_STATU	LOCK_DATA
97	test	test	(Null)	(Null)	(Null)	7296393848	TABLE	IX	GRANTED	(Null)
97	test	test	(Null)	(Null)	b	7296391064	RECORD	X	GRANTED	3, 5
97	test	test	(Null)	(Null)	PRIMARY	7296391408	RECORD	X,REC_NOT_(	GRANTED	5
97	test	test	(Null)	(Null)	b	7296391752	RECORD	X,GAP	GRANTED	6, 7

发现 a = 5 确实被加上了行记录锁。

那么为什么要加这个行记录锁呢？由于间隙锁锁住的仅仅只是内存位置之间的间隙，虽然不能插入删除，但是是可以对已存在的叶子进行修改的，这并不会改变叶子之间的距离（间隙），所以这需要额外上锁来组织这种行为，**以防止不可重复读问题。**

也许你会想，MVCC 版本快照不是解决了不可重复度问题吗？对，这没错，在 RR 级别下，的确是 MVCC 版本快照防止不可重复读问题，但是别忘了，我们的例子中 SELECT 语句可是加了 FOR UPDATE 的，这意味着我们希望它们锁上而不是通过读取版本快照！

AUTO-INC Locking

自增长锁，在InnoDB引擎中，每个表都会维护一个表级别的自增长计数器，当对表进行插入的时候，会通过以下的命令来获取当前的自增长的值。

```
SELECT MAX(auto_inc_col)  FROM user FOR UPDATE;
```

插入操作会在这个基础上加1得到即将要插入的自增长id，然后在一个事务内设置id。

为了提高插入性能，自增长的锁不会等到事务提交之后才释放，而是在**相关插入sql语句完成后立刻就释放**，这也导致了一些事务回滚之后，**id不连续**。

由于自增长会申请写锁，尽管不用等到事务结束，但仍然降低了数据库的性能，5.1.2版本后InnoDB支持互斥量的方式来实现自增长，通过互斥量可以对内存中的计数器进行累

加操作，比AUTO-INC Locking(表锁)要快些。

全文完。