MySQL 事务与锁机制详解——青山

1 什么是数据库的事务？

1.1 事务的典型场景

比如下单，会操作订单表，资金表，物流表等等，这个时候我们需要让这些操作都在一个事务里面完成。在金融的系统里面事务配置是很常见的，比如行内转账的这种操作，如果我们把它简单地理解为一个账户的余额增加，另一个账户的余额减少的情况（当然实际上要比这复杂），那么这两个动作一定是同时成功或者同时失败的，否则就会造成银行的会计科目不平衡。

1.2 事务的定义

维基百科的定义：事务是数据库管理系统（DBMS）执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。

1.3 哪些存储引擎支持事务

InnoDB 支持事务。

1.4 事务的四大特性

第一个，原子性，Atomicity，也就是我们刚才说的不可再分，也就意味着我们对数据库的一系列的操作，要么都是成功，要么都是失败，不可能出现部分成功或者部分失败的情况。

全部成功比较简单，问题是如果前面一个操作已经成功了，后面的操作失败了，怎么让它全部失败呢？这个时候我们必须要回滚。

原子性,在 InnoDB 里面是通过 undo log 来实现的,它记录了数据修改之前的值(逻辑日志），一旦发生异常，就可以用 undo log 来实现回滚操作。

第二个，一致性，Consistency，指的是数据库的完整性约束没有被破坏，事务执行的前后都是合法的数据状态。比如主键必须是唯一的，字段长度符合要求。

除了数据库自身的完整性约束，还有一个是用户自定义的完整性。

第三个，隔离性，Isolation，我们有了事务的定义以后，在数据库里面会有很多的事务同时去操作我们的同一张表或者同一行数据，必然会产生一些并发或者干扰的操作，那么我们对隔离性的定义，就是这些很多个的事务，对表或者行的并发操作，应该是透明的，互相不干扰的。通过这种方式，我们最终也是保证业务数据的一致性。

最后一个叫做持久性，durability。我们对数据库的任意的操作，增删改，只要事务提交成功，那么结果就是永久性的，不可能因为我们系统宕机或者重启了数据库的服务器，它又恢复到原来的状态了。这个就是事务的持久性。

持久性是通过 redo log 来实现的，我们操作数据的时候，会先写到内存的 buffer pool 里面,同时记录 redo log,如果在刷盘之前出现异常,在重启后就可以读取 redo log 的内容，写入到磁盘，保证数据的持久性。

实际上还有一个双写缓冲的机制。

因为存储引擎的页和操作系统的页大小不一致。一个存储引擎 page 的数据要写 4 次，如果中间发生异常，或造成页数据的不可用。所以，必须把页的数据备份起来，这个就是双写缓冲（double write buffer）。

原子性，隔离性，持久性，最后都是为了实现一致性。

1.5 数据库什么时候会出现事务

这条更新语句，有事务吗?

```
update student set sname = '猫老公 111' where id=1;
```

它自动开启了一个事务，并且提交了。

这个是开启事务的第一种方式，自动开启和自动提交。

InnoDB 里面有一个 autocommit 的参数（分成两个级别， session 级别和 global 级别）。

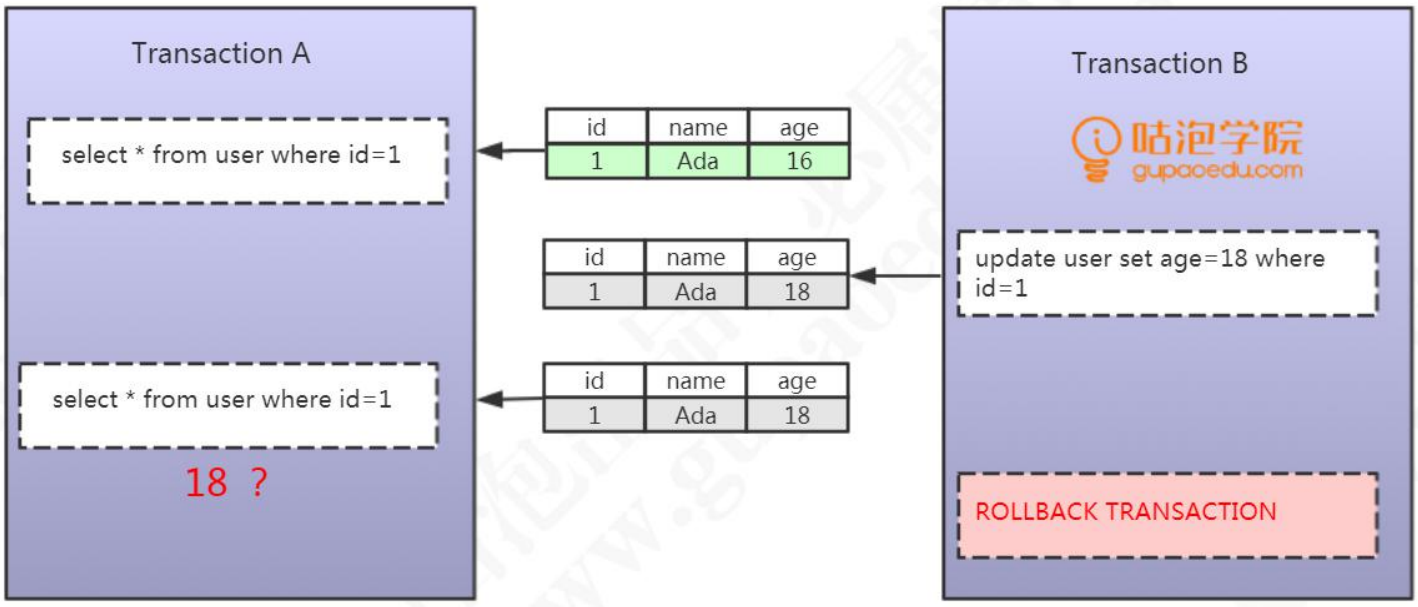
```
show  variables like 'autocommit';
```

它的默认值是 ON。autocommit 这个参数是什么意思呢？是否自动提交。如果它的值是 true/on 的话，我们在操作数据的时候，会自动开启一个事务，和自动提交事务。

手动开启事务也有几种方式，一种是用 begin；一种是用 start transaction。

那么怎么结束一个事务呢？我们结束也有两种方式，第一种就是提交一个事务，commit；还有一种就是 rollback，回滚的时候，事务也会结束。

1.6 事务并发会带来什么问题？

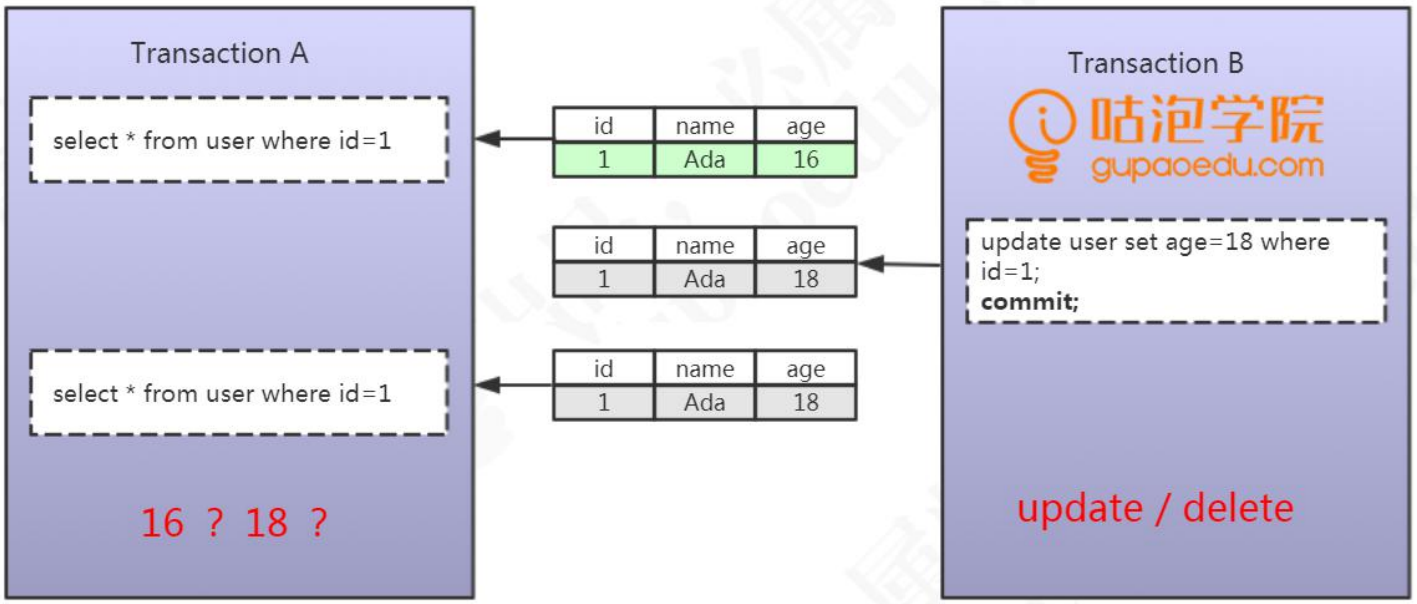


在第一个事务里面，它首先通过一个 where id=1 的条件查询一条数据，返回

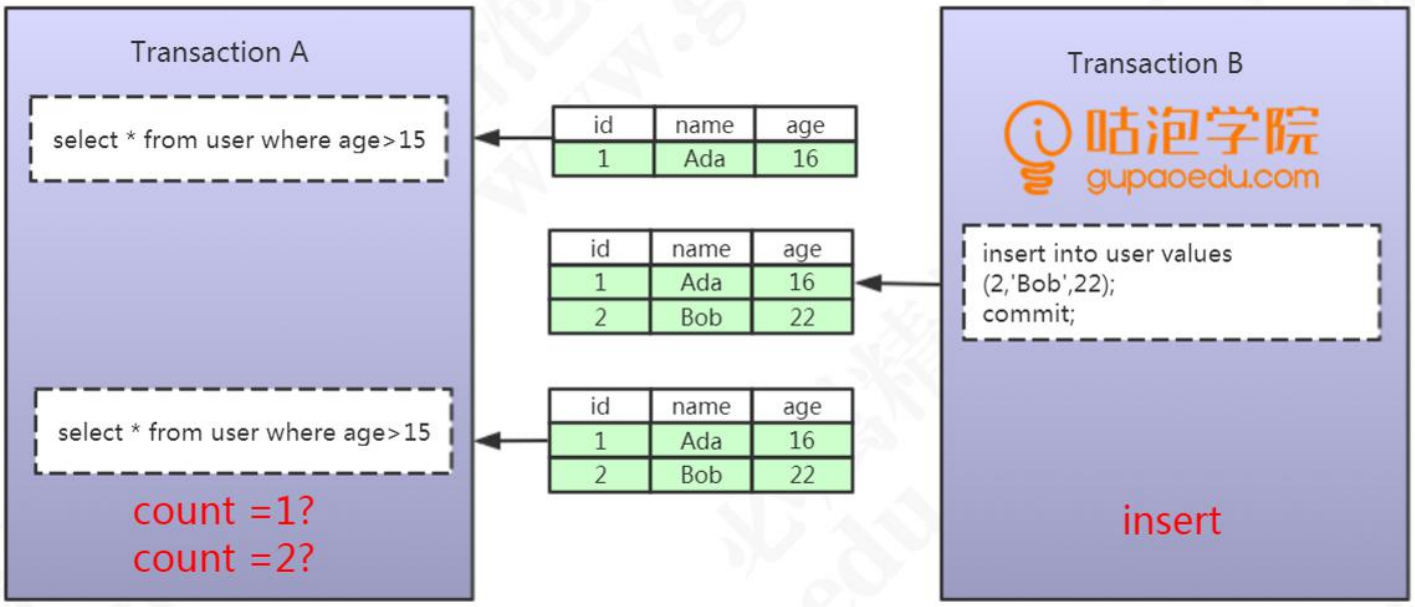
name=Ada，age=16 的这条数据。

第二个事务，它同样地是去操作 id=1 的这行数据，它通过一个 update 的语句，把这行 id=1 的数据的 age 改成了 18，没有提交。

这个时候，在第一个事务里面，它再次去执行相同的查询操作，发现数据发生了变化，获取到的数据 age 变成了 18。这种在一个事务里面，由于其他的时候修改了数据并且没有提交，而导致了前后两次读取数据不一致的情况，叫做脏读。



同样是两个事务，第一个事务通过 id=1 查询到了一条数据。然后在第二个事务里面执行了一个 update 操作，通过一个 commit 提交了修改。然后第一个事务读取到了其他事务已提交的数据导致前后两次读取数据不一致的情况。那么这种事务并发带来的问题，叫做不可重复读。



在第一个事务里面我们执行了一个范围查询，这个时候满足条件的数据只有一条。在第二个事务里面，它插入了一行数据，并且提交了。重点：插入了一行数据。在第一个事务里面再去查询的时候，它发现多了一行数据。

一个事务前后两次读取数据数据不一致，是由于其他事务插入数据造成的，这种情况我们把它叫做幻读。

1.7 SQL92 标准

http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

第一个隔离级别叫做：Read Uncommitted（未提交读），一个事务可以读取到其他事务未提交的数据，会出现脏读，所以叫做 RU，它没有解决任何的问题。

第二个隔离级别叫做：Read Committed（已提交读），也就是一个事务只能读取到其他事务已提交的数据，不能读取到其他事务未提交的数据，它解决了脏读的问题，但是会出现不可重复读的问题。

第三个隔离级别叫做：Repeatable Read (可重复读)，它解决了不可重复读的问题，也就是在同一个事务里面多次读取同样的数据结果是一样的，但是在这个级别下，没有

定义解决幻读的问题。

最后一个就是：Serializable（串行化），在这个隔离级别里面，所有的事务都是串行执行的，也就是对数据的操作需要排队，已经不存在事务的并发操作了，所以它解决了所有的问题。

1.8 MySQL InnoDB 对隔离级别的支持

在 MySQL InnoDB 里面，不需要使用串行化的隔离级别去解决所有问题。那我们来看一下 MySQL InnoDB 里面对数据库事务隔离级别的支持程度是什么样的。

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对 InnoDB 不可能
串行化（Serializable）	不可能	不可能	不可能

1.9 两大实现方案

1.9.1 LBCC

第一种，读取数据的时候，锁定我要操作的数据，不允许其他的事务修改就行了。这种方案我们叫做基于锁的并发控制 Lock Based Concurrency Control（LBCC）。

1.9.2 MVCC

https://dev.mysql.com/doc/refman/5.7/en/innodb-multi-versioning.html

在修改数据的时候给它建立一个备份或者叫快照，后面再来读取这个快照就行了。这种方案我们叫做多版本的并发控制 Multi Version Concurrency Control（MVCC）。

MVCC 的核心思想是： 我可以查到在我这个事务开始之前已经存在的已提交的数

据，即使它在后面被修改或者删除了。在我这个事务之后新增的数据，我是查不到的。

InnoDB 为每行记录都实现了两个隐藏字段（还加上一个 ROWID）：

DB_TRX_ID，6 字节：插入或更新行的最后一个事务的事务 ID，事务编号是自动递增的（我们把它理解为创建版本号，在数据新增或者修改为新数据的时候，记录当前事务 ID）。

DB_ROLL_PTR，7 字节：回滚指针（我们把它理解为删除版本号，数据被删除或记录为旧数据的时候，记录当前事务 ID）。

我们把这两个事务 ID 理解为版本号。

下面我们用一种简化的模型来理解一下 MVCC。

https://processon.com/view/5d29999ee4b07917e2e09294

第一个事务，初始化数据（检查初始数据）

Transaction 1
begin; insert into mvcctest values(NULL,'qingshan') ; insert into mvcctest values(NULL,'jack') ; commit;

此时的数据，创建版本是当前事务 ID，删除版本为空：

id	name	创建版本	删除版本
1	qingshan	1	undefined
2	jack	1	undefined

第二个事务，执行第 1 次查询，读取到两条原始数据，这个时候事务 ID 是 2：

Transaction 2
begin; select * from mvcctest ; -- (1) 第一次查询

第三个事务，插入数据：

Transaction 3
begin; insert into mvcctest values(NULL,'tom') ; commit;

此时的数据，多了一条 tom，它的创建版本号是当前事务编号，3：

id	name	创建版本	删除版本
1	qingshan	1	undefined
2	jack	1	undefined
3	tom	3	undefined

第二个事务，执行第 2 次查询：

Transaction 2
select * from mvcctest ; (2) 第二次查询

MVCC 的查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

也就是不能查到在我的事务开始之后插入的数据，tom 的创建 ID 大于 2，所以还是只能查到两条数据。

第四个事务，删除数据，删除了 id=2 jack 这条记录：

Transaction 4
begin; delete from mvcctest where id=2; commit;

此时的数据，jack 的删除版本被记录为当前事务 ID，4，其他数据不变：

id	name	创建版本	删除版本
1	qingshan	1	undefined
2	jack	1	4
3	tom	3	undefined

在第二个事务中，执行第 3 次查询：

Transaction 2
select * from mvcctest ; (3) 第三次查询

查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

也就是，在我事务开始之后删除的数据，所以 jack 依然可以查出来。所以还是这两条数据。

第五个事务，执行更新操作，这个事务事务 ID 是 5：

Transaction 4
begin; update mvcctest set name ='盆鱼宴' where id=1; commit;

此时的数据，更新数据的时候，旧数据的删除版本被记录为当前事务 ID 5（undo），产生了一条新数据，创建 ID 为当前事务 ID 5：

id	name	创建版本	删除版本
1	qingshan	1	5
2	jack	1	4
3	tom	3	undefined
1	盆鱼宴	5	undefined

第二个事务，执行第 4 次查询：

Transaction 2
select * from mvcctest ; (4) 第四次查询

查找规则：只能查找创建时间小于等于当前事务 ID 的数据，和删除时间大于当前事务 ID 的行（或未删除）。

因为更新后的数据 penyuyan 创建版本大于 2，代表是在事务之后增加的，查不出来。

而旧数据 qingshan 的删除版本大于 2，代表是在事务之后删除的，可以查出来。

通过以上演示我们能看到，通过版本号的控制，无论其他事务是插入、修改、删除，第一个事务查询到的数据都没有变化。

2 MySQL InnoDB 锁的基本类型

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html

2.1 共享锁

Shared Locks （共享锁），我们获取了一行数据的读锁以后，可以用来读取数据，所以它也叫做读锁。用 select …… lock in share mode; 的方式手工加上一把读锁。

释放锁有两种方式，只要事务结束，锁就会自动事务，包括提交事务和结束事务。

Transaction 1	Transaction 2
begin;	
SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE;	
	begin;
	SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE; // OK

2.2 排它锁

Exclusive Locks（排它锁），它是用来操作数据的，所以又叫做写锁。只要一个事务获取了一行数据的排它锁，其他的事务就不能再获取这一行数据的共享锁和排它锁。

增删改，都会默认加上一个排它锁。

手工加锁，用 FOR UPDATE 给一行数据加上一个排它锁，这个无论是在我们的代码里面还是操作数据的工具里面，都比较常用。

释放锁的方式跟前面是一样的。

排他锁的验证：

Transaction 1	Transaction 2
begin;	

UPDATE student SET sname = '猫老公 555' WHERE id=1;	
	begin;

	SELECT * FROM student WHERE id=1 LOCK IN SHARE MODE; // BLOCKED
	SELECT * FROM student where id=1 FOR UPDATE; // BLOCKED
	DELETE FROM student where id=1 ; // BLOCKED

2.3 意向锁

当我们给一行数据加上共享锁之前，数据库会自动在这张表上面加一个意向共享锁。

当我们给一行数据加上排他锁之前，数据库会自动在这张表上面加一个意向排他锁。

反过来说：

如果一张表上面至少有一个意向共享锁，说明有其他的事务给其中的某些数据行加上了共享锁。

如果一张表上面至少有一个意向排他锁，说明有其他的事务给其中的某些数据行加上了排他锁。

```
select * from t2  where id =4 for update;
```

```
TABLE LOCK table `gupao`.`t2` trx id 24467 lock mode IX
RECORD LOCKS space id 64 page no 3 n bits 72 index PRIMARY of table `gupao`.`t2` trx id 24467 lock_mode X locks rec but not gap
```

第一个，我们有了表级别的锁，在 InnoDB 里面就可以支持更多粒度的锁。

第二个作用，当我们准备给一张表加上表锁的时候，必须先要去判断有没其他的事务锁定了其中了某些行。如果有的话，肯定不能加上表锁。那么这个时候我们就要去扫描整张表才能确定能不能成功加上一个表锁，如果数据量特别大，比如有上千万的数据的时候，加表锁的效率是不是很低?

但是我们引入了意向锁之后就不一样了。我只要判断这张表上面有没有意向锁，如果有，就直接返回失败。如果没有，就可以加锁成功。所以 InnoDB 里面的表锁，我们可以把它理解成一个标志。就像火车上厕所有没有人使用的灯，是用来提高加锁的效率

的。

Transaction 1	Transaction 2
begin;	
SELECT * FROM student where id=1 FOR UPDATE;	
	BEGIN;

	LOCK TABLES student WRITE; // BLOCKED
	UNLOCK TABLES; // 释放表锁的方式

3 行锁的原理

3.1 没有索引的表（假设锁住记录）

首先我们有三张表，一张没有索引的 t1，一张有主键索引的 t2，一张有唯一索引的 t3。

我们先假设 InnoDB 的锁锁住了是一行数据或者一条记录。

Transaction 1	Transaction 2
begin;	
SELECT * FROM t1 WHERE id =1 FOR UPDATE;	
	select * from t1 where id=3 for update; //blocked
	INSERT INTO `t1` (`id`, `name`) VALUES (5, '5'); //blocked

现在我们在两个会话里面手工开启两个事务。

在第一个事务里面，我们通过 where id =1 锁住第一行数据。

在第二个事务里面，我们尝试给 id=3 的这一行数据加锁，这个加锁的操作被阻塞了。

我们再来操作一条不存在的数据，插入 id=5。它也被阻塞了。

实际上这里整张表都被锁住了。所以，我们的第一个猜想被推翻了，InnoDB 的锁锁住的应该不是 Record。

3.2 有主键索引的表

Transaction 1	Transaction 2
begin;	
select * from t2 where id=1 for update;	
	select * from t2 where id=1 for update; // blocked
	select * from t2 where id=4 for update; // OK

第一种情况，使用相同的 id 值去加锁，冲突；使用不同的 id 加锁，可以加锁成功。那么，既然不是锁定一行数据，有没有可能是锁住了 id 的这个字段呢?

3.3 唯一索引（假设锁住字段）

Transaction 1	Transaction 2
begin;	
select * from t3 where name= '4' for update;	
	select * from t3 where name = '4' for update; // blocked
	select * from t3 where id = 4 for update; // blocked

在第一个事务里面，我们通过 name 字段去锁定值是 4 的这行数据。

在第二个事务里面，尝试获取一样的排它锁，肯定是失败的，这个不用怀疑。

在这里我们怀疑 InnoDB 锁住的是字段，所以这次我换一个字段，用 id=4 去给这行数据加锁，又被阻塞了，说明锁住的是字段的这个推测也是错的，否则就不会出现第一个事务锁住了 name，第二个字段锁住 id 失败的情况。

既然锁住的不是 record，也不是 column，InnoDB 里面锁住的到底是什么呢？其实答案就是索引。InnoDB 的行锁，就是通过锁住索引记录来实现的。

索引又是什么东西？为什么它可以被锁住?

还有两个问题没有解决：

1、为什么表里面没有索引的时候，锁住一行数据会导致锁表?

或者说，如果锁住的是索引，一张表没有索引怎么办?

所以，一张表有没有可能没有索引?

1) 如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引。

2) 如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引作为主键索引。

3) 如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐藏的聚集索引，它会随着行记录的写入而主键递增。

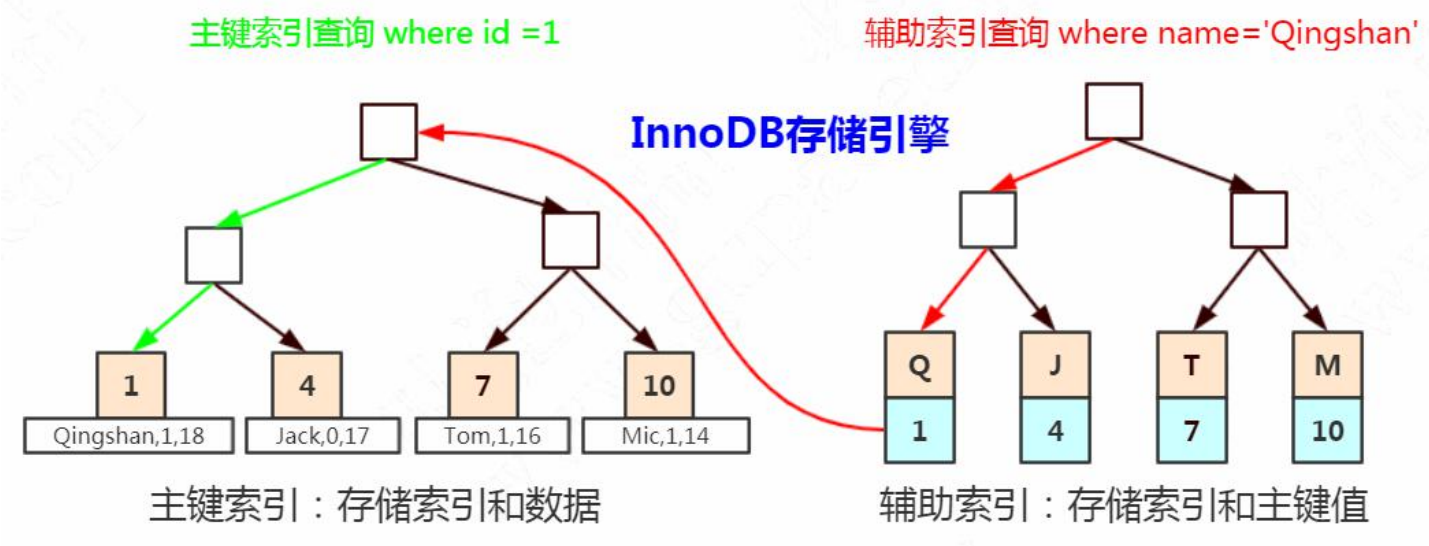
所以，为什么锁表，是因为查询没有使用索引，会进行全表扫描，然后把每一个隐藏的聚集索引都锁住了。

2、为什么通过唯一索引给数据行加锁，主键索引也会被锁住?

在 InnoDB 里面，当我们使用辅助索引的时候，它是怎么检索数据的？辅助索引的叶子节点存储的是什么内容?

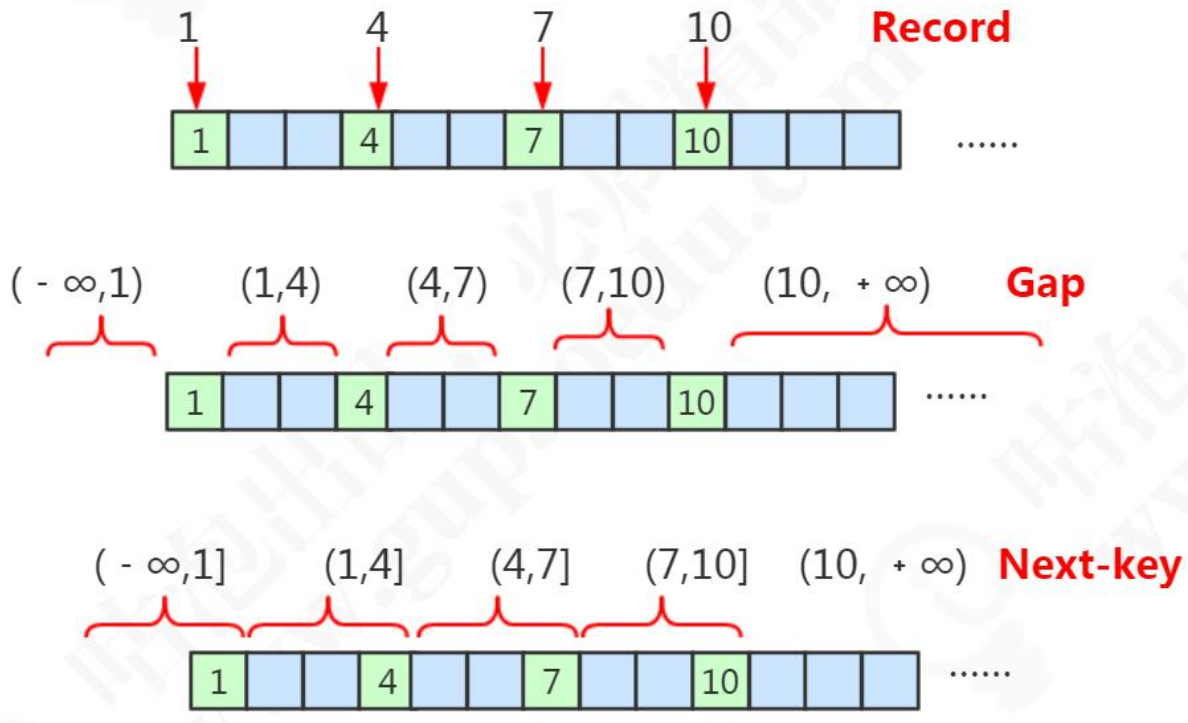
在辅助索引里面,索引存储的是二级索引和主键的值。比如 name=4,存储的是 name 的索引和主键 id 的值 4。

而主键索引里面除了索引之外，还存储了完整的数据。所以我们通过辅助索引锁定一行数据的时候，它跟我们检索数据的步骤是一样的，会通过主键值找到主键索引，然后也锁定。



4 锁的算法

t2 这张表有一个主键索引。



这些数据库里面存在的主键值，我们把它叫做 Record，记录，那么这里我们就有 4 个 Record。

根据主键，这些存在的 Record 隔开的数据不存在的区间，我们把它叫做 Gap，间隙，它是一个左开右开的区间，如果还有同学分不清开区间和闭区间的区别。

间隙（Gap）连同它左边的记录（Record），我们把它叫做临键的区间，它是一个左开右闭的区间。

4.1 记录锁

当我们对于唯一性的索引（包括唯一索引和主键索引）使用等值查询，精准匹配到一条记录的时候，这个时候使用的就是记录锁。

比如 where id = 1 4 7 10 。

4.2 间隙锁

当我们查询的记录不存在，没有命中任何一个 record，无论是用等值查询还是范围查询的时候，它使用的都是间隙锁。

举个例子，where id >4 and id <7，where id = 6。

Transaction 1	Transaction 2
begin;	

	INSERT INTO `t2` (`id`, `name`) VALUES (5, '5'); // BLOCKED
	INSERT INTO `t2` (`id`, `name`) VALUES (6, '6'); // BLOCKED
	select * from t2 where id =6 for update; // OK

select * from t2 where id >20 for update;	
	INSERT INTO `t2` (`id`, `name`) VALUES (11, '11'); // BLOCKED

注意，间隙锁主要是阻塞插入 insert。相同的间隙锁之间不冲突。

Gap Lock 只在 RR 中存在，如果要关闭间隙锁，就是把事务隔离级别设置成 RC，并且把 innodb_locks_unsafe_for_binlog 设置为 ON。

这种情况下除了外键约束和唯一性检查会加间隙锁，其他情况都不会用间隙锁。

4.3 临键锁

当我们使用了范围查询，不仅仅命中了 Record 记录，还包含了 Gap 间隙，在这种情况下我们使用的就是临键锁，它是 MySQL 里面默认的行锁算法，相当于记录锁加上间隙锁。

比如我们使用>5 <9， 它包含了记录不存在的区间，也包含了一个 Record 7。

Transaction 1	Transaction 2
begin;	
select * from t2 where id >5 and id < 9 for update;	
	begin;

	select * from t2 where id =4 for update; // OK
	INSERT INTO `t2` (`id`, `name`) VALUES (6, '6'); // BLOCKED
	INSERT INTO `t2` (`id`, `name`) VALUES (8, '8'); // BLOCKED
	select * from t2 where id =10 for update; // BLOCKED

临键锁，锁住最后一个 key 的下一个左开右闭的区间。

```
select * from t2 where id >5 and id <=7 for update;  -- 锁住(4,7]和(7,10]
select * from t2 where id >8 and id <=10 for update;  -- 锁住 (7,10]，(10,+∞)
```

为什么要锁住下一个左开右闭的区间？——就是为了解决幻读的问题。

4.4 小结：InnoDB 隔离级别的实现

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对 InnoDB 不可能
串行化（Serializable）	不可能	不可能	不可能

4.4.1 Read Uncommited

RU 隔离级别：不加锁。

4.4.2 Serializable

Serializable 所有的 select 语句都会被隐式的转化为 select ... in share mode，会和 update、delete 互斥。

这两个很好理解，主要是 RR 和 RC 的区别?

4.4.3 Repeatable Read

RR 隔离级别下，普通的 select 使用快照读(snapshot read)，底层使用 MVCC 来实现。

加锁的 select(select ... in share mode / select ... for update)以及更新操作 update, delete 等语句使用当前读（current read），底层使用记录锁、或者间隙锁、临键锁。

4.4.4 Read Commited

RC 隔离级别下，普通的 select 都是快照读，使用 MVCC 实现。

加锁的 select 都使用记录锁，因为没有 Gap Lock。

除了两种特殊情况——外键约束检查(foreign-key constraint checking)以及重复键检查(duplicate-key checking)时会使用间隙锁封锁区间。

所以 RC 会出现幻读的问题。