> 二狗：不多BB，先怼几道常问的大题目。MySQL 的事务隔离级别有哪些？分别用于解决什么问题？

主要用于解决脏读、不可重复读、幻读。

脏读：一个事务读取到另一个事务还未提交的数据。

不可重复读：在一个事务中多次读取同一个数据时，结果出现不一致。

幻读：在一个事务中使用相同的 SQL 两次读取，第二次读取到了其他事务新插入的行。

不可重复读注重于数据的修改，而幻读注重于数据的插入。

隔离级别	脏读	不可重复读	幻读
读未提交（Read Uncommitted）	有	有	有
读已提交（Read Committed）	无	有	有
可重复读（Repeatable Read）	无	无	有
串行化（Serializable）	无	无	

> 二狗：MySQL 的可重复读怎么实现的？

使用 MVCC 实现的，即 Mutil-Version Concurrency Control，多版本并发控制。关于 MVCC，比较常见的说法如下，包括《高性能 MySQL》也是这么介绍的。

InnoDB 在每行记录后面保存两个隐藏的列，分别保存了数据行的**创建版本号**和**删除版本号**。每开始一个新的事务，系统版本号都会递增。事务开始时刻的版本号会作为事务的版本号，用来和查询到的每行记录的版本号对比。在可重复读级别下，MVCC是如何操作的：

SELECT：必须同时满足以下两个条件，才能查询到。1）只查版本号早于当前版本的数据行；2）行的删除版本要么未定义，要么大于当前事务版本号。

INSERT：为插入的每一行保存当前系统版本号作为创建版本号。

DELETE：为删除的每一行保存当前系统版本号作为删除版本号。

UPDATE：插入一条新数据，保存当前系统版本号作为创建版本号。同时保存当前系统版本号作为原来的数据行删除版本号。

MVCC 只作用于 RC（Read Committed）和 RR（Repeatable Read）级别，因为 RU（Read Uncommitted）总是读取最新的数据版本，而不是符合当前事务版本的数据行。而 Serializable 则会对所有读取的行都加锁。这两种级别都不需要 MVCC 的帮助。

最初我也是坚信这个说法的，但是后面发现在某些场景下这个说法其实有点问题。

举个简单的例子来说：如果线程1和线程2先后开启了事务，事务版本号为1和2，如果在线程2开启事务的时候，线程1还未提交事务，则此时线程2的事务是不应该看到线程1的事务修改的内容的。

但是如果按上面的这种说法，由于线程1的事务版本早于线程2的事务版本，所以线程2的事务是可以看到线程1的事务修改内容的。

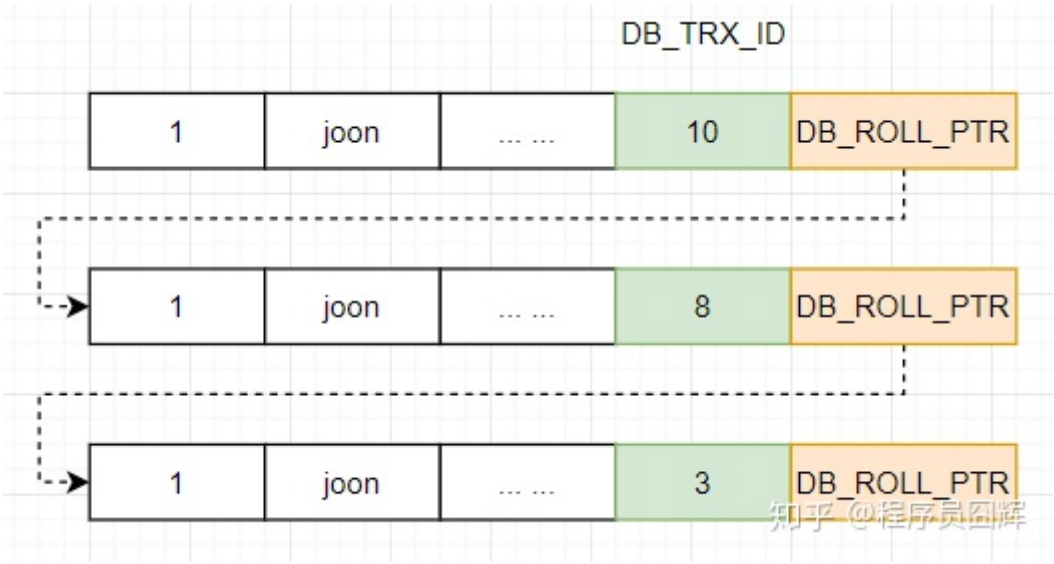
> 二狗：好像是有这个问题，那究竟是怎么实现的？

实际上，InnoDB 会在每行记录后面增加三个隐藏字段：

DB_ROW_ID：行ID，随着插入新行而单调递增，如果有主键，则不会包含该列。

DB_TRX_ID：记录插入或更新该行的事务的事务ID。

DB_ROLL_PTR：回滚指针，指向 undo log 记录。每次对某条记录进行改动时，该列会存一个指针，可以通过这个指针找到该记录修改前的信息 。当某条记录被多次修改时，该行记录会存在多个版本，通过 DB_ROLL_PTR 链接形成一个类似版本链的概念。



接下来进入正题，以 RR 级别为例：每开启一个事务时，系统会给该事务会分配一个事务 Id，在该事务执行第一个 select 语句的时候，会生成一个当前时间点的事务快照 ReadView，主要包含以下几个属性：

- trx_ids：生成 ReadView 时当前系统中活跃的事务 Id 列表，就是还未执行事务提交的。
- up_limit_id：低水位，取 trx_ids 中最小的那个，trx_id 小于该值都能看到。
- low_limit_id：高水位，生成 ReadView 时系统将要分配给下一个事务的id值，trx_id 大于等于该值都不能看到。
- creator_trx_id：生成该 ReadView 的事务的事务 Id。

有了这个ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

1）如果被访问版本的trx_id与ReadView中的creator_trx_id值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。

2）如果被访问版本的trx_id小于ReadView中的up_limit_id值，表明生成该版本的事务在当前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。

3）如果被访问版本的trx_id大于ReadView中的low_limit_id值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。

4）如果被访问版本的trx_id属性值在ReadView的up_limit_id和low_limit_id之间，那就需要判断一下trx_id属性值是不是在trx_ids列表中。如果在，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。

在进行判断时，首先会拿记录的最新版本来比较，如果该版本无法被当前事务看到，则通过记录的 DB_ROLL_PTR 找到上一个版本，重新进行比较，直到找到一个能被当前事务看到的版本。

而对于删除，其实就是一种特殊的更新，InnoDB 用一个额外的标记位 delete_bit 标识是否删除。当我们在进行判断时，会检查下 delete_bit 是否被标记，如果是，则跳过该版本，通过 DB_ROLL_PTR 拿到下一个版本进行判断。

以上内容是对于 RR 级别来说，而对于 RC 级别，其实整个过程几乎一样，唯一不同的是生成 ReadView 的时机，RR 级别只在事务开始时生成一次，之后一直使用该 ReadView。而 RC 级别则在每次 select 时，都会生成一个 ReadView。

> 二狗：那 MVCC 解决了幻读了没有?

幻读：在一个事务中使用相同的 SQL 两次读取，第二次读取到了其他事务新插入的行，则称为发生了幻读。

例如：

1）事务1第一次查询：select * from user where id < 10 时查到了 id = 1 的数据

2）事务2插入了 id = 2 的数据

3）事务1使用同样的语句第二次查询时，查到了 id = 1、id = 2 的数据，出现了幻读。

谈到幻读，首先我们要引入“当前读”和“快照读”的概念，聪明的你一定通过名字猜出来了：

快照读：生成一个事务快照（ReadView），之后都从这个快照获取数据。普通 select 语句就是快照读。

当前读：读取数据的最新版本。常见的 update/insert/delete、还有 select ... for update、select ... lock in share mode 都是当前读。

对于快照读，MVCC 因为因为从 ReadView 读取，所以必然不会看到新插入的行，所以天然就解决了幻读的问题。

而对于当前读的幻读，MVCC 是无法解决的。需要使用 Gap Lock 或 Next-Key Lock（Gap Lock + Record Lock）来解决。

其实原理也很简单，用上面的例子稍微修改下以触发当前读：select * from user where id < 10 for update，当使用了 Gap Lock 时，Gap 锁会锁住 id < 10 的整个范围，因此其他事务无法插入 id < 10 的数据，从而防止了幻读。

> 二狗：那经常有人说 Repeatable Read 解决了幻读是什么情况?

SQL 标准中规定的 RR 并不能消除幻读，但是 MySQL 的 RR 可以，靠的就是 Gap 锁。在 RR 级别下，Gap 锁是默认开启的，而在 RC 级别下，Gap 锁是关闭的。

> 二狗：小伙子不错，大活都给你搞下来了，接下来看下基础扎不扎实。什么是索引?

MySQL 官方对索引的定义为：索引（Index）是帮助 MySQL 高效获取数据的数据结构。简单的理解，索引类似于字典里面的目录。

> 二狗：常见的索引类型有哪些?

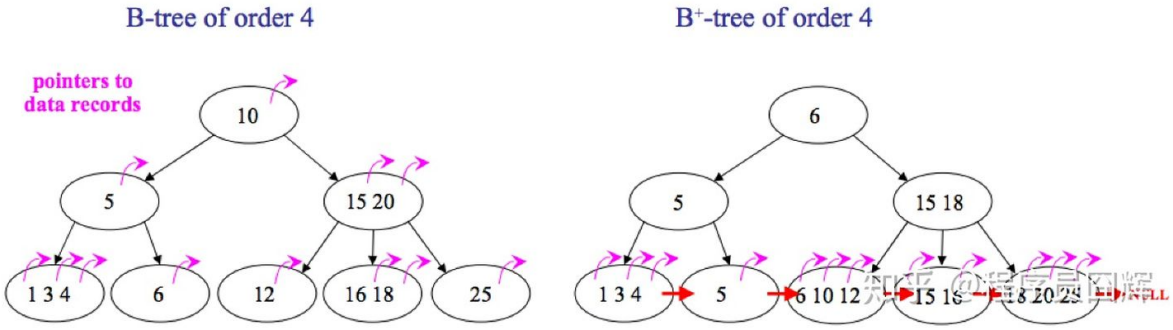
常见的索引类型有：hash、b树、b+树。

hash：底层就是 hash 表。进行查找时，根据 key 调用hash 函数获得对应的 hashcode，根据 hashcode 找到对应的数据行地址，根据地址拿到对应的数据。

B树：B树是一种多路搜索树，n 路搜索树代表每个节点最多有 n 个子节点。每个节点存储 key + 指向下一层节点的指针+ 指向 key 数据记录的地址。查找时，从根结点向下进行查找，直到找到对应的key。

B+树：B+树是b树的变种，主要区别在于：B+树的非叶子节点只存储 key + 指向下一层节点的指针。另外，B+树的叶子节点之间通过指针来连接，构成一个有序链表，因此对整棵树的遍历只需要一次线性遍历叶子结点即可。

- A B⁺-tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves



> 二狗：为什么MySQL数据库要用B+树存储索引？而不用红黑树、Hash、B树?

红黑树：如果在内存中，红黑树的查找效率比B树更高，但是涉及到磁盘操作，B树就更优了。因为红黑树是二叉树，数据量大时树的层数很高，从树的根结点向下寻找的过程，每读1个节点，都相当于一次IO操作，因此红黑树的I/O操作会比B树多的多。

hash 索引：如果只查询单个值的话，hash 索引的效率非常高。但是 hash 索引有几个问题：1）不支持范围查询；2）不支持索引值的排序操作；3）不支持联合索引的最左匹配规则。

B树索引：B树索相比于B+树，在进行范围查询时，需要做局部的中序遍历，可能要跨层访问，跨层访问代表着要进行额外的磁盘I/O操作；另外，B树的非叶子节点存放了数据记录的地址，会导致存放的节点更少，树的层数变高。

> 二狗：MySQL 中的索引叶子节点存放的是什么?

MyISAM和InnoDB都是采用的B+树作为索引结构，但是叶子节点的存储上有些不同。

MyISAM：主键索引和辅助索引（普通索引）的叶子节点都是存放 key 和 key 对应数据行的地址。在 MyISAM 中，主键索引和辅助索引没有任何区别。

InnoDB：主键索引存放的是 key 和 key 对应的数据行。辅助索引存放的是 key 和 key 对应的主键值。因此在使用辅助索引时，通常需要检索两次索引，首先检索辅助索引获得主键值，然后用主键值到主键索引中检索获得记录。

> 二狗：什么是聚簇索引（聚集索引）？

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。聚簇索引将索引和数据行放到了一块，找到索引也就找到了数据。因为无需进行回表操作，所以效率很高。

InnoDB 中必然会有，且只会有一个聚簇索引。通常是主键，如果没有主键，则优先选择非空的唯一索引，如果唯一索引也没有，则会创建一个隐藏的row_id 作为聚簇索引。至于为啥会只有一个聚簇索引，其实很简单，因为我们的数据只会存储一份。

而非聚簇索引则将数据存储和索引分开，找到索引后，需要通过对应的地址找到对应的数据行。MyISAM 的索引方式就是非聚簇索引。

> 二狗：什么是回表查询?

InnoDB 中，对于主键索引，只需要走一遍主键索引的查询就能在叶子节点拿到数据。

而对于普通索引，叶子节点存储的是 key + 主键值，因此需要再走一次主键索引，通过主键索引找到行记录，这就是所谓的回表查询，先定位主键值，再定位行记录。

> 二狗：走普通索引，一定会出现回表查询吗?

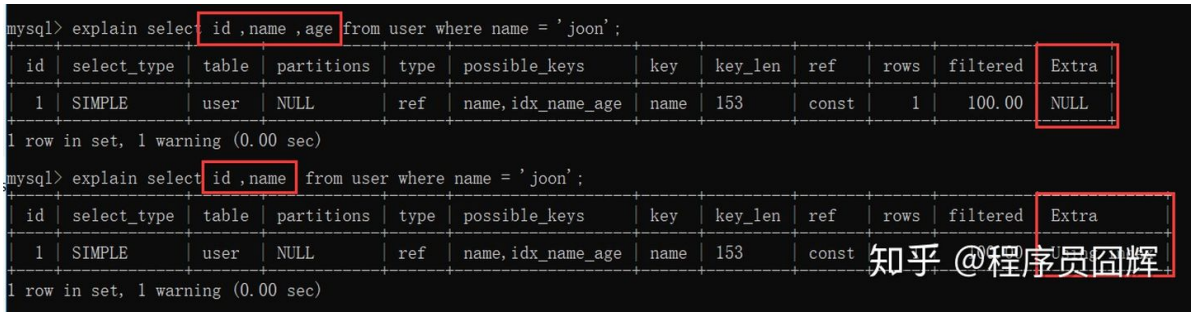
不一定，如果查询语句所要求的字段全部命中了索引，那么就不必再进行回表查询。

很容易理解，有一个 user 表，主键为 id，name 为普通索引，则再执行：select id, name from user where name = 'joonwhee' 时，通过name 的索引就能拿到 id 和 name了，因此无需再回表去查数据行了。

> 二狗：那你知道什么是覆盖索引（索引覆盖）吗?

覆盖索引是 SQL-Server 中的一种说法，上面讲的例子其实就实现了覆盖索引。具体的：当索引上包含了查询语句中的所有列时，我们无需进行回表查询就能拿到所有的请求数据，因此速度会很快。

当explain的输出结果Extra字段为Using index时，则代表触发覆盖索引。以上面的例子为例：



> 二狗：联合索引（复合索引）的底层实现？最佳左前缀原则？

联合索引底层还是使用B+树索引，并且还是只有一棵树，只是此时的排序会：首先按照第一个索引排序，在第一个索引相同的情况下，再按第二个索引排序，依次类推。

这也是为什么有“最佳左前缀原则”的原因，因为右边（后面）的索引都是在左边（前面）的索引排序的基础上进行排序的，如果没有左边的索引，单独看右边的索引，其实是无序的。

还是以字典为例，我们如果要查第2个字母为 k 的，通过目录是无法快速找的，因为首字母 A - Z 里面都可能包含第2个字母为 k 的。

> 二狗：union 和 union all 的区别

union all：对两个结果集直接进行并集操作，记录可能有重复，不会进行排序。

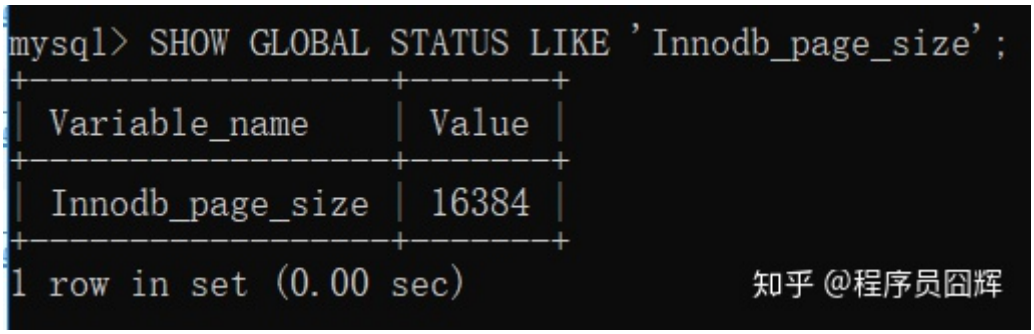
union：对两个结果集进行并集操作，会进行去重，记录不会重复，按字段的默认规则排序。

因此，从效率上说，UNION ALL 要比 UNION 更快。

> 二狗：B+树中一个节点到底多大合适？

1页或页的倍数最为合适。因为如果一个节点的大小小于1页，那么读取这个节点的时候其实也会读出1页，造成资源的浪费。所以为了不造成浪费，所以最后把一个节点的大小控制在1页、2页、3页等倍数页大小最为合适。

这里说的“页”是 MySQL 自定义的单位（和操作系统类似），MySQL 的 Innodb 引擎中1页的默认大小是16k，可以使用命令SHOW GLOBAL STATUS LIKE 'Innodb_page_size' 查看。



> 二狗：那 MySQL 中B+树的一个节点大小为多大呢？

在 MySQL 中 B+ 树的一个节点大小为“1页”，也就是16k。

> 二狗：为什么一个节点为1页就够了？

Innodb中，B+树中的一个节点存储的内容是：

- 非叶子节点：key + 指针
- 叶子节点：数据行（key 通常是数据的主键）

对于叶子节点：我们假设1行数据大小为1k（对于普通业务绝对够了），那么1页能存16条数据。

对于非叶子节点：key 使用 bigint 则为8字节，指针在 MySQL 中为6字节，一共是14字节，则16k能存放 16 * 1024 / 14 = 1170个。那么一颗高度为3的B+树能存储的数据为：1170 * 1170 * 16 = 21902400（千万级）。

所以在 InnoDB 中B+树高度一般为3层时，就能满足千万级的数据存储。在查找数据时一次页的查找代表一次IO，所以通过主键索引查询通常只需要1-3次 IO 操作即可查找到数据。千万级别对于一般的业务来说已经足够了，所以一个节点为1页，也就是16k是比较合理的。

> 二狗：什么是 Buffer Pool?

Buffer Pool 是 InnoDB 维护的一个缓存区域，用来缓存数据和索引在内存中，主要用来加速数据的读写，如果 Buffer Pool 越大，那么 MySQL 就越像一个内存数据库，默认大小为 128M。

InnoDB 会将那些热点数据和一些 InnoDB 认为即将访问到的数据存在 Buffer Pool 中，以提升数据的读取性能。

InnoDB 在修改数据时，如果数据的页在 Buffer Pool 中，则会直接修改 Buffer Pool，此时我们称这个页为脏页，InnoDB 会以一定的频率将脏页刷新到磁盘，这样可以尽量减少磁盘I/O，提升性能。

> 二狗：InnoDB 四大特性知道吗?

插入缓冲（insert buffer）：

索引是存储在磁盘上的，所以对于索引的操作需要涉及磁盘操作。如果我们使用自增主键，那么在插入主键索引（聚簇索引）时，只需不断追加即可，不需要磁盘的随机 I/O。但是如果我们使用的是普通索引，大概率是无序的，此时就涉及到磁盘的随机 I/O，而随机I/O的性能是比较差的（Kafka 官方数据：磁盘顺序I/O的性能是磁盘随机I/O的4000~5000倍）。

因此，InnoDB 存储引擎设计了 Insert Buffer ，对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池（Buffer pool）中，若在，则直接插入；若不在，则先放入到一个 Insert Buffer 对象中，然后再以一定的频率和情况进行 Insert Buffer 和辅助索引页子节点的 merge（合并）操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索引插入的性能。

插入缓冲的使用需要满足以下两个条件：1）索引是辅助索引；2）索引不是唯一的。

因为在插入缓冲时，数据库不会去查找索引页来判断插入的记录的唯一性。如果去查找肯定又会有随机读取的情况发生，从而导致 Insert Buffer 失去了意义。

二次写（double write）：

脏页刷盘风险：InnoDB 的 page size一般是16KB，操作系统写文件是以4KB作为单位，那么每写一个 InnoDB 的 page 到磁盘上，操作系统需要写4个块。于是可能出现16K的数据，写入4K 时，发生了系统断电或系统崩溃，只有一部分写是成功的，这就是 partial page write（部分页写入）问题。这时会出现数据不完整的问题。

这时是无法通过 redo log 恢复的，因为 redo log 记录的是对页的物理修改，如果页本身已经损坏，重做日志也无能为力。

doublewrite 就是用来解决该问题的。doublewrite 由两部分组成，一部分为内存中的 doublewrite buffer，其大小为2MB，另一部分是磁盘上共享表空间中连续的128个页，即2个区(extent)，大小也是2M。

为了解决 partial page write 问题，当 MySQL 将脏数据刷新到磁盘的时候，会进行以下操作：

1）先将脏数据复制到内存中的 doublewrite buffer

2）之后通过 doublewrite buffer 再分2次，每次1MB写入到共享表空间的磁盘上（顺序写，性能很高）

3）完成第二步之后，马上调用 fsync 函数，将doublewrite buffer中的脏页数据写入实际的各个表空间文件（离散写）。

如果操作系统在将页写入磁盘的过程中发生崩溃，InnoDB 再次启动后，发现了一个 page 数据已经损坏，InnoDB 存储引擎可以从共享表空间的 doublewrite 中找到该页的一个最近的副本，用于进行数据恢复了。

自适应哈希索引(adaptive hash index）：

哈希（hash）是一种非常快的查找方法，一般情况下查找的时间复杂度为 O(1)。但是由于不支持范围查询等条件的限制，InnoDB 并没有采用 hash 索引，但是如果能在一些特殊场景下使用 hash 索引，则可能是一个不错的补充，而 InnoDB 正是这么做的。

具体的，InnoDB 会监控对表上索引的查找，如果观察到某些索引被频繁访问，索引成为热数据，建立哈希索引可以带来速度的提升，则建立哈希索引，所以称之为自适应（adaptive）的。自适应哈希索引通过缓冲池的 B+ 树构造而来，因此建立的速度很快。而且不需要将整个表都建哈希索引，InnoDB 会自动根据访问的频率和模式来为某些页建立哈希索引。

预读（read ahead）：

InnoDB 在 I/O 的优化上有个比较重要的特性为预读，当 InnoDB 预计某些 page 可能很快就会需要用到时，它会异步地将这些 page 提前读取到缓冲池（buffer pool）中，这其实有点像空间局部性的概念。

空间局部性（spatial locality）：如果一个数据项被访问，那么与他地址相邻的数据项也可能很快被访问。

InnoDB使用两种预读算法来提高I/O性能：线性预读（linear read-ahead）和随机预读（randomread-ahead）。

其中，线性预读以 extent（块，1个 extent 等于64个 page）为单位，而随机预读放到以 extent 中的 page 为单位。线性预读着眼于将下一个extent 提前读取到 buffer pool 中，而随机预读着眼于将当前 extent 中的剩余的 page 提前读取到 buffer pool 中。

线性预读（Linear read-ahead）：线性预读方式有一个很重要的变量 innodb_read_ahead_threshold，可以控制 Innodb 执行预读操作的触发阈值。如果一个 extent 中的被顺序读取的 page 超过或者等于该参数变量时，Innodb将会异步的将下一个 extent 读取到 buffer pool 中，innodb_read_ahead_threshold 可以设置为0-64（一个 extend 上限就是64页）的任何值，默认值为56，值越高，访问模式检查越严格。

随机预读（Random read-ahead）: 随机预读方式则是表示当同一个 extent 中的一些 page 在 buffer pool 中发现时，Innodb 会将该 extent 中的剩余 page 一并读到 buffer pool中，由于随机预读方式给 Innodb code 带来了一些不必要的复杂性，同时在性能也存在不稳定性，在5.5中已经将这种预读方式废弃。要启用此功能，请将配置变量设置 innodb_random_read_ahead 为ON。

> 二狗：说说共享锁和排他锁?

共享锁又称为读锁，简称S锁，顾名思义，共享锁就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。

排他锁又称为写锁，简称X锁，顾名思义，排他锁就是不能与其他锁并存，如一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是获取排他锁的事务可以对数据就行读取和修改。

常见的几种 SQL 语句的加锁情况如下：

select * from table：不加锁

update/insert/delete：排他锁

select * from table where id = 1 for update：id为索引，加排他锁

select * from table where id = 1 lock in share mode：id为索引，加共享锁

> 二狗：说说数据库的行锁和表锁?

行锁：操作时只锁某一（些）行，不对其它行有影响。开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高。

表锁：即使操作一条记录也会锁住整个表。开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突概率高，并发度最低。

页锁：操作时锁住一页数据（16kb）。开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度一般。

InnoDB 有行锁和表锁，MyIsam 只有表锁。

> 二狗：InnoDB 的行锁是怎么实现的?

InnoDB 行锁是通过索引上的索引项来实现的。意味者：只有通过索引条件检索数据，InnoDB 才会使用行级锁，否则，InnoDB将使用表锁！

对于主键索引：直接锁住锁住主键索引即可。

对于普通索引：先锁住普通索引，接着锁住主键索引，这是因为一张表的索引可能存在多个，通过主键索引才能确保锁是唯一的，不然如果同时有2个事务对同1条数据的不同索引分别加锁，那就可能存在2个事务同时操作一条数据了。

> 二狗：InnoDB 锁的算法有哪几种?

Record lock：记录锁，单条索引记录上加锁，锁住的永远是索引，而非记录本身。

Gap lock：间隙锁，在索引记录之间的间隙中加锁，或者是在某一条索引记录之前或者之后加锁，并不包括该索引记录本身。

Next-key lock：Record lock 和 Gap lock 的结合，即除了锁住记录本身，也锁住索引之间的间隙。

> 二狗：MySQL 如何实现悲观锁和乐观锁?

乐观锁：更新时带上版本号（cas更新）

悲观锁：使用共享锁和排它锁，select...lock in share mode，select...for update。

> 二狗：InnoDB 和 MyISAM 的区别?

对比项	InnoDB	MyIsam
事务	支持	不支持
锁类型	行锁、表锁	表锁
缓存	缓存索引和数据	只缓存索引
主键	必须有，用于实现聚簇索引	可以没有
索引	B+树，主键是聚簇索引	B+树，非聚簇索引
select count(*) from table	较慢，扫描全表	贼快，用一个变量保存了表的行数，只需读出该变量即可
hash索引	支持	不支持
记录存储顺序	按主键大小有序插入	按记录插入顺序保存
外键	支持	不支持
全文索引	5.7 支持	支持
关注点	事务	性能

> 二狗：存储引擎的选择？

没有特殊情况，使用 InnoDB 即可。如果表中绝大多数都只是读查询，可以考虑 MyISAM。

> 二狗：explain 用过吗，有哪些字段分别是啥意思？

explain 字段有：

- id：标识符
- select_type：查询的类型
- table：输出结果集的表
- partitions：匹配的分区
- type：表的连接类型
- possible_keys：查询时，可能使用的索引
- key：实际使用的索引
- key_len：使用的索引字段的长度
- ref：列与索引的比较
- rows：估计要检查的行数
- filtered：按表条件过滤的行百分比
- Extra：附加信息

> 二狗：type 中有哪些常见的值？

按类型排序，从好到坏，常见的有：const > eq_ref > ref > range > index > ALL。

- const：通过主键或唯一键查询，并且结果只有1行（也就是用等号查询）。因为仅有一行，所以优化器的其余部分可以将这一行中的列值视为常量。
- eq_ref：通常出现于两表关联查询时，使用主键或者非空唯一键关联，并且查询条件不是主键或唯一键的等号查询。
- ref：通过普通索引查询，并且使用的等号查询。
- range：索引的范围查找（>=、<、in 等）。
- index：全索引扫描。
- All：全表扫描

> 二狗：explain 主要关注哪些字段？

主要关注 type、key、row、extra 等字段。主要是看是否使用了索引，是否扫描了过多的行数，是否出现 Using temporary、Using filesort 等一些影响性能的主要指标。

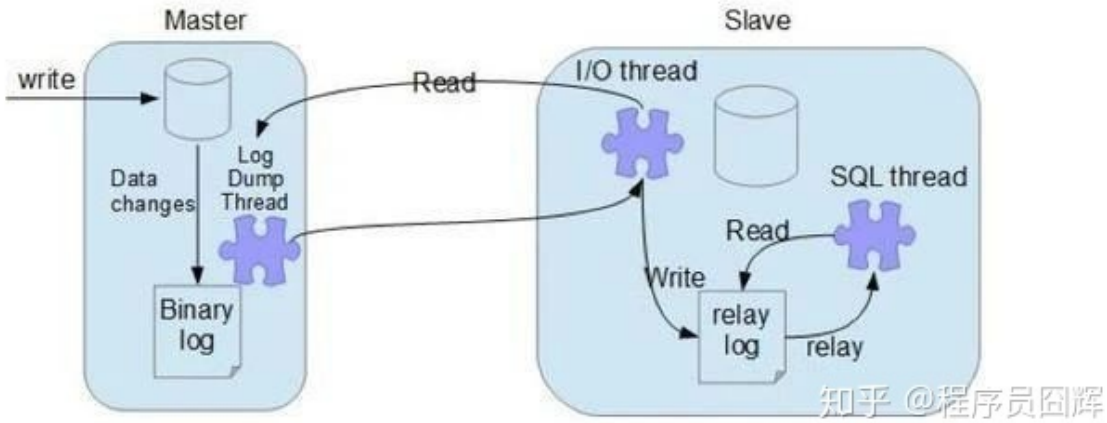
> 二狗：如何做慢 SQL 优化？

首先要搞明白慢的原因是什么：是查询条件没有命中索引？还是 load 了不需要的数据列？还是数据量太大？所以优化也是针对这三个方向来的。

- 首先用 explain 分析语句的执行计划，查看使用索引的情况，是不是查询没走索引，如果可以加索引解决，优先采用加索引解决。
- 分析语句，看看是否存在一些导致索引失效的用法，是否 load 了额外的数据，是否加载了许多结果中并不需要的列，对语句进行分析以及重写。
- 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行垂直拆分或者水平拆分。

> 二狗：说说 MySQL 的主从复制?

MySQL主从复制涉及到三个线程，一个运行在主节点（Log Dump Thread），其余两个（I/O Thread，SQL Thread）运行在从节点，如下图所示



主从复制默认是异步的模式，具体过程如下。

1）从节点上的I/O 线程连接主节点，并请求从指定日志文件（bin log file）的指定位置（bin log position，或者从最开始的日志）之后的日志内容；

2）主节点接收到来自从节点的 I/O请求后，读取指定文件的指定位置之后的日志信息，返回给从节点。返回信息中除了日志所包含的信息之外，还包括本次返回的信息的 bin-log file 以及 bin-log position；从节点的 I/O 进程接收到内容后，将接收到的日志内容更新到 relay log 中，并将读取到的 bin log file（文件名）和position（位置）保存到 master-info 文件中，以便在下一次读取的时候能够清楚的告诉 Master “我需要从某个bin-log 的哪个位置开始往后的日志内容”；

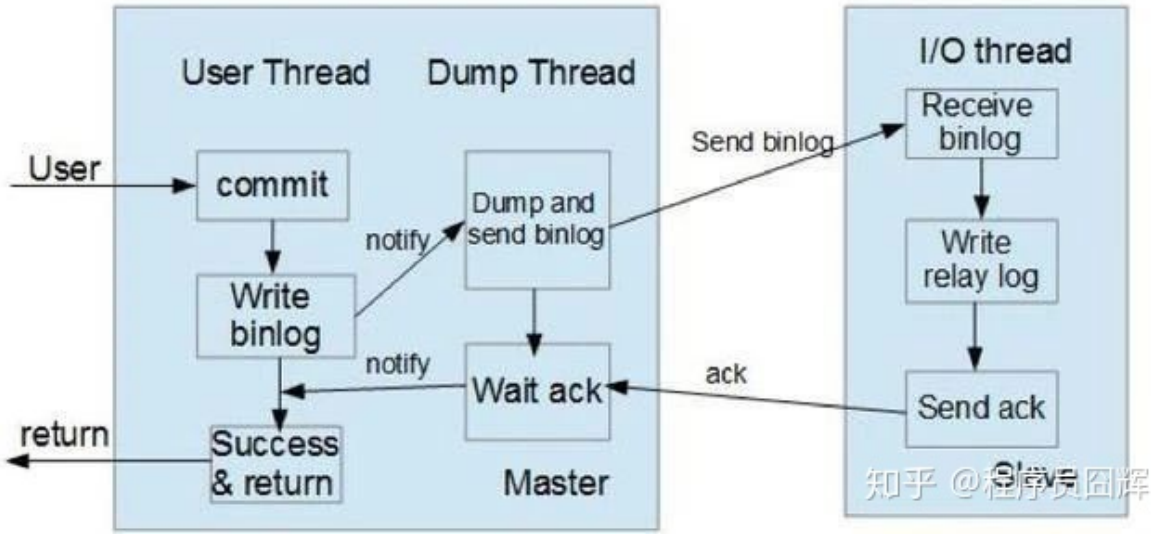
3）从节点的 SQL 线程检测到 relay-log 中新增加了内容后，会解析 relay-log 的内容，并在本数据库中执行。

> 二狗：异步复制，主库宕机后，数据可能丢失?

可以使用半同步复制或全同步复制。

半同步复制：

修改语句写入bin log后，不会立即给客户端返回结果。而是首先通过log dump 线程将 binlog 发送给从节点，从节点的 I/O 线程收到 binlog 后，写入到 relay log，然后返回 ACK 给主节点，主节点 收到 ACK 后，再返回给客户端成功。



半同步复制的特点：

- 确保事务提交后 binlog 至少传输到一个从库
- 不保证从库应用完这个事务的 binlog
- 性能有一定的降低，响应时间会更长
- 网络异常或从库宕机，卡主主库，直到超时或从库恢复

全同步复制：主节点和所有从节点全部执行了该事务并确认才会向客户端返回成功。因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。

> 二狗：主库写压力大，从库复制很可能出现延迟？

可以使用并行复制（并行是指从库多个SQL线程并行执行 relay log），解决从库复制延迟的问题。

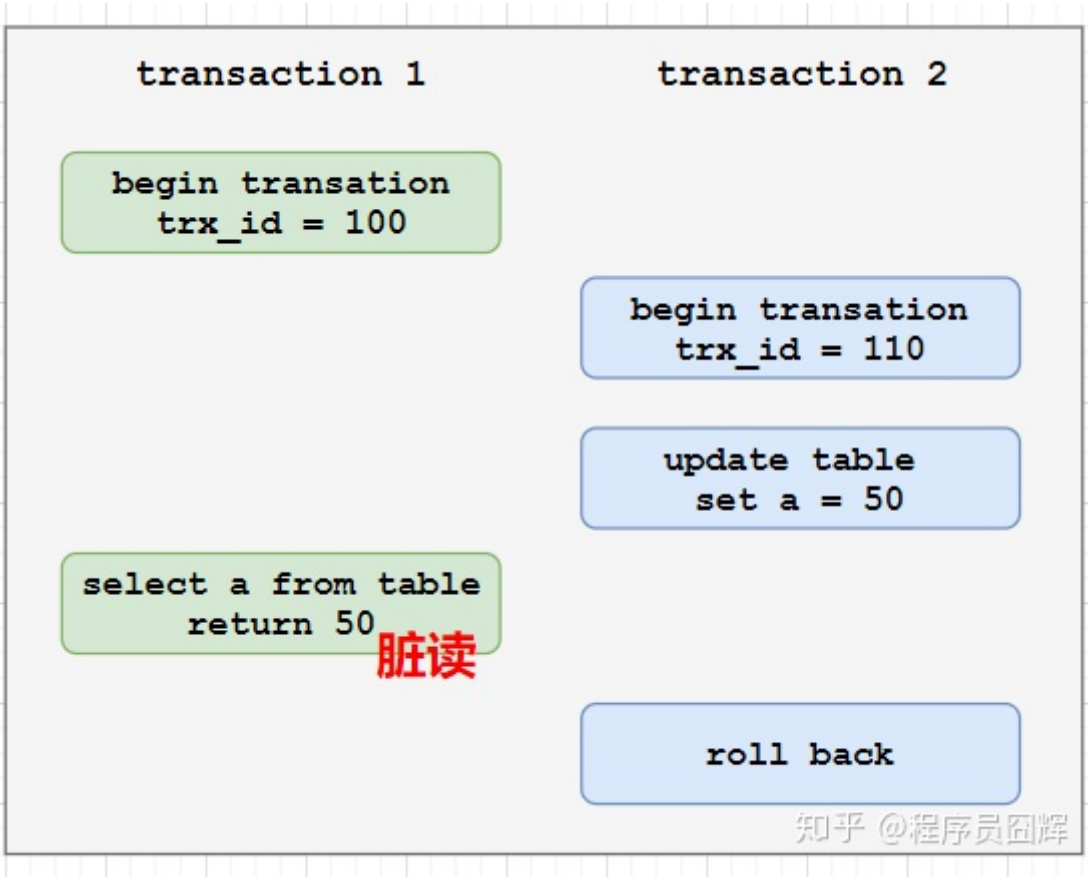
MySQL 5.7 中引入基于组提交的并行复制，其核心思想：一个组提交的事务都是可以并行回放，因为这些事务都已进入到事务的 prepare 阶段，则说明事务之间没有任何冲突（否则就不可能提交）。

判断事务是否处于一个组是通过 last_committed 变量，last_committed 表示事务提交的时候，上次事务提交的编号，如果事务具有相同的 last_committed，则表示这些事务都在一组内，可以进行并行的回放。

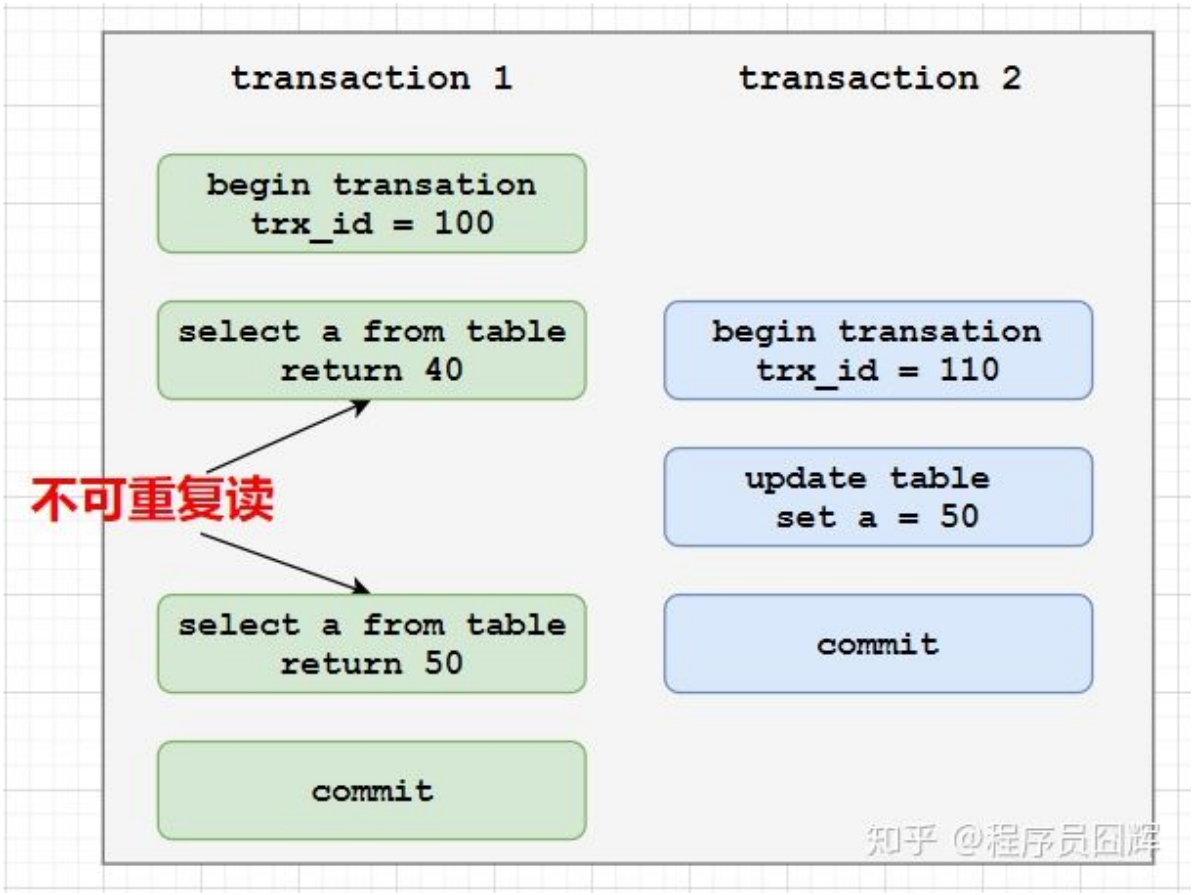
基础概念

并发事务带来的问题（现象）

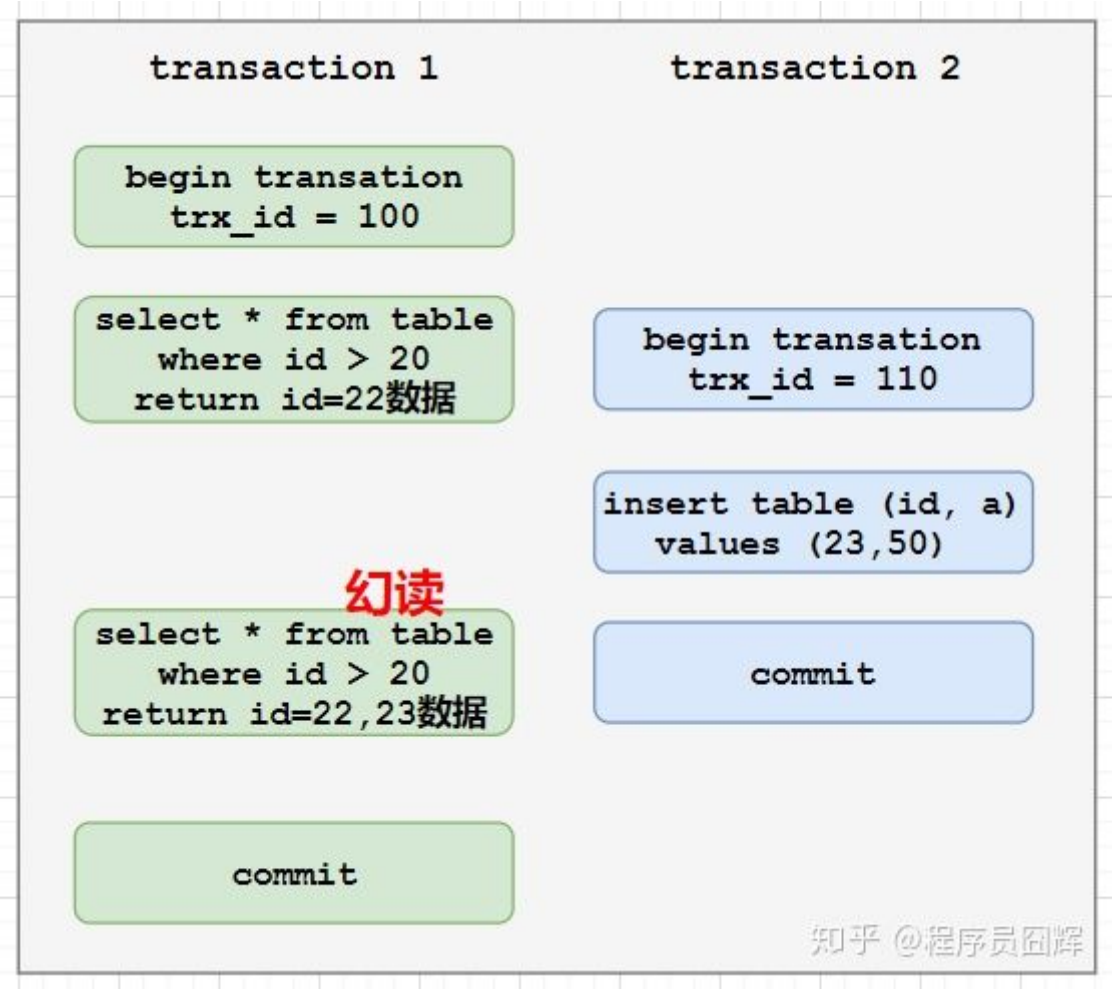
脏读：一个事务读取到另一个事务更新但还未提交的数据，如果另一个事务出现回滚或者进一步更新，则会出现问题。



不可重复读：在一个事务中两次次读取同一个数据时，由于在两次读取之间，另一个事务修改了该数据，所以出现两次读取的结果不一致。



幻读：在一个事务中使用相同的 SQL 两次读取，第二次读取到了其他事务新插入的行。



要解决这些并发事务带来的问题，一个比较简单粗暴的方法是加锁，但是加锁必然会带来性能的降低，因此 MySQL 使用了 MVCC 来提升并发事务下的性能。

MVCC 带来的好处?

试想，如果没有 MVCC，为了保证并发事务的安全，一个比较容易想到的办法就是加读写锁，实现：读读不冲突、读写冲突、写读冲突，写写冲突，在这种情况下，并发读写的性能必然会收到严重影响。

而通过 MVCC，我们可以做到读写之间不冲突，我们读的时候只需要将当前记录拷贝一份到内存中（ReadView），之后该事务的查询就只跟 ReadView 打交道，不影响其他事务对该记录的写操作。

事务隔离级别

读未提交（Read Uncommitted）：最低的隔离级别，会读取到其他事务还未提交的内容，存在脏读。

读已提交（Read Committed）：读取到的内容都是已经提交的，可以解决脏读，但是存在不可重复读。

可重复读（Repeatable Read）：在一个事务中多次读取时看到相同的内容，可以解决不可重复读，但是存在幻读。但是在 InnoDB 中不存在幻读问题，对于快照读，InnoDB 使用 MVCC 解决幻读，对于当前读，InnoDB 通过 gap locks 或 next-key locks 解决幻读。

串行化（Serializable）：最高的隔离级别，串行的执行事务，没有并发事务问题。

InnoDB MVCC 实现

核心数据结构

trx_sys_t：事务系统中央存储器数据结构

```
struct trx_sys_t {
  TrxSysMutex mutex; /*! 互斥锁 */

  MVCC *mvcc;    /*!  mvcc */

  volatile trx_id_t max_trx_id; /*! 要分配给下一个事务的事务id*/

  std::atomic<trx_id_t> min_active_id; /*! 最小的活跃事务Id */

  // 省略...

  trx_id_t rw_max_trx_id; /*!< 最大读写事务Id */

  // 省略...

  trx_ids_t rw_trx_ids; /*! 当前活跃的读写事务Id列表 */

  Rsegs rsegs; /*!< 回滚段 */

  // 省略...
};
```

MVCC：MVCC 读取视图管理器

```
class MVCC {
 public:
  // 省略...

  /** 创建一个视图 */
  void view_open(ReadView *&view, trx_t *trx);

  /** 关闭一个视图 */
  void view_close(ReadView *&view, bool own_mutex);

  /** 释放一个视图 */
  void view_release(ReadView *&view);

 // 省略...

  /** 判断视图是否处于活动和有效状态 */
  static bool is_view_active(ReadView *view) {
    ut_a(view != reinterpret_cast<ReadView *>(0x1));

    return (view != NULL && !(intptr_t(view) & 0x1));
  }
```

```
 // 省略...

 private:
  typedef UT_LIST_BASE_NODE_T(ReadView) view_list_t;

  /** 空闲可以被重用的视图*/
  view_list_t m_free;

  /**  活跃或者已经关闭的 Read View 的链表 */
  view_list_t m_views;
};
```

ReadView：视图，某一时刻的一个事务快照

```
class ReadView {

  // 省略...

 private:
  /** 高水位，大于等于这个ID的事务均不可见*/
  trx_id_t m_low_limit_id;

  /** 低水位：小于这个ID的事务均可见 */
  trx_id_t m_up_limit_id;

  /** 创建该 Read View 的事务ID*/
  trx_id_t m_creator_trx_id;

  /** 创建视图时的活跃事务id列表*/
  ids_t m_ids;

  /** 配合purge，标识该视图不需要小于m_low_limit_no的UNDO LOG，
   * 如果其他视图也不需要，则可以删除小于m_low_limit_no的UNDO LOG*/
  trx_id_t m_low_limit_no;

  /** 标记视图是否被关闭*/
  bool m_closed;

  // 省略...
};
```

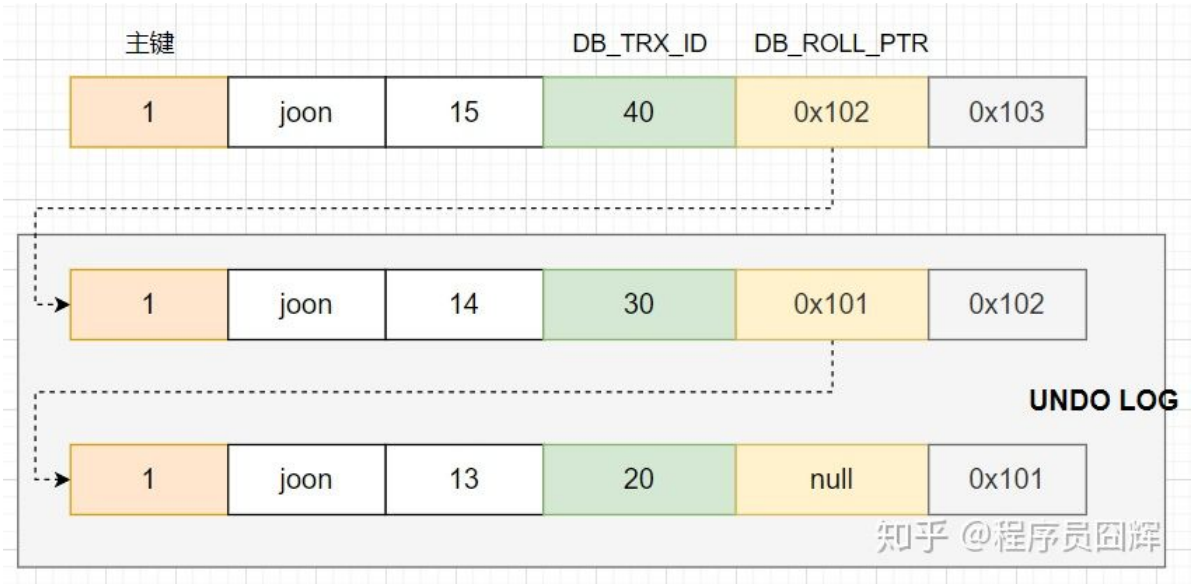
增加隐藏字段

为了实现 MVCC，InnoDB 会向数据库中的每行记录增加三个字段：

DB_ROW_ID：行ID，6字节，随着插入新行而单调递增，如果有主键，则不会包含该列。

DB_TRX_ID：事务ID，6字节，记录插入或更新该行的最后一个事务的事务标识，也就是事务ID。

DB_ROLL_PTR：回滚指针，7字节，指向写入回滚段的 undo log 记录。每次对某条记录进行更新时，会通过 undo log 记录更新前的行内容，更新后的行记录会通过 DB_ROLL_PTR 指向该 undo log 。当某条记录被多次修改时，该行记录会存在多个版本，通过DB_ROLL_PTR 链接形成一个类似版本链的概念，大致如下图所示。



源码分析

在源码中，添加这3个字段的方法在：/storage/innobase/dict/http://dict0dict.cc 的 dict_table_add_system_columns 方法中，核心部分如下图。

```
/** Adds system columns to a table object. 添加系统列到表对象*/
void dict_table_add_system_columns(dict_table_t *table, /*!< in/out: table */
                                   mem_heap_t *heap) /*!< in: temporary heap */
{
  // 1.添加行Id
  dict_mem_table_add_col(table, heap, "DB_ROW_ID", DATA_SYS,
                         DATA_ROW_ID | DATA_NOT_NULL, DATA_ROW_ID_LEN);
  // 2.添加事务Id
  dict_mem_table_add_col(table, heap, "DB_TRX_ID", DATA_SYS,
                         DATA_TRX_ID | DATA_NOT_NULL, DATA_TRX_ID_LEN);

  if (!table->is_intrinsic()) {
      // 3.添加回滚指针（table->is_intrinsic()：用于判断是否为内部表）
    dict_mem_table_add_col(table, heap, "DB_ROLL_PTR", DATA_SYS,
                           DATA_ROLL_PTR | DATA_NOT_NULL, DATA_ROLL_PTR_LEN);
  }
}
```


增删改的底层操作

当我们更新一条数据，InnoDB 会进行如下操作：

1. 加锁：对要更新的行记录加排他锁
2. 写 undo log：将更新前的记录写入 undo log，并构建指向该 undo log 的回滚指针 roll_ptr
3. 更新行记录：更新行记录的 DB_TRX_ID 属性为当前的事务Id，更新 DB_ROLL_PTR 属性为步骤2生成的回滚指针，将此次要更新的属性列更新为目标值
4. 写 redo log：DB_ROLL_PTR 使用步骤2生成的回滚指针，DB_TRX_ID 使用当前的事务Id，并填充更新后的属性值
5. 处理结束，释放排他锁

删除操作：在底层实现中是使用更新来实现的，逻辑基本和更新操作一样，几个需要注意的点：1）写 undo log 中，会通过 type_cmpl 来标识是删除还是更新，并且不记录列的旧值；2）这边不会直接删除，只会给行记录的 info_bits 打上删除标识（REC_INFO_DELETED_FLAG），之后会由专门的 purge 线程来执行真正的删除操作。

插入操作：相比于更新操作比较简单，就是新增一条记录，DB_TRX_ID 使用当前的事务Id，同样会有 undo log 和 redo log。

源码分析

更新行记录的核心源码在：/storage/innobase/btr/http://btr0cur.cc/btr_cur_update_in_place 方法，核心部分如下图。

```
dberr_t btr_cur_update_in_place(
    ulint flags,         /*!< in: undo logging and locking flags */
    btr_cur_t *cursor,
    ulint *offsets,      /*!< in/out: offsets on cursor->page_cur.rec */
    const upd_t *update, /*!< in: update vector */
    ulint cmpl_info,
    que_thr_t *thr,
    trx_id_t trx_id,     /*!< in: transaction id */
    mtr_t *mtr)
{
  // 1.通过游标获取记录指针
  rec = btr_cur_get_rec(cursor);
  // 省略...

  // 2.写undo log
  err = btr_cur_upd_lock_and_undo(flags, cursor, offsets, update, cmpl_info,
                                  thr, mtr, &roll_ptr);
  // 省略...
  if (!(flags & BTR_KEEP_SYS_FLAG) && !index->table->is_intrinsic()) {
    // 3.更新rec的trx_id、roll_ptr属性值
    row_upd_rec_sys_fields(rec, NULL, index, offsets, thr_get_trx(thr),
                           roll_ptr);
  }

  // 4.将rec要更新的属性更新为update的新值
  row_upd_rec_in_place(rec, index, offsets, update, page_zip);

  // 省略...

  // 5.写redo log
  btr_cur_update_in_place_log(flags, rec, index, update, trx_id, roll_ptr, mtr);

  // 省略...
  return (err);
}
```

构建一致性读取视图（ReadView）

当我们的隔离级别为 RR 时：每开启一个事务，系统会给该事务会分配一个事务 Id，在该事务执行第一个 select 语句的时候，会生成一个当前时间点的事务快照 ReadView，核心属性如下：

- m_ids：创建 ReadView 时当前系统中活跃的事务 Id 列表，可以理解为生成 ReadView 那一刻还未执行提交的事务，并且该列表是个升序列表。
- m_up_limit_id：低水位，取 m_ids 列表的第一个节点，因为 m_ids 是升序列表，因此也就是 m_ids 中事务 Id 最小的那个。
- m_low_limit_id：高水位，生成 ReadView 时系统将要分配给下一个事务的 Id 值。
- m_creator_trx_id：创建该 ReadView 的事务的事务 Id。

源码分析

MVCC 模式下的普通查询主方法入口在：/storage/innobase/row/http://row0sel.cc 的 row_search_mvcc 方法中，之后的所有源码分析基本都在该方法内。

具体创建视图的方法在 ReadView::prepare，调用链如下：

row_search_mvcc -> trx_assign_read_view -> MVCC::view_open ->

ReadView::prepare，源码如下：

```
void ReadView::prepare(trx_id_t id) {
  ut_ad(mutex_own(&trx_sys->mutex));
  // 1.当前ReadView创建版本号赋值为事务id
  m_creator_trx_id = id;
  // 2.高水位、低水位赋值为全局系统事务的版本号（系统分配给下一个事务的id）
  m_low_limit_no = m_low_limit_id = m_up_limit_id = trx_sys->max_trx_id;

  // 3.当前活跃事务列表处理
  if (!trx_sys->rw_trx_ids.empty()) {
    // 如果事务系统的活跃事务列表不为空，则将其拷贝至m_ids，该方法会通过
    // “m_up_limit_id = m_ids.front()”将m_ids中最小的事务id赋值给m_up_limit_id,
    // m_ids是个升序列表，因此头结点时最小的，并且后续查找trx_id是否在m_ids可以利
    // 用有序的特性进行二分查找
    copy_trx_ids(trx_sys->rw_trx_ids);
  } else {
    // 否则清空 Read View 的活跃事务列表
    m_ids.clear();
  }

  // ... 省略部分代码 ...

  // 4.视图是否关闭赋值为false
  m_closed = false;
}
```


最后，会将这个创建的 ReadView 添加到 MVCC 的 m_views 中。

视图可见性判断：SQL 查询走聚簇索引

有了这个 ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

1. 如果被访问版本的 trx_id 与 ReadView 中的 m_creator_trx_id 值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
2. 如果被访问版本的 trx_id 小于 ReadView 中的 m_up_limit_id（低水位），表明被访问版本的事务在当前事务生成 ReadView 前已经提交，所以该版本可以被当前事务访问。
3. 如果被访问版本的 trx_id 大于等于 ReadView 中的 m_low_limit_id（高水位），表明被访问版本的事务在当前事务生成 ReadView 后才开启，所以该版本不可以被当前事务访问。
4. 如果被访问版本的 trx_id 属性值在 ReadView 的 m_up_limit_id 和 m_low_limit_id 之间，那就需要判断 trx_id 属性值是不是在 m_ids 列表中，这边会通过二分法查找。如果在，说明创建 ReadView 时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建 ReadView 时生成该版本的事务已经被提交，该版本可以被访问。

在进行判断时，首先会拿记录的最新版本来比较，如果该版本无法被当前事务看到，则通过记录的 DB_ROLL_PTR 找到上一个版本，重新进行比较，直到找到一个能被当前事务看到的版本。

而对于删除，其实就是一种特殊的更新，InnoDB 在 info_bits 中用一个标记位 delete_flag 标识是否删除。当我们在进行判断时，会检查下 delete_flag 是否被标记，如果是，则会根据情况进行处理：1）如果索引是聚簇索引，并且具有唯一特性（主键、唯一索引等），则返回 DB_RECORD_NOT_FOUND；2）否则，会寻找下一条记录继续流程。

其实很容易理解，如果是唯一索引查询，必然只有一条记录，如果被删除了则直接返回空，而如果是普通索引，可能存在多个相同值的行记录，该行不存在，则继续查找下一条。

以上内容是对于 RR 级别来说，而对于 RC 级别，其实整个过程几乎一样，唯一不同的是生成 ReadView 的时机，RR 级别只在事务第一次 select 时生成一次，之后一直使用该 ReadView。而 RC 级别则在每次 select 时，都会生成一个 ReadView。

源码分析

走聚簇索引的核心流程在 row_search_mvcc 方法，如下：

```
else if (index == clust_index) {
// 索引是聚簇索引

// 判断rec是否在ReadView可见
if (srv_force_recovery < 5 &&
    !lock_clust_rec_cons_read_sees(rec, index, offsets,
                                   trx_get_read_view(trx))) {
  rec_t *old_vers;
  // 获取rec的上一个版本，赋值给old_vers
  err = row_sel_build_prev_vers_for_mysql(
      trx->read_view, clust_index, prebuilt, rec, &offsets, &heap,
      &old_vers, need_vrow ? &vrow : NULL, &mtr,
      prebuilt->get_lob_undo());

  if (err != DB_SUCCESS) {
    goto lock_wait_or_error;
  }

  if (old_vers == NULL) {
    /* The row did not exist yet in
    the read view  */
    goto next_rec;
  }
  // rec赋值为老版本，用于下一次判断
  rec = old_vers;
  prev_rec = rec;
}
```


视图可见性判断在方法：changes_visible，调用链如下：

row_search_mvcc -> lock_clust_rec_cons_read_sees ->

changes_visible，源码如下：

```
bool changes_visible(trx_id_t id, const table_name_t &name) const
    MY_ATTRIBUTE((warn_unused_result)) {
  ut_ad(id > 0);

  // id：被访问的记录版本的事务id，该方法在ReadView里，
  // 所以可以直接访问m_up_limit_id、m_creator_trx_id等属性

  // 1.事务id小于视图“低水位” 或 事务id等于创建该视图的事务id，则可见
  if (id < m_up_limit_id || id == m_creator_trx_id) {
    return (true);
  }
  // 检查id是否有效
  check_trx_id_sanity(id, name);
  // 2.事务id大于等于“高水位”，则不可见
  if (id >= m_low_limit_id) {
    return (false);

  } else if (m_ids.empty()) {
    // 3.如果当前视图的活跃事务id列表为空，则可见
    return (true);
  }

  const ids_t::value_type *p = m_ids.data();
  // 4.当事务id在m_up_limit_id和m_low_limit_id之间时，
  // 利用二分查找搜索活跃事务列表m_ids，如果id在m_ids数组中，
  // 表明ReadView创建时候，事务处于活跃状态，因此记录不可见，否则可见
  return (!std::binary_search(p, p + m_ids.size(), id));
}
```


判断记录是否被打上 delete_flag 标的方法在：/storage/innobase/include/rem0rec.ic 的 rec_get_deleted_flag 方法中，如下图。

```
/** The following function tells if record is delete marked.   记录是否被打上了删除标记
 @return nonzero if delete marked 返回非0代表被打上删除标记，返回0代表没有*/
UNIV_INLINE
ulint rec_get_deleted_flag(const rec_t *rec, /*!< in: physical record */
                           ulint comp) /*!< in: nonzero=compact page format*/
{
  if (comp) {
    return (rec_get_bit_field_1(rec, REC_NEW_INFO_BITS, REC_INFO_DELETED_FLAG,
                                REC_INFO_BITS_SHIFT));
  } else {
    return (rec_get_bit_field_1(rec, REC_OLD_INFO_BITS, REC_INFO_DELETED_FLAG,
                                REC_INFO_BITS_SHIFT));
  }
}
```


获取记录的上一个版本

获取记录的上一个版本，主要是通过 DB_ROLL_PTR 来实现，核心流程如下：

1. 获取记录的回滚指针 DB_ROLL_PTR、获取记录的事务Id
2. 通过回滚指针拿到对应的 undo log
3. 解析 undo log，并使用 undo log 构建用于更新向量 UPDATE
4. 构建记录的上一个版本：先用记录的当前版本填充，然后使用 UPDATE（undo log）进行覆盖。

源码解析

构建记录的上一个版本：trx_undo_prev_version_build，调用链如下：

row_search_mvcc -> row_sel_build_prev_vers_for_mysql -> row_vers_build_for_consistent_read -> trx_undo_prev_version_build，源码如下：

```
// 1.获取Record的回滚段指针roll_ptr
roll_ptr = row_get_rec_roll_ptr(rec, index, offsets);

// 2.获取Record的事务ID
rec_trx_id = row_get_rec_trx_id(rec, index, offsets);

// 3.通过回滚指针获取undo log记录，拷贝到heap中，并且填充给undo_rec
if (trx_undo_get_undo_rec(roll_ptr, rec_trx_id, heap, is_temp,
                          index->table->name, &undo_rec)) {
  // 省略...
}

// 4.以下开始解析undo log，填充给对应的参数（参考undo log的构造图，这边会按构造顺序依次读取填充），
type_cmpl_t type_cmpl;
// 4.1、通过undo_rec读取type、undo_no、cmpl_info、table_id的值，并填充到入参的变量中，
// 最后将读取这些值后，undo log记录的剩余部分返回，也就是赋值给ptr
ptr = trx_undo_rec_get_pars(undo_rec, &type, &cmpl_info, &dummy_extern,
                            &undo_no, &table_id, type_cmpl);
// 4.2、从ptr读取trx_id、roll_ptr、info_bits的值，并填充到入参的变量中，最后将读取这些值后，
// undo log记录的剩余部分返回，也就是赋值给ptr
ptr = trx_undo_update_rec_get_sys_cols(ptr, &trx_id, &roll_ptr, &info_bits);
// 4.3、undo log跳过行引用
ptr = trx_undo_rec_skip_row_ref(ptr, index);
// 4.4、通过undo log的剩余部分（roll_ptr、info_bts等）构建update（就是填充到update变量中）
ptr = trx_undo_update_rec_get_update(ptr, index, type, trx_id, roll_ptr,
                                     info_bits, NULL, heap, &update, lob_undo,
                                     type_cmpl);
if (row_upd_changes_field_size_or_external(index, offsets, update)) {
  // 省略...
} else {
  // 5.构建old_vers：先用rec填充，再用update覆盖（也就是rec的roll_ptr指向的undo log的内容）
  // 5.1、将rec拷贝到buf中，同时返回指向buf数据部分的指针，赋值给old_vers
  *old_vers = rec_copy(buf, rec, offsets);
  rec_offs_make_valid(*old_vers, index, offsets);
  // 5.2、使用update覆盖old_vers，相当于先将old_vers赋值为rec（当前版本），然后用undo log来覆盖
  row_upd_rec_in_place(*old_vers, index, offsets, update, NULL);
}
```


视图可见性判断：SQL 查询走普通（二级）索引

面试必问的 MySQL，你懂了吗？ 只分析了走聚簇索引的情况，本文简单的介绍下走普通（二级）索引的情况。

当走普通索引时，判断逻辑如下：

1. 判断被访问索引记录所在页的最大事务 Id 是否小于 ReadView 中的 m_up_limit_id（低水位），如果是则代表该页的最后一次修改事务 Id 在 ReadView 创建前以前已经提交，则必然可以访问；如果不是，并不代表一定不可以访问，道理跟走聚簇索引一样，事务 Id 大的也可能提交比较早，所以需要做进一步判断，见步骤2。
2. 使用 ICP（Index Condition Pushdown）根据索引信息来判断搜索条件是否满足，这边主要是在使用聚簇索引判断前先进行过滤，这边有三种情况：a）ICP 判断不满足条件但没有超出扫描范围，则获取下一条记录继续查找；b）如果不满足条件并且超出扫描返回，则返回 DB_RECORD_NOT_FOUND；c）如果 ICP 判断符合条件，则会获取对应的聚簇索引来进行可见性判断。

源码分析

普通（非聚簇）索引的视图可见性判断在方法：lock_sec_rec_cons_read_sees，调用链如下：

row_search_mvcc -> lock_sec_rec_cons_read_sees，源码如下：

```
bool lock_sec_rec_cons_read_sees(
    const rec_t *rec,           /*!< in: user record which
                                should be read or passed over
                                by a read cursor */
    const dict_index_t *index, /*!< in: index */
    const ReadView *view)       /*!< in: consistent read view */
{
  ut_ad(page_rec_is_user_rec(rec));

  /* NOTE that we might call this function while holding the search
  system latch. */

  if (recv_recovery_is_on()) {
    return (false);

  } else if (index->table->is_temporary()) {
    /* Temp-tables are not shared across connections and multiple
    transactions from different connections cannot simultaneously
    operate on same temp-table and so read of temp-table is
    always consistent read. */

    return (true);
  }
  // 1.取索引页上的PAGE_MAX_TRX_ID字段（索引页上的最大事务Id）
  trx_id_t max_trx_id = page_get_max_trx_id(page_align(rec));

  ut_ad(max_trx_id > 0);
  // 2.判断max_trx_id是否小于m_up_limit_id
  return (view->sees(max_trx_id));
}
```

```
/**
@param id   transaction to check
@return true if view sees transaction id */
bool sees(trx_id_t id) const { return (id < m_up_limit_id); }
```

扩展理解

ICP（Index Condition Pushdown）

ICP 是 MySQL 5.6 引入的一个优化，根据官方的说法：ICP 可以减少存储引擎访问基表的次数 和 MySQL 访问存储引擎的次数，这边涉及到 MySQL 底层的处理逻辑，不是本文重点，这边不进行细讲。

这边用官方的例子简单介绍下，我们有张 people 表，索引定义为：INDEX (zipcode, lastname, firstname)，对于以下这个 SQL：

```
SELECT *
FROM people
WHERE zipcode='95054' AND lastname LIKE '%etrunia%' AND address LIKE '%Main
Street%';
```

当没有使用 ICP 时：此查询会使用该索引，但是必须扫描 people 表所有符合 zipcode='95054' 条件的记录。

当使用 ICP 时：不仅会使用 zipcode 的条件来进行过滤，还会使用 （lastname LIKE '%etrunia%'）来进行过滤，这样可以避免扫描符合 zipcode 条件而不符合 lastname 条件匹配的记录行 。

ICP 的官方文档：https://dev.mysql.com/doc/refman/8.0/en/index-condition-pushdown-optimization.html

当前读和快照读

当前读：官方叫做 Locking Reads（锁定读取），读取数据的最新版本。常见的 update/insert/delete、还有 select ... for update、select ... lock in share mode 都是当前读。

官方文档：https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html

快照读：官方叫做 Consistent Nonlocking Reads（一致性非锁定读取，也叫一致性读取），读取快照版本，也就是 MVCC 生成的 ReadView。用于普通的 select 的语句。

官方文档：https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html

MVCC 解决了幻读了没有?

MVCC 解决了部分幻读，但并没有完全解决幻读。

对于快照读，MVCC 因为因为从 ReadView 读取，所以必然不会看到新插入的行，所以天然就解决了幻读的问题。

而对于当前读的幻读，MVCC 是无法解决的。需要使用 Gap Lock 或 Next-Key Lock（Gap Lock + Record Lock）来解决。

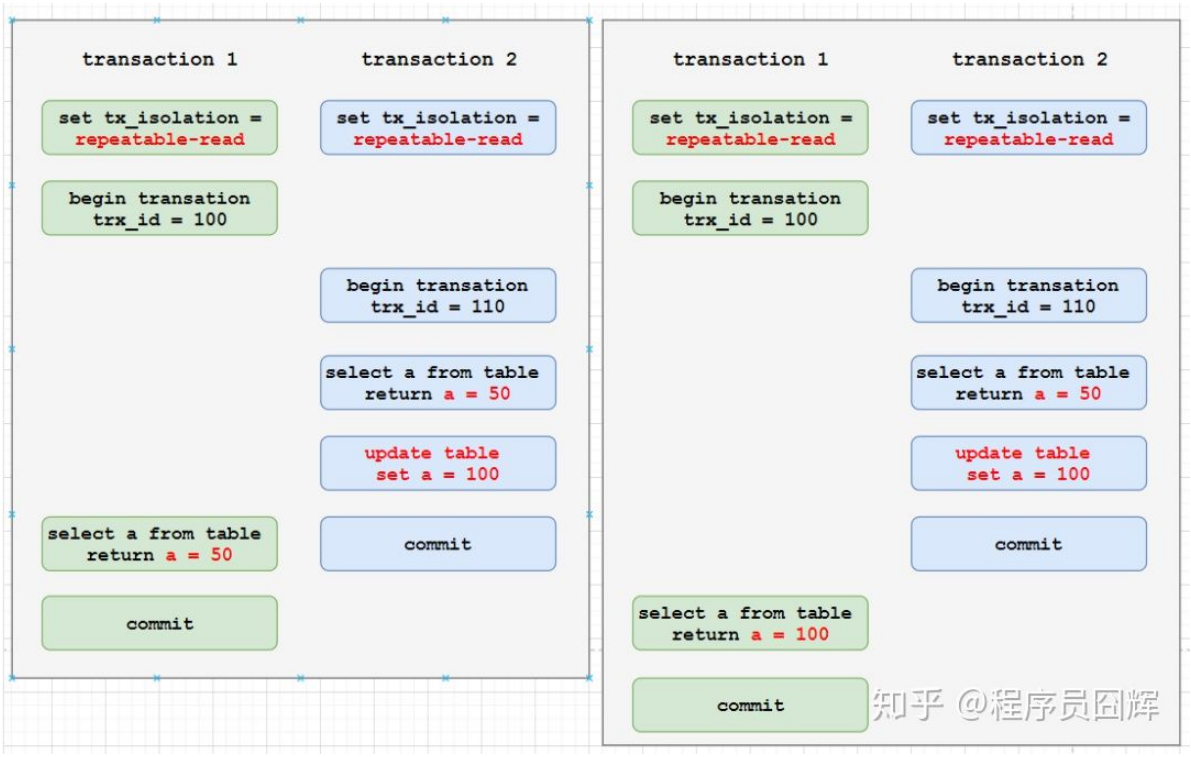
其实原理也很简单，用上面的例子稍微修改下以触发当前读：select * from user where id < 10 for update，当使用了 Gap Lock 时，Gap 锁会锁住 id < 10 的整个范围，因此其他事务无法插入 id < 10 的数据，从而防止了幻读。

Repeatable Read 解决了幻读是什么情况?

SQL 标准中规定的 RR 并不能消除幻读，但是 MySQL InnoDB 的 RR 可以，靠的就是 Gap 锁。在 RR 级别下，Gap 锁是默认开启的，而在 RC 级别下，Gap 锁是关闭的。

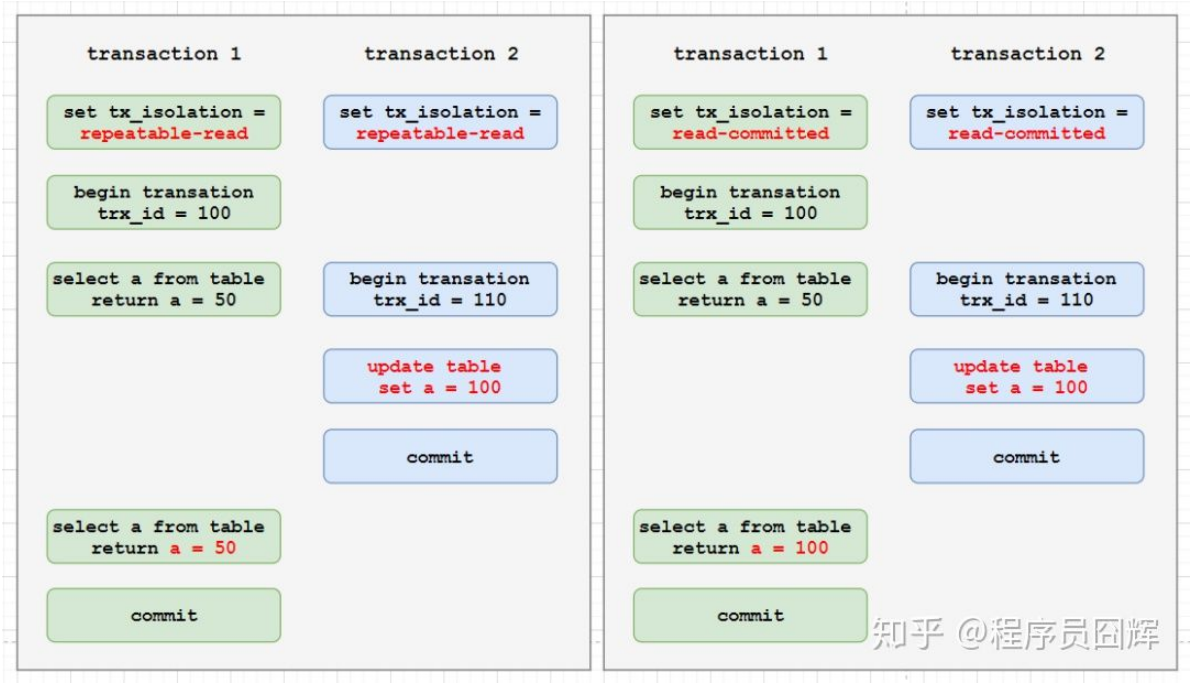
几个例子

例子1：RR（RC） 真正生成 ReadView 的时机



解析：RR 生成 ReadView 的时机是事务第一个 select 的时候，而不是事务开始的时候。右边的例子中，事务1在事务2提交了修改后才执行第一个 select，因此生成的 ReadView 中，a 的是 100 而不是事务1刚开始时的 50。

例子2：RR 和 RC 生成 ReadView 的区别



解析：RR 级别只在事务第一次 select 时生成一次，之后一直使用该 ReadView。而 RC 级别则在每次 select 时，都会生成一个 ReadView，所以 在第二次 select 时，读取到了事务2对于 a 的修改值。