Redo Undo MVCC

Content/内容

1. Undo Log
2. Redo Log
3. MySQL实例恢复
4. MySQL中的MVCC

没有Undo Log会是什么样？

- 用户发起插入操作

Begin
Insert
Update
Delete
Commit

只有Undo Log 没有Redo Log情况下

- 用户发起插入操作

Begin
记录A=1到Undo Log中
修改记录A=3
记录B=1到Undo Log中
修改记录B=2
写入Undo Log到磁盘中
将数据写入磁盘
Commit

Undo Log

Undo Log应遵从以下两个条件:

1. 事务T修改了数据库元素X，形式如<T, X, v>的日志记录必须在X的新值写到磁盘之前写入磁盘中。

2. 如果事务提交，则COMMIT日志记录必须在事务改变的所有数据库元素先写到磁盘之后写到磁盘中，但应尽快

问题点

Undo Log解决了原子性问题但是在性能方面有以下两点:

1. 每次数据库的修改还是必须要将数据以同步的方式写入磁盘。

2. 数据写入磁盘属于随机IO，性能方面较差。

Content/内容

1. Undo Log
2. Redo Log
3. MySQL实例恢复
4. MySQL中的MVCC

Redo Log

- 用户发起插入操作

Begin
记录A=1到Undo Log中
修改记录A=3
写日志到Redo Log中
记录B=1到Undo Log中
修改记录B=2
写日志到Redo Log中
写入Redo Log到磁盘中
Commit

引入Redo日志带来的好处

1. 数据写入磁盘不用同步写入，可以后台线程以异步的方式进行写入。

2. Redo是以追加的方式记录，将以前数据写入磁盘的随机IO，转换成顺序IO。

3. 为了实现上的方便undo日志也一样写入到Redo日志中。

Redo Log物理还是逻辑？

1. 物理日志:
   a) 新旧对象(块、页)都放入到日志中
   b) 只记录对象改变的部分

```
Struct value_log_record_for_page_update
{
    int          opcode;
    filename fname;
    long         pageno;
    int          offset;
    char         old_value[PAGESIZE];
    char         new_value[PAGESIZE];
}
```

Redo Log物理还是逻辑？

1. 逻辑日志:
   a) <插入操作，表名，记录值>

部分操作：

部分执行的事务需要使用Undo撤销，例如事务分为A B C三个步骤，逻辑日志可能A完成了或者A B完成了，但是C没有完成。

操作一致性:

多条消息或多次持续写入的操作就没有必要时原子的。所以在下次启时就会出现Undo和Redo不一致状态。例如事务分为A B C三个操作，那么{A,B,C}的任意子集在重启时都可能是持久的

Redo Log物理还是逻辑？

1. 物理-逻辑日志:
   a) 物理到页级别，但实际记录的还是逻辑日志

<insert op, tablename =A , record value = r>

<insert op, base filename = A ,page number = 502 , record value = r>
<insert op, index1 filename = B ,page number = 72 , index1 record value = keyB of r =s
<insert op, index1 filename = C ,page number = 50 , index2 record value = keyC of r =t

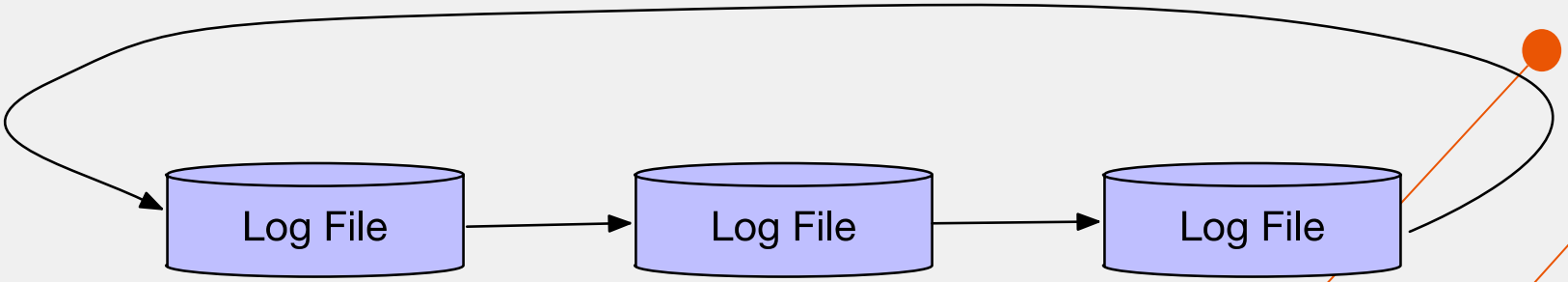
CheckPoint

CheckPoint:

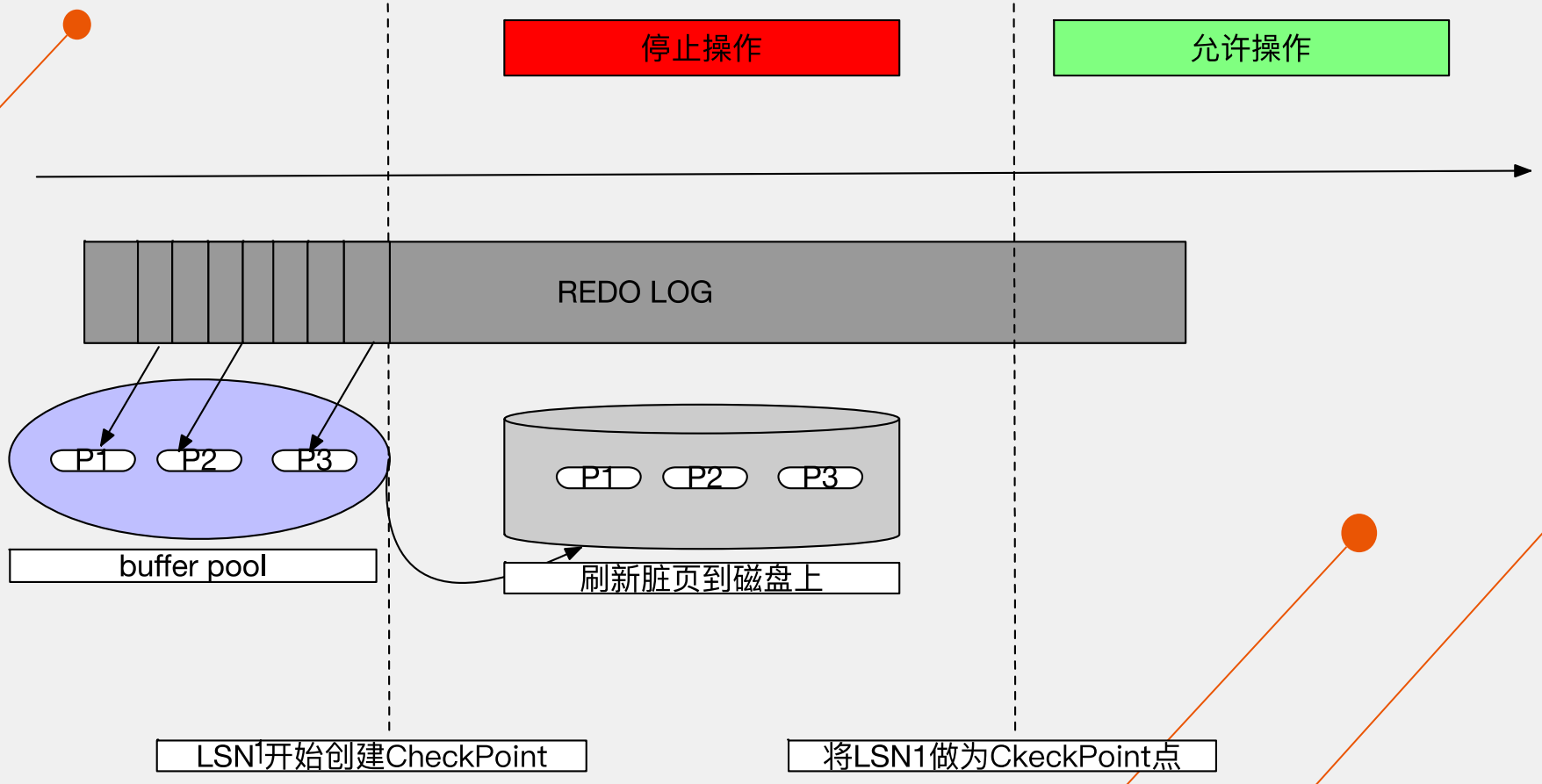
引入checkpoint机制减少实例恢复的时间。

脏页:

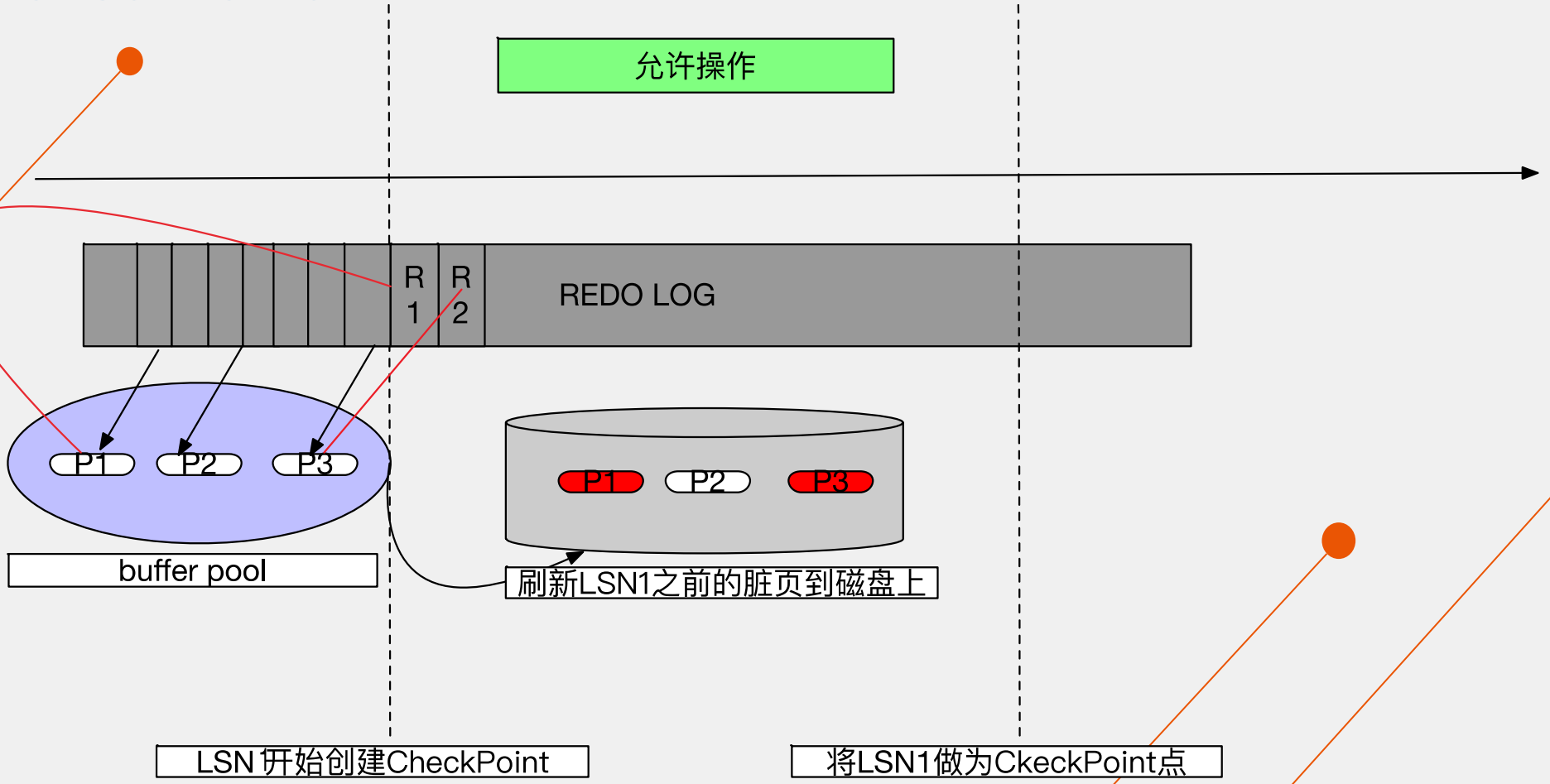
内存中的数据与磁盘上的数据不一样则成为脏页



CheckPoint



CheckPoint



Content/内容

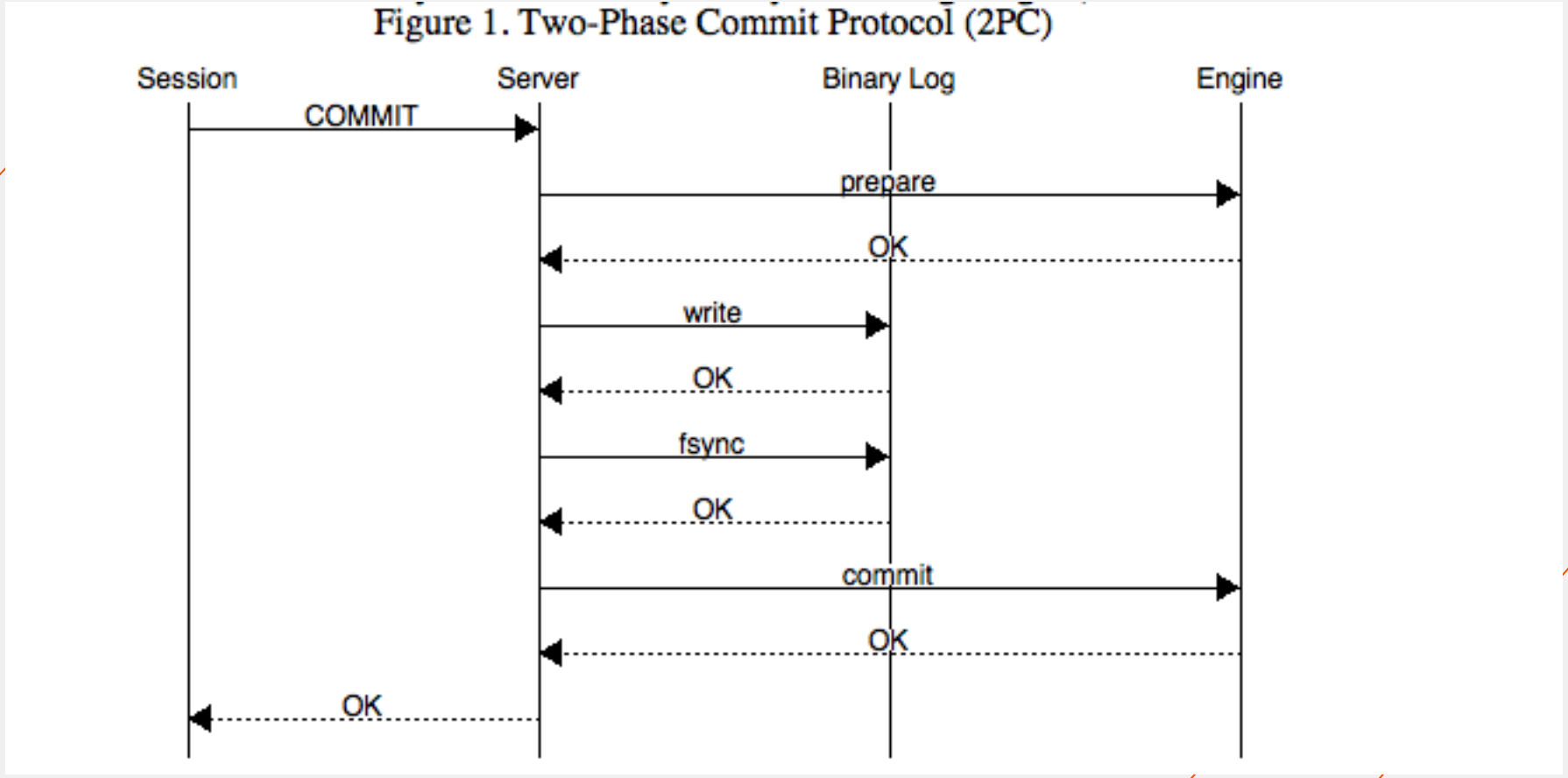
1. Undo Log
2. Redo Log
3. MySQL实例恢复
4. MySQL中的MVCC

实例恢复

MySQL Innodb实例恢复与以往数据库一样，先前滚Redo日志再利用Undo日志回滚但是完全采用这种方式会带来主从直接数据不一致问题。

当事务完成需要提交时，为了和BINLOG做XA，InnoDB的commit被划分成了两个阶段：prepare阶段和commit阶段

实例恢复



实例恢复

MySQL中实例恢复将存储引擎层与Server层分开，先恢复Innodb层恢复已提交的事务
Server层恢复分为两部分:

对于状态是Active状态的事务进行回滚
对于Prepare状态的事务，如果该事务对应的binlog已经记录则提交，
否则回滚事务

Content/内容

1. Undo Log
2. Redo Log
3. MySQL实例恢复
4. MySQL中的MVCC

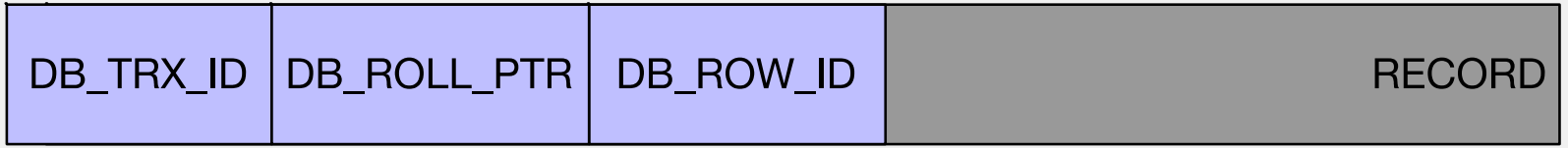
Multiversion concurrency control

If someone is reading from a database at the same time as someone else is writing to it, it is possible that the reader will see a half-written or inconsistent piece of data. There are several ways of solving this problem, known as concurrency controlmethods. The simplest way is to make all readers wait until the writer is done, which is known as a lock. This can be very slow, so MVCC takes a different approach: each user connected to the database sees a *snapshot* of the database at a particular instant in time. Any changes made by a writer will not be seen by other users of the database until the changes have been completed (or, in database terms: until the transaction has been committed.)

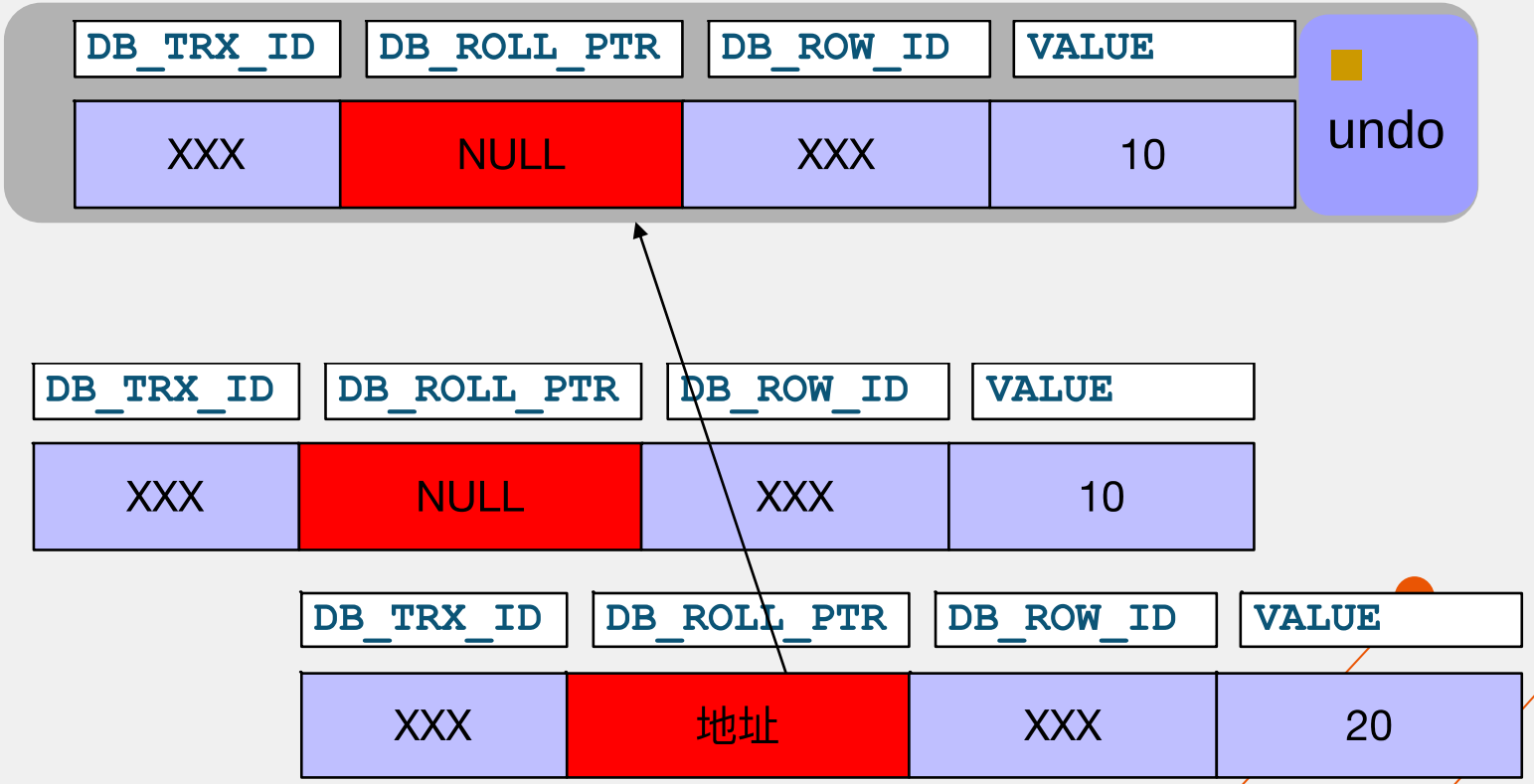
MySQL中的MVCC

Innodb表在记录行上会多出三个字段:

a) 6字节的事务ID
b) 7字节的指向回滚段的指针
c) 6字节的ROW_ID



MySQL中的MVCC



在RR隔离级别下

- MySQL内部并不是通过事务Id比较来判断数据的可见性

trx_id = 100

session1	session2
Start transaction;	
Select c1 from t ; return 2	Start transaction;
	Select c1 from t ; return 2
Insert into t values (3);	
	Select c1 from t; return 2
Commit;	
	Select c1 from t; return 2

trx_id = 101

如果通过trx_id比较 session1 的trx_id < session2的trx_id 则应该可以读取出3这条记录，但实际上是查询不出来。

MySQL中的MVCC

MySQL中当事务开始时候，会根据当前活跃的事务构造出一个列表(Read View).当读取一行记录时会根据行记录上的TRX_ID与Read View中的最大TRX_ID，最小TRX_ID比较来判断是否可见

1.比较TRX_ID 是否< read view中的最小TRX_ID
   a) 如果是则说明此事务早于read view中的所有事务结束则可以输出返回
   b) 如果否则判断TRX_ID 是否 > 最大TRX_ID
      - 如果是则根据行上的回滚指针找到回滚段中的对应的记录取出TRX_ID赋值给当前TRX_ID并执行步骤1
      - 如果否则判断TRX_ID是否在read view中
        - ✓ 如果是则根据行上的回滚指针找到回滚段中的对应的记录取出TRX_ID
        - ✓ 如果否则返回记录

```
函数row_search_mvcc->lock_clust_rec_cons_read_sees
bool
lock_clust_rec_cons_read_sees(
/*==========================*/
const rec_t*    rec,    /*!< in: user record which should be read or
passed over by a read cursor */
dict_index_t*   index,  /*!< in: clustered index */
const ulint*    offsets,/*!< in: rec_get_offsets(rec, index) */
ReadView*   view)   /*!< in: consistent read view */
{
ut_ad(index->is_clustered());
ut_ad(page_rec_is_user_rec(rec));
ut_ad(rec_offs_validate(rec, index, offsets));

/* 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. */
//只读事务或者临时表是不需要一致性读的判断
if (srv_read_only_mode || index->table->is_temporary()) {
ut_ad(view == 0 || index->table->is_temporary());
return(true);
}

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

trx_id_t    trx_id = row_get_rec_trx_id(rec, index, offsets); //获取记录上的TRX_ID这里需要解释下，我们一个查询可能满足的记录数有多个。那我们每读取一条记录的时候就要根据这条记录上的TRX_ID判断这条记录是否可见
return(view->changes_visible(trx_id, index->table->name)); //判断记录可见性
}
```

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

//如果ID小于Read View中最小的，则这条记录是可以看到。说明这条记录是在select这个事务开始之前就结束的
if (id < m_up_limit_id || id == m_creator_trx_id) {

return(true);
}

check_trx_id_sanity(id, name);

//如果比Read View中最大的还要大，则说明这条记录是在事务开始之后进行修改的，所以此条记录不应查看到
if (id >= m_low_limit_id) {

return(false);

} else if (m_ids.empty()) {

return(true);
}

const ids_t::value_type*    p = m_ids.data();

return(!std::binary_search(p, p + m_ids.size(), id)); //判断是否在Read View中， 如果在说明在创建Read View时 此条记录还处于活跃状态则不应该查询到，否则说明创建Read View是此条记录已经是不活跃状态则可以查询到
}
```

•READ-COMMITTED
事务内的每个查询语句都会重新创建Read View，这样就会产生不可重复读现象发生

•REPEATABLE-READ
事务内开始时创建Read View， 在事务结束这段时间内 每一次查询都不会重新重建Read View， 从而实现了可重复读。

参考资料:

《MySQL数据库InnoDB存储引擎Log漫游(1)》
《MySQL数据库InnoDB存储引擎Log漫游(2)》
《MySQL数据库InnoDB存储引擎Log漫游(3)》
《唐成－2016PG大会-数据库多版本实现内幕》
《事务处理概念与技术》
《数据库系统实现》

Thanks

董红禹
高级数据库工程师

LET DATA DRIVEN
数据 · 驱动之源