第八章　并发控制

基于时间戳的协议

事务时间戳的分配

- 每个事务 T_i 进入系统被分配一个时间戳 $TS(T_i)$
- 如果 T_j 晚于 T_i 进入系统， $TS(T_i) < TS(T_j)$
- 回滚的事务重新启动，分配新的时间戳

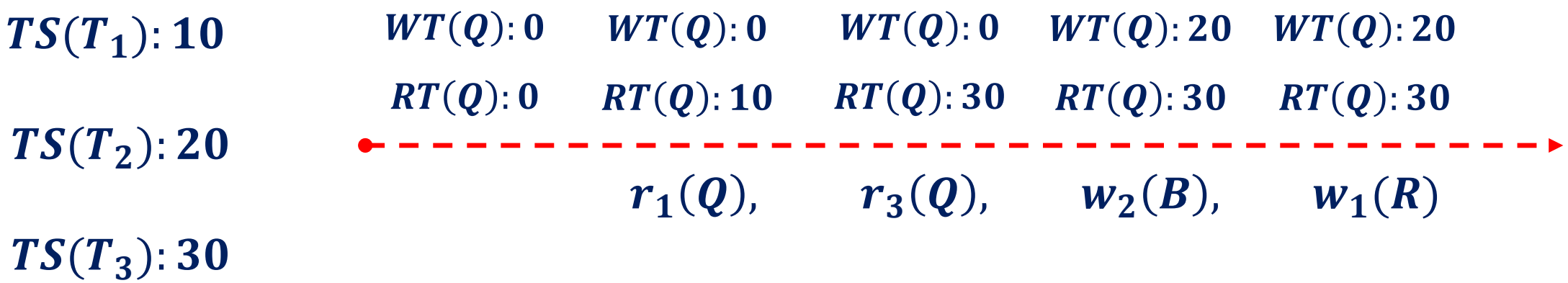
- 时间戳顺序决定了串行化顺序
- 回滚违反发出串行性操作的事务

$$start(t_1), r_1(A), start(t_2), r_2(B), w_2(B), commit(t_2), w_1(R), commit(t_1)$$

基于时间戳的协议

每个数据项 Q 有两个时间戳与之联系

- $WT(Q)$: 所有执行$write(Q)$的事务中最大的时间戳
- $RT(Q)$: 所有执行$read(Q)$的事务中最大的时间戳



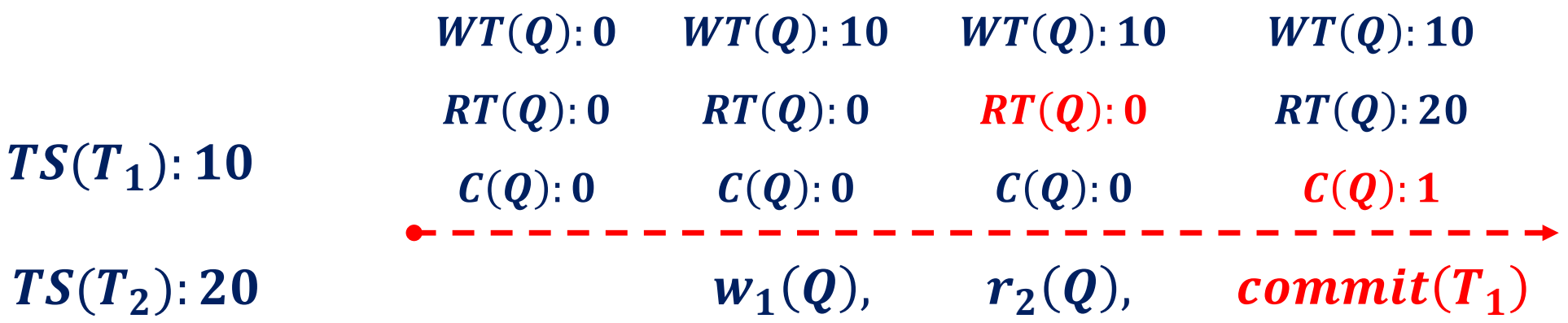
时间戳协议中的脏读

可能的脏读

$start(t_1), w_1(Q), start(t_2), r_2(Q), commit(t_2), rollback(t_1)$

为数据项 Q 设置提交位 $C(Q)$：

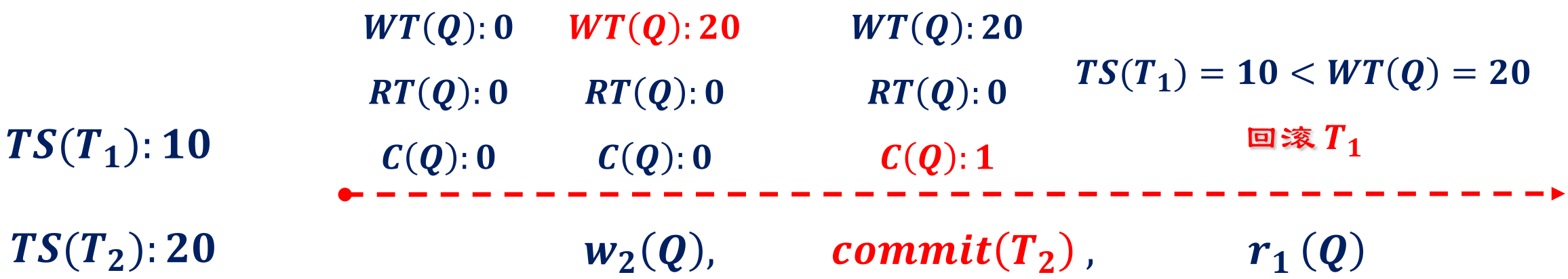
表示拥有 Q 上写时间戳的事务是否提交



假定事务 T_i 发出 $read(Q)$：过晚的读

如果 $TS(T_i) < WT(Q)$

T_i 需读入的值已经被覆盖，$read(Q)$操作被拒绝，回滚 T_i



假定事务T_i发出$read(Q)$：正常的读

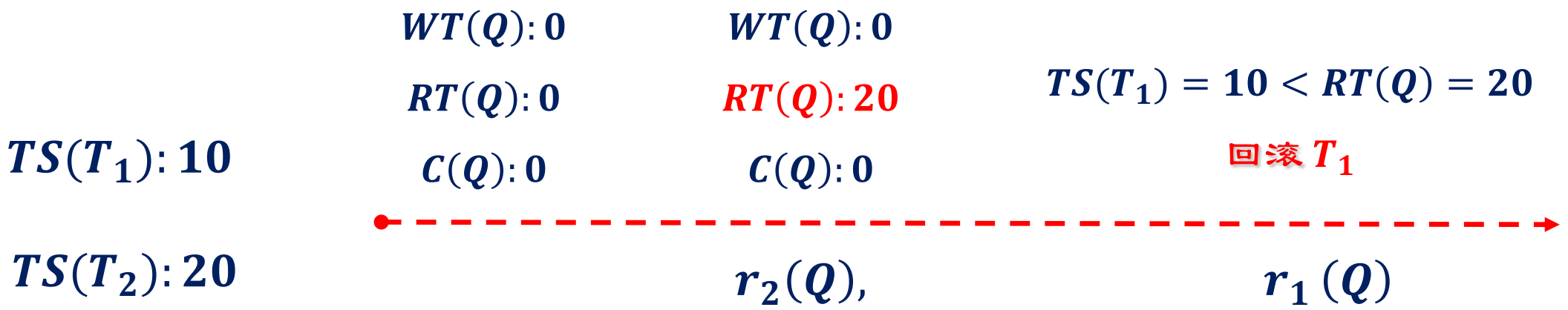
如果$TS(T_i) \geq WT(Q)$

- 若$C(Q)$为真则执行$read(Q)$操作，$RT(Q) = max(RT(Q), TS(T_i))$
- 若$C(Q)$为假则推迟到$C(Q)$为真或写Q的事务中止

假定事务 T_i 发出 $read(Q)$：过晚的写

如果 $TS(T_i) < RT(Q)$

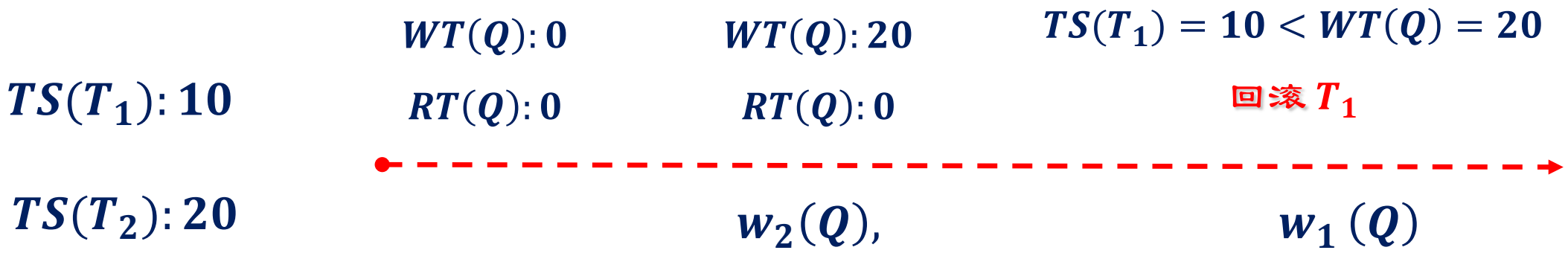
T_i 产生的值是先前所需要的值，$write\ (Q)$操作被拒绝，回滚 T_i



假定事务 T_i 发出 $write(Q)$：跳过的写

如果 $TS(T_i) < WT(Q)$

T_i 产生的值已经被其后的写事务覆盖，跳过 T_i 的 $write(Q)$



Thomas写规则：写操作在更晚的写操作已经发生时可以跳过

假定事务T_i发出$read(Q)$：正常的写

如果 $TS(T_i) > RT(Q) \wedge TS(T_i) > WT(Q)$

则 执行 T_i的$write\ (Q)$, $WT(Q) = TS(T_i)$

$TS(T_1): 10$

$TS(T_2): 20$

$TS(T_3): 30$

$WT(Q): 0$

$RT(Q): 0$

$WT(Q): 10$

$RT(Q): 10$

$WT(Q): 20$

$RT(Q): 0$

$TS(T_3) = 30 > WT(Q) = 20$

$TS(T_3) = 30 > RT(Q) = 10$

执行$w_3(Q), WT(Q) = 30$

$r_1(Q)$,

$w_2\ (Q)$,

$w_3\ (Q)$

基于时间戳协议的练习

T_1	T_2	T_3	A	B	C
200	150	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
$r_1(B)$				RT=200	
	$r_2(A)$		RT=150		
		$r_3(C)$			RT=175
$w_1(B)$				WT=200	
$w_1(A)$			WT=200		
	$w_2(C)$				
	中止				
		$w_3(A)$			


时间戳协议
有效性检查协议
MVCC

有效性检查协议

读阶段

事务T_i在这一阶段中执行。数据项被读入并保存在T_i的局部变量中。所有$write$操作都是对局部临时变量进行的，并不对数据库进行真正更新

有效性检查阶段

T_i进行有效性检查，通过与其他事务的读写集合进行比较，来判定是否可以将$write$操作所更新的临时局部变量值拷入数据库而不违反可串行性

写阶段

若T_i通过有效性检查，则进行实际的数据库更新，否则回滚T_i

有效性检查协议

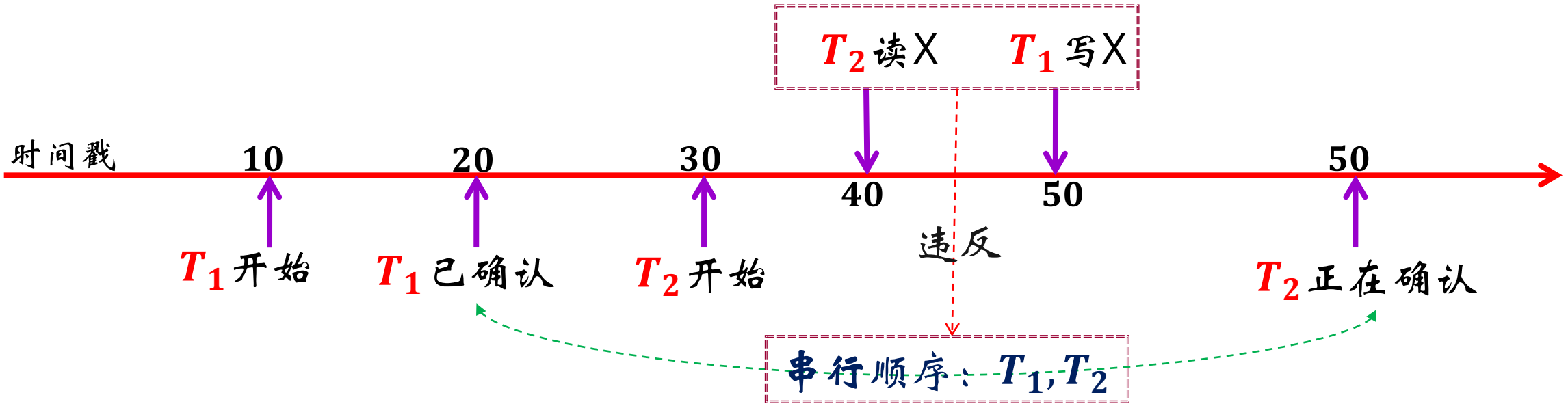
每个事务T_i有3个时间戳与之联系

- $Start(T_i)$：T_i开始执行的时间
- $Validation(T_i)$：T_i进入其有效性检查阶段的时间
- $Finish(T_i)$：T_i完成其写阶段的时间

令$TS(T_i) = Validation(T_i)$

等价的串行顺序与有效性确认时间戳一致

违反串行性的情况



$RS(T_i)$：T_i的读集合

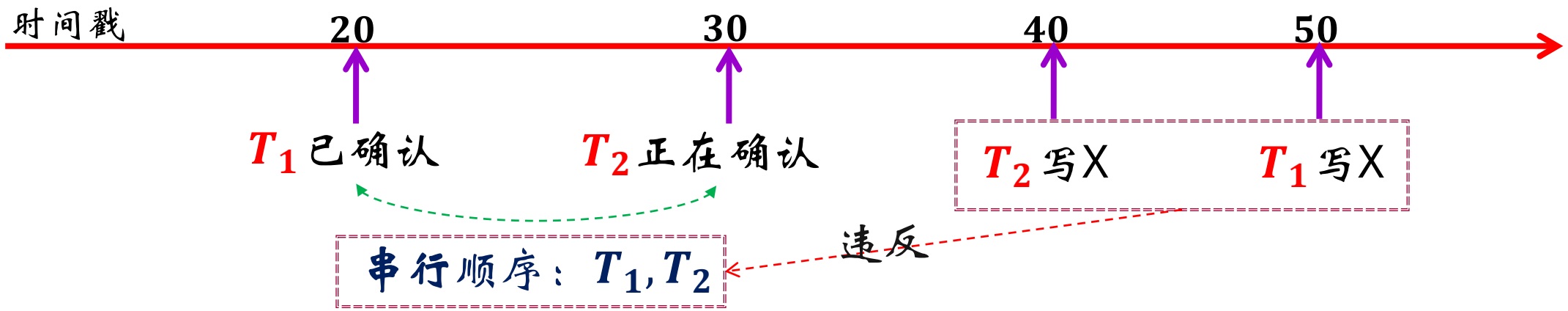
$WS(T_i)$：T_i的写集合

有效性确认检查条件

if $start(T_2) < finish(T_1)$

then $RS(T_2) \cap WS(T_1) = \Phi$

违反串行性的情况

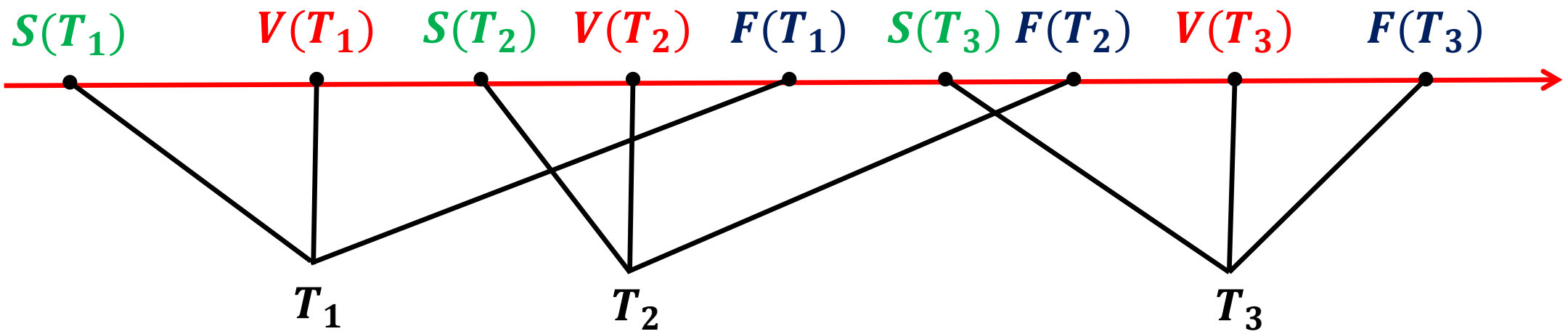


有效性确认检查条件

$$if \quad finish(T_1) > validation(T_2)$$

$$then \quad WS(T_1) \cap WS(T_2) = \Phi$$

有效性检查协议的例子

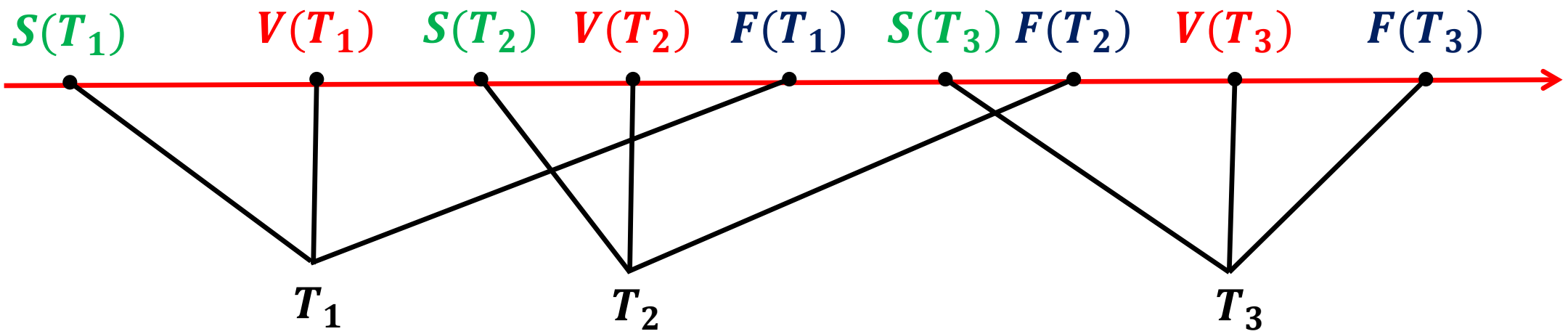


确认T_2：$S(T_2) < F(T_1)$，T_2可能先于T_1的写入进行读写

检查$WS(T_1) \cap WS(T_2) = \Phi$是否成立

检查$RS(T_2) \cap WS(T_1) = \Phi$是否成立

有效性检查协议的例子



确认T_3：$S(T_3) > F(T_1)$，不需要判定T_3和T_1的相交性

确认T_3：$S(T_3) < F(T_2)$，检查$RS(T_3) \cap WS(T_2) = \Phi$是否成立

思考：需要检查$WS(T_1) \cap WS(T_2) = \Phi$是否成立吗？

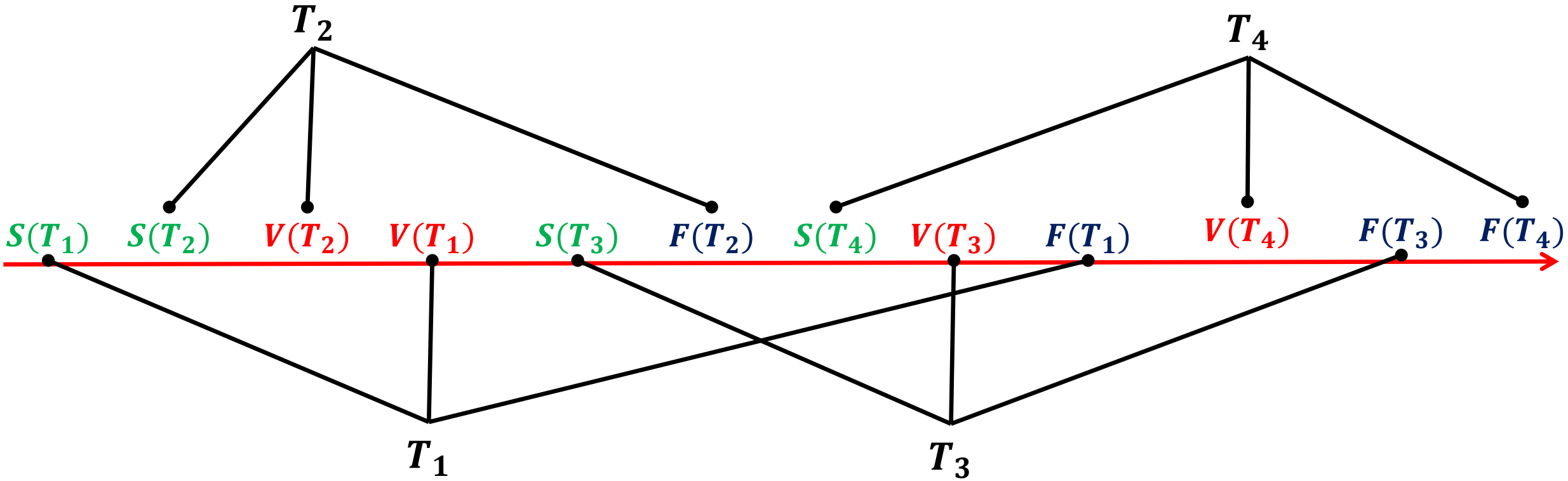
有效性检查协议的另外一个例子

$RS(T_2) = \{B\}$

$WS(T_2) = \{D\}$

$RS(T_4) = \{A, D\}$

$WS(T_4) = \{A, C\}$



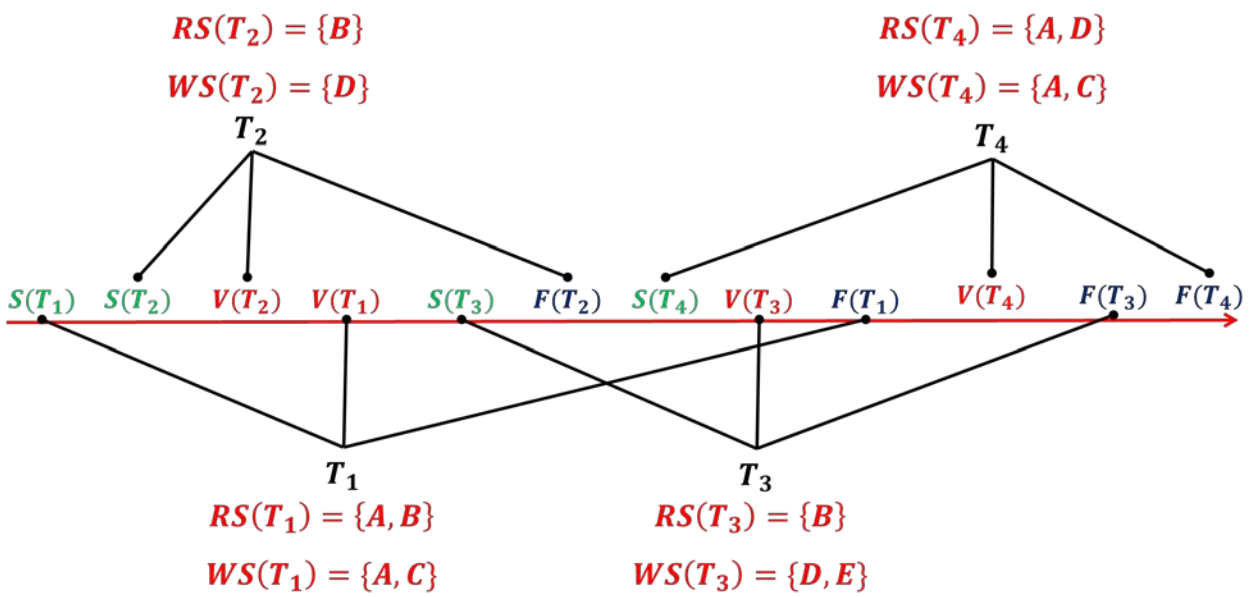
$RS(T_1) = \{A, B\}$

$WS(T_1) = \{A, C\}$

$RS(T_3) = \{B\}$

$WS(T_3) = \{D, E\}$

有效性检查协议的例子



确认T_2：没有其他确认事务

T_2确认成功，写入D

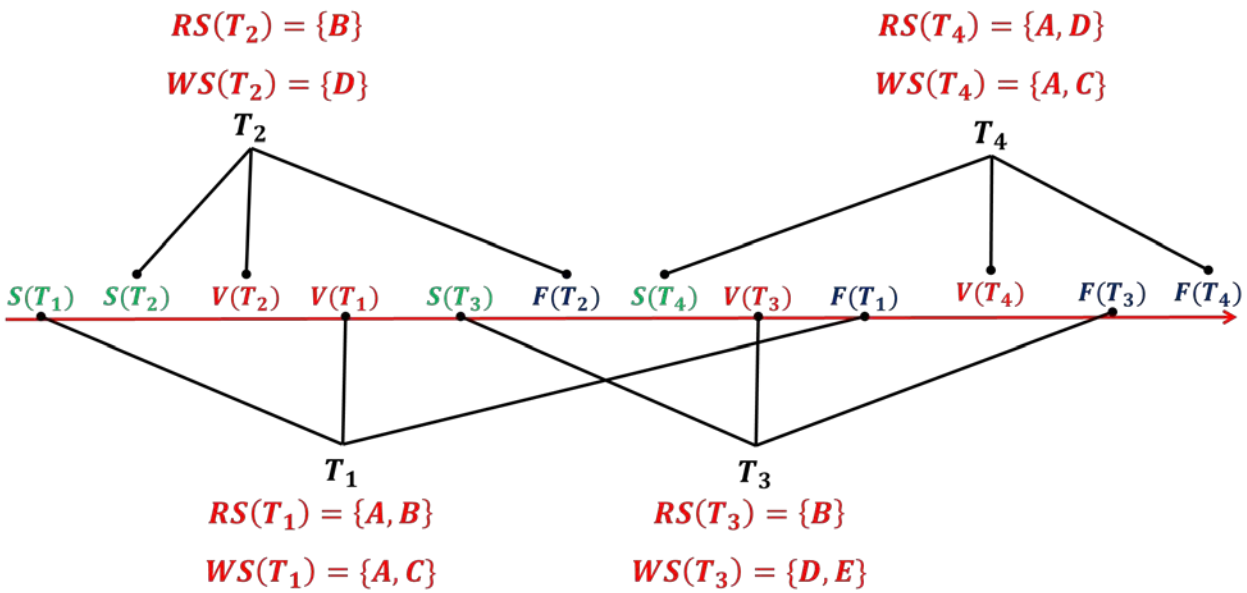
确认T_1：确认T_1时，T_2已确认但尚未执行

$V(T_1) < F(T_2)$，检查$RS(T_1) \cap WS(T_2) = \{A, B\} \cap \{D\} = \Phi$

检查$WS(T_1) \cap WS(T_2) = \{A, C\} \cap \{D\} = \Phi$

T_1确认成功，写入A, B

有效性检查协议的例子



确认T_3：确认T_3时，T_2已完成

T_1已确认但尚未执行

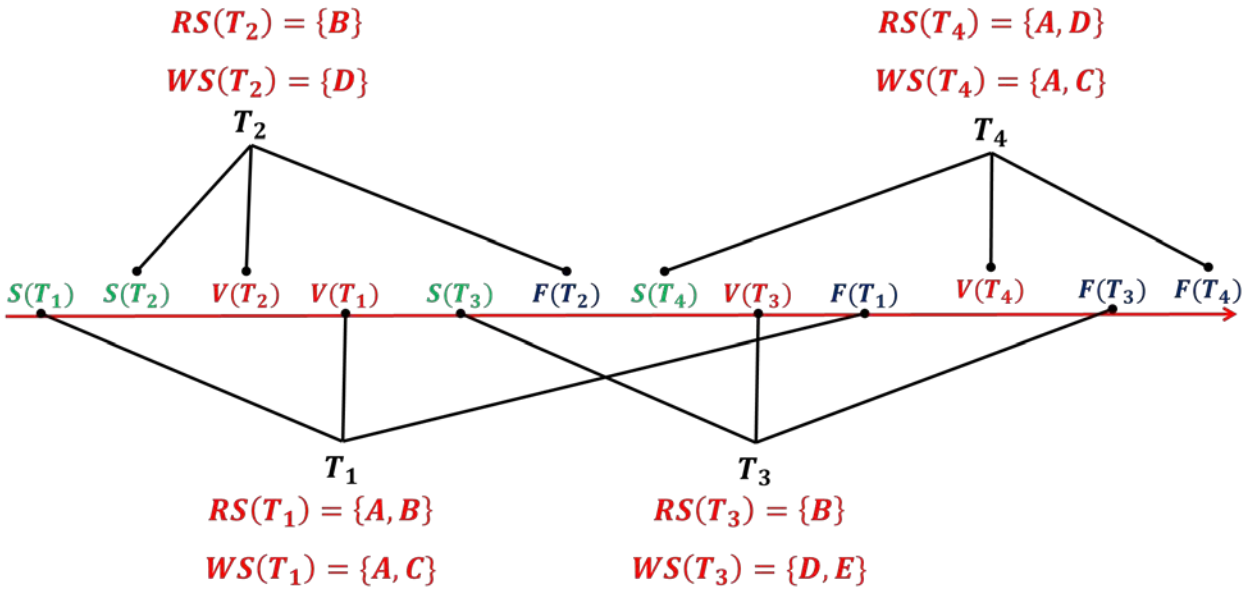
$S(T_3) < F(T_2)$，检查 $RS(T_3) \cap WS(T_2) = \{B\} \cap \{D\} = \Phi$

$V(T_3) < F(T_1)$，检查 $RS(T_3) \cap WS(T_1) = \{B\} \cap \{A, C\} = \Phi$

检查 $WS(T_3) \cap WS(T_1) = \{D, E\} \cap \{A, C\} = \Phi$

T_3确认成功，写入D, E

有效性检查协议的例子



确认T_4：

$S(T_4) > F(T_2)$,无需考虑两者读写集合之间的相交性

$S(T_4) < F(T_1)$，检查 $RS(T_4) \cap WS(T_1) = \{A,D\} \cap \{A,C\} = \{A\}$

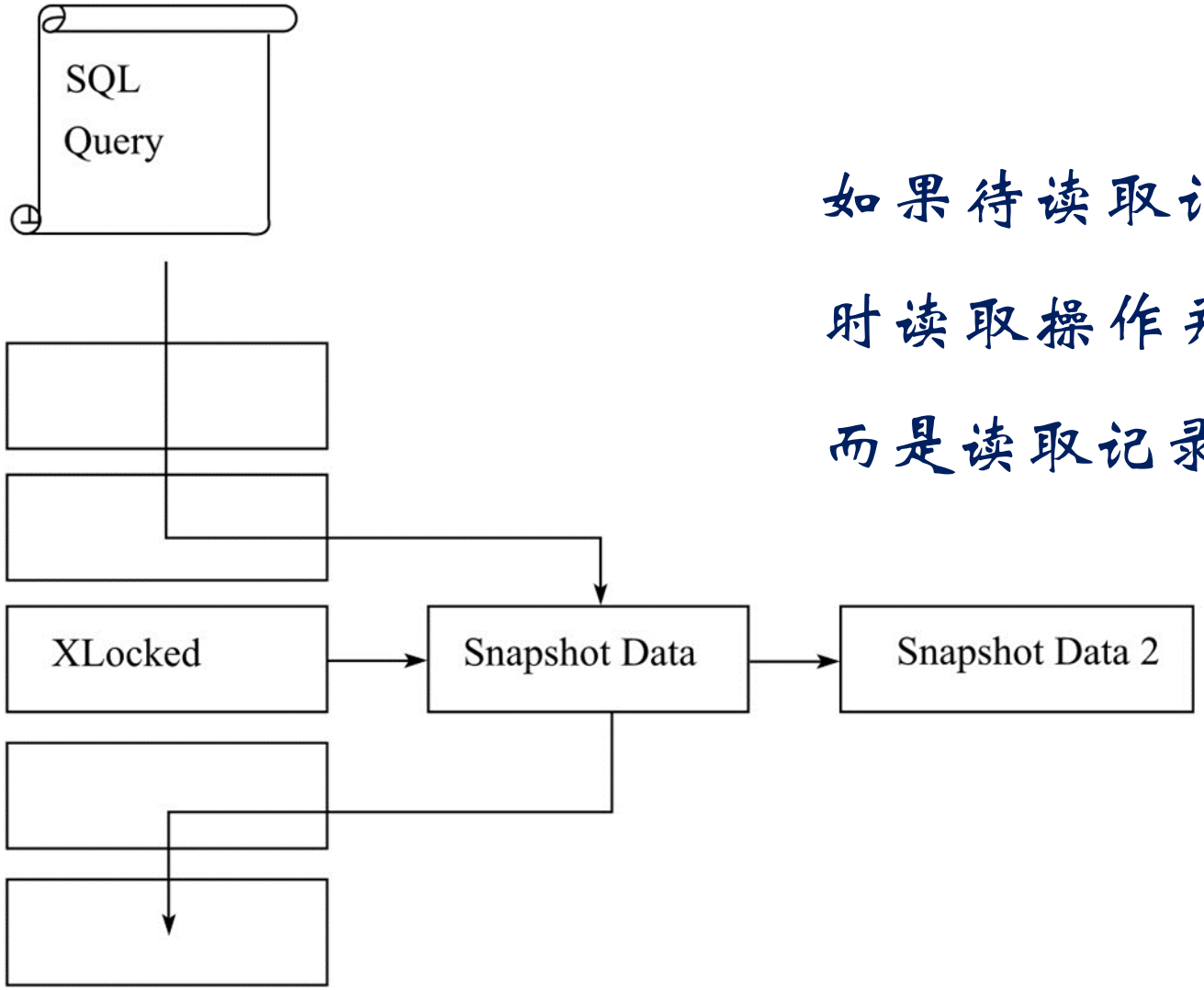
$V(T_3) < F(T_3)$，检查 $RS(T_4) \cap WS(T_3) = \{A,D\} \cap \{D,E\} = \{D\}$

检查 $WS(T_4) \cap WS(T_3) = \{A,C\} \cap \{D,E\} = \Phi$

T_4确认失败


时间戳协议
有效性检查协议
MVCC

MySQL一致性非锁定读（consistent nonlocking read）



如果待读取记录被施加了写锁，这时读取操作并不会等待锁的释放，而是读取记录的某个快照版本

MySQL一致性非锁定读与隔离性级别的关系

- 在read committed和repeatable read下InnoDB使用非锁定的一致性读
- read committed的非一致性读总是读取被锁定行的最新一份快照数据
- repeatable read的非一致性读总是读取事务开始时的行数据版本

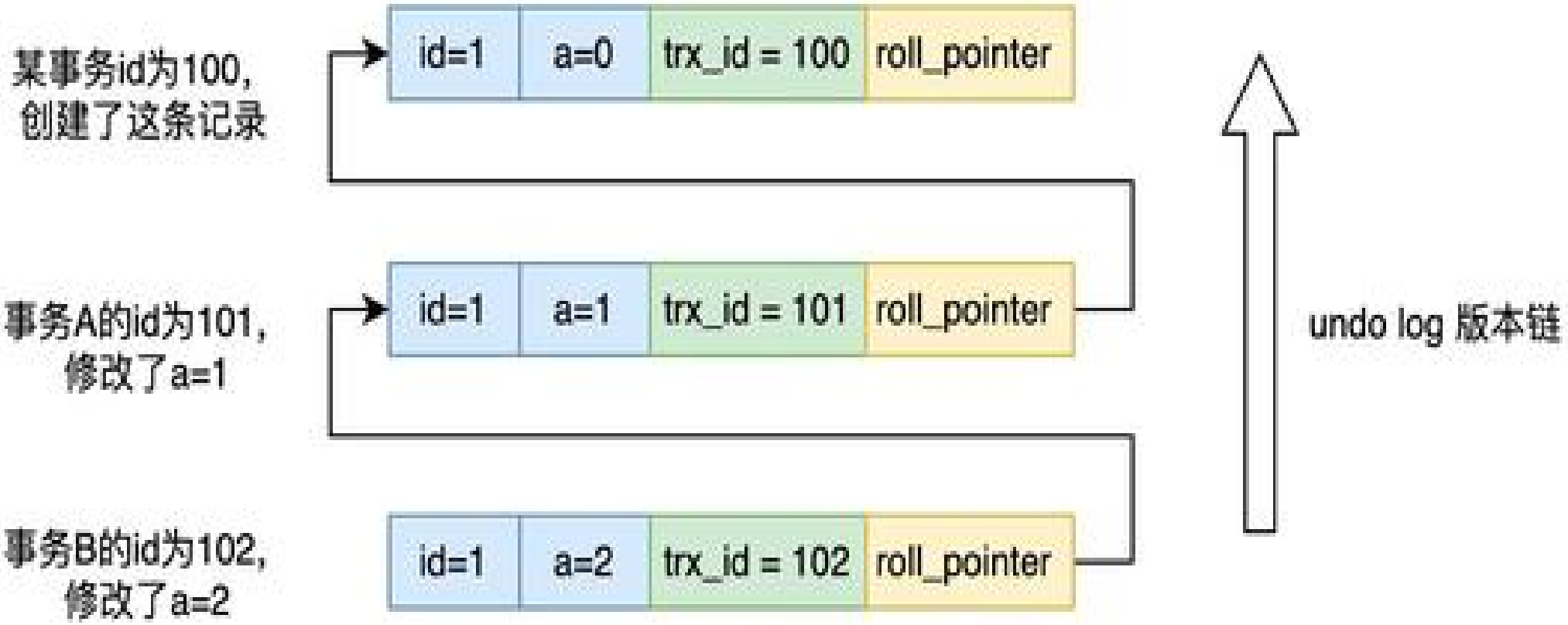
MySQL MVCC实现要点：数据结构

快照存放在日志undo段中

InnoDB为每行数据增加三个隐藏列用于实现MVCC

- db_trx_id：插入或更新行的最后一个事务的全局标识符（每个事务创建都会分配id，全局递增）
- db_roll_ptr：指向当前记录的前一个undo log版本
- db_row_id：行标识（隐藏单调自增id）

MySQL MVCC实现要点：数据结构



InnoDB新版本的数据是叶子结点的值，老版本的数据则通过undo记录存储在回滚段（Rollback Segment）中

MySQL MVCC实现要点：读视图

$read\_view$

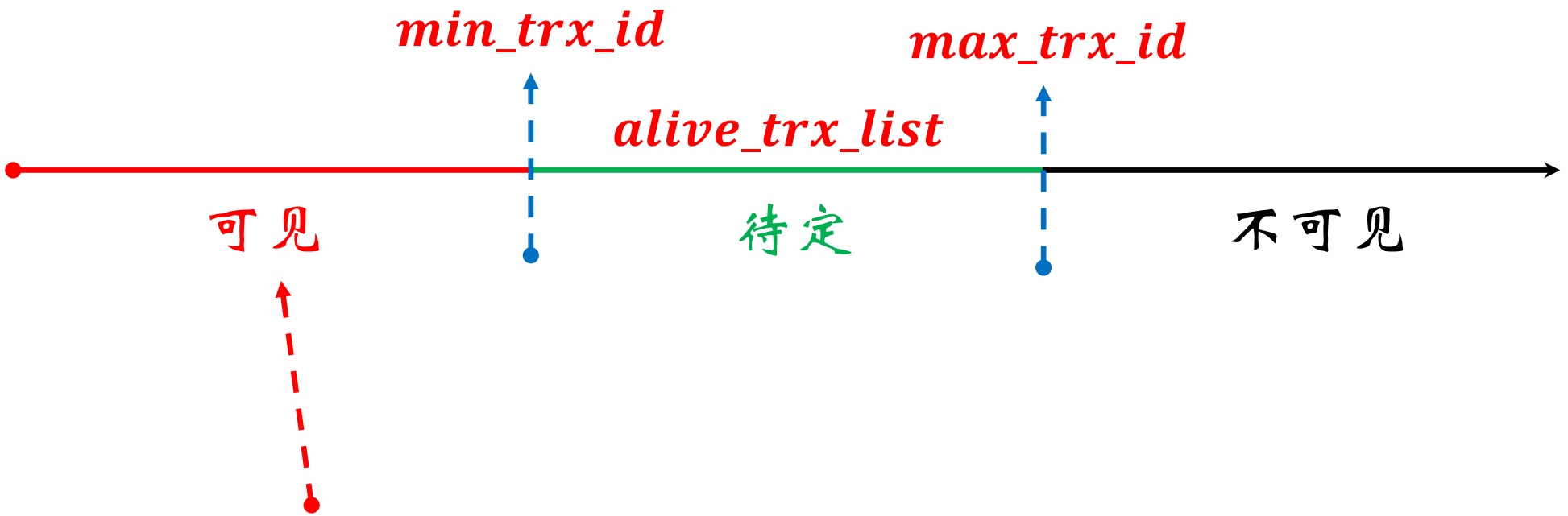
事务在进行快照读的时候会创建一个读视图

- $current\_trx\_id$ ： 当前事务的id
- $alive\_trx\_list$：读视图生成时刻系统中正在活跃的事务id
- min_trx_id：上面的 $alive\_trx\_list$ 中的最小事务id
- max_trx_id ：读视图生成时刻目前已创建过的事务id最大值 + 1

MySQL MVCC实现要点：可见性算法

当一个事务读取某条记录$record_i$时会追溯其undo log版本链，找到第一个可以访问的版本，而该记录的某一个版本$db_trx_id(record_i)$是否能被这个事务读取到遵循如下可见性算法规则：

MySQL MVCC实现要点：可见性算法



if $db\_trx\_id(record_i) < \boldsymbol{min\_trx\_id}$

$then$ $record_i$ 对当前事务可见

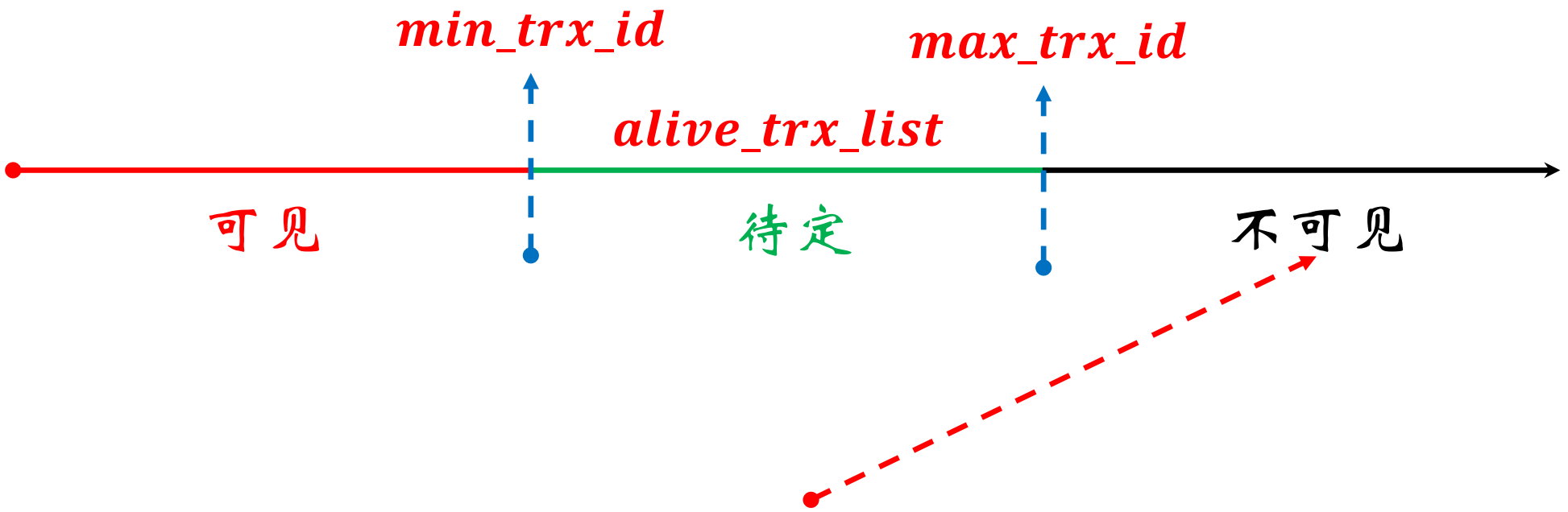
(db_trx_id在当前事务之前提交)

if $db\_trx\_id(record_i) = current\_trx\_id$

$then$ $record_i$ 对当前事务可见

($record_i$是被当前事务生成或修改的)

MySQL MVCC实现要点：可见性算法



$if \quad db\_trx\_id(record_i) \geq \boldsymbol{max\_trx\_id}$

$then\ record_i$ 对当前事务不可见

(db_trx_id在当前事务之后提交)

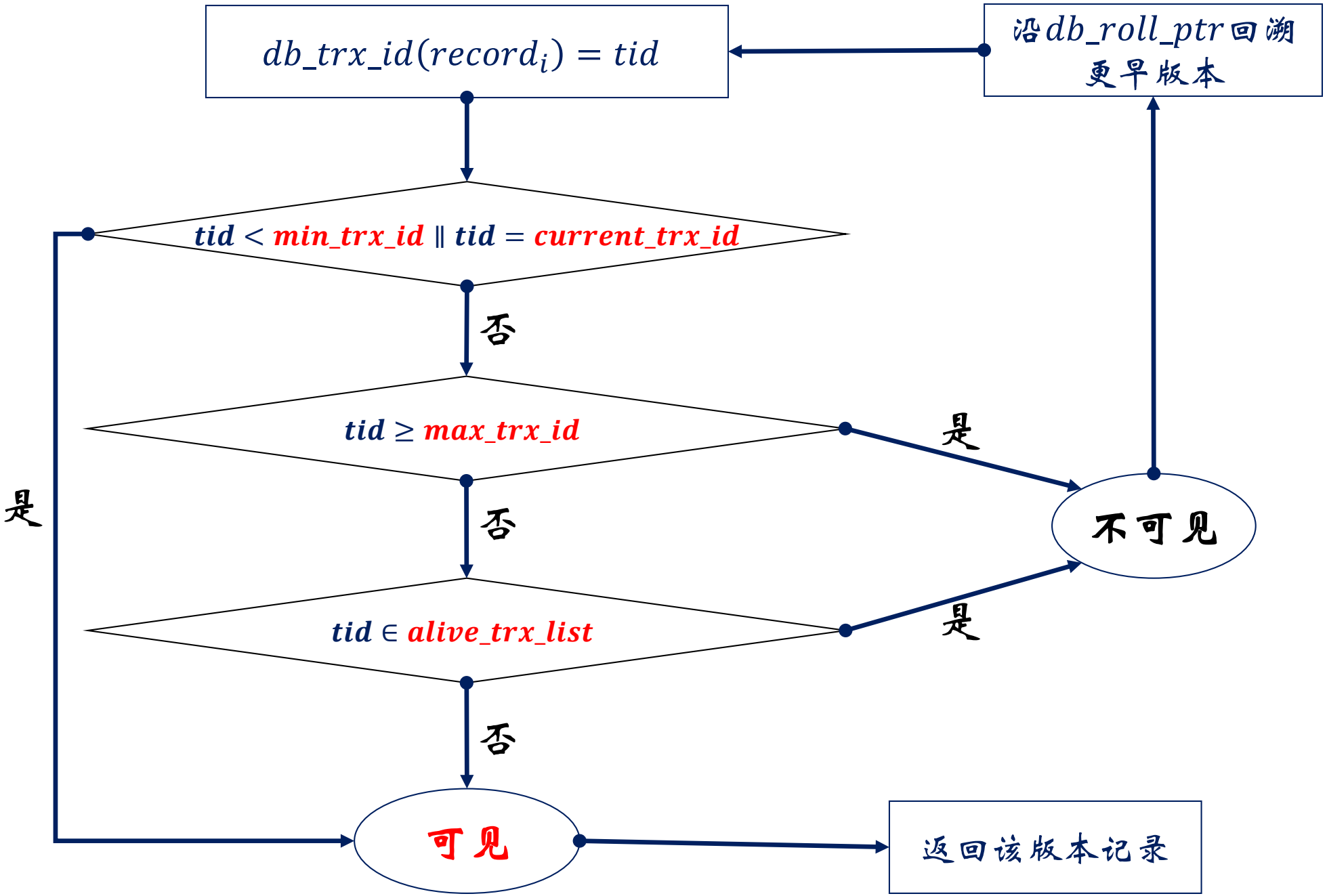
MySQL MVCC实现要点：可见性算法

$$\boldsymbol{min\_trx\_id} \leq db\_trx\_id(record_i) < \boldsymbol{max\_trx\_id}$$

$$if \quad db\_trx\_id(record_i) \in alive\_trx\_list$$

$$then\ record_i \text{ 对当前事务不可见}$$

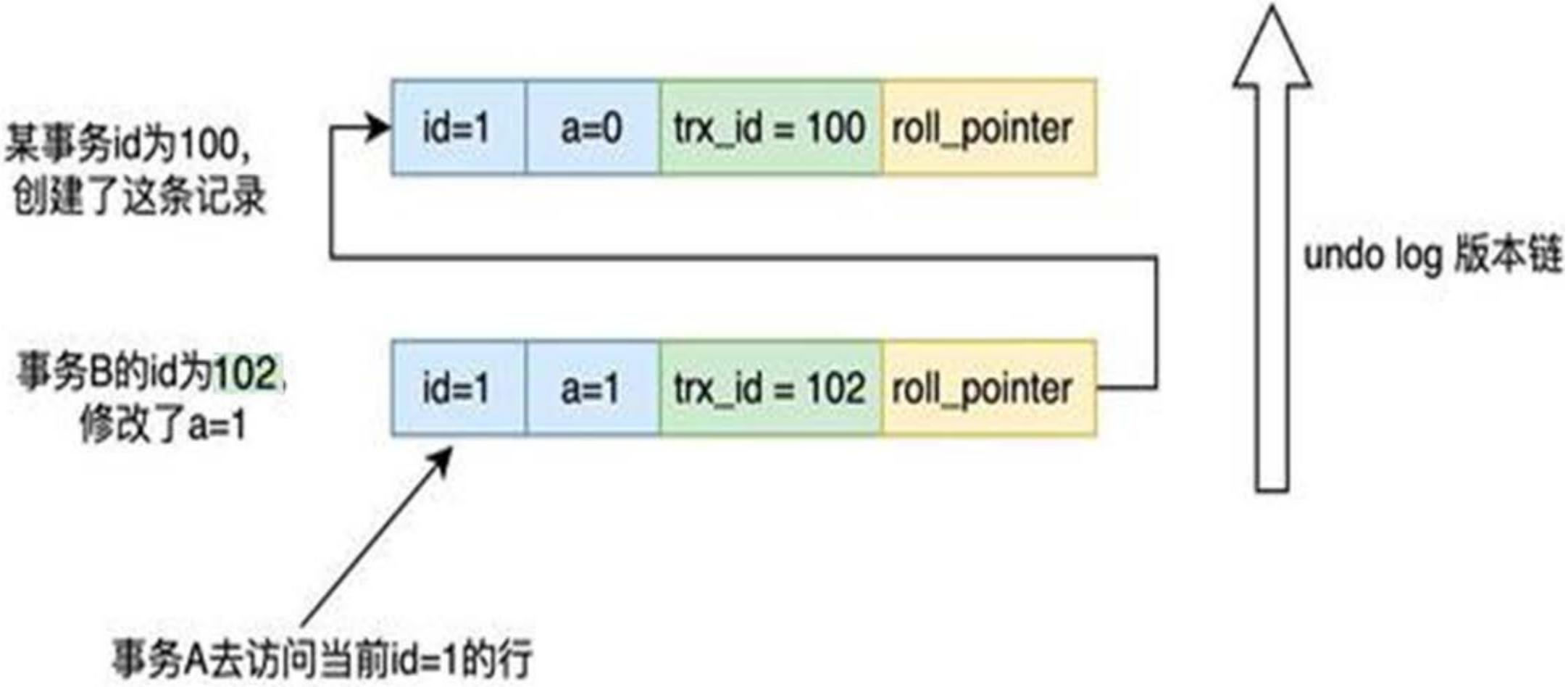
- 属于则说明这条记录还未提交，对于当前操作的事务是不可见的
- 如果不属于则说明已经提交，则是可见的


$db\_trx\_id(record_i) = tid$
沿db_roll_ptr回溯更早版本
$tid < min\_trx\_id \parallel tid = current\_trx\_id$
否
$tid \geq max\_trx\_id$
是
否
$tid \in alive\_trx\_list$
是
否
是
不可见
可见
返回该版本记录

MySQL MVCC在不同隔离性级别下的读视图

T_A	T_B
$begin(trx\_id = 101)$	
	$begin(trx\_id = 102)$
$select\ \ a\ \ \ldots where\ id = 1$	
	$update \ldots set\ a = 1\ where\ id = 1$
	$commit$
$select\ \ a\ \ \ldots where\ id = 1$	
$commit$	

MySQL MVCC在不同隔离性级别下的读视图



假定T_A处于*repeatable read*

T_A第一次读取时的*read_view*

- ***current_trx_id*** : **101**
- ***alive_trx_list*** : **[101, 102]**
- ***min_trx_id*** : **101**
- ***max_trx_id*** : **103**

T_B此时尚未提交

$id = 1$的这一行的$db_trx_id = 100$

$db\_trx\_id(record_{id=1}) < min\_trx\_id$

它对T_A可见，返回a=0

假定T_A处于*repeatable read*

T_A第二次读取时的*read_view*

- *current_trx_id* : **101**
- *alive_trx_list* : **[101, 102]**
- *min_trx_id* : **101**
- *max_trx_id* : **103**

T_B此时已经提交

$id = 1$的这一行的$db_trx_id = 102$

$db\_trx\_id(record_{id=1}) \in$ *alive_trx_list*

它对T_A不可见，继续沿着db_roll_ptr回溯其他版本

记住：*repeatable read*下事务的*read_view*始终保持不变

假定T_A处于*read committed*

T_A第一次读取时的*read_view*

- ***current_trx_id*** ： **101**
- ***alive_trx_list*** ： **[101, 102]**
- ***min_trx_id*** ： **101**
- ***max_trx_id*** ： **103**

T_B此时尚未提交

$id = 1$的这一行的$db_trx_id = 100$

$db\_trx\_id(record_{id=1}) <$ ***min_trx_id***

它对T_A可见，返回a=0

假定T_A处于*read committed*

T_A第二次读取时的*read_view*

- *current_trx_id* : **101**
- *alive_trx_list* : **[101]**
- *min_trx_id* : **101**
- *max_trx_id* : **103**

T_B此时已经提交

$id = 1$的这一行的$db_trx_id = 102$

$db\_trx\_id(record_{id=1}) \notin$ *alive_trx_list*

它对T_A可见，返回a=1

记住：*read committed*下事务每次读取都会生成新的*read_view*

PG与MySQL写操作的不同

- PG对写操作也是乐观并发控制
  - 在表中保存同一行数据记录的多个不同版本，每次写操作，都是创建，而回避更新
  - 在事务提交时，按版本号检查当前事务提交的数据是否存在写冲突，抛异常告知用户，回滚事务

- innodb只对读无锁，写操作仍是上锁的悲观并发控制
  - 每行数据只在表中保留一份，在更新数据时上行锁，同时将旧版数据写入 undo log
  - 表和undo log中行数据都记录着事务ID，检索时只读取来自当前已提交事务的行数据

MVCC的关键设计要点

An Empirical Evaluation of In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

- 并发控制(Concurrency Control)
- 版本存储 (Version Storage)
- 垃圾回收(Garbage Collection)
- 索引管理(Index management)