

西北工业大学

Northwestern Polytechnical University

数据库系统原理

Database System

第十二章 并发控制

2024.11

■ 本章重点

- 事务的概念和性质
- 事务故障、系统故障和介质故障的恢复技术与原理
- Redo & Undo

■ 本章难点

- 具有检查点的恢复技术

12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

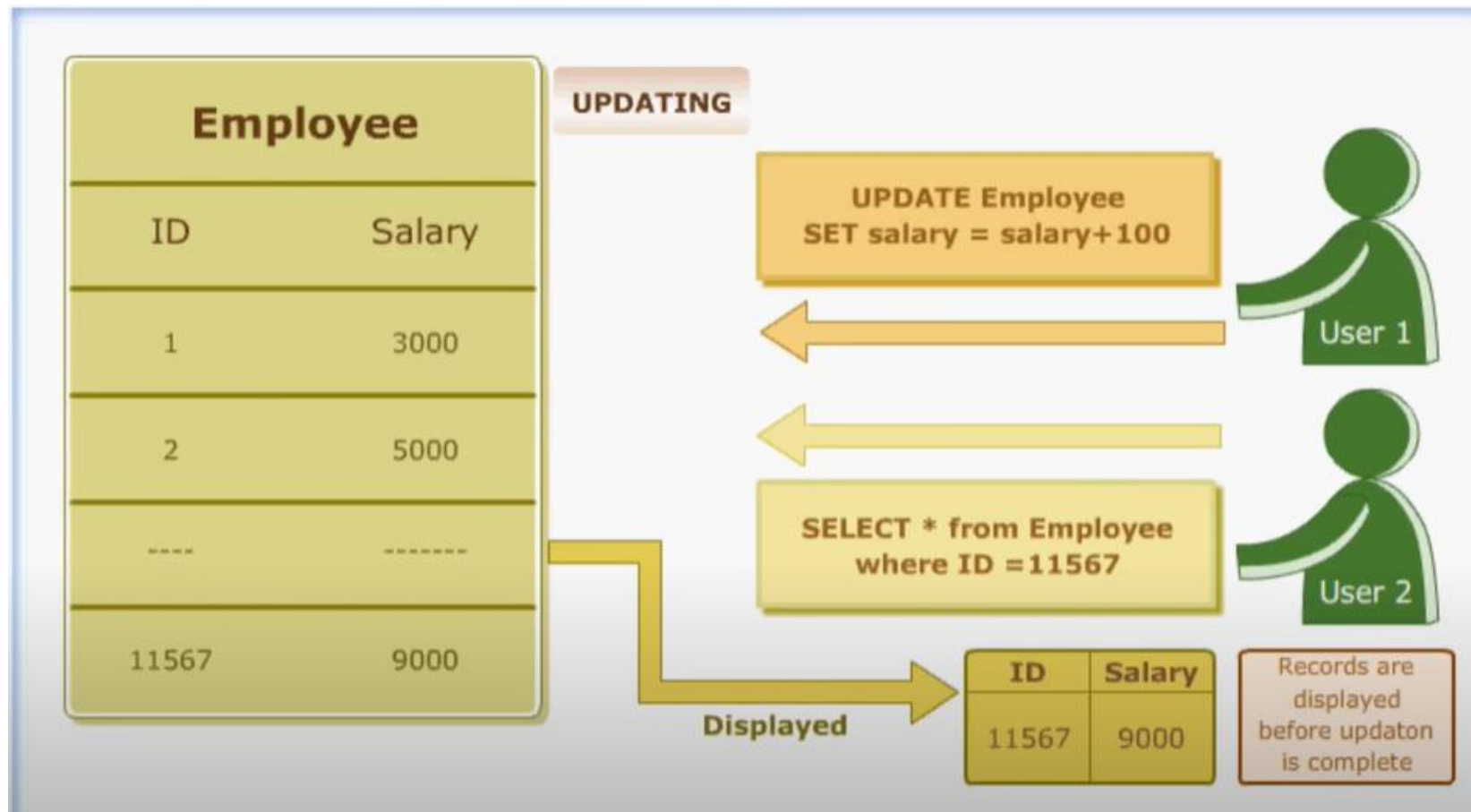
12.8 MVCC



- 事务是并发控制的基本单位
- 并发控制的任務
 - 对并发操作进行正确调度
 - 保证事务的隔离性
 - 保证数据的一致性



12.1 并发控制概述

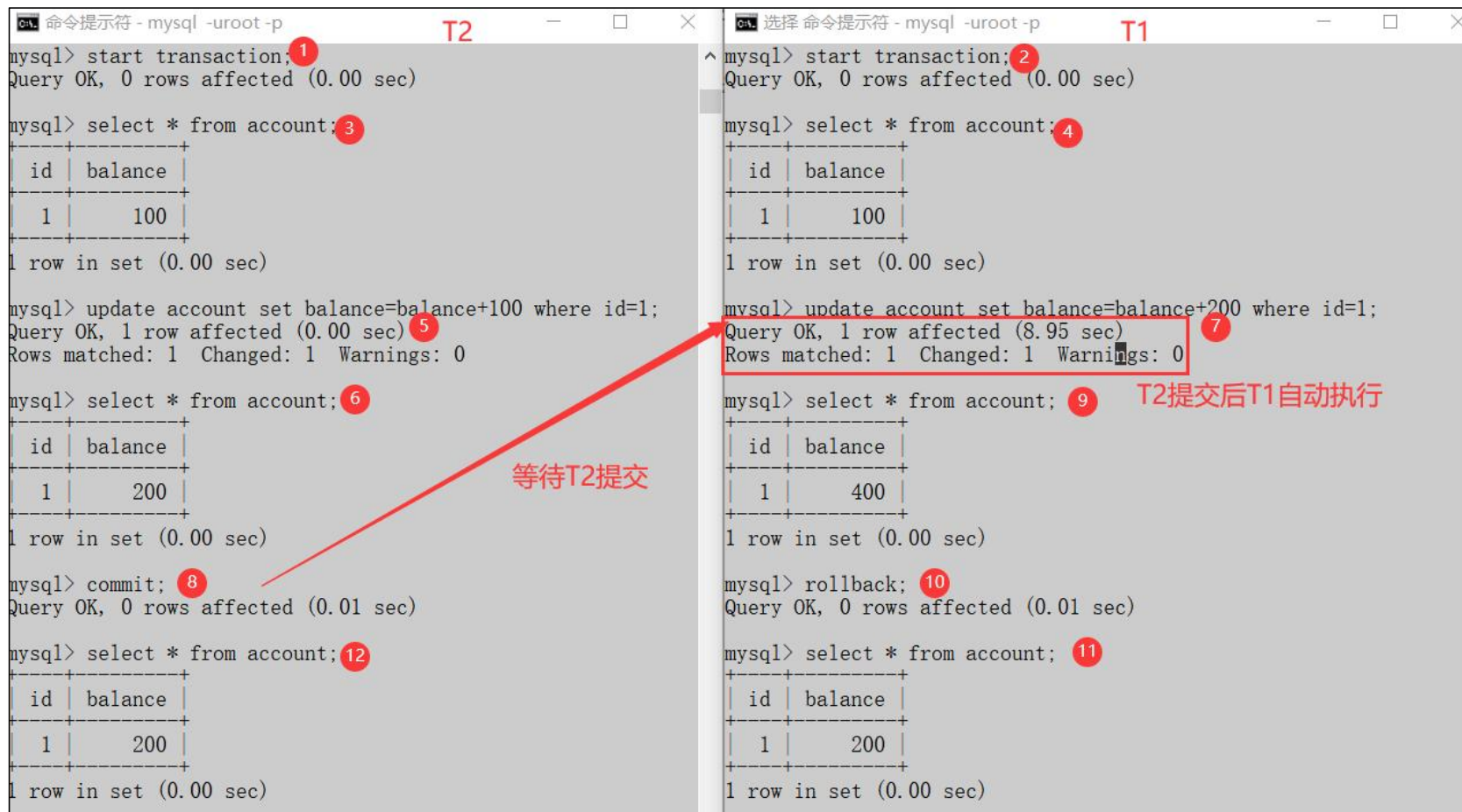


User2: 谁告诉我，为什么本次涨工资没有我？

- 并发操作带来的数据不一致性
 - 丢失修改 (lost update)
 - T2回滚的结果覆盖了T1提交的结果 (第一类错误)
 - T2提交的结果覆盖了T1提交的结果 (第二类错误)
 - 脏数据 (dirty read)
 - T1修改写回, T2读, T1回滚, 此时T2读数据为脏数据
 - 不可重复读 (non-repeatable read)
 - T1读, T2更新, T1再次读 (与第一次读取数据不同)
- 该类错误中还包括: 幻读 (phantom row)
 - T1读后, T2删除/插入部分记录, T1再读时不同

■ 丢失修改 (lost update)

- T1回滚的结果覆盖了T2提交的结果（第一类错误：目前DBMS不出现）



T2 Window:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;
+----+-----+
| id | balance |
+----+-----+
| 1  | 100     |
+----+-----+
1 row in set (0.00 sec)

mysql> update account set balance=balance+100 where id=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> select * from account;
+----+-----+
| id | balance |
+----+-----+
| 1  | 200     |
+----+-----+
1 row in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from account;
+----+-----+
| id | balance |
+----+-----+
| 1  | 200     |
+----+-----+
1 row in set (0.00 sec)
```

T1 Window:

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account;
+----+-----+
| id | balance |
+----+-----+
| 1  | 100     |
+----+-----+
1 row in set (0.00 sec)

mysql> update account set balance=balance+200 where id=1;
Query OK, 1 row affected (8.95 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> select * from account;
+----+-----+
| id | balance |
+----+-----+
| 1  | 400     |
+----+-----+
1 row in set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from account;
+----+-----+
| id | balance |
+----+-----+
| 1  | 200     |
+----+-----+
1 row in set (0.00 sec)
```

等待T2提交

T2提交后T1自动执行

- 丢失修改 (lost update)
 - T2提交的结果覆盖了T1提交的结果（第二类错误）

命令提示符 - mysql -uroot -p

```
mysql> show session variables like 'transaction_isolation';
```

Variable_name	Value
transaction_isolation	READ-UNCOMMITTED

1 row in set, 1 warning (0.00 sec)

```
mysql> start transaction; 1
```

Query OK, 0 rows affected (0.01 sec)

```
mysql> select * from account; 2
```

id	balance
1	120

1 row in set (0.01 sec)

```
mysql> UPDATE account SET balance = 200 WHERE id = 1; 8
```

Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

```
mysql> commit; 9
```

Query OK, 0 rows affected (0.01 sec)

```
mysql> select * from account; 10
```

id	balance
1	200

1 row in set (0.00 sec)

选择 命令提示符 - mysql -uroot -p

```
mysql> show session variables like 'transaction_isolation';
```

Variable_name	Value
transaction_isolation	READ-UNCOMMITTED

1 row in set, 1 warning (0.00 sec)

```
mysql> start transaction; 3
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> select * from account; 4
```

id	balance
1	120

1 row in set (0.00 sec)

```
mysql> UPDATE account SET balance = 50 WHERE id = 1; 5
```

Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

```
mysql> commit; 6
```

Query OK, 0 rows affected (0.01 sec)

```
mysql> select * from account; 7
```

id	balance
1	50

1 row in set (0.00 sec)

■ 脏数据(dirty read)

– T1修改写回，T2读，T1回滚，此时T2读数据为脏数据

T ₁	T ₂
① R(C)=100	
C←C*2	
W(C)=200	
②	R(C)=200
③ ROLLBACK	
C恢复为100	
读“脏”数据	

- T1将C值修改为200， T2读到C为200
- T1由于某种原因撤销，其修改作废， C恢复原值100
- 这时T2读到的C为200，与数据库内容不一致，就是“脏”数据

■ 不可重复读(non-repeatable read)

– T1读, T2更新, T1再次读 (与第一次读取数据不同)

T_1	T_2
① $R(A)=50$	
$R(B)=100$	
求和=150	
②	$R(B)=100$
	$B \leftarrow B * 2$
	$W(B)=200$
③ $R(A)=50$	
$R(B)=200$	
求和=250	
(验算不对)	
不可重复读	

■ T1读取B=100进行运算

■ T2读取同一数据B, 对其进行修改后将B=200写回数据库

■ T1为了对读取值校对重读B, B已为200, 与第一次读取值不一致

- **幻读:** T1读后, T2删除/插入部分记录, T1插入同ID报错

命令提示符 - mysql -uroot -p

```
mysql> select @@transaction_isolation;
+-----+
| @@transaction_isolation |
+-----+
| REPEATABLE-READ         |
+-----+
1 row in set (0.00 sec)

mysql> begin; 1
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account; 3
+----+-----+
| id | balance |
+----+-----+
| 1  |    100  |
+----+-----+
1 row in set (0.01 sec)

mysql> select * from account; 7
+----+-----+
| id | balance |
+----+-----+
| 1  |    100  |
+----+-----+
1 row in set (0.00 sec)

mysql> insert into account values(2,200); 8
ERROR 1062 (23000): Duplicate entry '2' for key 'account.PRIMARY'
mysql> select * from account; 9
+----+-----+
| id | balance |
+----+-----+
| 1  |    100  |
+----+-----+
1 row in set (0.00 sec)
```

T1

命令提示符 - mysql -uroot -p

```
mysql> select @@transaction_isolation;
+-----+
| @@transaction_isolation |
+-----+
| REPEATABLE-READ         |
+-----+
1 row in set (0.00 sec)

mysql> begin; 2
Query OK, 0 rows affected (0.00 sec)

mysql> select * from account; 4
+----+-----+
| id | balance |
+----+-----+
| 1  |    100  |
+----+-----+
1 row in set (0.00 sec)

mysql> insert into account values(2,200); 5
Query OK, 1 row affected (0.01 sec)

mysql> commit; 6
Query OK, 0 rows affected (0.01 sec)

mysql>
```

T2

12.1 并发控制 —— MySQL常用操作



功能	例子
查看当前活动事务	<pre>select * from information_schema.innodb_trx;</pre>
设置/查看是否自动提交	<pre>set session autocommit=0; set @@global.autocommit=0; set @@autocommit=0; set global autocommit=OFF; show global/session variables like 'autocommit'; (省略global/session 修饰符时, 默认为session范围)</pre>
设置/查看全局隔离级别	<pre>set global transaction isolation level repeatable read; show global variables like 'transaction_isolation';</pre>
设置/查看当前连接(session)隔离级别	<pre>set transaction isolation level repeatable read; set session transaction isolation level repeatable read; show variables like 'transaction_isolation'; show session variables like 'transaction_isolation' ; select @@transaction_isolation;</pre>



- 并发控制的主要技术
 - 封锁 (Locking)
 - 多版本并发控制 (MVCC)
 - 时间戳 (Timestamp)
 - 乐观控制法

12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

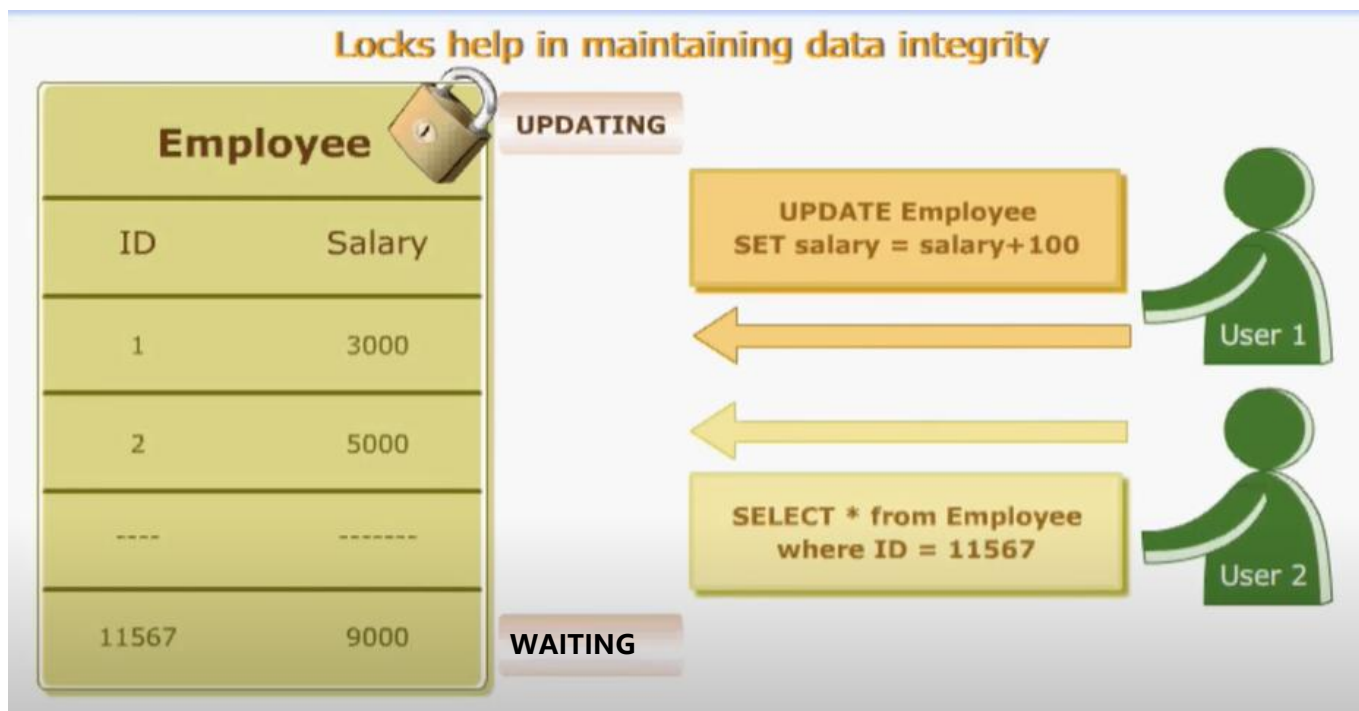
12.7 封锁的粒度

12.8 MVCC



■ 什么是封锁

- 封锁：事务T在对某个数据对象(例如表、记录等)操作之前，先向系统发出请求，**对其加锁**。



■ 基本封锁类型

- 排它锁 (Exclusive lock, 简记为X锁, 又称为写锁)
 - 若事务T对数据对象A加上X锁, 则只允许T读取和修改A, 其它任何事务都不能再对A加任何类型的锁, 直到T释放A上的锁。
- 共享锁 (Share lock, 简记为S锁, 又称为读锁)
 - 若事务T对数据对象A加上S锁, 则其它事务只能再对A加S锁, 而不能加X锁, 直到T释放A上的S锁。

■ 锁的相容矩阵

$T_2 \backslash T_1$	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

Y=Yes, 相容的请求
N=No, 不相容的请求

12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

12.8 MVCC



■ 什么是封锁协议

在运用**X**锁和**S**锁对数据对象加锁时，需要约定一些规则，这些规则为封锁协议（**Locking Protocol**）。

- 何时申请**X**锁或**S**锁
- 持锁时间
- 何时释放

■ 不同的规则所形成的各种不同的封锁协议，它们分别在不同的程度上为并发操作的正确调度提供一定的保证。

- 一级封锁协议
- 二级封锁协议
- 三级封锁协议

■ 一级封锁协议

事务T在**修改**数据R之前必须先对其加X锁，直到事务结束才释放。
事务结束包括**commit**和**rollback**。**注意读数据无需加锁！**

T_1	T_2
① Xlock A	
② $R(A)=16$	
	Xlock A
③ $A \leftarrow A-1$	等待
$W(A)=15$	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	$R(A)=15$
	$A \leftarrow A-1$
⑤	$W(A)=14$
	Commit
	Unlock A

可以解决：

- 修改丢失

无法解决：

- 不可重复读
- 读脏数据

■ 二级封锁协议

一级封锁协议 + 事务**T**在读取数据**R**之前必须先对其加**S**锁，读完后即可释放**S**锁。

T_1	T_2
① Xlock C	
R(C)=100	
$C \leftarrow C * 2$	
W(C)=200	
②	Slock C
	等待
③ ROLLBACK	等待
(C恢复为100)	等待
Unlock C	等待
④	获得Slock C
	R(C)=100
⑤	Unlock C
	Commit C

可以解决：

- 修改丢失
- 读脏数据

无法解决：

- 不可重复读

■ 三级封锁协议

一级封锁协议 + 事务T在读取数据R之前必须先对其加S锁，直到**事务结束**才释放。

T ₁	T ₂
① Slock A	
Slock B	
R(A)=50	
R(B)=100	
求和=150	
②	Xlock B
	等待
③ R(A)=50	等待
R(B)=100	等待
求和=150	等待
Commit	等待
Unlock A	等待
Unlock B	等待
④	获得XlockB
	R(B)=100
	B←B*2
⑤	W(B)=200
	<u>Commit</u>
	Unlock B

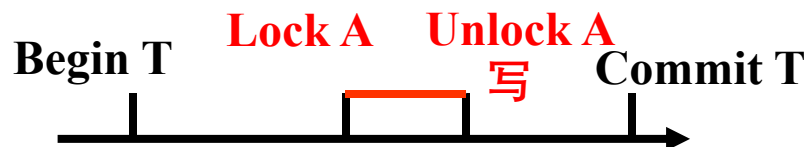
可以解决:

- 修改丢失
- 读脏数据
- 不可重复读

无法解决:

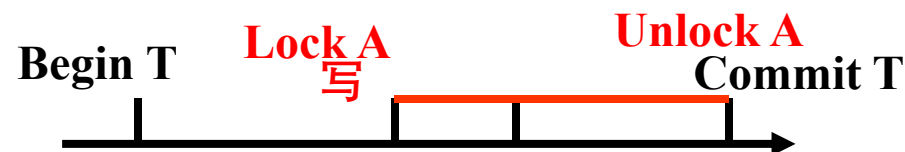
■ 封锁协议的加锁/解锁时机

0级协议



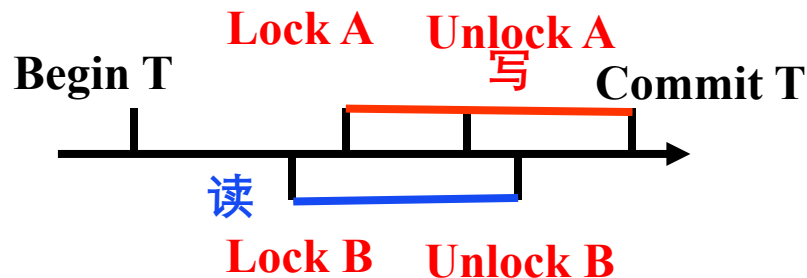
写锁：写完释放；读锁：无

1级协议



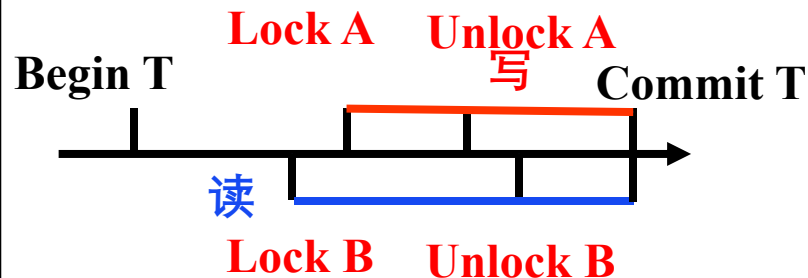
写锁：事务结束释放；读锁：无

2级协议



写锁：事务结束释放
读锁：读完释放

3级协议



写锁：事务结束释放
读锁：事务结束释放

■ 封锁协议小结

- 各级封锁区别：什么操作需要申请封锁以及何时释放锁
- 封锁协议级别越高，一致性程度越高

	X锁		S锁		一致性保证			隔离性级别 保证
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不丢失修改	不“脏”读	可重复读	
一级封锁协议		√			√			读未提交
二级封锁协议		√	√		√	√		读已提交
三级封锁协议		√		√	√	√	√	可串行化

分析示例：
为什么二级封锁协议（**读完释放**）会发生不可重复读？

Transaction 1	Transaction 2	Effect
scan relation R	insert new row into R commit	T_1 locks all rows
scan relation R		T_2 locks new row T_2 's lock released reads new row too!



注意：事务隔离级别
并非越高越好

■ 隔离级别

- 读未提交: Read Uncommitted
- 读已提交: Read Committed
- 可重复读: Repeatable Read
- 可串行化: Serializable

读未提交

读已提交

可重复读

可串行化

数据一致性增强

系统代价增高

默认隔离级别：MySQL是**可重复读**；KingBase、SQL Server、Oracle等为**读已提交**

MySQL常用: `set session transaction isolation level read uncommitted`
`select @@transaction_isolation`

■ 隔离级别与数据不一致关系

表 12.1 事务隔离级别与数据不一致性的关系

事务隔离级别	数据不一致性			
	丢失修改	脏读	不可重复读	* 幻读
读未提交	否	是	是	是
读已提交	否	否	是	是
可重复读	否	否	否	是
可串行化	否	否	否	否

[动画演示](#)



12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

12.8 MVCC

12.9 小结



■ 封锁机制带来的问题

- 死锁
- 活锁



T ₁	T ₂	T ₃	T ₄
Lock R	•	•	•
•	•	•	•
•	Lock R	•	•
•	等待	Lock R	•
Unlock R	等待	•	Lock R
•	等待	•	等待
•	等待	Lock R	等待
•	等待	•	等待
•	等待	Unlock	等待
•	等待	•	等待
•	等待	•	Lock R
•	等待	•	•
•	等待	•	•

(a)活 锁

活锁:

T1、T2、T3、T4相继申请R的锁，可是T2可能永远处于等待之中。。。

解决方案:

依据请求封锁的先后次序对事务排队，首先批准申请队列中第一个事务获得锁。

活锁： T1、T2、T3、T4相继申请R的锁，可是T2可能永远处于等待之中。。。

解决方案1： 依据请求封锁的先后次序对事务排队，首先批准申请队列中第一个事务获得锁。

解决方案2： 排他锁实现方式

1. 将exclusive_bit设置为1；
2. 等待持有共享锁的所有用户解锁完成；
3. 如果第2步无法在超时时间内完成，加锁失败，将exclusive_bit设置为0。

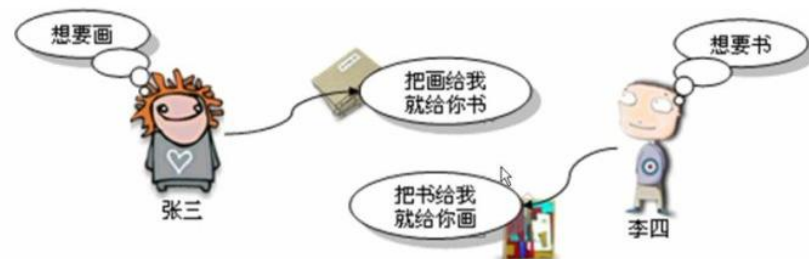
该步执行后，新来的用户无法获取共享锁，就可避免饿死

死锁

T ₁	T ₂
Lock R ₁	•
	•
	•
•	Lock R ₂
•	•
•	•
Lock R ₂	•
等待	
等待	
等待	Lock R ₁
等待	等待
等待	等待
	•
	•
	•

(b) 死锁

T1在等待T2释放R2，而T2又在等待T1释放R1，**T1和T2两个事务永远不能结束，形成死锁。**



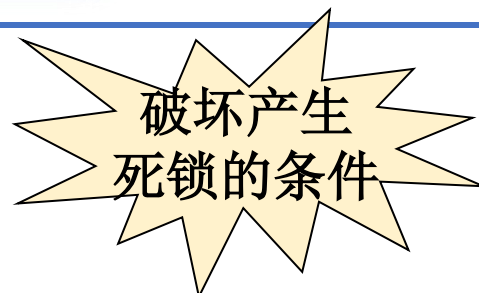
12.4 死锁



序	事务1	事务2
1	begin;	begin;
2	select * from t where id = 1 for update; 此时事务1持有id=1的行锁	select * from t where id = 2 for update; 此时事务2持有id=2的行锁
3	select * from t where id = 2 for update; 此时在等待id=2的行锁释放	
4		select * from t where id = 1 for update; 此时事务2在等待id=1的行锁，与事务1形成循环等待，产生死锁
5		ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

这是mysql返回的，此时事务2会释放id=2的锁和取消id=1的竞争

■ 解决死锁 - 预防死锁发生



1. 一次封锁法

要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行。

如右图：T1一次对R1和R2加锁

引起问题：

- 1) 扩大封锁范围，降低系统并发度。
- 2) 由于数据的动态性，难以事先精确确定需要加锁的数据对象
→ 扩大封锁范围，系统并发度再次降低

T ₁	T ₂
Lock R ₁	•
	•
	•
•	Lock R ₂
•	•
•	•
Lock R ₂	•
等待	
等待	
等待	Lock R ₁
等待	等待
等待	等待
	•
	•
	•

(b) 死锁

■ 解决死锁 - 预防死锁发生


2. 顺序封锁法

预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁。如树结构中，封锁顺序必须是从根节点开始。

引起问题：

1) 维护成本高：

- 需要封锁的数据对象极多，
- 随数据的插入、删除等操作而不断地变化。



死锁的预防
不太适用

2) 难以实现：

事务的封锁请求可以随着事务的执行而动态地决定，很难事先确定每一个事务要封锁哪些对象，因此也就很难**按规定的顺序**去施加封锁。

■ 解决死锁 - 死锁的诊断与解除

1. 超时法

如果一个事务的等待时间超过了规定的时限，就认为发生了死锁。



优点：实现简单

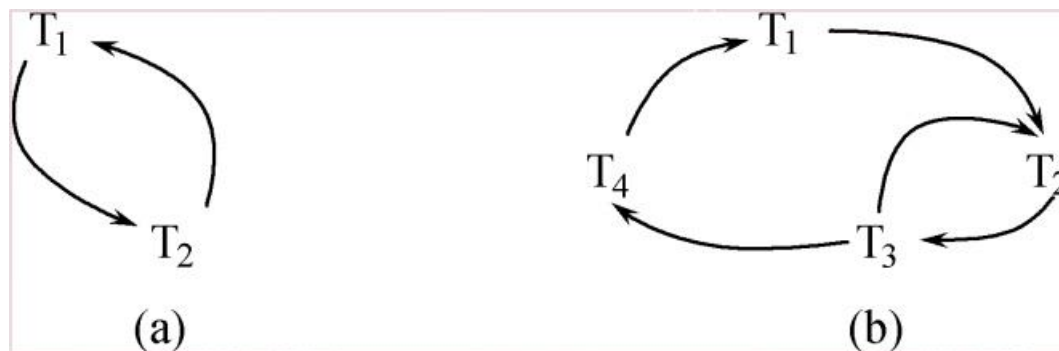
缺点：

- 有可能误判死锁
- 时限若设置得太长，死锁发生后不能及时发现

■ 解决死锁 - 死锁的诊断与解除

2. 等待图法

用事务等待图动态反映所有事务的等待情况。



- 若T1等待T2，则T1， T2之间划一条有向边，从T1指向T2
- 定期检测，如果发现该有向图中存在回路，则认为存在死锁

解除死锁：选择一个处理死锁代价最小的事务，将其撤消，释放其所持有的锁，使其它事务能继续运行下去。

12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

12.8 MVCC

12.9 小结



■ 什么样的并发操作调度是正确的？

- 并行调度是随机的，不同的调度可能会产生不同的结果。
- 将所有事务串行的调度策略一定是正确的调度策略。
- 以不同的顺序串行执行事务可能会产生不同的结果，但由于不会将数据库置于不一致状态，都可认为是正确的。

可串行化（Serializable）的调度策略：
几个事务的并行执行是正确的，当且仅当其结果与按某一次序串行地执行它们时的结果相同。

■ 可串行化调度

- 事务T1: 读B; $A=B+1$; 写回A (A, B初值都为2)
- 事务T2: 读A; $B=A+1$; 写回B

T ₁	T ₂
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
A=Y+1=3	
W(A)	
Unlock A	
	Slock A
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

串行调度(a)

A=3
B=4

✓

T ₁	T ₂
	Slock A
	X=R(A)=2
	Unlock A
	Xlock B
	B=X+1=3
	W(B)
	Unlock B
Slock B	
Y=R(B)=3	
Unlock B	
Xlock A	
A=Y+1=4	
W(A)	
Unlock A	

串行调度(b)

A=4
B=3

✓

不可串行化调度

- 事务T1: 读B; $A=B+1$; 写回A
- 事务T2: 读A; $B=A+1$; 写回B

T ₁	T ₂
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

不可串行化的调度

→ A=3
B=3

✗

T ₁	T ₂
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

可串行化的调度

→ A=3
B=4

✓

■ 冲突可串行化调度：判断可串行化的方法之一

- 一个比可串行化更严格的条件
- 商用系统中的调度器采用

■ 冲突操作：

不同的事务对同一数据的读写操作和写写操作

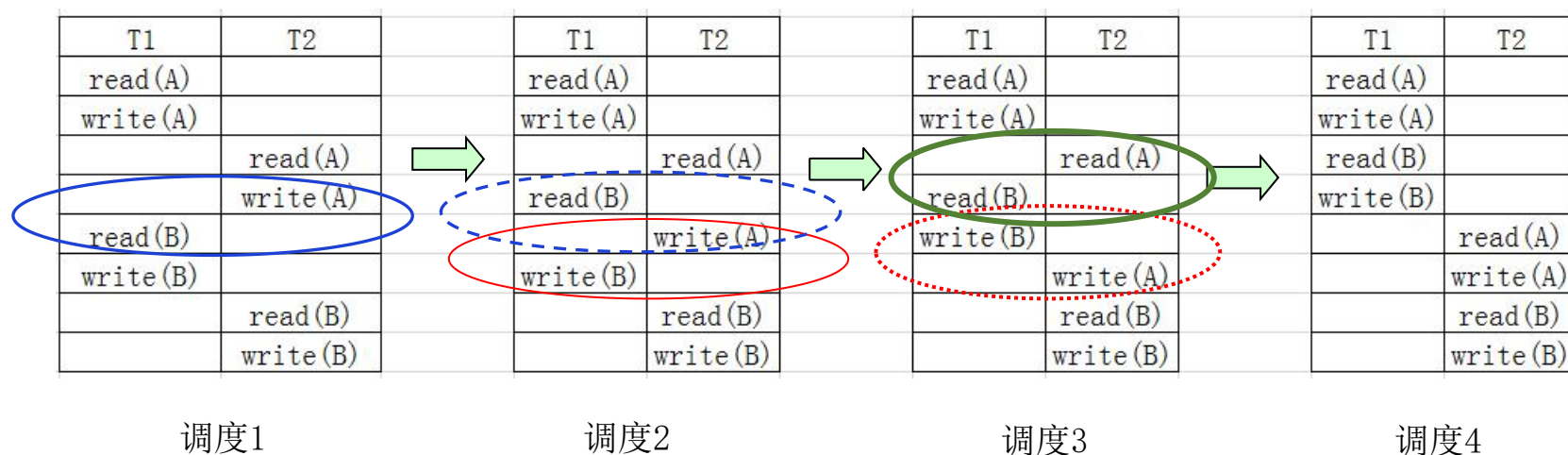
$R_i(x)$ 与 $W_j(x)$ /*事务 T_i 读 x , T_j 写 x , 其中 $i \neq j$ */

$W_i(x)$ 与 $W_j(x)$ /*事务 T_i 写 x , T_j 写 x , 其中 $i \neq j$ */

■ 不同事务的冲突操作是不能交换的

■ 冲突可串行化调度

一个调度 S_c 在保证冲突操作的次序不变的情况下，通过交换两个事务不冲突操作的次序得到另一个调度 S_c' ，如果 S_c' 是串行的，称调度 S_c 是冲突可串行化的调度。

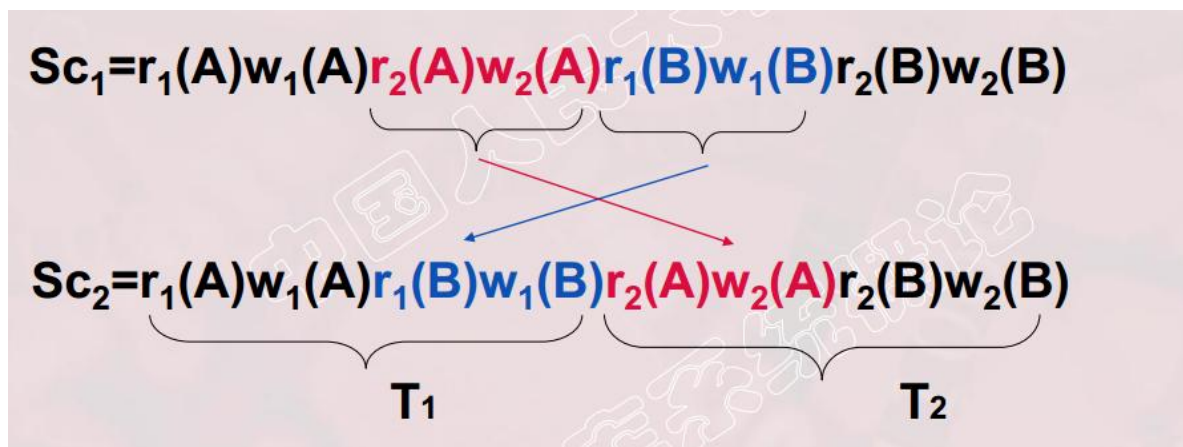


因为调度1等价于一个串行调度 $T1 \rightarrow T2$ ，所以调度1是一个冲突可串行化的调度。

12.5 冲突可串行化调度



- 若一个调度是冲突可串行化，则一定是可串行化的调度。



因为 Sc_2 等价于一个串行调度 $T_1 \rightarrow T_2$ ，
所以 Sc_1 是冲突可串行化的调度。
 Sc_1 也是一个可串行化调度。

调度： $R_1(A) W_1(A) R_2(A) W_2(A) R_2(B) W_2(B) R_1(B) W_1(B)$ 是否可串行化？

这两组均是不能交换的

- 冲突可串行化调度是可串行化调度的充分条件，不是必要条件。还有不满足冲突可串行化条件的可串行化调度
- 例如：有三个事务

$T_1=W_1(Y)W_1(X)$, $T_2=W_2(Y)W_2(X)$, $T_3=W_3(X)$

- 调度 $L_1=W_1(Y)W_1(X)W_2(Y)W_2(X)W_3(X)$ 是一个串行调度。
- 调度 $L_2=W_1(Y)W_2(Y)W_2(X)W_1(X)W_3(X)$ 不满足冲突可串行化。

但是调度 L_2 是可串行化的，因为 L_2 执行的结果与调度 L_1 相同， Y 的值都等于 T_2 的值， X 的值都等于 T_3 的值。

如何保证并发操作调度正确性？

- 封锁方法：两段锁协议
- 乐观方法



12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

12.8 MVCC

12.9 小结





OceanBase数据库大赛初赛（10.17-11.6）阶段，满分（520分）队伍有多少个？

- A 9
- B 19
- C 39
- D 56**

1.basic	11.unique	21.complex-sub-query
2.date	12.null	22.update-mvcc
3.drop-table	13.update-select	23.big-query
4.update	14.expression	24.big-write
5.aggregation-func	15.alias	25.big-order-by
6.like	16.text	
7.join-tables	17.order-by	
8.simple-sub-query	18.group-by	
9.function	19.create-view	
10.multi-index	20.create-table-select	

OceanBase 社区

数据库大赛英雄榜

排名	团队名称	学校名称	成绩	最优成绩提交日
No.1	白天写论文，晚上写bug	西北工业大学	520.000	2023-10-21
No.2	接哈你眼不跟	中国人民大学	520.000	2023-10-23
No.3	你说得对，但是OceanBa...	中国人民大学	520.000	2023-10-23
No.4	神锋无影	华东师范大学	520.000	2023-10-27
No.5	TokTik	四川大学	520.000	2023-10-28
No.6	五点下班	西北工业大学	520.000	2023-10-29

有效提测的同学有平时加分！

提交

■ “两段”锁的含义（Two-Phase Locking, 简称2PL）

事务的两个阶段

1. 第一阶段是获得封锁，也称为扩展阶段；

事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁

2. 第二阶段是释放封锁，也称为收缩阶段。

事务可以释放任何数据项上的任何类型的锁，但是不能再申请任何锁。

遵守2PL



Slock A	Slock B	Xlock C	Unlock B	Unlock A	Unlock C;
←	扩展阶段	→	←	收缩阶段	→

不遵守2PL



Slock A	Unlock A	Slock B	Xlock C	Unlock C	Unlock B;
---------	----------	---------	---------	----------	-----------

■ 结论：

若并行执行的所有事务均遵守两段锁协议，则对这些事务的所有并行调度策略都是可串行化的。即：

(1) 所有遵守两段锁协议的事务，其并行执行的结果一定是正确的。

(2) 事务遵守两段锁协议是可串行化调度的充分条件，而不是必要条件(即可串行化的调度中，不一定所有事务都必须符合两段锁协议)。

试举例：是可串行化的调度，但不符合两段锁协议！

12.6 两段锁协议



事务T ₁	事务T ₂
Slock A R(A)=260	
	Slock C R(C)=300
Xlock A W(A)=160	
	Xlock C W(C)=250
Slock B R(B)=1000 Xlock B W(B)=1100 Unlock A	Slock A 等待 等待 等待 等待 等待 R(A)=160 Xlock A
Unlock B	W(A)=210 Unlock C
遵守两段锁协议的可串行化调度	

左图的调度是遵守两段锁协议的，因此一定是一个可串行化调度。

■ 两段锁协议与防止死锁的一次封锁法的比较:

- 一次封锁法要求每个事务**必须一次将所有要使用的数据全部加锁**，否则就不能继续执行，因此一次封锁法遵守两段锁协议；
- 但是**两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁**，因此遵守两段锁协议的事务可能发生死锁。

例：遵守两段锁协议的事务发生死锁

T_1	T_2
Slock B $R(B)=2$	Slock A $R(A)=2$
Xlock A 等待 等待	
	Xlock B 等待

■ 两段锁协议与三级封锁协议的关系：

- 两段锁协议： **封锁的顺序** => 确保调度可串行
- 三级封锁协议： **封锁的类型和时间**

例如： 下图满足两段锁协议的并发事务T1和T2，
可以满足：

一级： 只对写加锁

二级： 写锁+读锁（读完就释放）

三级： 写锁+读锁（事务完释放）

T1	T2
XLock A	
XLock B	
	XLock A
	wait
	wait
Unlock B	wait
Unlock A	wait
	...
	Unlock A

12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

12.8 MVCC

12.9 小结



■ 封锁粒度 (Granularity)

- X锁和S锁都是加在某一个数据对象上
- 封锁粒度即指封锁对象的大小

封锁的对象

- 逻辑单元
- 物理单元

例：在关系数据库中，封锁对象可以是：

- 逻辑单元：属性值、属性值的集合、元组、关系、索引项、整个索引、整个数据库等
- 物理单元：页（数据页或索引页）、物理记录等

■ 封锁粒度的关系：

比较对象	变化趋势
封锁的粒度	大 → 小
系统被封锁的对象个数	少 → 多
并发度	小 → 高
系统开销	小 → 大

■ 选择封锁粒度：

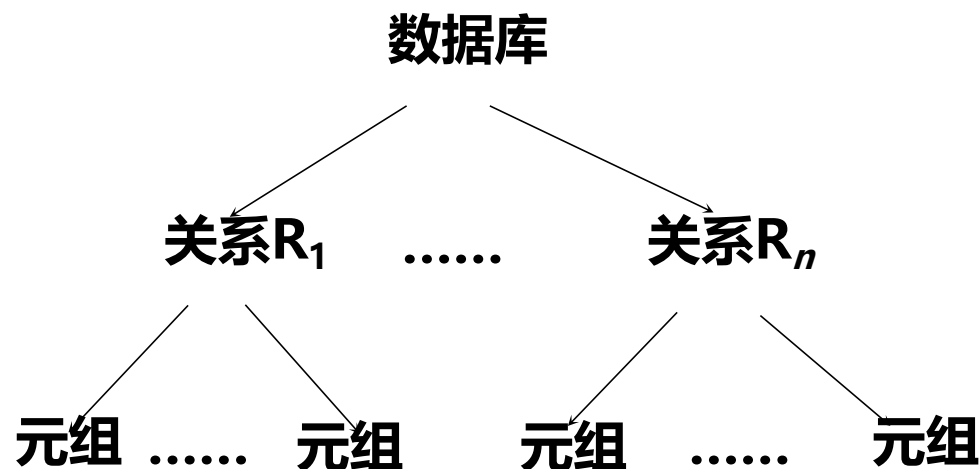
考虑**封锁对象**和**并发度**两个因素, 对系统开销与并发度进行权衡。

- 需要处理多个关系的大量元组的用户事务：以数据库为封锁单位；
- 需要处理大量元组的用户事务：以关系为封锁单元；
- 只处理少量元组的用户事务：以元组为封锁单位

■ 多粒度封锁 (Multiple Granularity Locking):

在一个系统中同时支持多种封锁粒度供不同的事务选择

- **多粒度树**: 以树形结构来表示多级封锁粒度, 根结点是整个数据库, 表示最大的数据粒度; 叶结点表示最小的数据粒度。



■ 多粒度封锁协议：

- 允许多粒度树中的每个结点被独立地加锁；
- 对一个结点加锁意味着这个结点的所有后裔结点也被加以同样类型的锁；
- 在多粒度封锁中一个数据对象可能是显式或者隐式封锁
 - 显式封锁：直接加到数据对象上的封锁
 - 隐式封锁：该数据对象没有独立加锁，是由于其上级结点加锁而使该数据对象加上了锁

■ 对某个数据对象加锁，系统要检查：

- 该数据对象

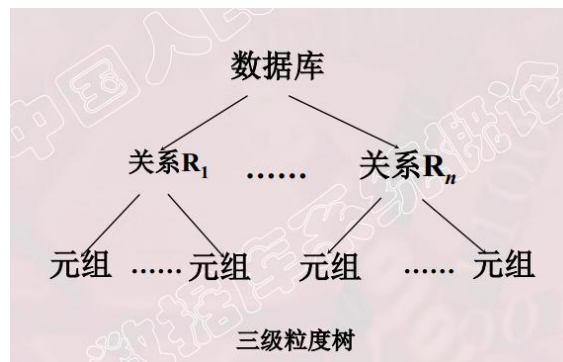
- ✓ 有无显式封锁与之冲突

- 所有上级结点

- ✓ 检查本事务的显式封锁是否与该数据对象上的**隐式封锁**冲突（由上级结点已加的封锁造成的）

- 所有下级结点

- ✓ 看上面的**显式封锁**是否与本事务的隐式封锁（将加到下级结点的封锁）冲突



如此检查，效率太低！

⇒ 意向锁 (intention lock)

■ 意向锁（intention lock）：

- 如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁
- 对任一结点加基本锁，必须先对它的上层结点加意向锁

如：对任一元组加锁，必须先对它所在的数据库和关系加意向锁

■ 常用意向锁

- 意向共享锁(Intent Share Lock, 简称IS锁)
- 意向排它锁(Intent Exclusive Lock, 简称IX锁)
- 共享意向排它锁(Share Intent Exclusive Lock, 简称SIX锁)

■ 意向共享锁(Intent Share Lock, 简称IS锁)

- 表示它的后裔结点拟加S锁

例如：事务T1要对数据库D中的关系R中的某个元组加S锁，
则要首先对关系R和数据库D加IS锁

■ 意向排它锁(Intent Exclusive Lock, 简称IX锁)

- 表示它的后裔结点拟加X锁

例如：事务T1要对数据库D中的关系R中的某个元组加X锁，
则要首先对关系R和数据库D加IX锁

■ 共享意向排它锁(Share Intent Exclusive Lock, 简称SIX锁)

- 如果对一个数据对象加SIX锁，表示对它加S锁，再加IX锁，
即 $SIX = S + IX$

例如：对某个表加SIX锁，则表示该事务要：

- 读整个表（所以要对该表加S锁）
- 同时会更新个别元组（所以要对该表加IX锁）

■ 意向锁相容矩阵

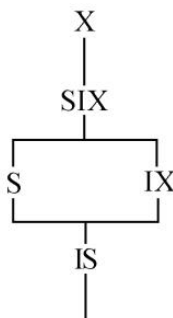
$T_1 \backslash T_2$	S	X	IS	IX	SIX	-
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
-	Y	Y	Y	Y	Y	Y

Y=Yes, 表示相容的请求 N=No, 表示不相容的请求

(a) 数据锁的相容矩阵

- 锁的强度： 锁的强度是指它对其他锁的排斥程度

锁强度
偏序关系



一个事务在申请封锁时
以强锁代替弱锁是安全
的，反之则不然！

- 具有意向锁的多粒度封锁方法（广泛应用在DBMS中）

- 申请封锁：自上而下
- 释放封锁：自下而上

例：事务T1要对关系R1加S锁

1. 要首先对上级节点数据库加IS锁
2. 检查上级节点数据库和自身R1是否已加了不相容的锁(X或IX)
3. **不再**需要搜索和检查下级节点R1中的**元组**是否加了不相容的锁(X锁)

■MySQL支持的锁机制

引擎	表锁	页锁	行锁
InnoDB	✓	×	✓
MyISAM	✓	×	×
BDB	✓	✓	×

BDB: BerkeleyDB存储引擎

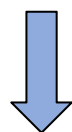
- 表锁：一次锁整个表
- 行锁：一次锁相关行
- 页锁：一次锁一个物理页面

比较项目	表锁	页锁	行锁
加锁粒度	大	中	小
开销	小	中	大
加锁速度	快	中	慢
有无死锁	无	有	有
并发度	低	一般	高

■MySQL中的表锁使用

```
1 # 获取表锁
2 LOCK TABLES
3     tbl_name [[AS] alias] lock_type
4     [, tbl_name [[AS] alias] lock_type] ...
5
6 lock_type:
7     READ [LOCAL]
8     | [LOW_PRIORITY] WRITE
9
10 # 释放表锁
11 UNLOCK TABLES
```

MyISAM: 在执行查询前, 会自动执行表的加锁、解锁操作。仅在需要手动控制的时候, 使用该命令。



使用例子

```
LOCK TABLES t1 read, t2 read;
select count(t1.id1) as 'count' from t1;
select count(t2.id1) as 'count' from t2;
UNLOCK TABLES;
```

■ InnoDB的锁机制

	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

IS：意向共享锁。

IX：意向排他锁。

意向锁：不会阻塞全表扫描之外的任何请求，主要目的是为了表示是否有人请求锁定表中的某一行数据。

■查看锁的方式:

- `show engine innodb status;`
- MySQL 8.0 的系统表: `innodb_trx`, `data_locks`, `innodb_lock_waits`.
`select * from information_schema.innodb_trx`
`select * from performance_schema.data_locks`
`select * from sys.innodb_lock_waits;`

information_schema.innodb_trx表 (事务信息表, `trx_id`)

字段名	说明
<code>trx_id</code>	InnoDB 存储引擎内部唯一的事务 ID
<code>trx_state</code>	当前事务的状态
<code>trx_started</code>	事务的开始时间
<code>trx_requested_lock_id</code>	等待事务的锁 ID。如 <code>trx_state</code> 的状态为 LOCK WAIT, 那么该值代表当前的事务等待之前事务占用锁资源的 ID。若 <code>trx_state</code> 不是 LOCK WAIT, 则该值为 NULL
<code>trx_wait_started</code>	事务等待开始的时间
<code>trx_weight</code>	事务的权重, 反映了一个事务修改和锁住的行数。在 InnoDB 存储引擎中, 当发生死锁需要回滚时, InnoDB 存储引擎会选择该值最小的进行回滚
<code>trx_mysql_thread_id</code>	MySQL 中的线程 ID, SHOW PROCESSLIST 显示的结果
<code>trx_query</code>	事务运行的 SQL 语句

■查看锁的方式:

sys.innodb_lock_waits表

字段	说明	字段	说明
requesting_trx_id	申请锁资源的事务 ID	blocking_trx_id	阻塞的事务 ID
requesting_lock_id	申请的锁的 ID	blocking_lock_id	阻塞的锁的 ID

performance_schema.data_locks表

```
mysql> desc performance_schema.data_locks;
```

Field	Type	Null	Key	Default	Extra
ENGINE	varchar(32)	NO	PRI	NULL	
ENGINE_LOCK_ID	varchar(128)	NO	PRI	NULL	
ENGINE_TRANSACTION_ID	bigint unsigned	YES	MUL	NULL	
THREAD_ID	bigint unsigned	YES	MUL	NULL	
EVENT_ID	bigint unsigned	YES		NULL	
OBJECT_SCHEMA	varchar(64)	YES	MUL	NULL	
OBJECT_NAME	varchar(64)	YES		NULL	
PARTITION_NAME	varchar(64)	YES		NULL	
SUBPARTITION_NAME	varchar(64)	YES		NULL	
INDEX_NAME	varchar(64)	YES		NULL	
OBJECT_INSTANCE_BEGIN	bigint unsigned	NO		NULL	
LOCK_TYPE	varchar(32)	NO		NULL	
LOCK_MODE	varchar(32)	NO		NULL	
LOCK_STATUS	varchar(32)	NO		NULL	
LOCK_DATA	varchar(8192)	YES		NULL	

LOCK_TYPE:record/table

LOCK_MODE:S[, GAP], X[, GAP], I
S[, GAP], IX[, GAP], AUTO_INC,
UNKNOWN

LOCK_STATUS:GRANTED/WAITING

LOCK_DATA:与锁关联的数据
(表锁时, 该值为NULL)。

■ InnoDB加锁方式

- 意向锁：自动加，不需用户干预。
- UPDATE、DELETE和INSERT语句：自动给涉及数据加排他锁（X）；
- 普通SELECT 语句：不加任何锁
- DDL语句：自动加表锁

可以通过以下语句显式加共享锁或排他锁：

- 共享锁（S）：

```
SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE。
```

其他session仍然可以查询记录，也可对该记录加share mode锁。但是当前事务需要对该记录进行更新操作，则很有可能造成死锁。

- 排他锁（X）：

```
SELECT * FROM table_name WHERE ... FOR UPDATE
```

其他session可以查询该记录，但是不能对该记录加共享锁或排他锁，而是等待获得锁。

■ InnoDB支持的锁

锁类别	描述
Shared and Exclusive Locks	共享锁/排他锁
Intention Locks	意向共享锁/意向排他锁
Record Locks	行锁
Gap Locks	间隙锁
Next-Key Locks	临键锁
Insert Intention Locks	插入意向锁
AUTO-INC Locks	自增锁
Predicate Locks for Spatial Indexes	空间索引使用，本处暂不讨论

■ InnoDB的行锁

- **Record Lock**: 单个行记录上的锁

Record Lock总是锁住索引记录，若没有索引表会使用隐式的主键来进行锁定。

- **Gap Lock**: 间隙锁，锁定一个范围，不包含记录本身。
在索引记录间隙上的锁，或者是第一条索引记录之前、最后一条索引记录之后上的间隙锁。

- **Next-Key Lock**: Gap Lock + Record Lock
锁定一个范围，并且锁定记录本身。
前开后闭区间，如(5, 11】。

■ InnoDB: 间隙锁

```
create table t (  
  id int(11) not null,  
  c int(11),  
  d int(11),  
  primary key (id),  
  key c(c) )  
ENGINE = InnoDB;
```

```
insert into t values  
  (0, 0, 0),  
  (5, 5, 5),  
  (15, 15, 15);
```

间隙锁冲突：是“往这个间隙中插入一个记录”。

0

5

15

$(-\infty, 0)$

$(0, 5)$

$(5, 15)$

$(15, +\infty)$

```
select * from t where c=5 for update;
```

限制插入c值=5的左右两个间隙锁范围内的值：[0, 15]

12.7.3 MySQL中的并发控制

■InnoDB: 间隙锁

primary key (id),
key c(c))
ENGINE = InnoDB;

insert into
t(id, c, d)
values
(0, 0, 0),
(5, 5, 5),
(15, 15, 15);

begin; T1
select * from t where c = 5 for update;

```
mysql> select * from information_schema.innodb_trx\G
***** 1 row *****
      trx_id: 4729
      trx_state: RUNNING
```

```
mysql> select * from performance_schema.data_locks\G
***** 1. row *****
      ENGINE: INNODB
ENGINE_LOCK_ID: 1261878693768:1161:1261843426456
ENGINE_TRANSACTION_ID: 4729
      THREAD_ID: 62
      EVENT_ID: 36
OBJECT_SCHEMA: test
OBJECT_NAME: t
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: NULL
OBJECT_INSTANCE_BEGIN: 1261843426456
      LOCK_TYPE: TABLE
      LOCK_MODE: IX
      LOCK_STATUS: GRANTED
      LOCK_DATA: NULL
***** 2. row *****
      ENGINE: INNODB
ENGINE_LOCK_ID: 1261878693768:100:5:3:1261843423672
ENGINE_TRANSACTION_ID: 4729
      THREAD_ID: 62
      EVENT_ID: 36
OBJECT_SCHEMA: test
OBJECT_NAME: t
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: c
OBJECT_INSTANCE_BEGIN: 1261843423672
      LOCK_TYPE: RECORD
      LOCK_MODE: X
      LOCK_STATUS: GRANTED
      LOCK_DATA: 5, 5
```

```
***** 3. row *****
      ENGINE: INNODB
ENGINE_LOCK_ID: 1261878693768:100:4:3:1261843424016
ENGINE_TRANSACTION_ID: 4729
      THREAD_ID: 62
      EVENT_ID: 36
OBJECT_SCHEMA: test
OBJECT_NAME: t
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 1261843424016
      LOCK_TYPE: RECORD
      LOCK_MODE: X, REC_NOT_GAP
      LOCK_STATUS: GRANTED
      LOCK_DATA: 5
***** 4. row *****
      ENGINE: INNODB
ENGINE_LOCK_ID: 1261878693768:100:5:4:1261843424360
ENGINE_TRANSACTION_ID: 4729
      THREAD_ID: 62
      EVENT_ID: 36
OBJECT_SCHEMA: test
OBJECT_NAME: t
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: c
OBJECT_INSTANCE_BEGIN: 1261843424360
      LOCK_TYPE: RECORD
      LOCK_MODE: X, GAP
      LOCK_STATUS: GRANTED
      LOCK_DATA: 15, 15
4 rows in set (0.00 sec)
```

12.7.3 MySQL中的并发控制

■ InnoDB: 间隙锁

```
primary key (id),  
key c(c) )  
ENGINE = InnoDB;
```

```
insert into  
t(id, c, d)  
values  
(0, 0, 0),  
(5, 5, 5),  
(15, 15, 15);
```

0

5

15

$(-\infty, 0)$

$(0, 5)$

$(5, 15)$

$(15, +\infty)$

T1:

```
begin;  
select * from t where c = 5 for update;
```

T2的以下插入都能成功吗？需要等待锁吗？

T2:

```
insert into t values (3,5,5); ×  
insert into t values (4,4,4); ×  
insert into t values (17,17,17); ✓  
insert into t values (8,13,6); ×
```

- C列有索引信息，所以会用到该索引的间隙锁： $(0, 15]$ 之间都不能插入。
- 只与C列有关，与id, d列的值无关。

12.7.3 MySQL中的并发控制



■ InnoDB: 间隙锁

```
primary key (id),  
key c(c) )  
ENGINE = InnoDB;
```

```
insert into  
t(id, c, d)  
values  
(0, 0, 0),  
(5, 5, 5),  
(15, 15, 15);
```

0

5

15

$(-\infty, 0)$

$(0, 5)$

$(5, 15)$

$(15, +\infty)$

T1:

```
begin;  
select * from t where d=5 for update;
```

T2:

```
insert into t values (1, 1, 5);  
insert into t values (100, 100, 100);  
insert into t values (-1, -1, -1);
```

T2的3个插入需要等待锁吗?

因为列d没有索引，
select * from t where d
= 5 for update; 的时候会
一行一行的查找，一行一行
的加锁，直到锁定全表
(涉及多行则锁全表)

12.7.3 MySQL中的并发控制



■ InnoDB: 间隙锁

因为列d没有索引，走按照主键的全表扫描

T1:

```
begin;  
select * from t where d=5 for update;
```

```
mysql> select * from performance_schema.data_locks\G  
***** 1. row *****  
ENGINE: INNODB  
ENGINE_LOCK_ID: 1261878693768:1163:1261843426456  
ENGINE_TRANSACTION_ID: 4826  
THREAD_ID: 62  
EVENT_ID: 77  
OBJECT_SCHEMA: test  
OBJECT_NAME: t  
PARTITION_NAME: NULL  
SUBPARTITION_NAME: NULL  
INDEX_NAME: NULL  
OBJECT_INSTANCE_BEGIN: 1261843426456  
LOCK_TYPE: TABLE  
LOCK_MODE: IX  
LOCK_STATUS: GRANTED  
LOCK_DATA: NULL  
***** 2. row *****  
ENGINE: INNODB  
ENGINE_LOCK_ID: 1261878693768:102:4:1:1261843423672  
ENGINE_TRANSACTION_ID: 4826  
THREAD_ID: 62  
EVENT_ID: 77  
OBJECT_SCHEMA: test  
OBJECT_NAME: t  
PARTITION_NAME: NULL  
SUBPARTITION_NAME: NULL  
INDEX_NAME: PRIMARY  
OBJECT_INSTANCE_BEGIN: 1261843423672  
LOCK_TYPE: RECORD  
LOCK_MODE: X  
LOCK_STATUS: GRANTED  
LOCK_DATA: supremum pseudo-record
```

```
***** 3. row *****  
ENGINE: INNODB  
ENGINE_LOCK_ID: 1261878693768:102:4:2  
ENGINE_TRANSACTION_ID: 4826  
THREAD_ID: 62  
EVENT_ID: 77  
OBJECT_SCHEMA: test  
OBJECT_NAME: t  
PARTITION_NAME: NULL  
SUBPARTITION_NAME: NULL  
INDEX_NAME: PRIMARY  
OBJECT_INSTANCE_BEGIN: 1261843423672  
LOCK_TYPE: RECORD  
LOCK_MODE: X  
LOCK_STATUS: GRANTED  
LOCK_DATA: 0  
***** 4. row *****  
ENGINE: INNODB  
ENGINE_LOCK_ID: 1261878693768:102:4:3:1261843423672  
ENGINE_TRANSACTION_ID: 4826  
THREAD_ID: 62  
EVENT_ID: 77  
OBJECT_SCHEMA: test  
OBJECT_NAME: t  
PARTITION_NAME: NULL  
SUBPARTITION_NAME: NULL  
INDEX_NAME: PRIMARY  
OBJECT_INSTANCE_BEGIN: 1261843423672  
LOCK_TYPE: RECORD  
LOCK_MODE: X  
LOCK_STATUS: GRANTED  
LOCK_DATA: 5
```

```
***** 5. row *****  
ENGINE: INNODB  
ENGINE_LOCK_ID: 1261878693768:102:4:4:1261843423672  
ENGINE_TRANSACTION_ID: 4826  
THREAD_ID: 62  
EVENT_ID: 77  
OBJECT_SCHEMA: test  
OBJECT_NAME: t  
PARTITION_NAME: NULL  
SUBPARTITION_NAME: NULL  
INDEX_NAME: PRIMARY  
OBJECT_INSTANCE_BEGIN: 1261843423672  
LOCK_TYPE: RECORD  
LOCK_MODE: X  
LOCK_STATUS: GRANTED  
LOCK_DATA: 15  
5 rows in set (0.00 sec)
```

12.1 并发控制概述

12.2 封锁

12.3 封锁协议

12.4 活锁和死锁

12.5 并发调度的可串行性

12.6 两段锁协议

12.7 封锁的粒度

12.8* MVCC

12.9 小结



多版本并发控制 (MVCC)：保留数据的多个版本。写数据时，旧版本的数据并不删除，写一个新的版本，这样并发的读操作还能读到旧版本的数据。读写不阻塞，并发度高。

事务T1	事务T2
XLOCK A	
R (A)=5	
W(A)=6	
	SLOCK A
	等待
COMMIT	等待
UNLOCK A	等待
	SLOCK A
	R (A)=6
	COMMIT
	UNLOCK A

传统封锁方式

事务T1	事务T2
R(A)=5	BEGIN TRANSACTION
W(A)=6	
创建新版本A'=6	
COMMIT	
	R(A')=6
	COMMIT

MVCC

1. 在T₁准备写A的时候，为A生成一个新版本(A')，T₂事务不用等待，可继续在A'上执行。
2. T₂准备提交时，确认事务T₁是否已经完成。
3. 如果T₁已经完成了，T₂就可以放心地提交。
4. 如果T₁还没有完成，那么T₂必须等待直到T₁完成。

TS(T): 事务T的时间戳, $TS(T_i) < TS(T_j)$ 表示事务 T_i 在事务 T_j 之前开始执行。有如下说明:

- write(Q) 操作: 创建Q的一个新版本
- W-timestamp(Q): 数据项Q上成功执行write(Q)操作的所有事务中的最大时间戳
- R-timestamp(Q): 在数据项Q上成功执行read(Q)操作的所有事务中的最大时间戳
- 每一个版本 Q_k 保存:
 - ✓ 该版本的值
 - ✓ 创建 Q_k 的事务的时间戳W-timestamp(Q_k)
 - ✓ 成功读取 Q_k 的事务的最大时间戳R-timestamp(Q_k)

MVCC协议描述

（假设版本 Q_k 具有小于或等于 $TS(T)$ 的最大时间戳）

- 若事务 T 发出 $read(Q)$ ，则返回版本 Q_k 的内容
- 若事务 T 发出 $write(Q)$ ，则：
 - 当 $TS(T) < R\text{-timestamp}(Q_k)$ 时，回滚 T ；
 - 当 $TS(T) = W\text{-timestamp}(Q_k)$ 时，覆盖 Q_k 的内容
 - $TS(T) > R\text{-timestamp}(Q_k)$ ，创建 Q 的新版本
- 若一个数据对象的两个版本 Q_k 和 Q_l ，其 $W\text{-timestamp}$ 都小于系统中最老的事务的时间戳，那么这两个版本中较旧的那个版本将不再被用到，可以从系统中删除。

改进的MVCC——MV2PL（混合协议）

- 只读事务：发生冲突的可能性很小，可以采用多版本时间戳
- 更新事务：采用较保守的两阶段封锁(2PL)协议
- 实现方式之一：引入验证锁C

T ₁	T ₂		
	S	X	C
S	Y	Y	N
X	Y	N	N
C	N	N	N

注：Y=Yes,表示相容的请求；N=No,表示不相容的请求

图 12.14 验证锁的相容矩阵

采用该MV2PL：
Oracle、KingBase等

实现MVCC的两种方法：

- 写新数据时，把旧数据移到一个单独的地方，如回滚段中；其他读数据时，从回滚段中把旧版本数据读出来。

(MySQL中的innodb引擎)

https://blog.csdn.net/weixin_39723544/article/details/98200870

- 写新数据时，旧版本的数据不删除，而是把新数据插入。
(PostgreSQL)

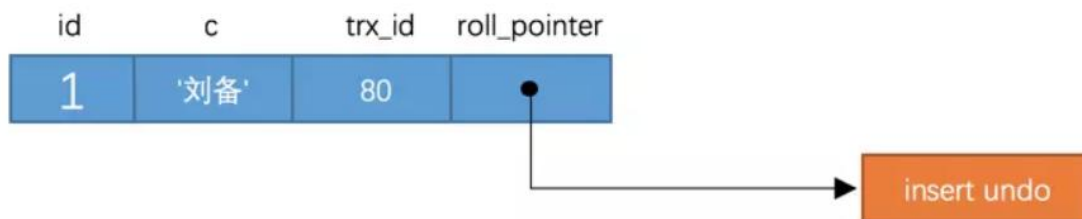
PostgreSQL实现该功能，需要在每张表上添加四个系统字段 **tmin**、**tmax**、**cmin**、**cmax**，通过这四个字段区分并发，记录不同数据的版本，和事务标识。当删除时，只会标记记录，而不会从数据块中删除，空间也没有立即释放。通过运行vacuum进程来进行回收之前的存储空间。

12.8 MVCC —— Innodb的MVCC机制



Innodb中的MVCC机制：用`trx_id`和`roll_pointer`表示数据的版本信息。

```
1 mysql> SELECT * FROM t;
2 +----+-----+
3 | id | c      |
4 +----+-----+
5 | 1 | 刘备   |
6 +----+-----+
7 1 row in set (0.01 sec)
```



show extended full columns from tablename;

```
mysql> show extended full columns from campus_card;
```

Field	Type	Collation	Null	Key	Default	Extra	Privileges	Comment
sno	varchar(8)	utf8mb4_0900_ai_ci	YES		NULL		select,insert,update,references	
balance	int	NULL	YES		NULL		select,insert,update,references	
DB_ROW_ID		NULL	NO		NULL		select,insert,update,references	
DB_TRX_ID		NULL	NO		NULL		select,insert,update,references	
DB_ROLL_PTR		NULL	NO		NULL		select,insert,update,references	

5 rows in set (0.00 sec)

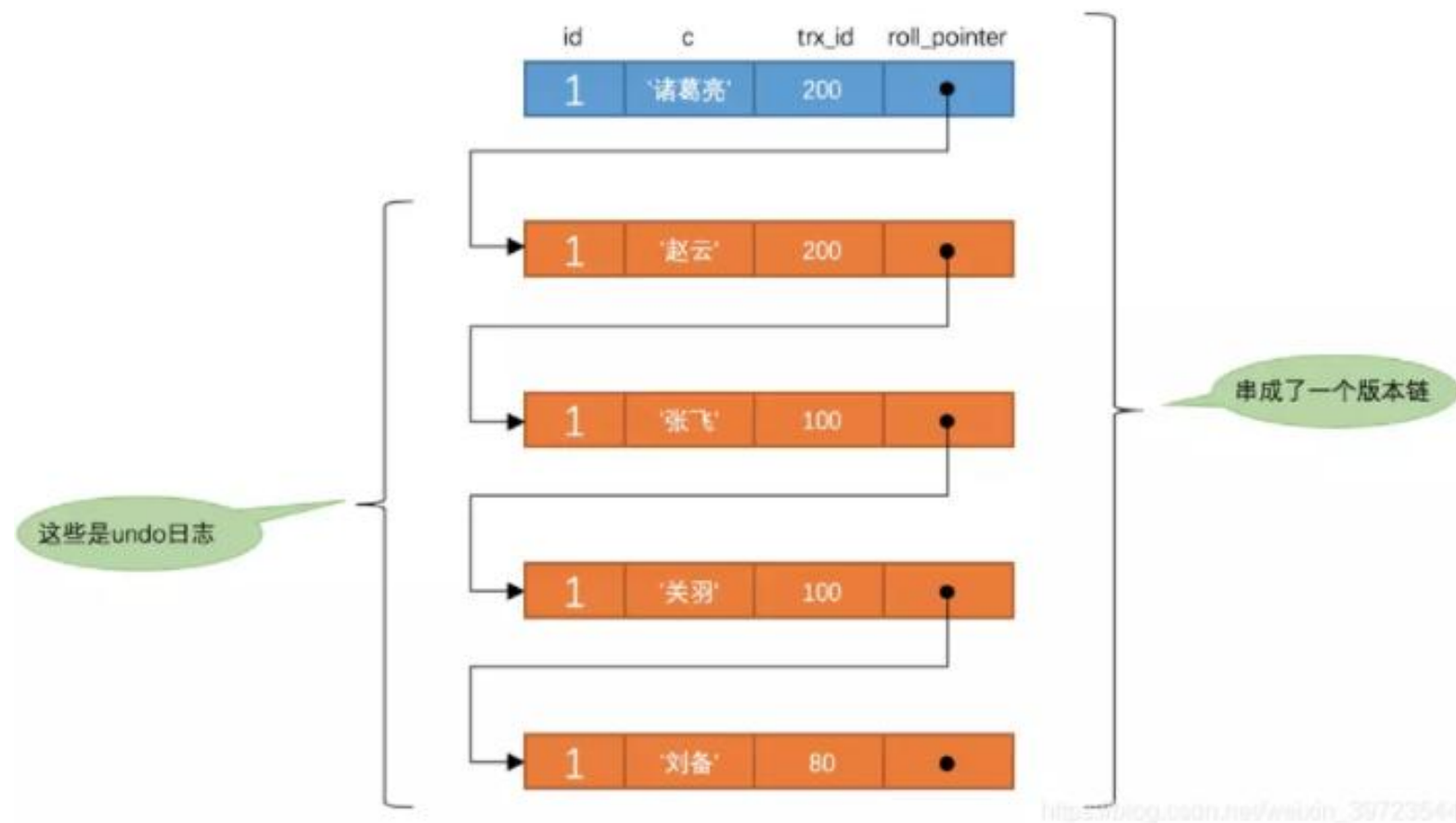
本部分例子引用：https://blog.csdn.net/weixin_39723544/article/details/98200870

12.8 MVCC —— Innodb的MVCC机制



发生时间编号	trx 100	trx 200
①	BEGIN;	
②		BEGIN;
③	UPDATE t SET c = '关羽' WHERE id = 1;	
④	UPDATE t SET c = '张飞' WHERE id = 1;	
	COMMIT;	
⑤		UPDATE t SET c = '赵云' WHERE id = 1;
⑥		UPDATE t SET c = '诸葛亮' WHERE id = 1;
		COMMIT;

MySQL Innodb中的MVCC机制



不同隔离级别下具体使用这个数据版本链呢？

核心问题：需要判断版本链中的哪些版本是当前事务可见？

readview

m_ids: 当前活跃事务的列表

min_trx_id: 生成readview时, m_ids的最小值

max_trx_id: 生成readview时, 系统应该分配的下一个trx_id值

版本链中每条数据的可见性判断规则：

- 如果当前记录的 $\text{trx_id} < \text{min_trx_id}$, 肯定可见。
- 如果当前记录的 $\text{trx_id} > \text{max_trx_id}$, 肯定不可见。
- 如果当前记录的 trx_id 列在 min_trx_id 和 max_trx_id 之间, 判断该 trx_id 在不在m_ids列表中, 如果在不可见, 否则可见。

12.8 MVCC —— Innodb的MVCC机制



- READ UNCOMMITTED隔离级别的实现：读最新的版本
- READ COMMITTED隔离级别的实现

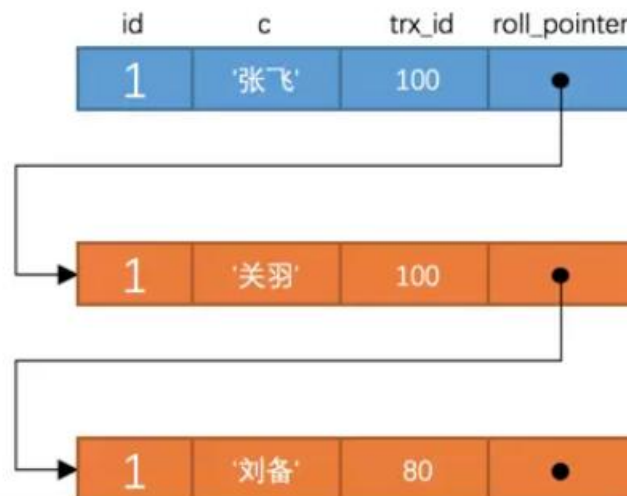
```
1 # Transaction 100
2 BEGIN;
3
4 UPDATE t SET c = '关羽' WHERE id = 1;
5
6 UPDATE t SET c = '张飞' WHERE id = 1;
```

1

```
1 # Transaction 200
2 BEGIN;
3
4 # 更新了一些别的表的记录
5 ...
6
```

2

id=1的数据版本链



注意：事务执行中，只有第一次真正修改记录时（如INSERT、DELETE、UPDATE语句），才会被分配一个单独的事务id，事务id是递增的。

12.8 MVCC —— Innodb的MVCC机制

READ COMMITTED隔离级别的实现（续）

注意：在每次读取数据时生成一个ReadView

```
1 # 类似于第三个客户端，开启了一个新的事务
2 # 使用READ COMMITTED隔离级别的事务
3 BEGIN;
4
5 # SELECT1: Transaction 100、200未提交
6 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
```

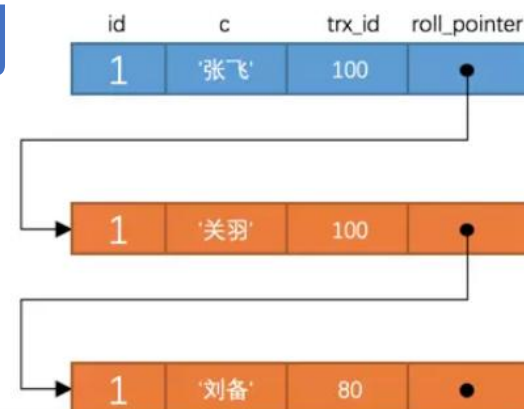
select1: select * from t where id=1

生成
readview

m_ids: [100, 200]

min_trx_id: 100

max_trx_id: 300



执行select1的时候，生成readview，判断该数据的版本链中应该显示哪个版本：

1. 最新内容：张飞，trx_id=100 在[min_trx_id, max_trx_id]之间，确认100是不是属于当前活跃事务列表m_ids中？是 => 不可见；
2. 顺roll-pointer继续：关羽，trx_id=100，也前面判断相同，不可见；
3. 顺roll-pointer继续：刘备，trx_id=80 < min_trx_id，可见，返回刘备。
(含义：如果是比本事务开始还早并提交的事务所更新的值，就应该读取)

READ COMMITTED隔离级别的实现（续）

```
1 # Transaction 100
2 BEGIN;
3
4 UPDATE t SET c = '关羽' WHERE id = 1;
5
6 UPDATE t SET c = '张飞' WHERE id = 1;
7
8 COMMIT;
```

4

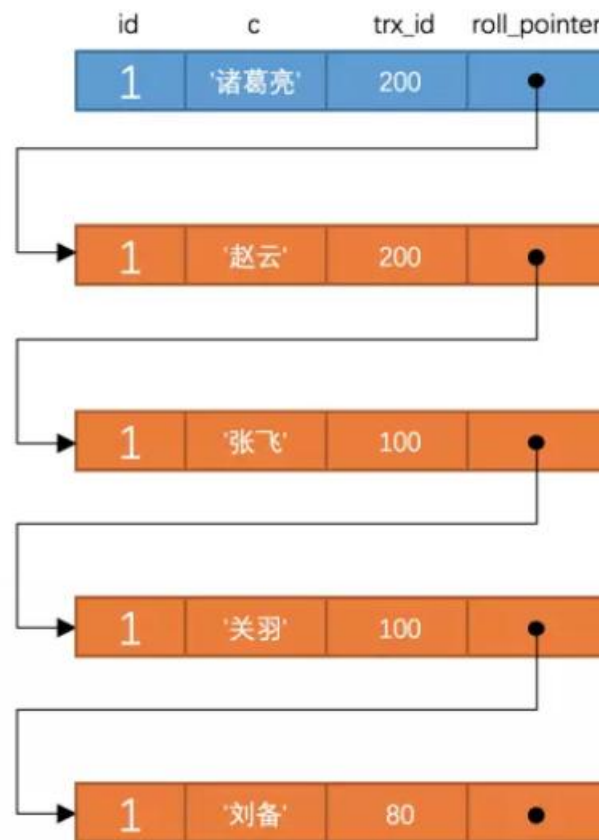
事务提交



```
1 # Transaction 200
2 BEGIN;
3
4 # 更新了一些别的表的记录
5 ...
6
7 UPDATE t SET c = '赵云' WHERE id = 1;
8
9 UPDATE t SET c = '诸葛亮' WHERE id = 1;
```

5

新事务



h

12.8 MVCC —— Innodb的MVCC机制

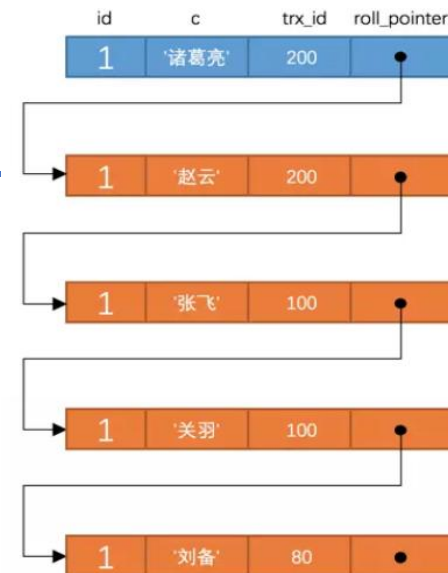
READ COMMITTED隔离级别的实现（续）

```
1 # 新开的一个客户端字段
2 # 使用READ COMMITTED隔离级别的事务
3 BEGIN;
4
5 # SELECT1: Transaction 100、200均未提交
6 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'刘备'
7
8 # SELECT2: Transaction 100提交, Transaction 200未提交
9 SELECT * FROM t WHERE id = 1; # 得到的列c的值为'张飞'
```

select2: select * from t where id=1

注意：在每次读取数据时生成一个ReadView

第二次查询生成新readview



m_ids: [200]

min_trx_id: 200

max_trx_id: 300

执行select2的时候，根据规则判断该数据的版本链中应该显示哪个版本：

1. 最新内容：诸葛亮， trx_id=200 在[min_trx_id, max_trx_id]之间，确认200是不是属于当前活跃事务列表m_ids中，是 => 不可见；
2. 顺roll-pointer继续：赵云， trx_id=200 ，也前面判断相同，不可见；
3. 顺roll-pointer继续：张飞， trx_id=100 < min_trx_id，可见，返回张飞。
(含义：如果是比本事务开始还早的事务更新的值，就应该读取)

如果T200也提交了，可知：m_ids=[], max_trx_id=300，第一个数据诸葛亮OK

12.8 MVCC —— Innodb的MVCC机制

READ REPEATABLE READ 隔离级别的实现:

注意: 仅在第一次读取数据时生成一个ReadView

```
1 # 新开的一个客户端字段
2 # 使用READ COMMITTED隔离级别的事务
3 BEGIN;
4
5 # SELECT1: Transaction 100、200均未提交
6 SELECT * FROM t WHERE id = 1; # 得到的列c的值为 '刘备'
7
8 # SELECT2: Transaction 100提交, Transaction 200未提交
9 SELECT * FROM t WHERE id = 1; # 得到的列c的值为 '张飞'
```

select2: select * from t where id=1

维持原有
readview

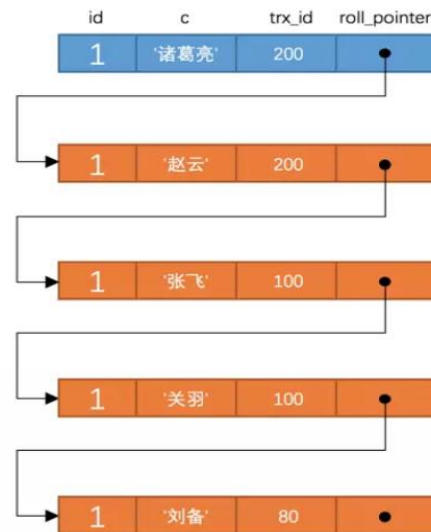
m_ids: [100, 200]

min_trx_id: 100

max_trx_id: 300

执行select2的时候, 根据规则判断该数据的版本链中应该显示哪个版本:

1. 最新内容: 诸葛亮, trx_id=200, 在[min_trx_id, max_trx_id]之间, 确认200是不是属于当前活跃事务列表m_ids中, 是 => 不可见;
2. 顺roll-pointer继续: 赵云, trx_id=200, 也前面1判断相同, 不可见;
3. 顺roll-pointer继续: 张飞, trx_id=100, 在[min_trx_id, max_trx_id]之间, 确认100是不是属于当前活跃事务列表m_ids中, 是 => 不可见
4. 顺roll-pointer继续: 关羽, trx_id=100, 也前面3判断相同, 不可见;
5. 顺roll-pointer继续: 刘备, trx_id=80 < min_trx_id, 可见, 返回刘备。



■ 并发操作带来的数据不一致性问题

- 修改丢失、脏数据、不可重复读、幻读

■ 数据库的并发控制通常使用封锁机制

- 基本封锁（S锁和X锁）
- 封锁协议（三级封锁协议所能解决的数据不一致性问题）
- 活锁和死锁
- 多粒度封锁（多粒度树、意向锁）
- MySQL中的隔离级别与锁

■ 并发事务调度的正确性

- 可串行性：两段锁协议
- 冲突可串行性

■ MVCC理解

作业：并发控制
本章：10, 11, 14, 15



