分布式数据库系统及其应用

徐喜荣
(xirongxu@dlut.edu.cn)

第6章 分布式数据库中的并发控制

1. 并发控制的概念和理论
2. 分布式数据库系统并发控制的封锁技术
3. 分布式数据库系统中的死锁处理
4. 分布式数据库系统并发控制的时标技术
5. 分布式数据库系统并发控制的多版本技术
6. 分布式数据库系统并发控制的乐观方法

1 并发控制的概念和理论

1.1 并发控制的概念

- 通常，数据库总有若干个事务在运行，这些事务可能并发地存取相同的数据，称为**事务的并发操作**。
- 当数据库中有**多个事务并发执行**时，系统必须对并发事务之间的相互作用加以控制，这是通过**并发控制机制**来实现。
- **并发控制**负责**正确协调并发事务执行**，保证这种并发存取操作不至于破坏数据库的完整性和一致性，确保并发执行的**多个事务能够正确地运行**并**获得正确的结果**。

1 并发控制的概念和理论

1.1 并发控制的概念

■以一个实例，说明并发操作带来的数据的不一致性问题。

例如：飞机订票系统中的一个活动序列

1.甲售票点（甲事务）读出某航班的机票余额A, 设A=16.

2.乙售票点（乙事务）读出同一航班的机票余额A, 也为16.

3.甲售票点卖出一张机票，修改余额A←A-1。所以A为15, 把A写回数据库。

4.乙售票点也卖出去一张机票，修改余额A←A-1。所以A为15，把A 写回数据库。

结果明明卖出两张机票，数据库中机票余额只减少1。

1 并发控制的概念和理论

1.1 并发控制的概念

在并发操作情况下，对甲、乙两个事务的操作序列的**调度是随机的**。若按上面调度序列执行，**甲事务的修改就被丢失**。原因：第4步中乙事务修改 A 并写回后覆盖了甲事务的修改。这种情况称为**数据库的不一致性**是由**并发操作引起**的。

- **并发操作带来的数据不一致性包括三类：**
  - 丢失修改（Lost Update）
  - 不可重复读（Non-repeatable Read）
  - 读“脏”数据（Dirty Read）

1 并发控制的概念和理论

1.1 并发控制的概念

并发控制问题之一----丢失修改：两个事务T_1和T_2读入同一数据并修改，T_2提交的结果破坏了T_1提交的结果，导致T_1的修改被丢失。

时间	更新事务T_1	数据库中X的值	更新事务T_2
t_0		100	
t_1	FIND x		
t_2			FIND x
t_3	x:=x-30		
t_4			x:=x*2
t_5	**UPDATE x**		
t_6		70	**UPDATE x**
t_7		200	

注：其中FIND表示从数据库中读值，UPDATE表示把值写回到数据库，T_1T_2结果140，T_2T_1结果170，但得到的结果是200，显然是不对的，T_1在t_7丢失更新操作。

1 并发控制的概念和理论

1.1 并发控制的概念

并发控制问题之二----不一致性读：指事务T_1读取数据X后，事务T_2读取数据X。之后事务T_1更新了数据X的值，此时事务T_2使用的X值仍是原来的数据X的值。

时间	更新事务T1	数据库中A的值	更新事务T2
t_0		100	
t_1	FIND x		
t_2			FIND x
t_3	x:=x-30		
t_4	**UPDATE x**		
t_5		70	

注：在时间t_5事务T_2仍认为x的值是100

1 并发控制的概念和理论

1.1 并发控制的概念

并发控制问题之三----读脏数据（依赖于未提交更新）:事务T_1修改某一数据，并将其写回磁盘。事务T_2读取同一数据后，T_1由于某种原因被撤消。事务T_1已经修改过的数据恢复原值，T_2读到的数据就与数据库中的数据不一致。

时间	更新事务T1	数据库中A的值	更新事务T2
t_0		100	
t_1	FIND x		
t_2	x:=x-10		
t_3	**UPDATE x**		
t_4		90	FIND x
t_5	**ROLLBACK**		
t_6		100	

1 并发控制的概念和理论

1.1 并发控制的概念

- 产生上述三类数据不一致性的**主要原因**是：
  - 并发操作**破坏了事务的隔离性**。
- **并发控制**就是要**用正确的方式调度并发操作**，使一个用户事务的执行不受其它事务的干扰，**避免造成数据的不一致性**。

1 并发控制的概念和理论

1.1 并发控制的概念

■并发控制的**主要技术**是：

- **封锁**（Locking）、**时间戳**和**乐观控制法**；
- 商用的DBMS一般都采用**封锁方法**。

■例如，甲事务要修改数据项A，若在读出A前先**锁住A**，其他事务就不能再读取和修改A了，直到甲修改并写回A后**解除了对A的封锁**为止。这样就不会丢失甲的修改。

1 并发控制的概念和理论

1.1 并发控制的概念

- **分布式数据库中的并发控制**主要解决**多个分布式事务**对数据并发执行的正确性，**保证数据库的完整性和一致性**。
- 分布式数据库中，允许数据被复制在多个站点上，当需要对数据执行更新操作时，也必须**同时正确地更新它的所有副本**。当来自同一站点或/和不同站点的多个事务对数据进行并发操作时，如果不能正确处理， 数据库的完整性和一致性很容易遭到破坏。因此，**分布式并发控制比集中式并发控制更复杂**。

1 并发控制的概念和理论

1.2 事务可串行化理论

- 对一组**并发的分布式事务**可能存在多种正确调度，分布式数据库管理系统的**事务管理器并发控制机制**应该采用代价最小的正确调度。
- 与集中式数据库系统一样，**可串行化调度**也是分布式事务能否正确执行的**基本方法**。

1 并发控制的概念和理论

1.2 事务可串行化理论

- **事务的可串行性**是指若干个事务**并发执行的结果**与按**希望的顺序执行的结果相同**时，称**诸事务是可串行的**。
- 如果事务的并发执行能够通过以一定顺序串行执行就可使数据库处于新的一致状态，那么诸如丢失更新的问题就可能得到解决，这就是串行化理论的观点。

1 并发控制的概念和理论

1.2 事务可串行化理论的基本概念

一、分布式事务的调度定义

在数据库系统中，事务访问数据库中数据的方式是通过发出**读操作**和**写操作**原语来实现的。

通常以 T_i 表示某个事务，以 $R_i(x)$ 表示事务T_i对**数据项x**的**读操作**，以$W_i(x)$表示该事务T_i对**数据项x**的**写操作**。

事务的一个**操作序列**称为一个**调度**(Schedule也称history)，一般以字母S表示。

例如，关于两个事务的一个调度：

S： $R_1(x)$，$R_2(y)$，$W_2(y)$，$R_2(x)$，$W_1(x)$，$W_2(x)$

1 并发控制的概念和理论

1.2 事务可串行化理论的基本概念

二、操作冲突定义

两个**同时访问**同一**数据项*x***的操作，如果其中至少有一个是**写操作**，那么称这**两个操作是冲突的**。注意两点：

第一，只有两种冲突：**读-写**冲突(或**写-读**冲突)，**写-写**冲突。

第二，两个操作可以属于**同一事务**或者两个**不同的事务**，后者的情况称为**两个事务冲突**。

如果有两个事务T_i和T_j，T_i的所有操作都先于T_j的操作，则这两个事务为**串行执行**的，必定不会有冲突。

1 并发控制的概念和理论

1.2 事务可串行化理论的基本概念

三、分布式事务串行调度定义

设有一组事务$T=\{T_1, T_2, \ldots\ldots, T_n\}$，如果事务 T_i **的所有操作**都**先于事务T_j 的操作**，记为 $T_i < T_j$。

若一个**调度S**，其每个事务的执行均有$T_i<T_j$，$i\neq j$，记为：

$$S = \{ \ldots\ldots < T_i < T_j < \ldots\ldots \}$$

称 S 是一个**串行调度**。

1 并发控制的概念和理论

1.2 事务可串行化理论的基本概念

- 对一个**串行调度S**来说，它总是可以**正确地执行**，执行它可以使数据库保持**一致状态**。原因如下：

（1）如果**S正确执行完成**，则S中的每一个事务都被提交，由于事务的原子性，**保证了数据库的一致性**。

（2）如果**S在执行时发生故障**，若**T_k之前的事务都已提交**，则夭折T_k，使数据库的状态恢复到T_k前的状态。该状态的数据库也是一致的，因为T_k之前的事务都已提交。

（3）如果**S在执行时发生故障**，若**T_k之前的事务有被夭折的**，则夭折T_k，**重做**T_k以前已被提交的事务，**撤销**T_k以前被夭折的事务，此时数据库也是一致的。

所以**串行调度可使数据库保持一致**。但系统运行效率低。

1 并发控制的概念和理论

1.2 事务可串行化理论的基本概念

四、可串行化调度

可串行化调度是让**有冲突的操作串行执行**，**非冲突的操作并行执行**，所以**可串行化调度**是事务**并发控制**要寻求的**基本方法**。

因为分布式事务之间的冲突最终分解，转换为同一站点上子事务间的冲突操作，而且由于分布式数据库中数据的复制，会使冲突的几率比集中式更小，从而使并行执行的程度更高。

因此，**分布式事务可串行化调度可以转化为子事务的可串行化调度**，但涉及多副本选择时，分布式事务调度要多做一个选择副本的操作，以避免冲突操作。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

一、事务的定义

一个**事务**是一个**偏序集**：$T_i = \{ \sum_i , <_i \}$，其中：

(1) $\sum_i$ ： 操作符集合,包含$\{R_i[x], W_i[x] \mid x$为数据项$\} \cup \{ A_i , C_i \}$，A_i, C_i 是$\sum_i$ 中**最后一个操作符**，且**只能出现其中之一个**；A_i为撤销(abort), C_i 为提交(commit)；

$<_i$ ： 偏序关系，即(冲突)操作**有先后次序执行**。

(2) 如果$R_i[x], W_i[x] \in \sum_i$，则它们必满足 $R_i(x) <_i W_i(x)$ 或 $W_i(x) <_i R_i(x)$。

(3) $R_i[x]$, $W_i[x]$，A_i，C_i 都是事务T_i操作符序列中的一个操作。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

二、冲突动作

如果有两个操作P和Q对同一个**数据A**进行操作，其中有一个是**写操作**W(A)，则 P和Q称为**冲突操作**。

$R_1(A)$ < **$W_2(A)$**　　**$W_2(A)$** < $R_1(A)$　　**$W_1(A)$** < **$W_2(A)$**

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

三、并发事务的一个调度(简称并发调度)定义

令$T=\{T_1,T_2,\ldots,T_n\}$是一组**并发**执行事务。**T上的调度 S** 是具有如下顺序关系 $<_T$ 的**偏序集**，即 $S=\{\sum_T, <_T\}$：

(1) $\sum_T = \bigcup_{i=1}^{N} \sum_i$

(2) $<_T \supseteq \bigcup_{i=1}^{N} <_i$

(3) 对任意两个冲突操作 $p,q \in S$, 存在 **$p < q$ 或 $q < p$**关系。

第一种情况简单地说明了调度的域是每个事务域的并集。
第二种情况定义**偏序关系**为每个事务偏序关系的超集，这保证了每一个事务内部的操作的顺序。
最后一种情况定义了冲突操作的执行顺序。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

四、串行调度

如果一个调度S中的任意两个事务T_i和T_j，$i \neq j$，若

$$\cup_{i=1}^{n}\Sigma_i < \cup_{j=1}^{n}\Sigma_j \quad \text{或者} \quad \cup_{j=1}^{n}\Sigma_j < \cup_{i=1}^{n}\Sigma_i$$

则称调度S为**串行调度**。

- 即一个事务的第一个动作是在另一个事务的最后一个动作完成后开始。
- 即一个调度中**不同事务**的**各个操作不会**互相**交叉**，每个事务是**相继执行**的。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

五、一致性调度

- 如果执行一个调度S，可以使得数据库从**一个一致性状态**转变为**另一个一致性状态**，则称调度S为**一致性调度**。
- 显然，串行调度是一致性调度。

六、调度等价

- 调度S_1与S_2是**等价**的**充分条件**是：对于两个有冲突的操作O_i和O_j，若 O_i, $O_j \in S_1$, 且 **$O_i < O_j$ 在S_1中**成立, 则 O_i, $O_j \in S_2$, 且也有**$O_i < O_j$ 在S_2中**也成立。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

七、可串行化调度

- 如果一个调度**等价于某个串行调度**，则该调度称为**可串行化调度**。
- 也即该调度可以通过一系列**非冲突动作的交换操作**使其成为串行调度。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

例1. 考虑两个事务，分别定义如下：

T_1:

1. Read(x)
2. x=x+10
3. Write(x)
4. Read(y)
5. y=y-15
6. Write(y)
7. commit

T_2:

1. Read(x)
2. x=x-20
3. Write(x)
4. Read(y)
5. y=y*2
6. Write(y)
7. commit

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

T_1:	T_2:
1. Read(x)	1. Read(x)
2. x=x+10	2. x=x-20
3. Write(x)	3. Write(x)
4. Read(y)	4. Read(y)
5. y=y-15	5. y=y*2
6. Write(y)	6. Write(y)
7. commit	7. commit

①$R_1(x)$	③$R_2(x)$
$W_1(x)$	$W_2(x)$
②$R_1(y)$	④$R_2(y)$
$W_1(y)$	$W_2(y)$

满足序关系：

① < ②

③ < ④

可产生五种调度方式：

S_1. ① ② ③ ④

S_2. ① ③ ② ④

S_3. ① ③ ④ ②

S_4. ③ ④ ① ②

S_5. ③ ① ④ ②

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

S_1	S_2	S_3	S_4	S_5
$R_1(x)$,	$R_1(x)$,	$R_1(x)$,	$R_2(x)$,	$R_2(x)$,
x=x+10,	x=x+10,	x=x+10,	x=x-20,	x=x-20,
$W_1(x)$,	$W_1(x)$,	$W_1(x)$,	$W_2(x)$,	$W_2(x)$,
$R_1(y)$,	$R_2(x)$,	$R_2(x)$,	$R_2(y)$,	$R_1(x)$,
y=y-15,	x=x-20,	x=x-20,	y=y*2,	x=x+10,
$W_1(y)$,	$W_2(x)$,	$W_2(x)$,	$W_2(y)$,	$W_1(x)$,
C_1,	$R_1(y)$,	$R_2(y)$,	C_2,	$R_2(y)$,
$R_2(x)$,	y=y-15,	y=y*2,	$R_1(x)$,	y=y*2,
x=x-20,	$W_1(y)$,	$W_2(y)$,	x=x+10,	$W_2(y)$,
$W_2(x)$,	C_1,	C_2 ,	$W_1(x)$,	C_2 ,
$R_2(y)$,	$R_2(y)$,	$R_1(y)$,	$R_1(y)$,	$R_1(y)$,
y=y*2,	y=y*2,	y=y-15,	y=y-15,	y=y-15,
$W_2(y)$,	$W_2(y)$,	$W_1(y)$,	$W_1(y)$,	$W_1(y)$,
C_2	C_2	C_1	C_1	C_1

- 如果将事务提交延迟到两个事务操作完成之后执行有：
  - 调度S_1和S_4是**串行调度**，也是一致性调度；
  - 调度S_2和S_1的冲突操作具有相同的顺序，因此是**等价调度**；S_2是**可串行化调度**，也是一致性调度；
  - 调度S_3虽是一致调度，但是它不与S_1或S_4等价，所以**S_3不是可串行化调度**；
  - 调度S_5和S_4等价，所以S_5是一致调度，也是**可串行化调度**。

1 并发控制的概念和理论

1.3 分布式事务的可串行化理论

- 有以下推论：
  - **一个可串行化调度必定与某个串行调度等价**，且是一致性调度；
  - 一致性调度不一定是可串行化调度；
  - 同一事务集**几个可串行化调度**，它们的**结果未必相同**。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

1. 使用优先图 P(S)判别可串行化调度

- 调度 S 的**优先图**是一个**有向图G(N, E)**，其中
  - N：一组**节点**$N=\{T_1 T_2,\ldots,T_n\}$, T_i 是S中的**事务**；
  - E：一组有向边$E=\{e_1,e_2,\ldots,e_n\}$, **每条边 e_i** 形如$T_i \rightarrow T_j$，$1\le i\le n$，$1\le j\le n$，其中T_i是e_i 的始节点，T_j是e_i 终节点。

    如果调度中T_i的**一个操作**出现在T_j的某个**冲突操作**前，那么就创建这样的一条边。即：当且仅当$\exists\ p \in T_i, q \in T_j$ 使得p, q 冲突，并且$p <_S q$。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

算法5.1 测试调度S的可串行化

- 对于调度 S 中的事务T_i，在图中创建一个节点T_i。
- 对于每一种这样的情形：如果S中的在**T_i执行了W(X)**操作后**执行T_j的R(X)操作**，那么在优先图中创建一条边（$T_i \rightarrow T_j$）；
- 对于每一种这样的情形：如果S中的在**T_i执行了R(X)**操作后**执行T_j的W(X)操作**，那么在优先图中创建一条边（$T_i \rightarrow T_j$）；
- 对于每一种这样的情形：如果S中的在**T_i执行了W(X)**操作后**执行T_j的W(X)操作**，那么在优先图中创建一条边（$T_i \rightarrow T_j$）；
- 当且仅当**优先图中没有闭环**时，调度S是**可串行化**的调度。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

测试调度S的可串行化

- 如果优先图中**存在环路**，说明**调度是不可串行化的**，

  否则是可串行化的。

- 环路是指有向图中的一个边序列C={($T_j \rightarrow T_k$)，($T_k \rightarrow T_p$)，……，($T_i \rightarrow T_j$)}。 每条边的起始节点(第一条边除外)都与前一条边的终止节点相同，第一条边的起始节点与最后一条边的终止节点相同，即事务序列是以同一个节点作为开始和结束的。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

测试调度S的可串行化

- 在优先图中，一条从T_i到T_j的边意味着调度S中事务T_i在事务T_j之前，与S等价的调度中T_i也必须在T_j之前。如果优先图中不存在环路，就可以**创建与S等价的串行调度S′**，并按如下方式对S中的事务进行排序：

  只要在优先图中存在从T_i到T_j的边，则在等价串行调度S′中T_i就必须出现在T_j前。某个数据项X导致了调度中的一条边的生成，那么就可用该数据项名X来标注优先图中的这条边$T_i \rightarrow T_j$。
- **如果调度S的优先图不存在环路，则就可能存在若干个与S等价的串行调度S’**。但如果优先图存在环路，则不可能创建任何等价的串行调度，从而S是不可串行的。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

- 例如, 考虑如下3个事务:

  T_1: Read(x); Write(x); Commit;

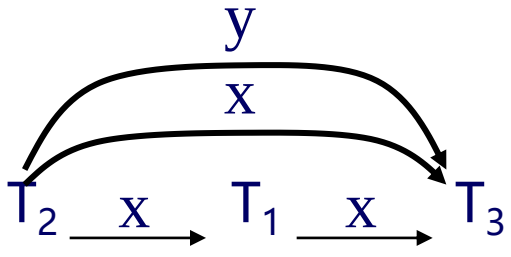
  T_2: Write(x); Write(y); Read(z); Commit;

  T_3: Read(x); Read(y); Read(z); Commit;

这3个事务的一个调度:

S={ **$W_2(x)$**, **$W_2(y)$**, $R_2(z)$, C_2, **$R_1(x)$**, **$W_1(x)$**, C_1, **$R_3(x)$**, **$R_3(y)$**, $R_3(z)$, C_3 }

优先图:



无环, S是串行调度。

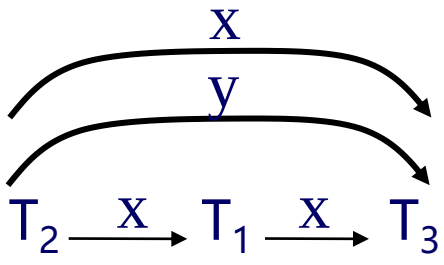
1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

另外一个调度S′：

S′ ={**$W_2(x)$**, $R_1(x)$, **$W_1(x)$**, C_1, **$R_3(x)$**, **$W_2(y)$**, **$R_3(y)$**, $R_2(z)$, C_2, $R_3(z)$, C_3 }

优先图：



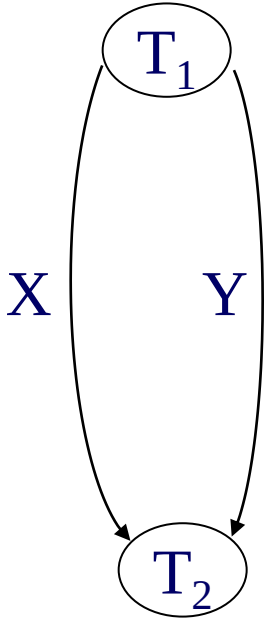
无环，S′是可串调度。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

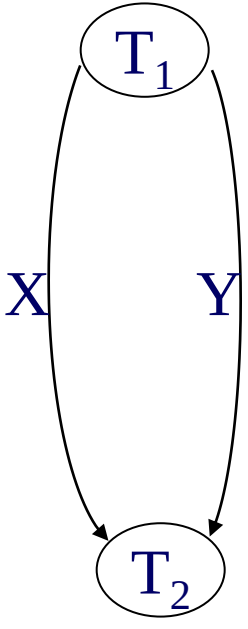
$R_1(x)$,
x=x+10,
$W_1(x)$,
$R_1(y)$,
y=y-15,
$W_1(y)$,
C_1,
$R_2(x)$,
x=x-20,
$W_2(x)$,
$R_2(y)$,
y=y*2,
$W_2(y)$,
C_2

S1的优先图



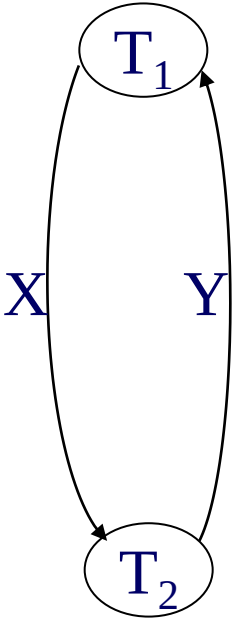
$R_1(x)$,
x=x+10,
$W_1(x)$,
$R_2(x)$,
x=x-20,
$W_2(x)$,
$R_1(y)$,
y=y-15,
$W_1(y)$,
C_1,
$R_2(y)$,
y=y*2,
$W_2(y)$,
C_2

S2的优先图



$R_1(x)$,
x=x+10,
$W_1(x)$,
$R_2(x)$,
x=x-20,
$W_2(x)$,
$R_2(y)$,
y=y*2,
$W_2(y)$,
C_2,
$R_1(y)$,
y=y-15,
$W_1(y)$,
C_1

S3的优先图



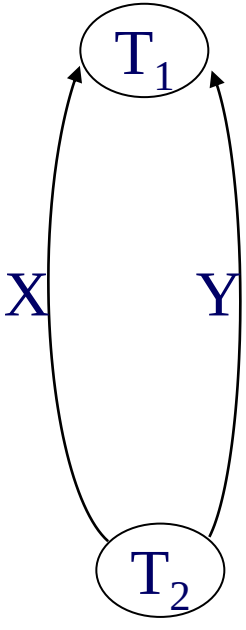
存在环路

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

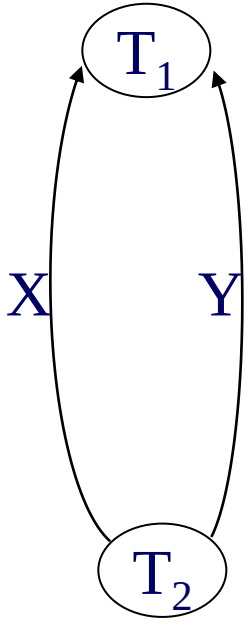
$R_2(x)$,
x=x-20,
$W_2(x)$,
$R_2(y)$,
y=y*2,
$W_2(y)$,
C_2,
$R_1(x)$,
x=x+10,
$W_1(x)$,
$R_1(y)$,
y=y-15,
$W_1(y)$,
C_1

S4的优先图



$R_2(x)$,
x=x-20,
$W_2(x)$,
$R_1(x)$,
x=x+10,
$W_1(x)$,
$R_2(y)$,
y=y*2,
$W_2(y)$,
C_2 ,
$R_1(y)$,
y=y-15,
$W_1(y)$,
C_1

S5的优先图



1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

2. 分布式数据库可串行性理论扩展

- 可串行性理论可以**直接扩展到无重复副本**的**分布式数据库**中。
  - 事务在每个站点上的执行调度称作**局部调度**，涉及多个站点上的调度称为**全局调度**。
  - 如果分布式数据库中**数据没有副本**，并且**每个局部调度都是可串行化调度**，只要这些**局部调度的顺序一致**，则它们的**并（全局调度）也是可串行化调度**。
  - 在一个**有复制副本**的分布式数据库上，可串行化理论的扩展就比较**复杂**。可能局部调度是可串行化的，而分布式数据库的相互一致性却仍不能保证。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

3. 单副本可串行化

- 相互一致性要求所有数据项副本的值都是相同的，能维持相互一致性的调度称作**副本可串行化的调度**。
- **一个单副本可串行化的全局调度**必须满足以下条件：
  - **每一个局部调度**必须是**可串行化的**。
  - **两个冲突操作**在它们同时出现的**各个局部调度**中，必须具有**相同的相对顺序**。

  条件二保证了任何同时执行冲突事务站点上的可串行化顺序都相同。

  在有副本的分布式数据库中，还需做的是**保证单副本的串行性**。

  这是**副本控制协议**的职责。

1 并发控制的概念和理论

1.4 分布式事务的可串行化调度测试

4. 读一个/写全部副本控制协议

- 假定一个**数据项x**有若干副本x_1, x_2, ……, x_n，则称**x**为**逻辑数据项**，它的**副本x_1,x_2,…,x_n**为**物理数据项**。如果复制是透明的，那么用户事务就可以对逻辑数据项x进行读写操作。
- “读一个/写全部”副本控制协议：
  - 对于逻辑数据项x上的一个**读操作[Read(x)]**，只映射到x的**某一个**物理数据项x_j上，即[Read(x_j)]，而对于逻辑数据项x的**写操作[Write(x)]**，则映射到x的物理数据项的**全集**x_1,x_2,……,x_n上。所以这一协议通常称为**“读一个/写全部”(ROWA)协议**。
  - 用于**分布式两阶段锁协议**的实现方法中。

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

- 使用协议或规则保证调度可串行化

大多数并发控制机制不是真正通过测试确定调度是否为可串行化的，而是**使用协议或规则来保证一个调度是可串行化的**。

实际应用中，测试调度的可串行化非常困难，很难为确保可串行化，而事先确定调度中的操作如何交错。

大多数商业DBMS中采取的方法是**设计协议(规则的集合)**，如果协议被每个单独事务遵循，或者被一个DBMS并发控制子系统执行，将确保事务参与的所有调度都是可串行化的。

1 并发控制的概念和理论

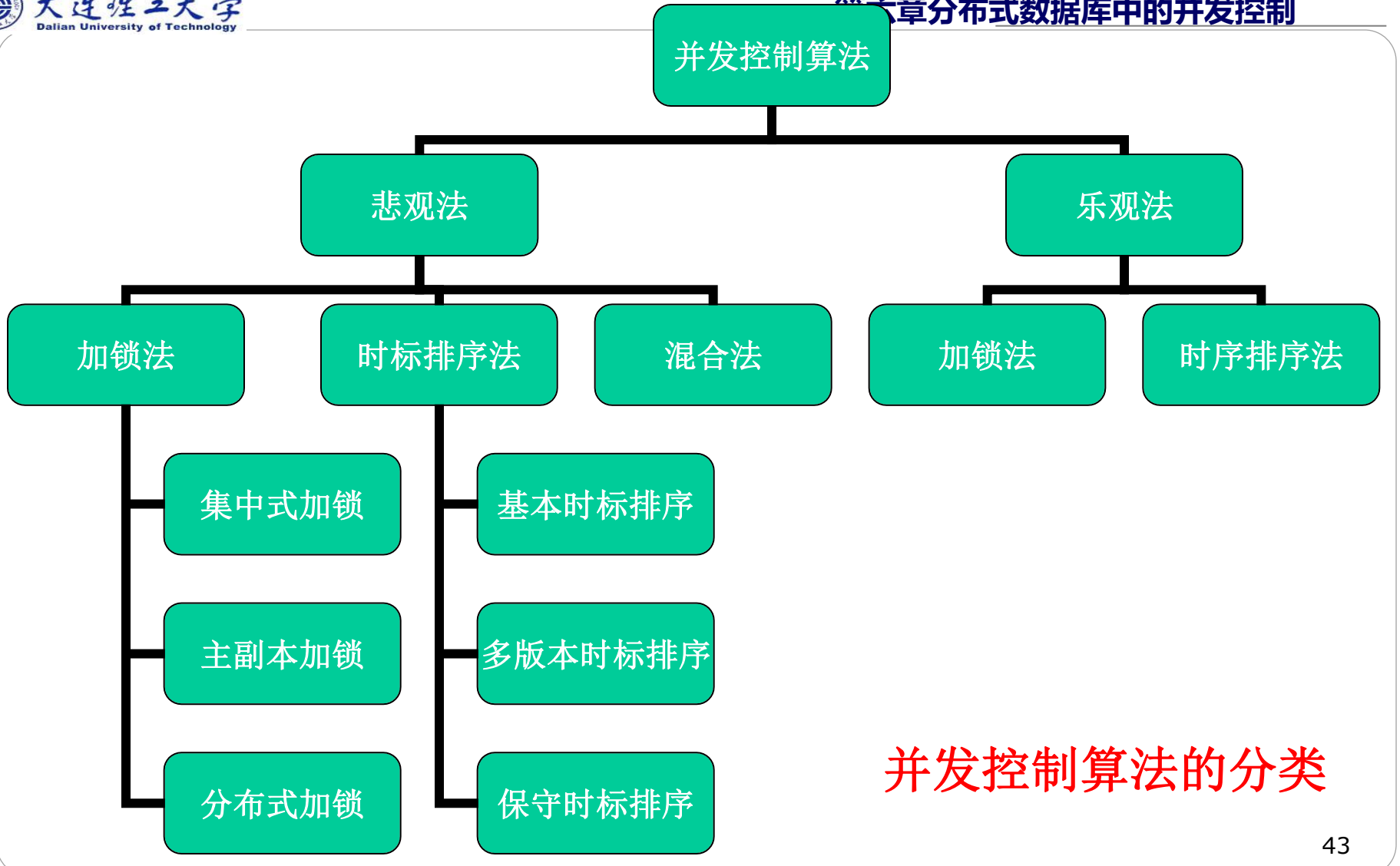
1.5 并发控制机制的常用方法及其分类

- 多种保证可串行化的并发控制协议：
  - **两阶段封锁协议**：它是基于**对数据项进行封锁**，以**阻止并发事务受到其他事务的干扰**，并且执行一个附加的确保可串行化的条件。大多数商业DBMS使用的都是这种技术。
  - **时间戳排序**：每个**事务**被指定一个**唯一的时间戳**，协议确保按照**事务时间戳的顺序**执行任何冲突操作；
  - **多版本协议**：对**数据项**的多个版本进行维护；
  - **最优化(也称为确认或证明)协议**：在事务终止后但在事务被允许提交前，检查是否有可能破坏可串行化。

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

- 并发控制机制划分为两种类型
  - **悲观并发控制法**
    - 悲观算法使事务的**并发执行**在执行生命周期的**开始就同步化**。
    - 悲观方法有**基于封锁**算法、**基于时标排序**(或事务排序)算法和**混合算法**。
  - **乐观并发控制法**
    - 乐观算法将**同步化延迟到事务执行周期的结束**。
    - 乐观算法可分为**基于封锁**或**基于时标排序**的算法。


并发控制算法
悲观法
乐观法
加锁法
时标排序法
混合法
加锁法
时序排序法
集中式加锁
主副本加锁
分布式加锁
基本时标排序
多版本时标排序
保守时标排序

并发控制算法的分类

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

- 1. 基于封锁的算法

在基于封锁的方法中，**事务的同步化**是通过对数据库的片段或者数据项进行物理或逻辑封锁来实现，**封锁对象的大小通常称为封锁粒度**。

封锁方法的类型可以根据在哪里进行封锁来进一步细分：

集中式封锁方法、主副本封锁方法、分布式封锁方法。

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

1. 基于封锁的算法

（1）集中式封锁方法

- 网络中的一个站点被指定为**主站点**，**存放**对整个分布式数据库的**封锁表**，并且**负责**对**全系统事务**进行**封锁**。

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

（2）主副本封锁法：

a. 如果**每一个封锁数据项**有多个副本，则**指定一个副本为主副本**，**必须对主副本进行封锁**，以访问此特定的数据项。

例如，如果封锁数据X在站点1、2和3上都有副本，若站点1上 的X被选作X的主副本，则站点1就是X的主站点，那么所有事务要想访问X都必须在访问X的副本前获得在站点1上的锁。

b. 如果数据项没有副本(即每个数据项只有一个)，那么主副本封锁机制就在这些**数据项所在的站点上进行封锁管理**。

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

(3) 分布式封锁法：

锁的管理是由网络中所有站点共享的。在此情况下，一个事务的 执行包括多于一个的站点上的调度器的参与与协调。

每个本地调度器负责该站点上的封锁数据。

1 并发控制的概念和理论

1.5 并发控制机制的常用方法及其分类

- 2. 时标排序的方法

在基于时标排序(TO)的方法中，**按时标排序的方法组织事务的执行顺序**，以维护相互之间和内部的一致性。

排序是通过对**事务**和**数据项**进行**分配时标**来实现的。

这类算法包括基本TO算法、多版本TO算法和保守TO算法。

3. 混合的方法

有些**基于封锁的算法**中，也**使用了时标**，这样做主要是为了提高效率及并发的程度，称这种方式为混合算法。

本算法还没有在任何一个分布式数据库的商业或研究原型上实现。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

一、锁的类型、操作和粒度

基于封锁的并发控制方法是一种最常见的并发控制算法，其基本思想是事务访问数据项之前要对该**数据项封锁**，如果已经被其他事务锁定，就要等待直到那个事务释放该锁为止。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

一、锁的类型、操作和粒度

- 1. 锁的类型：
  - **共享锁**：Share锁，S锁或者读锁；
  - **排它锁**：eXclusive锁，X锁，拒绝锁或写锁；
  - **更新锁**：Update锁，U锁。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

■ 排它锁：

- 排它锁又称为**写锁，X锁**。事务T对数据对象A加上X锁，则允许T读取和修改A，其它任何事务都不能再对A加任何类型锁，直到T释放A上的锁。
- 保证了其它事务在T释放A上的锁之前不能再读取和修改A。

■ 共享锁

- 共享锁又称为**读锁，S锁**。若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其它事务只能再对A加S锁，不能加X锁，直到T释放A上的S锁。
- 保证了其它事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

■ 更新锁

- 更新锁又称为**Update锁，U锁**。在Update语句中的FROM<表名>后加HOLDLOCK，表示该表数据将被更新，对它只能加S锁(读取)，不能加U锁(更新)或X锁(写)。 **当执行更新时系统自动将U锁升级为X锁**。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

例：使用封锁机制解决丢失修改问题

T_1	T_2
① **Xlock A**	
② R(A)=16	
	Xlock A
③ A←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←A-1
⑤	W(A)=14
	Commit
	Unlock A

没有丢失修改：

- 事务T1在读A进行修改之前先对A加X锁
- 当T2再请求对A加X锁时被拒绝
- T2只能等待T1释放A上的锁后T2获得对A的X锁
- 这时T2读到的A已经是T1更新过的值15
- T2按此新的A值进行运算，并将结果值A=14送回到磁盘。避免了丢失T1的更新。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

例

T_1	T_2
① Slock A	
Slock B	
R(A)=50	
R(B)=100	
求和=150	
②	Xlock B
	等待
	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
	Commit
	Unlock B

使用封锁机制解决不可重复读问题

- 事务T1在读A，B之前，先对A，B加S锁；其他事务只能再对A，B加S锁， 而不能加X锁，即其他事务只能读A，B，而不能修改；
- 当T2为修改B而申请对B的X锁时被拒绝只能等待T1释放B上的锁；
- T1为验算再读A，B，这时读出的B仍是100，求和结果仍为150，即可重复读；
- T1结束才释放A，B上的S锁。T2才获得对B的X锁。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

例 使用封锁机制解决读“脏”数据问题

T_1	T_2
① **Xlock C**	
R(C)=100	
C←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	Commit C
	Unlock C

- 事务T1在对C进行修改之前，先对C加X锁，修改其值后写回磁盘；
- T2请求在C上加S锁，因T1已在C上加了X锁，T2只能等待；
- T1因某种原因被撤销，C恢复为原值100；
- T1释放C上的X锁后T2获得C上的S锁，读C=100。避免了T2读“脏”数据。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **2. 锁的选择：**
  - 数据项既可以读也可以写，则要用X锁；
  - 如果数据项只可以读，则要用 S锁。
- **3. 锁的操作**

  在读/写封锁模式中，存在**三种锁的操作**：
  - **Read_lock(x)**：**读封锁**也被称为共享封锁(shared-locked)，因为它允许其他事务读同一个数据项；
  - **Write_lock(x)**：**写封锁**也被称为排他封锁(exclusive-locked)，因为在同一个数据项上只有单独的一个事务排他地持有该锁。
  - **Unlock(x)**：解锁。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **4. 数据项的状态：**

  一个锁和一个数据项X相关联，Lock(X)有三种状态：

  read_locked读封锁、**Write_locked写封锁**和**未封锁**。

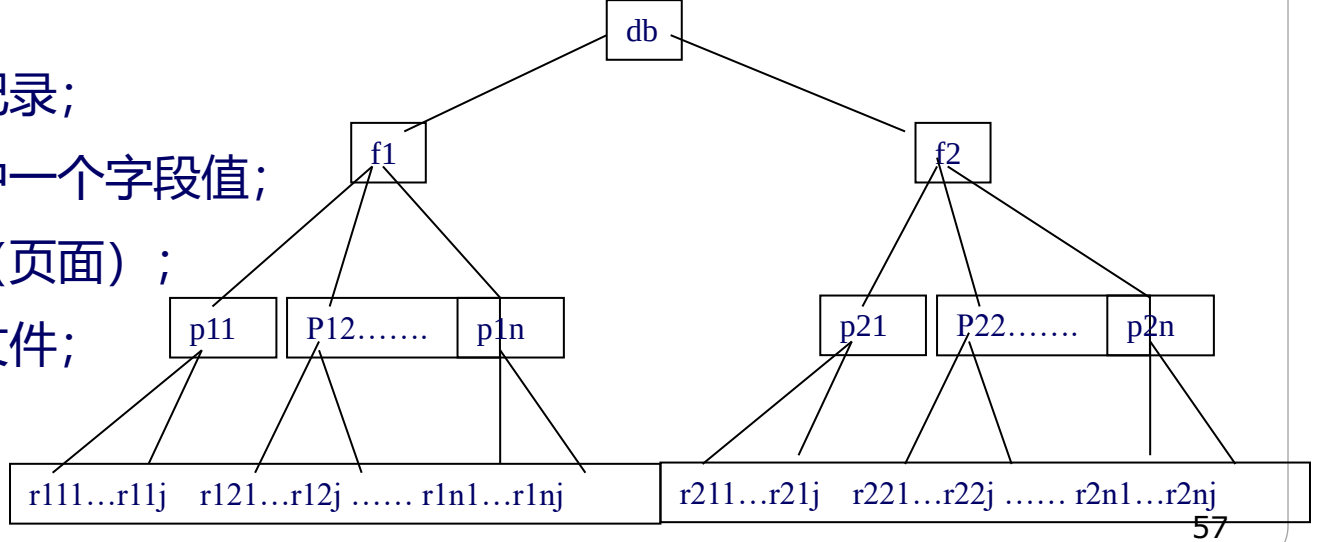
- **5. 实现读/写锁的三种操作的方法：**
  - 在**系统锁表**中记录关于锁的信息；
  - 系统锁表中**每条记录**有**四个字段**：

    <数据项名称，**锁状态**，读锁的数目，正在封锁该数据项的事务>；
  - 锁状态的值要么是**读封锁**，要么是**写封锁**，没有被封锁的数据项，在系统表中就没有记录。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **6. 锁的粒度**
  - **锁的粒度是指锁定数据项的范围**。所有的并发控制技术都假定，数据库是由许多命名的数据项组成。一个数据项可以是下列的任何一种：
    - 一条数据库记录；
    - 数据库记录中一个字段值；
    - 一个磁盘块（页面）；
    - 一个完整的文件；
    - 整个数据库。



2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- 7. 粒度对并发控制和恢复的影响
  - **粒度小，并发度高，锁开销大**
    - 数据项尺寸越小，数据项的数量越多，系统中的**锁管理器**处理更多数量的活动的锁，执行更多封锁和解锁操作，将导致系统开销增高。**锁表存储空间大**（如存储读写时间戳）。
  - **粒度大，并发度低，锁开销小**
    - 数据项尺寸越大，允许的并发程度越低。
    - 如果数据项的尺寸是磁盘块，封锁磁盘块中一条记录B的事务T必须封锁整个磁盘块。另外一个事务S如果要封锁另外一条不同的记录C，而C也在磁盘块中，由于磁盘块正在封锁中，S只能被迫等待。如果数据项的尺寸是一条记录的话，事务S就可以继续进行，不用等待了。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- 8. 如何来确定粒度
  - 取决于参与事务的类型。
  - 如果参与事务都**访问少量的记录**，那么选择**一个记录作为数据项粒度**较好；
  - 如果参与事务都**访问同一文件中大量的记录**，则最好**采用块或者文件作为粒度**，使得事务可以把这些记录 看作一个(或少数几个)数据项。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

二、封锁准则和锁的转换

- 1. 封锁准则/**锁的相容性规则**：

  采用共享/排他封锁模式时，系统实施下列规则：

  (1)事务T在执行任何**read_item(x)操作之前**，必须先执行read_lock(x)或者write_lock(x)操作（对数据项x加读锁或写锁）；

  (2)事务T在执行任何**write_item(x)操作之前**，必须先执行 write_lock(x)操作（对数据项x加写锁）；

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- 1. 封锁准则/**锁的相容性规则**:

(3)如果事务T **执行read_lock(x)操作**, 数据项x必须没有加锁或者已经加了读锁，否则事务T的这个操作不能执行;

(4)如果事务T **执行write_lock(x)操作**, 数据项x必须没有加锁，否则事务T的这个操作不能执行;

(5)事务T在**完成**所有**read_item(x)**和**write_item(x)操作之后**，必须执行unlock(x)操作;

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- 1. 封锁准则/**锁的相容性规则**:

(6)如果事务T**已经持有**数据项x上的一个**读锁**或者一个**写锁**，那么它不能再执行read_lock(x)操作；

(7)如果事务T**已经持有**数据项x上的一个**读锁**或者一个**写锁**，那么它不能再执行write_lock(x)操作；

(8)如果事务T**没有持有**数据项x上的一个**读锁**或者一个**写锁**，那么它不能执行unlock(x)操作。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

二、封锁准则和锁的转换

- 2. 锁的转换：

特定条件下，一个已经在数据项x上持有锁的事务T，允许将某种封锁状态转换为另外一种封锁状态。

① 一个事务T先执行了read_lock(x)操作，然后它可以通过执行write_lock(x)操作来**升级(Upgrade)该锁**。如果当事务T要执行write_lock(x)操作时，它是**持有数据项X上读锁的唯一事务**，那么该锁就可以被升级；否则，事务必须等待。

② 同样一个事务T也可以执行了write_lock(x)操作，之后它可以通过执行read_lock(x) 操作来**降级(Downgrade)该锁**。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

三、分布式数据库基本封锁算法

分布式数据库中封锁的难度比集中式大。因为在分布式数据库中，数据的分布导致执行的分布，**封锁消息将要在整个网络上传输**，其通信代价相当大；对多副本数据，要实现同步更新，原则上就得锁定所有副本。

因此，在**分布式数据库系统中封锁的方法**有许多种，常用的基本封锁算法有四种：**简单的分布式封锁方法**、**主站点封锁法**、**主副本封锁法**和**快照方法**。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **1. 简单的分布式封锁方法**
  - **数据更新**时，要将**同一数据的全部副本封锁**，然后对其进行更新，更新完成之后**解除全部上述封锁**。
  - **缺点是各站点间有相当大消息传输**，如果网络中有N个站点就有：
    - N个请求封锁的消息； N个封锁授权的消息；
      N个更新数据的消息； N个更新执行了的消息；
      N个解除封锁的消息；
    - 这有相当大的传输量，一般来说在分布式数据库系统中不宜采用此法。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **2. 主站点封锁法**
  - 主站点封锁法模拟集中式封锁方法，**选定一个站点定义为“主站点”**，负责系统全部封锁管理。
  - 所有站点都向主站点提出封锁和解锁请求，**所有封锁和解锁信息都被传送到那个主站点管理和保存**，然后由**主站点处理封锁事宜**。
  - 这种方式是集中式封锁方案的扩展。例如，如果所有的事务都遵守两阶段封锁协议，那么可以保证可串行化。
  - **优点**：它是集中式方案的简单扩展，因此不太复杂， 便于封锁管理，减少通信代价。
  - **缺点**：
    - 所有封锁请求都被送往单个站点，使那个**站点因超负荷造成瓶颈** 。
    - 主站点故障使系统瘫痪，封锁消息都在此，制约系统的可用性和可靠性。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **3. 主副本封锁法**
  - 不指定主站点，**对每个数据项指定一个主副本**，不同数据项的主副本放在不同的站点上。
  - 当处理程序对某个数据项进行操作时，**先对其主副本进行封锁**，再进行操作，对主副本封锁，**意味着对这个数据项的所有副本都被封锁**。
  - 主副本按使用情况，尽量就近分布。
  - 主副本方法不仅减轻了主站点的负荷，使得**各站点负荷比较均衡**，同时减少了站点间控制消息的传输量，是一个**比较好的并发控制方法**。
  - 缺点：**对只读操作要求过高**，可以采用**快照方法补充**。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

- **4. 快照方法**
  - 快照方法类似于视图的一种导出关系，但又与视图不同。它是实际数据的暂时凝聚，是数据库数据的一种存储方式。
  - 快照方法不考虑数据的复制，只考虑每一个数据的“主副本”和定义在这些“主副本”上的任意多个快照。
  - **快照可以定义为一个或多个“主副本”的部分拷贝**，也可以定义为某个或某些“主副本”的全拷贝。
  - 采用快照方法，可**完成复杂查询**而又**不影响更新**，因为快照中数据不受更新操作的影响，所以不会妨碍其他事务对有关数据的更新操作。

2 分布式数据库系统并发控制机制的封锁技术

2.1 基于封锁的并发控制方法概述

满足封锁规则不能保证产生串行化调度

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:= X+Y;	Y := Y + X;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

（a）两个事务T1和T2

初始值：X=20，Y=30

串行调度T1，T2的结果： X=50，Y=80

串行调度T2，T1的结果： X=70，Y=50

（b）T1和T2可能的串行调度的结果

T1	T2
read_lock(Y);	
read_item(Y); (Y=30)	
unlock(Y);	
	read_lock(X);
	read_item(X); (X=20)
	unlock(X);
	write_lock(Y);
	read_item(Y); (Y=30)
	Y := Y + X; (Y=50)
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);(X=20)	
X:= X+Y; (X=50)	
write_item(X);	
unlock(X);	

这个调度S的结果：

X=50，Y=50

(不可串行化)

（c）使用锁的一个不可串行化调度的结果

不遵守两阶段封锁协议的两个事务

2 分布式数据库系统并发控制机制的封锁技术

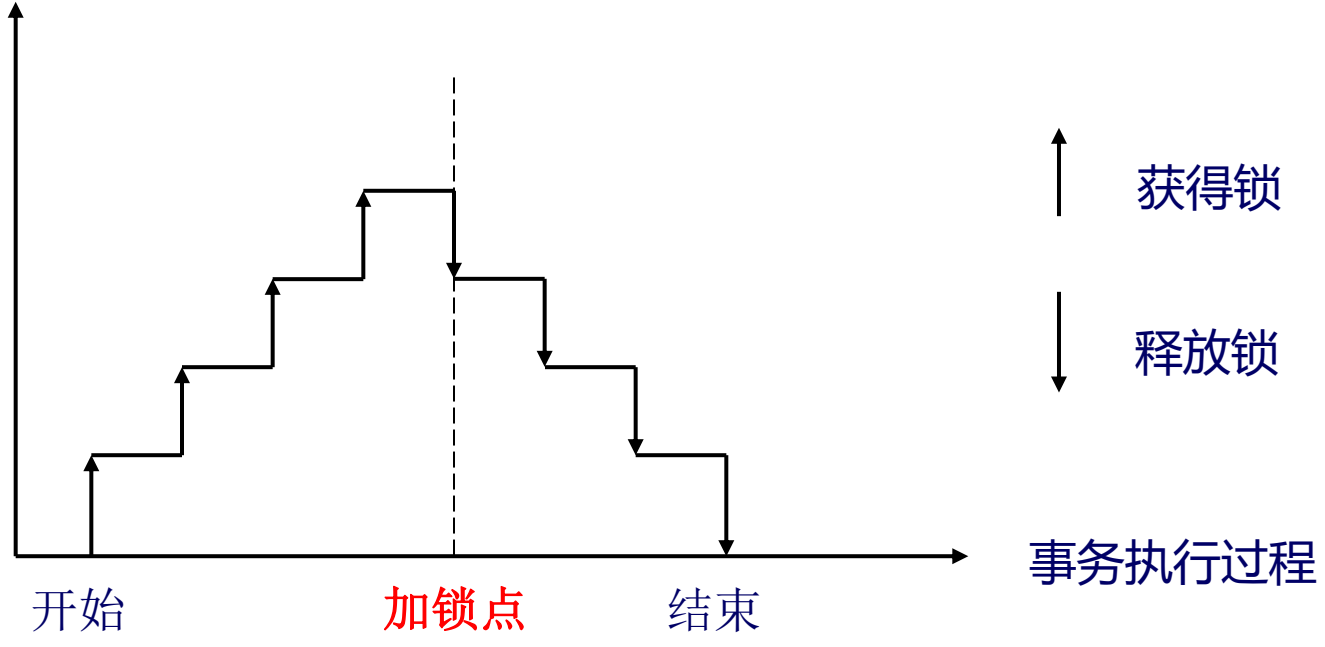
2.2 2PL协议（两阶段封锁协议）

一、基本2PL协议

- 如果一个事务**所有的封锁操作**(读封锁和写封锁)都**放在第一个解锁操作之前**，那么就说该事务遵守2PL协议。
- 事务的执行中封锁的管理分成**两个阶段**：
  - **第一阶段是扩张阶段或成长阶段**：事务只能**获得新的数据项锁**，而不能释放任何已持有的锁。
  - **第二阶段是收缩阶段或衰退阶段**：事务只能**释放已经持有的锁**，而不能获得任何的新锁。
- **封锁点**是指事务获得了它所要求的所有锁，且还没有开始释放任何一个锁的**时刻**。封锁点决定了一个事务生长阶段的结束和衰退阶段的开始。
- 如果允许锁的转换，那么**锁的升级** (从读锁转换到写锁)必须**在成长阶段完成**，而**锁的降级**(从写锁转换到读锁)必须**在锁的收缩阶段完成**。

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议（两阶段封锁协议）



两阶段封锁协议

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议（两阶段封锁协议）

一、基本2PL协议

- 一个很有名的理论[Eswaran et al.,1976]是：

  遵循了两段封锁规则的并发控制算法所产生的调度都是可串行化的。

- 可以证明，**如果调度中的每个事务都遵守两阶段封锁协议**，就可以保证**该调度是可串行化的**，不再需要检测调度的可串行性。

- **基本2PL协议保证事务执行的可串行性**。

- 实施两阶段封锁规则的封锁机制，也就实施了调度的可串行性。

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议（两阶段封锁协议）

T_1'	T_2'
read_lock(Y);	**read_lock(X);**
read_item(Y);	read_item(X);
write_lock(X);	**write_lock(Y);**
unlock(Y);	**unlock(X);**
read_item(X);	read_item(Y);
X:= X+Y;	Y := Y + X;
write_item(X);	write_item(Y);
unlock(X);	**unlock(Y);**

遵守2PL封锁协议的两个事务T1’和T2’

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议（两阶段封锁协议）

二、保守的、严格的、严酷的2PL协议

- 1. 保守的2PL协议
  - 要求事务在**开始执行之前**就**持有所有它要访问的数据项上的锁**。
  - 事务要**预先声明它的读集和写集**。
  - 一个事务的读集就是该事务要读的所有数据项的集合；一个事务的写集就是该事务要写的所有数据项的集合。
  - 如果有任何事先声明需要的数据项不能被封锁，则事务就不能封锁任何一个数据项；换句话说，**事务必须一直等待，直到所有的数据项都是可封锁的**。
  - **保守2PL是一个无死锁的协议**。但是在大多数情况下，事先声明读集和写集都是不可能的，所以保守2PL很难应用于实际。

2 分布式数据库系统并发控制机制的封锁技术

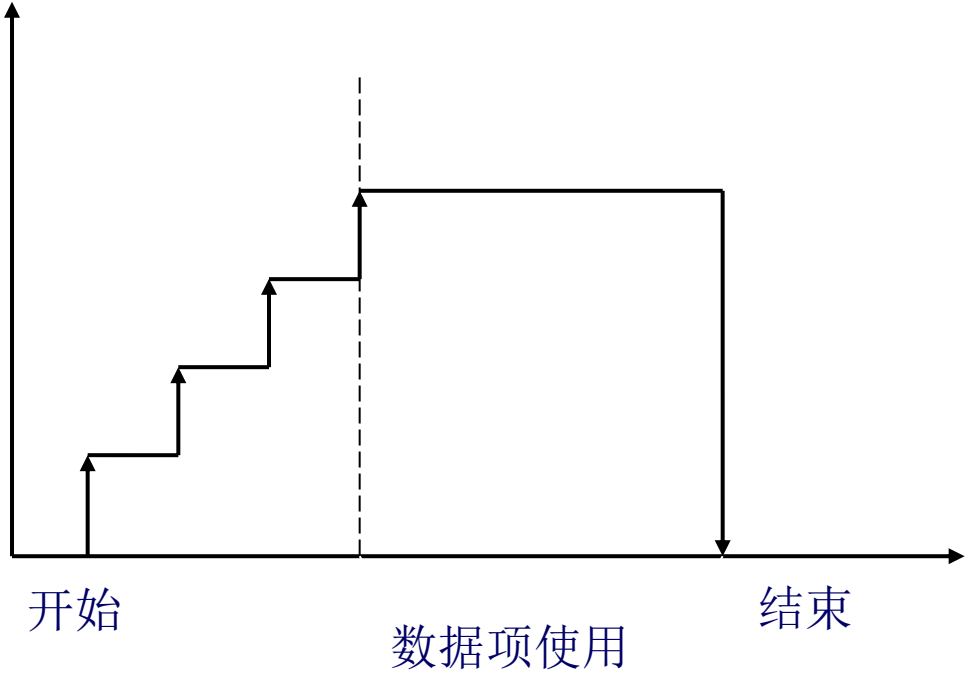
2.2 2PL协议（两阶段封锁协议）

二、保守的、严格的、严酷的2PL协议

- 2. 严格的2PL协议
  - 它是2PL的变型。
  - 事务在**提交或者撤销之前**，绝对**不释放任何一个写锁**；
  - 事务**结束时**（提交或者撤销），**同时释放所有的锁**。
  - 因此，除非事务T已经提交，否则任何其他事务都不可以读或写由事务T所写的数据项，从而产生了一个对可恢复性而言的严格的调度。
  - 严格2PL是不能避免死锁的。

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议（两阶段封锁协议）



严格2PL(Strict Two-phase Locking)协议

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议

二、保守的、严格的、严酷的2PL协议

- 3. 严酷的2PL协议
  - 严酷2PL协议是严格2PL协议的一种**更具限制性**的变型。
  - 在严酷2PL协议中，事务T在**提交或撤销之前，不能释放任何一个锁（写锁或者读锁），**因此它比严格2PL协议更容易实现。

2 分布式数据库系统并发控制机制的封锁技术

2.2 2PL协议

二、保守的、严格的、严酷的2PL协议

- 4. 保守2PL协议与严酷2PL协议之间的区别：
  - 保守2PL协议要求事务必须在**开始之前封锁**它所需要的**所有数据项**，因此，一旦**事务开始就处在收缩阶段**；
  - 严酷2PL协议要求直到**事务结束**（提交或者撤销）后才**开始释放锁**，因此，**事务一直处于扩张阶段**，直到结束。

2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

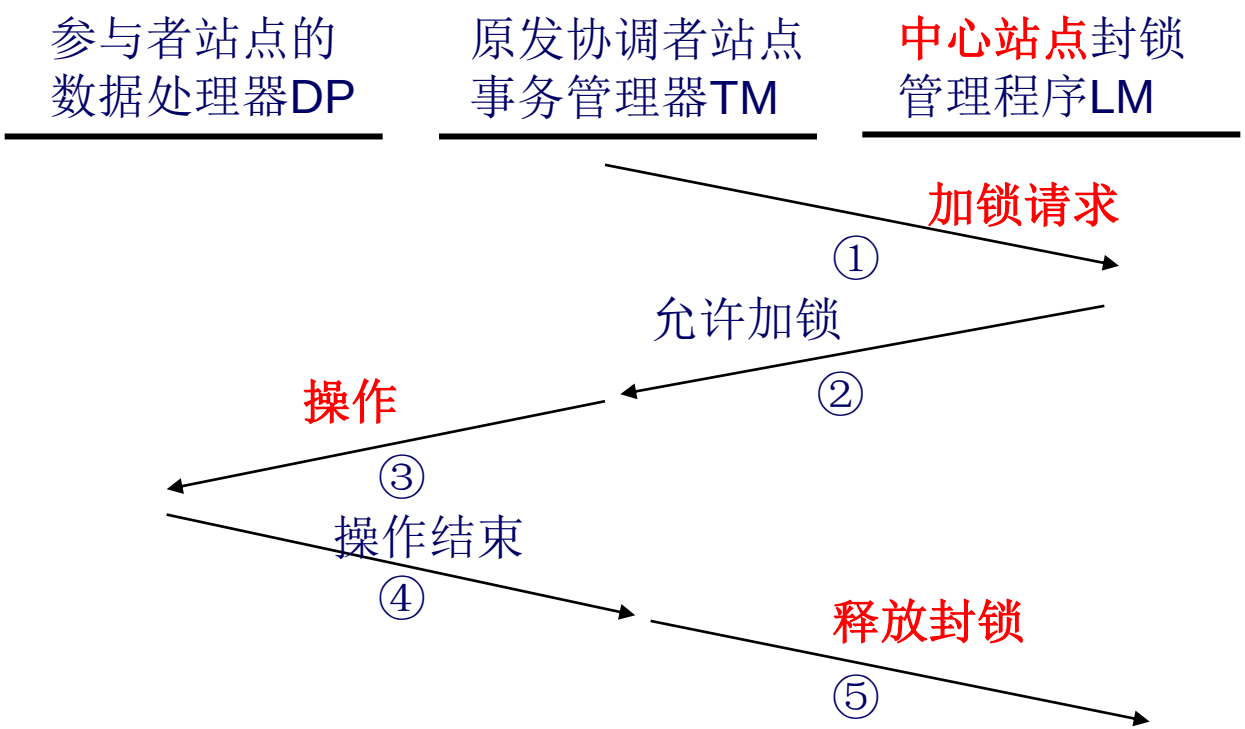
一、集中式2PL（主站点2PL）的实现方法

- **只有一个站点即主站点拥有封锁管理程序，其他站点上的事务管理程序**是**同这个封锁管理程序进行通信**，而不是同它们自己站点上的封锁管理程序进行通信。这种方法也被称作**主站点2PL算法**。
- 合作站点上的事务执行方式是通过集中式两段锁(C2PL)算法实现的。**通信发生**在事务被初始化站点上的**协调者的事务管理程序**与主站点上的**封锁管理程序**，以及其他参与站点上的**数据处理器(DP)**之间。参与站点是指操作被执行的站点。

2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

一、集中式2PL（主站点2PL）的实现方法



集中式2PL的通信结构

–协调事务管理器(coordinating TM)：事务原发站点

–数据处理器(data processor，DP)：其他参与站点

–中心站点LM：主站点锁管理器

中心站点LM不需要向DP发送操作

2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

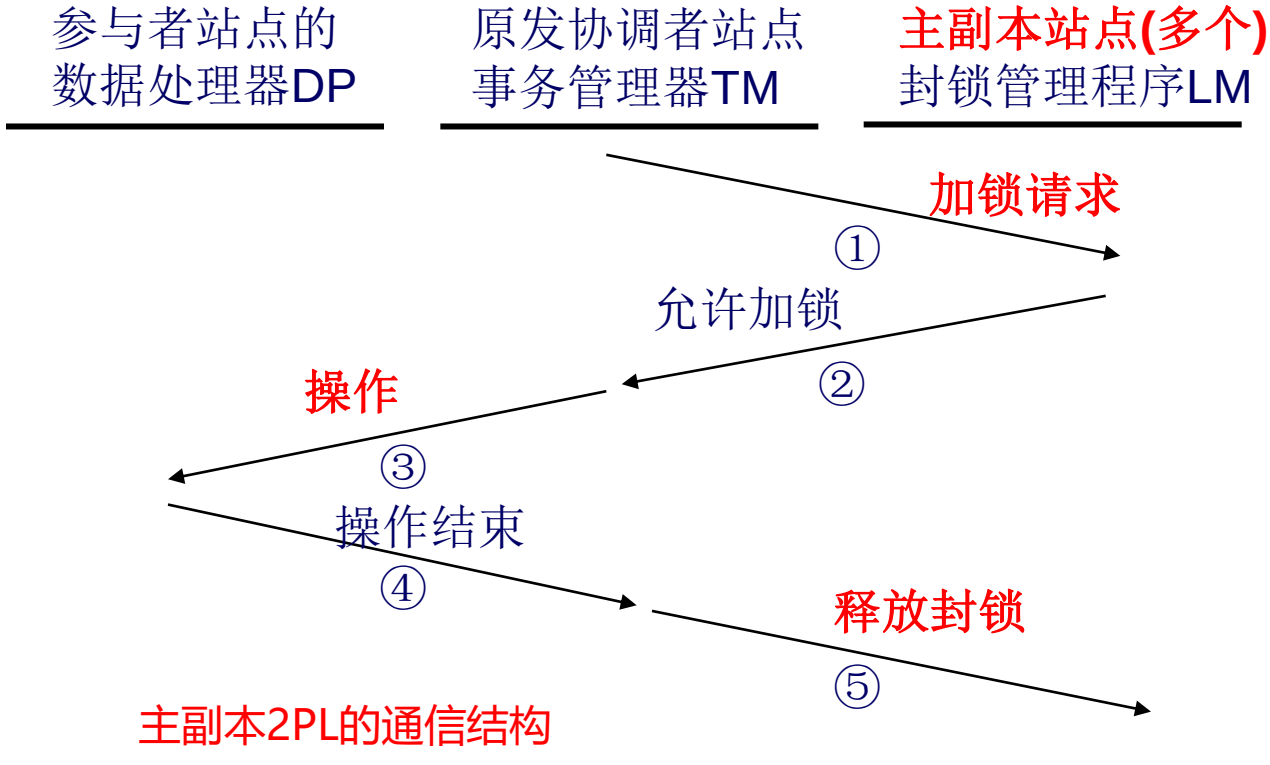
二、主副本2PL的实现方法

- 主副本2PL(PC2PL)实现方法是对集中式2PL实现方法的**向前扩展**。
  - 在**一些站点上实现封锁管理**，**每一个封锁管理程序**管理所指定的**一组封锁单元上的锁**。
  - **事务管理程序**向**封锁管理程序LM**发出对某一特定封锁单元的封锁和释放锁的请求。对主副本封锁**意味着对这个数据项的所有副本都被封锁**。这一算法把**每一个数据项的副本都作为其主要副本**。
  - 主副本2PL算法与集中式2PL的差别：
    必须先为**每一个数据项确定一个主副本站点**，然后事务管理程序再向**主副本站点上的封锁管理程序**发送**封锁或释放锁的请求**。
  - 主副本2PL减少了主站点上的负载，且不会增加事务管理程序与封锁管理程序之间的通信。

2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

二、主副本2PL的实现方法



主副本2PL的通信结构

2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

三、分布式2PL的实现方法

- 分布式两阶段封锁特点
  - 在**每个站点实现封锁管理程序的有效性**。
  - 如果数据没有被复制，分布式两段锁降级为主副本两阶段锁算法。
  - 如果数据已被复制，**事务将执行“读一个/写全部” (ROWA)副本控制协议**。

2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

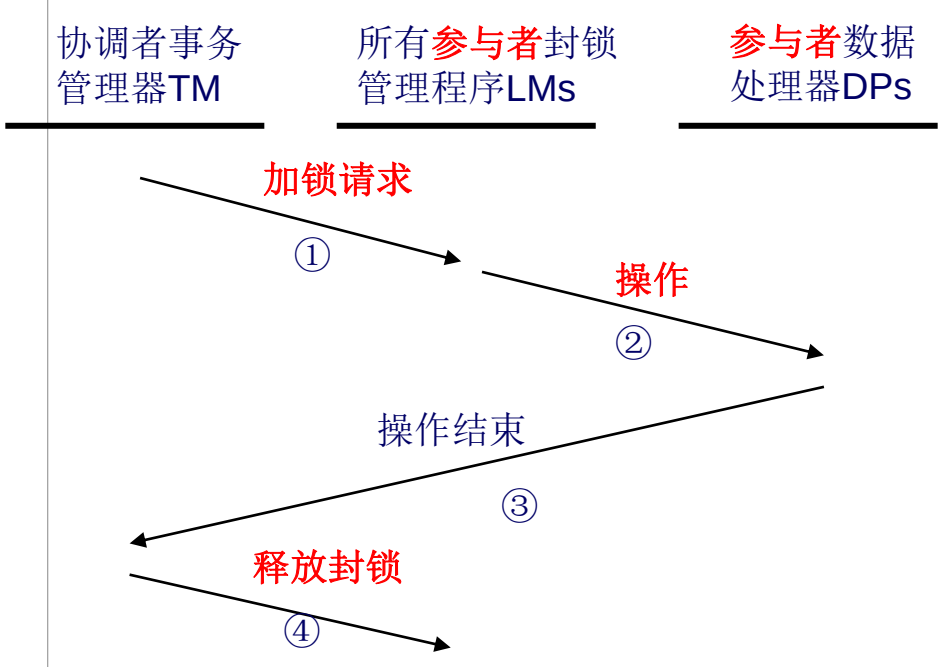
三、分布式2PL的实现方法

- 分布式2PL事务管理算法与C2PL-TM(集中式两阶段锁事务管理算法)相似，但有两处重大的改进。
  - 在集中式两段锁事务管理中，事务被初始化站点上的**协调者事务管理程序**向中心站点封锁管理程序LM**发送的封锁信息**，在分布式两段锁事务管理中，**将封锁信息发送给所有参与站点的封锁管理程序LM**；
  - 另外不同之处是**操作**不通过协调者事务管理程序传到参与者的数据处理器，而是**通过参与者的封锁管理程序**传到**参与者的数据处理器。参与者的数据处理器**向**协调者的事务管理程序发送“操作结束”信息。**

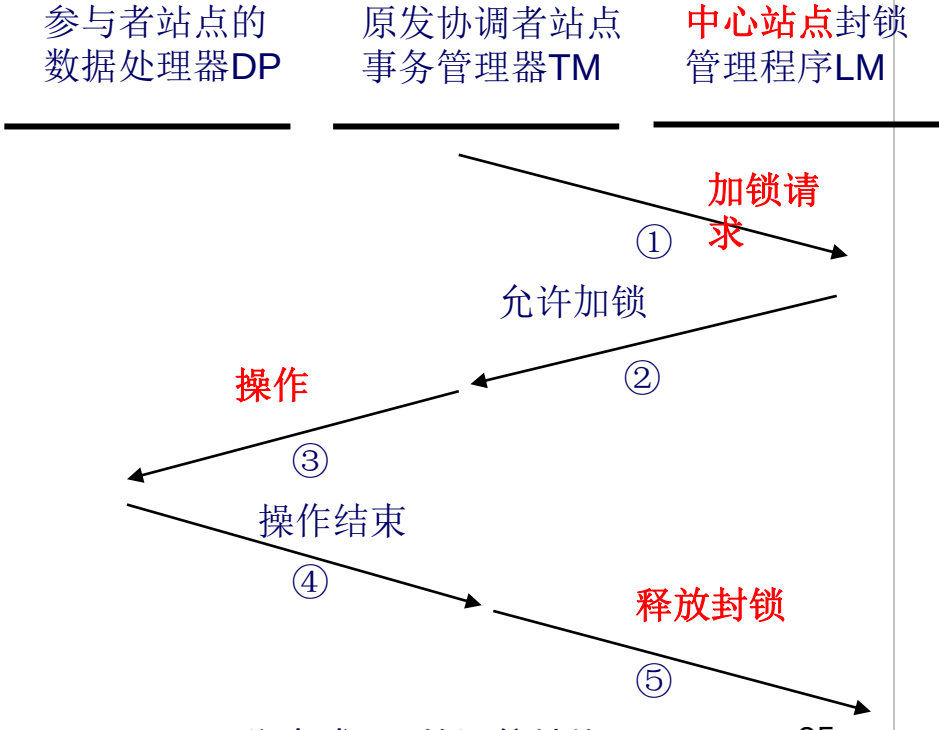
2 分布式数据库系统并发控制机制的封锁技术

2.3 2PL协议的实现方法

三、分布式2PL的实现方法



分布式2PL的通信结构



集中式2PL的通信结构

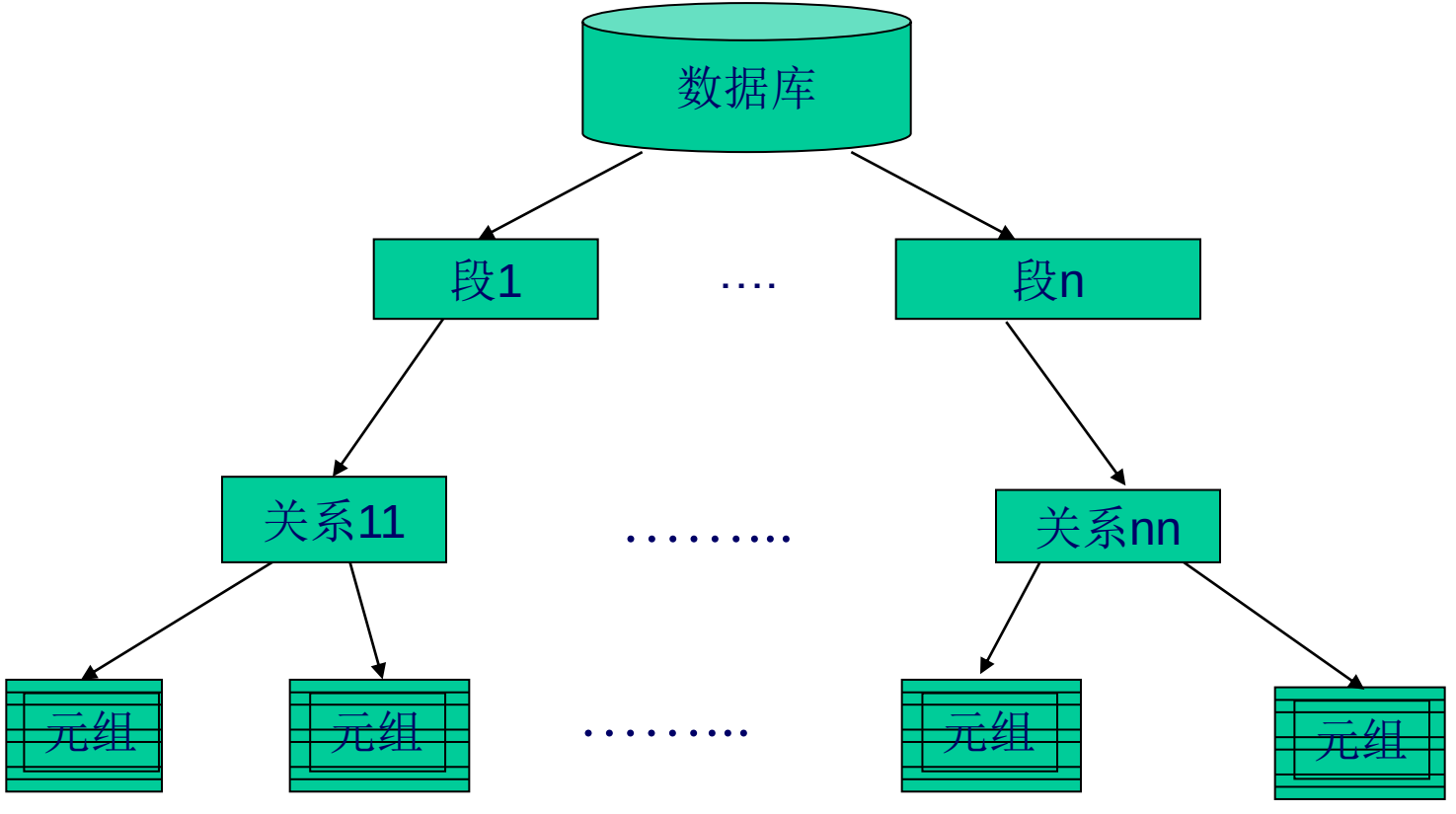
2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- 1. 多粒度封锁
  - 封锁的粒度不是单一的一种粒度，而是有多种粒度。可以定义**多粒度树**，**根节点**是整个数据库，**叶节点**表示最小的封锁粒度。
- 2. 多粒度封锁的封锁协议
  - 多粒度封锁的封锁协议允许多粒度树中的每个节点被独立地封锁。**对一个节点封锁**意味着**这个节点所有后裔节点也被加同类型的锁**。
  - 在多粒度封锁中一个数据项可能以两种方式封锁，**显式封锁**和**隐式封锁**。显式封锁是因事务的要求直接对数据对象封锁；隐式封锁是该数据对象没有直接独立封锁，是由于其上级节点封锁而使该数据对象加上了锁。
  - 多粒度封锁方法中，**显式封锁和隐式封锁的效果是一样**的，因此系统检查锁冲突时不仅要检查显式封锁还要检查隐式封锁。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

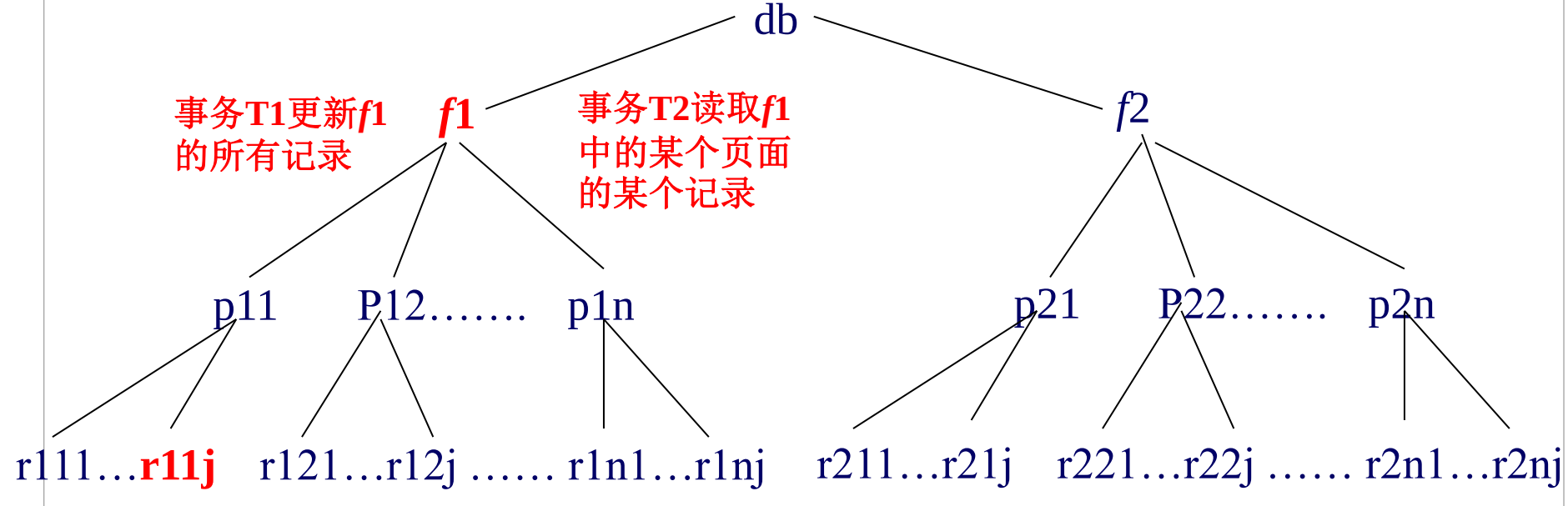


多级粒度树

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

下图给出了一个简单的粒度层次。它是一个包含两个文件的数据库，其中每个文件包含若干页，每页又包含若干记录。

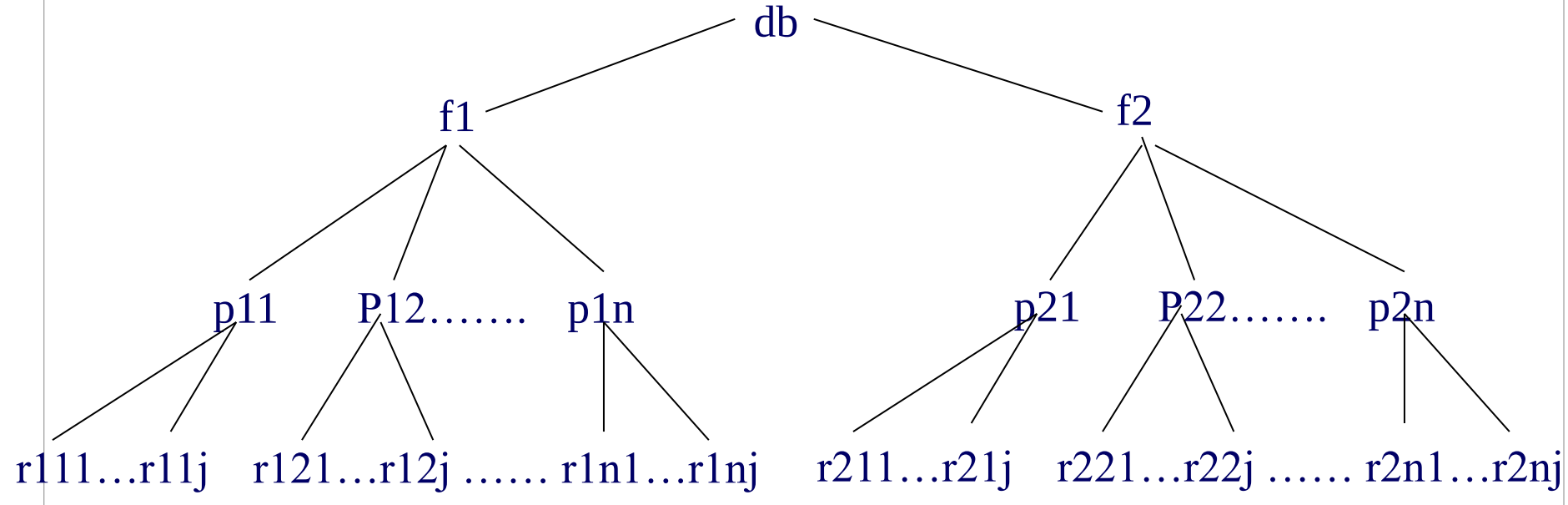


用来说明多粒度级别封锁的粒度层次结构

2 分布式数据库系统并发控制机制的封锁技术

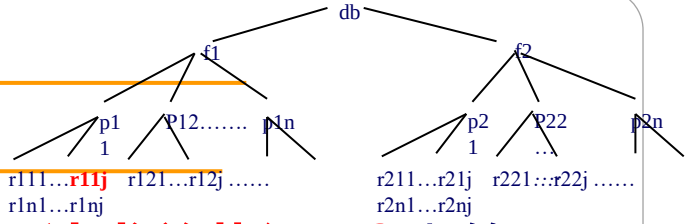
2.4 多粒度封锁与意向锁

下图给出了一个简单的粒度层次。它是一个包含两个文件的数据库，其中每个文件包含若干页，每页又包含若干记录。



2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁



- 例：假定事务T_1要**更新**文件f_1中的所有记录，**T_1请求并获得了f_1**上的一个**写锁**。那么f_1下面的**页面和记录**就获得了**隐式写锁**。
  - 若此刻事务T_2想从f_1中的某个页面中**读**某个记录，那么T_2就要申请该记录上的一个**记录级读锁**。但是这个数据库系统(锁管理器)必须确认这个读锁和已经存在锁的**相容性**，**确认方法是自下而上遍历该树**：从记录到页，到文件最后到数据库。如果任意时刻这些项中的任意一个上存在冲突锁，那么对记录的封锁请求被拒绝，T_2被阻止并且必须等待。
  - 如果**事务T_2的请求比事务T_1的请求先到**，则T_2对记录提出的共享**记录锁**得到批准。但是当T_1请求**文件级锁**时，让**锁管理器去检查节点f_1的所有后代节点(页和记录)**，查看是否存在封锁冲突则是十分困难的。这会使查询效率**非常低**，违背多粒度级别封锁目的。
  - 为此引进了一种**新型锁**，称为**意向锁**。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- **3. 意向锁**
  - 如果对一个节点加意向锁，则说明该节点的下层节点正在被封锁：**对任一节点封锁时，必须先对它的上层节点加意向锁**。
  - 意向锁的思想是：对于一个**事务**，在多粒度树中沿着从**根节点**到**目标节点**的**路径**，指出将在该目标节点**某个后代节点**上需要锁的类型(共享或排他锁)。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- 三种类型的意向锁

（1）意向共享锁(IS)：

指示在其**后代节点**上将会**请求共享锁**，即如果对某个数据对象加IS锁，表示它的后代节点拟加共享锁。

– 例如，要对某个元组加S锁，则要首先对关系和数据库加IS锁。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- 三种类型的意向锁

(2) 意向排它锁(IX):

指示在其**后代节点**上将会**请求排他锁**，即如果对某个数据对象加IX锁，表示它的后代节点拟加排他锁。

– 例如，要对某个元组加X锁，则要首先对关系和数据库加IX锁。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- 三种类型的意向锁

（3）共享意向排它锁(SIX)：

指示**当前节点处在共享方式的 封锁中**，但是在它的某些**后代节点中将会请求排他锁**。即如果**对一个数据对象加SIX锁,表示对它加共享锁，再加IX锁（SIX=S+IX）**。

– 例如：对某个关系（即表）加SIX锁，则表示该事务要读整个表（加S锁），同时会更新个别元组（加IX锁）。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

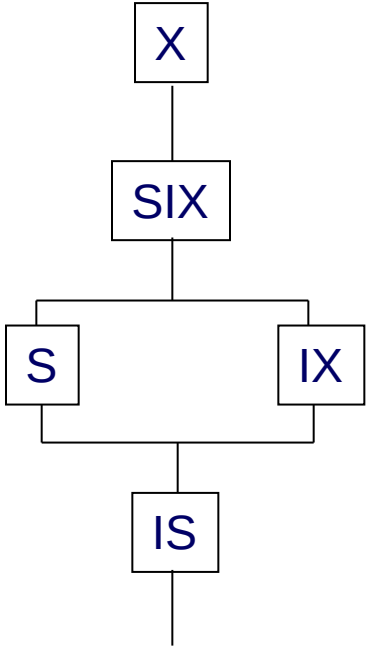
锁的相容矩阵：对称的矩阵

T_1 \ 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：对其它锁的排斥程度



(b)锁的强度的偏序关系

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- **4. 多粒度封锁协议的规则**
  1. 必须**遵守锁的相容性规则**；
  2. 必须**首先封锁树的根节点**，可以用任何一种方式的锁；
  3. 只有当节点N的**父节点**已经被事务T以**IS或IX方式封锁后**，**节点N**才可以被事务T以**S或者IS方式封锁**；
  4. 只有当节点N的**父节点**已经被事务T以**IX或SIX方式封锁后**，**节点N**才可以被事务T以**X, IX或者SIX方式封锁**；
  5. 只有当事务T还没有释放任何节点时，事务T才可以封锁一个节点；
  6. 只有当事务T当前没有封锁节点N的任何子节点时，T才可以为节点N解锁。

规则1简单地陈述了不能允许冲突锁。
规则2、3、4则陈述了一个事务以任意一种锁方式封锁给定节点的条件。
规则5和6实施了**2PL规则**，以产生可串行化调度。

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

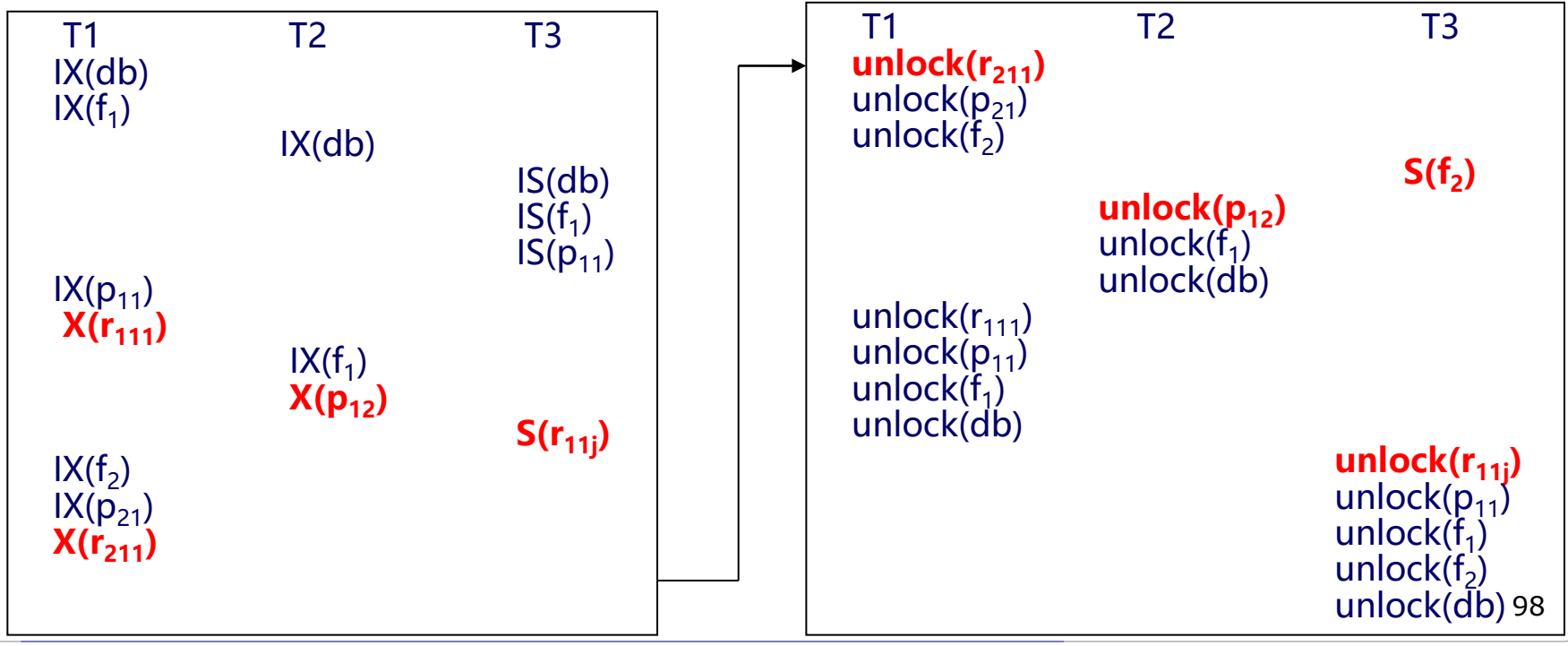
- 总结
  - 具有意向锁的多粒度加锁方法中，任意事务T要对一个**数据对象加锁**，**必须先对它的上层节点加意向锁**。
  - **申请封锁**时应该**按自上而下的次序**进行，**释放锁**时则应该**按自下而上的次序**进行。
  - 具有意向锁的多粒度加锁方法提高了系统的并发度，减少了加锁和释放锁的开销，它已经在实际的DBMS系统中广泛应用，例如新版Oracle数据库系统就采用了这种封锁方法。

- 例如：事务T1要对关系R1加S锁
  - 首先对数据库加IS锁
  - 检查数据库和关系R1是否已加了不相容的锁(X或IX)
  - 不再需要搜索和检查R1中的元组是否加了不相容的锁(X锁)

2 分布式数据库系统并发控制机制的封锁技术

2.4 多粒度封锁与意向锁

- 例 考虑以下三个事务：1. T_1 要**更新**记录r_{111}和记录r_{211}；2. T_2要**更新**页p_{12}中的所有记录； 3. T_3要**读取**记录r_{11j}和整个f_2文件。下图给出了对这三个事务的一种可能的可串行化调度，其中只列出了锁操作。



3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

一、活锁、死锁和全局死锁

- 封锁技术易于理解也很实用，利用封锁技术，可以避免由于并发冲突操作引起的数据错误，但也可能产生其他一些问题。
- 例如可能存在**某个事务永远处于等待状态**，得不到执行的机会，这种现象称为**活锁**。
- 活锁问题不仅在DBMS中可能出现，在一般的OS中也会遇到。

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

一、活锁、死锁和全局死锁

■活锁的情形：

- 事务T1封锁了数据R，事务T2又请求封锁R，于是**T2等待**。
- T3也请求封锁R，当T1释放了R上的封锁之后系统首先批准了T3的请求，**T2仍然等待**。
- T4又请求封锁R， 当T3释放了R上的封锁之后系统又批准了T4的请求，…，
- **T2有可能永远等待**。

T_1	T_2	T_3	T_4
lock R	.	.	.
.	lock R	.	.
.	等待	Lock R	.
Unlock	等待	.	Lock R
.	等待	Lock R	等待
.	等待	.	等待
.	等待	Unlock	等待
.	等待	.	Lock R
.	等待	.	.

活 锁

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

一、活锁、死锁和全局死锁

- 解决活锁的一种简单的方法是采用“**先来者先执行**”的控制策略，也就是简单的**排队方式**。

  当**多个事务**请求**封锁同一个数据对象**时，按**请求封锁的先后次序**对这些事务排队； 该数据对象上的锁一旦释放，首先批准**申请队列**中第一个事务获得锁。

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

一、活锁、死锁和全局死锁

- 死锁定义：
  - 在两个或多个事务的集合中，当每个事务T都在等待已经被该集合中另一个事务T’封锁的数据项时，即该集合中**每个事务**都在**等待**该集合中**另外一个事务释放**它所需要的**数据项上持有的锁**，它才能继续执行下去，结果**任何一个事务都无法继续执行**，这种现象称为**死锁**(deadlock)。

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

一、活锁、死锁和全局死锁

■死锁的形成：

- **事务T1封锁了数据R1，事务T2封锁了数据R2**；
- 事务T1又**请求**封锁数据R2，因为T2已封锁了R2，于是T1等待T2释放R2上的锁。
- 事务T2又**申请**封锁R1，因为T1已封锁了R1，T2也只能等待T1释放R1上的锁；
- 这样就出现了**T1在等待T2**，而**T2又在等待T1局面**；
- T1和T2两个事务永远不能结束，形成死锁。

T_1	T_2
lock R_1	•
•	Lock R_2
•	•
Lock R_2.	•
等待	•
等待	Lock R_1
等待	等待
等待	等待
	•

死　锁

3 分布式数据库系统中的死锁处理

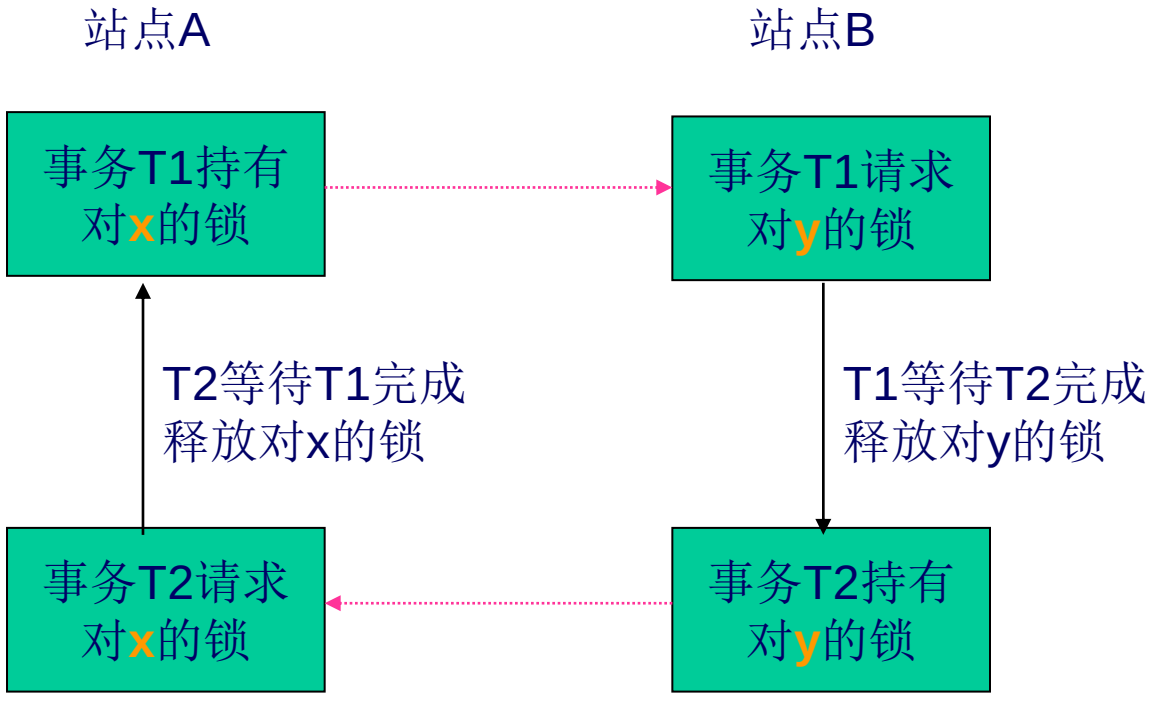
3.1 全局死锁与等待图

一、活锁、死锁和全局死锁

- 死锁发生的条件
  - 互斥条件：事务请求对资源的独占控制；
  - 等待条件：**事务已持有分配给它的资源，又去申请并等待别的资源**；
  - 非抢占条件：直到资源被持有它的事务释放前，**不可能将资源强制从持有它的事务夺去**；
  - 循环等待条件：**存在**事务互相等待的**等待圈**；
- 死锁分类
  - **局部死锁**：仅在**一个站点**上发生的死锁。
  - **全局死锁**：涉及**多个站点**的死锁(即**等待圈由多个站点组成**)。

3 分布式数据库系统中的死锁处理

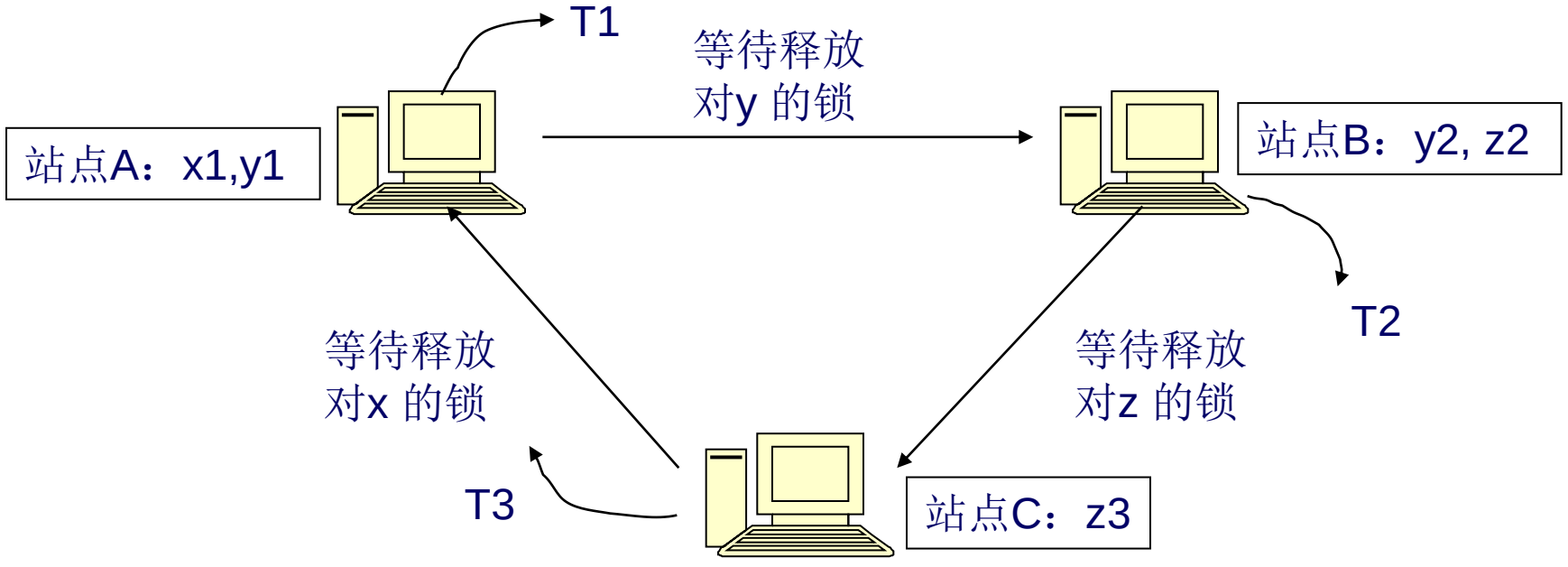
3.1 全局死锁与等待图



相互等待引起的全局死锁

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图



站点A：存储x和y的副本，　发出事务T1：read(x),write(y)
站点B：存储y和z的副本，　发出事务T2：read(y),write(z)
站点C：存储z的副本，　发出事务T3：read(z),write(x)

多副本引起的三个站点间的死锁

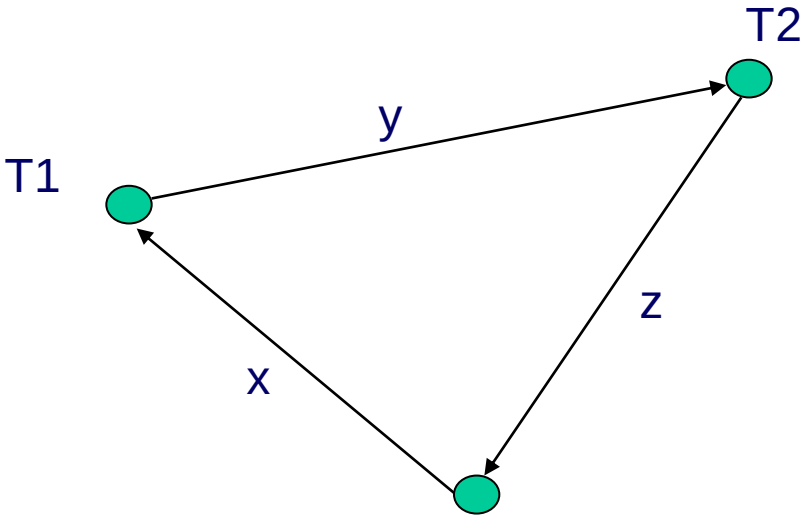
3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

- 等待图
  - 分析**死锁**的有用工具是**等待图(WFG)**。等待图是一种用来表示事务之间相互等待关系的**有向图**，其中指出哪一个事务在等待其他哪个或哪些事务释放锁。
  - 图中**节点表示事务**，带有箭头的**有向边表示“等待”关系**，箭头的方向就是等待方向。如果事务Ti等待事务Tj正拥有的锁，则从Ti向Tj画一条有向边，有向边旁也可标注数据项名，以表示等待的是对哪个数据项的锁。
  - **当且仅当等待图中至少包含一个回路，则存在一个死锁。**
- 等待图分类
  - 局部等待图(LWFG)
  - 全局等待图(GWFG)：是所有局部等待图的并。

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图



GWFG等待图

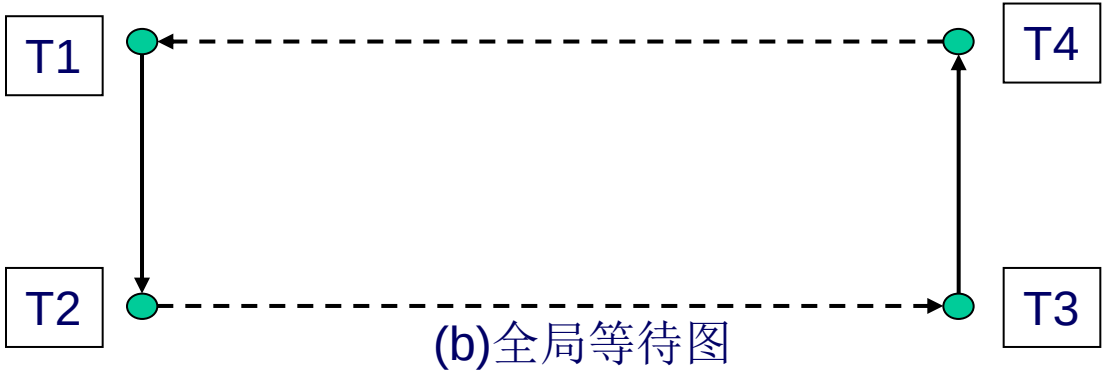
T1等待T2释放对y的共享锁(s)
T2等待T3释放对z的共享锁(s)
T3等待T1释放对x的共享锁(s)
全局等待图中包含回路，故存在死锁。

站点A：存储x和y的副本，事务T1：read(x),write(y)
站点B：存储y和z的副本，事务T2：read(y),write(z)
站点C：存储z的副本，事务T3：read(z),write(x)

3 分布式数据库系统中的死锁处理

3.1 全局死锁与等待图

(a)局部等待图



(b)全局等待图

例. 事务间有等待关系：

T1→T2→T3→T4→T1

通过检查全局等待图可以找出死锁。

LWFG和GWFG之间的不同

3 分布式数据库系统中的死锁处理

3.2 死锁的预防方法

预防死锁的方法是一种比较保守的方法。这类方法的想法是**当有发生死锁危险时，就中止并重新启动其中一个事务或让该事务等待**，但如果允许有等待的话，则绝不可能发生死锁。

- 形成死锁的原因：

  多个事务并发执行，互相等待另一事务所持有的锁，且形成等待回路而引起的。如果事务T1请求以资源，而该资源被另一事务T2所持有时，则进行一次**预防性测试**。

  若测试结果表明有死锁的危险时，则或者是不让T1进入等待状态**重新启动T1**，或者是**中止并重新启动T2**。

- 预防性测试的方法：

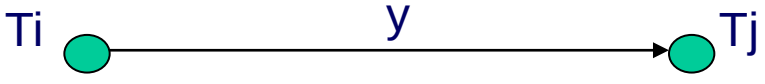
  按**事务开始时间排序**，预防死锁的方法有两种。

3 分布式数据库系统中的死锁处理

3.2 死锁的预防方法

- 非占先权方法(**排队在先者可能失去优先**)----也称**等待-死亡模式**。
  - 如果Ti对已被**Tj封锁的一数据项**请求封锁的话，则只有在**Ti**比Tj**年老**时（Ti<Tj），才允许**Ti等待**(失去优先)。

    如果**Ti**比Tj**年轻**（Ti>Tj），则**Ti被终止**并带有同一时间戳**重新启动**。
  - 这个方法的理论基础是：**最好总是重新启动较年轻的事务。允许较年老的事务**去**等待**已持有资源的**较年轻的事务**，但不允许较年轻的事务去等待较年老的事务。

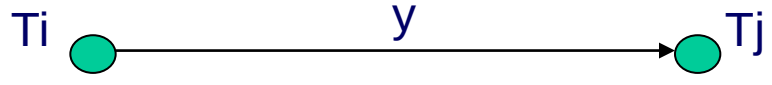


- 如果Ti<Tj （Ti比Tj年老），则允许**Ti等待**(失去优先)。
- 如果Ti>Tj （Ti比Tj年轻），则**Ti被终止**并带有同一时间戳**重新启动**。

3 分布式数据库系统中的死锁处理

3.2 死锁的预防方法

- 占先权方法(**排队在先者绝对优先**)----也称**受伤-等待模式**。
  - 如果Ti对已被Tj封锁的数据项请求封锁的话，则T_i比Tj年老（Ti<Tj），则**Tj被终止并带有同一时间戳重新启动**，而允许Ti获得锁执行；否则，只有在T_i比T_j年轻时（T_i>T_j），才允许T_i等待。
  - 这个方法的理论基础是：**要求终止较年轻的事务**，并且允许年老的事务绝对优于年轻的事务，因而只有**年轻的等待年老的**。



- 如果Ti<Tj （Ti比Tj年老），则**Tj被终止**并带有同一时间戳**重新启动**。
- 如果Ti>Tj （Ti比Tj年轻），则允许**Ti等待**。

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

一、死锁检测和解决的一般方法

- 死锁检测通过对**全局等待图中回路的形成**进行研究来实现。如果系统处于死锁状态，就必须撤销一些引起死锁的事务，以**打破GWFG中的回路**。
- 但是对一组陷入死锁的事务**选择最小总开销打破死锁**，可以考虑一些会影响这一**选择的因素**：
  - 调度器力图**避免撤销那些几乎要完成的事务**。
  - **最好撤销那些包含多个回路的事务**，因为撤销一个事务会打破包含该事务的所有回路。

一般地，应该**避免选择那些已经运行了很长时间的事务**，以及避免选择已经执行了许多更新操作的事务，应该**选择那些还没有执行许多更新操作的事务**。

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

二、分布式死锁检测方法

分布式死锁检测有三种方法：集中式、层次式及分布式死锁检测法。

- 集中式死锁检测法
  - **选择一个站点**负责整个系统的死锁检测，**死锁检测器**放到这个站点。
  - 其他每个站点的锁管理器**周期性**地将本站点的**LWFG**传送给死锁检测器，**死锁检测器构造GWFG**，并**在其中寻找回路**。
  - 或者，其它每个站点上的锁管理器周期性地把**记录**本站点上事务的开始时间，对锁的持有、请求情况变化的**动态表**，发送给负责处理封锁的站点，由它维护一张**全局封锁动态表**，**形成GWFG**，并**在其中寻找回路**。

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

表6.1 全局封锁动态表

事务	请求	持有	开始时间	站点
T1	X(D), S(B)	S(A)	9:30	1
T2	X(B), S(A)	S(D)	10:01	2
T3	X(C), X(E)	S(B), S(A)	10:20	3
T4	S(A)		10:21	3

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

- 集中式死锁检测法
  - 如果形成的**GWFG中至少包含一条回路**，它将**选择一个或者多个事务把它们撤销并恢复**，**释放被占资源**，使得其它事务继续运行。
  - 选择的标准是尽可能使**撤销并恢复的代价最小**，视具体系统而定。
  - 例如：
    - 可以撤销年轻的事务；
    - 可以撤销占有较少资源的事务；
    - 可以撤销具有最短运行时间的事务；
    - 可以撤销具有最长运行时间的事务等。

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

- 层次式死锁检测法
  - 以**层次方式组织**成员DBMS中的**死锁检测器**。
  - 死锁发生时，常常**只涉及部分站点**。
  - 层次检测的层次结构**与网络拓扑结构有关**。
  - **减少了对中心站点的依赖性**，从而**减少了传输开销**。

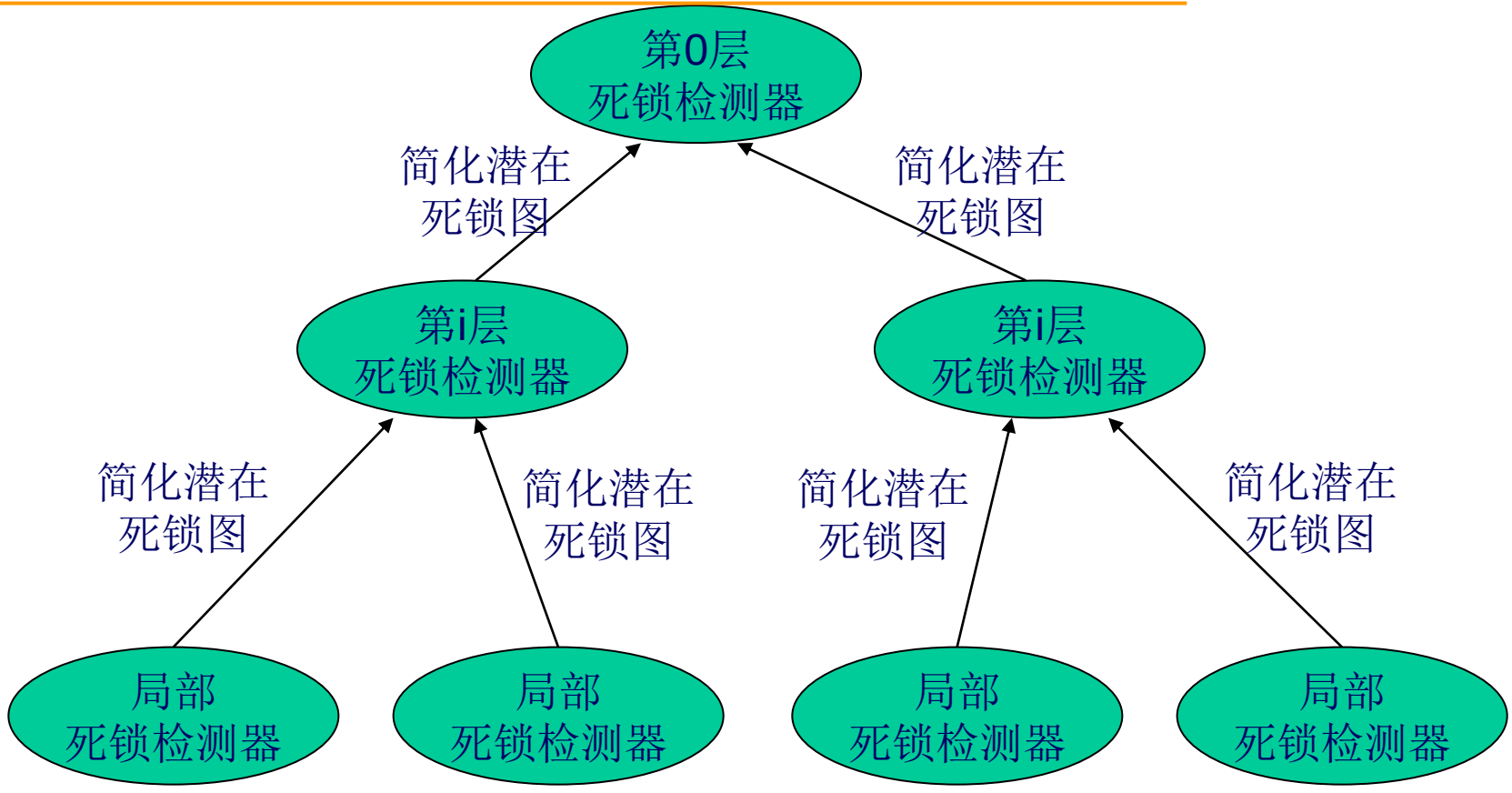
3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

- 层次式死锁检测的步骤
  - **树叶**是各站点**局部死锁检测器**，它们在**本站点建立局部等待图**；
  - **本站点死锁检测器**找出本站点**局部等待图**中的**任何回路**，并把有关**潜在全局回路的信息**发送给层次结构中紧挨的**上一层死锁检测器**；
  - 每个**非本地死锁检测器**只对它所涉及的**紧挨下层进行死锁检测**，**合并**这些接收到的**有关潜在全局回路的信息**，并**找出**任何**回路**；
  - 如果还有上层死锁检测器，将经过简化的有关潜在全局回路的信息发送给它上一层死锁检测器，由上一层死锁检测器再进行合并，找出任何全局回路。
  - 这样**逐层检测**，**直到最高层**。

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法



层次式死锁检测方法示意图

3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

- 分布式死锁检测法
  - 赋予**每个站点相同的检测死锁职责**。每个站点上的**本地死锁检测器互相传送各自的LWFG**(实际上只传送潜在的死锁回路)。
  - 每一站点的LWFG的形成及更新如下:

    (1)由于**每一站点接收**从其它站点传来的可能的死锁回路，因此向自己的**局部LWFG增加一些边**。

    (2)在站点的LWFG中，**被增加的**用于表示本地事务正在等待其它站点事务的**边**，同用于表示远程事务正在等待本站点事务的**边相连接**，该节点称为**外部节点(EX)**。

3 分布式数据库系统中的死锁处理

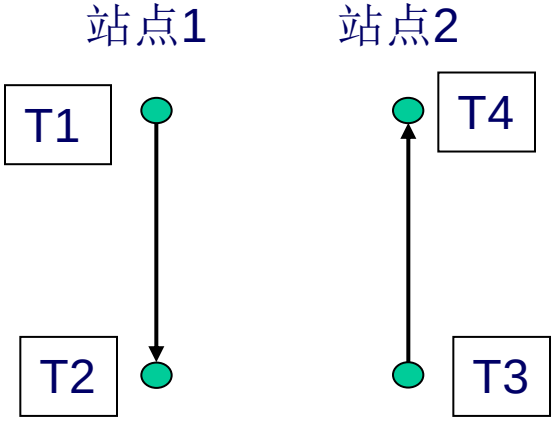
3.3 死锁的检测和解决方法

- 分布式死锁检测算法
  - 使用局部信息构造LWFG, 该LWFG包含EX节点;
  - 对每次**接收到的报文信息**, 执行如下对LWFG的修改:
    - 对报文信息中的**每个事务**, 若LWFG中不存在, 则将其加入;
    - 从EX节点开始, 按照报文所给的信息, **建立一个到下一个事务的边**;
  - 在新的LWFG中寻找**不含EX节点的回路**, 若存在, 则**检测到死锁**;
  - 在新的LWFG中找到**含有EX节点的回路**, 于是有**潜在的死锁**, 再按规定向外传送信息。

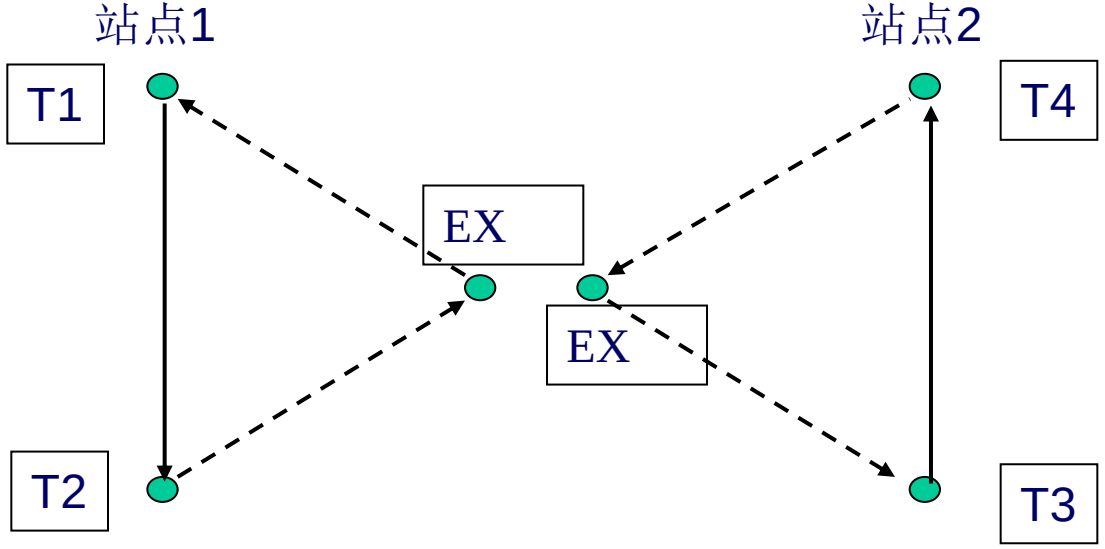
3 分布式数据库系统中的死锁处理

3.3 死锁的检测和解决方法

例事务间有等待关系：T1→T2→T3→T4→T1，
对(a)中的两个站点的LWFG的更新如(b)所示。



(a)局部等待图



(b)更新后的局部等待图

4 分布式数据库系统并发控制的时标技术

4.1 基于时标的并发控制方法

- **1. 基本概念**
  - 基于时标的并发控制算法并不试图通过互斥来支持串行性，而是选择一个事先的**串行次序**依次执行事务。
  - 为建立这个次序，在**每个事务初始化**时，事务管理器将给**每个事务T_i分配**一个在**整个系统中唯一的时标(时间戳)$Ts(T_i)$**。
  - **时标**是用来**唯一识别每个事务**并允许排序的**标识符**。
  - **唯一性**只是产生时标的属性之一，另外一个属性是**单调性**，同一事务管理程序所产生的两个时标将是单调递增的。因此，**时标是一个完整的全排序域**。
  - 如果 $ts(T_1)<ts(T_2)< \ldots < ts(T_n)$, 则调度器产生的序是: $T_1, T_2, \ldots T_n$

4 分布式数据库系统并发控制的时标技术

4.1 基于时标的并发控制方法

- **2. 时标排序规则定义如下：**
  - 已知两个冲突操作Q_{ij}与Q_{kl}，分别属于事务T_i和T_k，Q_{ij}在Q_{kl}之前执行当且仅当 **$ts(T_i) < ts(T_k)$**。这种情况下，**T_i**被称之为**年老的**(或年长的)事务，**T_k**被称为**年轻的**事务。

4 分布式数据库系统并发控制的时标技术

4.1 基于时标的并发控制方法

- **3. 时标分配方法**
  - **全局时标**
    - 使用全局(整个系统)的**单调递增**的**计数器**。全局计数器维护很难。
  - **局部时标**
    - **每个站点**基于其**本地计数器**自治地指定一个时标。
    - 为保持其**唯一性**，每一站点向计数器值附加其自身站点的标识符。这时**时标标识符**由两部分组成：**〈本地计数器值，站点标识符〉**
  - 站点标识符是次要的，主要是本地计数器值，因为它是用来为事务排序和调整时标的主要依据。
  - 如果每个站点能够访问其自身的系统时钟，那么可以**使用系统时钟来代替计数器值**也是可以的。

4 分布式数据库系统并发控制的时标技术

4.1 基于时标的并发控制方法

- 4. 时标法思想
  - 给**每个事务赋一个唯一的时标**，事务的执行等效于**按时标次序串行执行**。
  - 如果发生**冲突**，通过**撤销并重新启动一个事务**来解决的。
  - **事务重新启动时，则赋予新的时标**。

本方法的优点是没有死锁，不必设置锁。封锁和死锁检测引起的通信开销也避免了。

但这个方法**要求时标在全系统中是唯一的**。

4 分布式数据库系统并发控制的时标技术

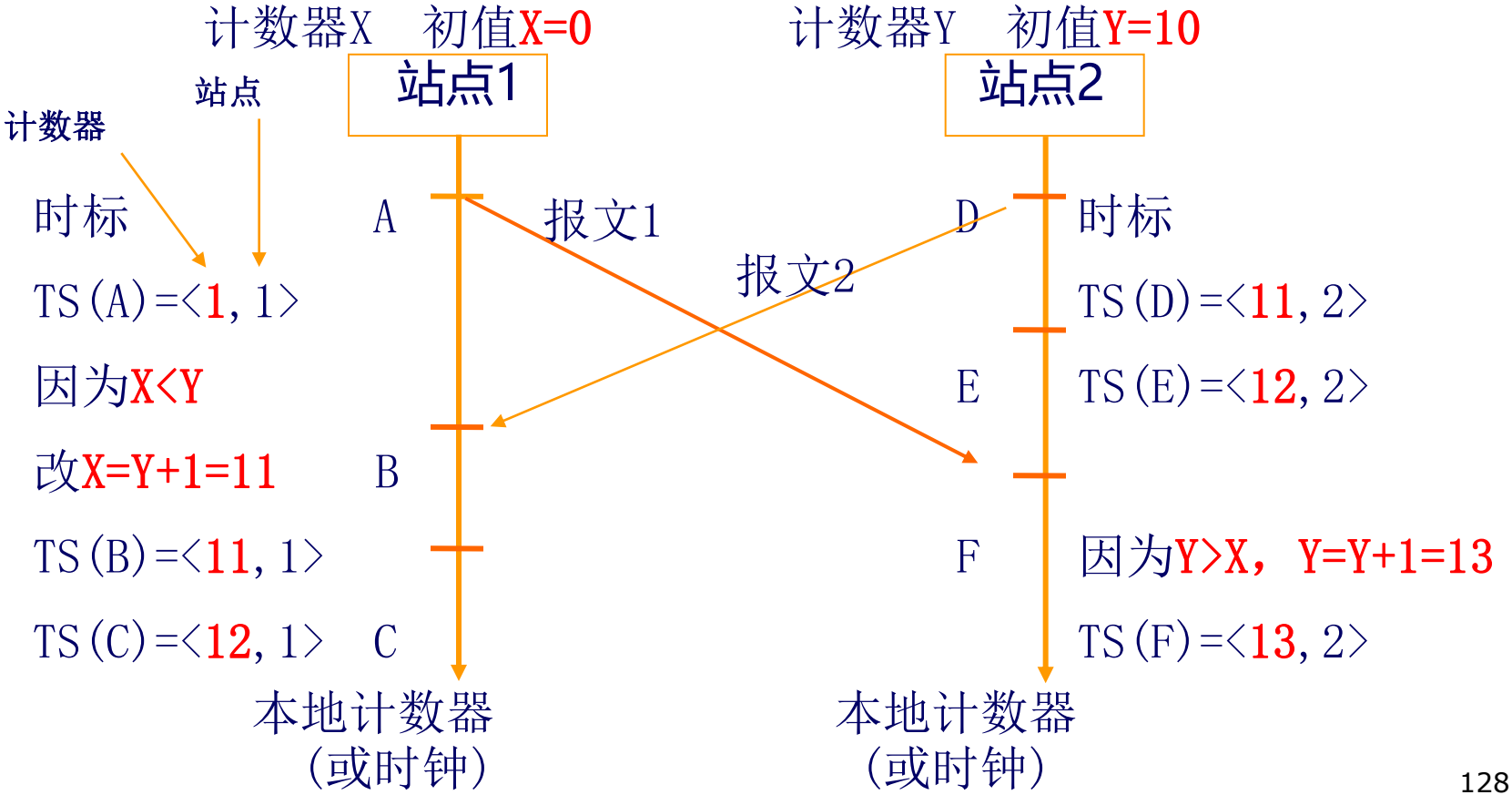
4.1 基于时标的并发控制方法

- 5. 全局唯一时标的形成与调整
  - **全局唯一时标**由**本地计数器值**和**站点标识符**构成。有两个站点，在**每个站点设置一个计数器**，每当**发生一个事务**，计数器加一，这解决了同一站点内事务的次序问题。
  - 对于不同站点，在**发送报文时把本站点的计数器值包含在报文中**，用以近似地**同步各站点的计数器值**。
  - 若站点1收到一个报文的计数器值为Y，它比本地现行计数器X的值大，即**Y>X**，则把本地计数器值**X改为Y+1**；否则，若**Y<X**，则本地计数器值为原值基础上**加1**(如站点2)。

用这种方法来协调不同站点的计数器值保持同步。

4 分布式数据库系统并发控制的时标技术

4.1 基于时标的并发控制方法



4 分布式数据库系统并发控制的时标技术

4.2 基本时标法

1. 基本时标法

- 基本时标法使用下述规则：
  - 每个事务在本站点**开始时赋予一个全局唯一时标**；
  - 在事务结束前，不对数据库进行物理更新；
  - 事务的**每个读操作**或**写操作**都具有**该事务的时标**；
  - 对于数据库中的**每个数据项x**，**记录**对其进行**读操作**和**写操作**的**最大时标**，分别记为**RTM(x)**和**WTM(x)**；
  - 如果事务被**重新启动**，则**被赋予新的时标**。

4 分布式数据库系统并发控制的时标技术

4.2 基本时标法

- 2. 基本时标法执行过程：
  - 令read_TS是事务对数据项x进行读操作时的时标，
    - 若**read_TS<WTM(x)**，则**拒绝该操作**，并使发出该操作的**事务用新时标重新启动**；
    - **否则**执行该操作，且令**RTM(x)=max{RTM(x), read_TS}。**
  - 令write_TS是对数据进行写操作时的时标，
    - 若**write_TS < RTM(x)** 或 **write_TS < WTM(x)** ,则**拒绝**该写操作，并使发出该操作的**事务用新时标重新启动。**
    - **否则**执行该写操作，令**WTM(x)=max{WTM(x), write_TS}。**

4 分布式数据库系统并发控制的时标技术

4.2 基本时标法

- **基本时标法执行过程**确保了**有冲突的操作**在所有站点都是**按时标顺序执行的**，因此是正确的。
- 基本时标法的特点是**不会产生死锁**，任何一个事务都不会阻塞，如果**某一操作不能执行就重新启动**，而不是等待。
- 这种方法**避免死锁是以重启动为代价的**，缺点是重启动多。

4 分布式数据库系统并发控制的时标技术

4.3 保守时标法

- 1. 保守时标法基本思想
  - 一种**消除重启动**的方法，通过**缓冲年轻的操作**，**直至年长的操作执行完成**，因此操作不会被拒绝，事务也绝不被重启动。
  - 为了执行一个被缓冲的操作，需知道什么时候已没有任何较年长事务且与之冲突的操作。

4 分布式数据库系统并发控制的时标技术

4.3 保守时标法

- 2. 保守时标法的规则
  - 规则1. **每个事务只在一个站点执行**, 它不能激活远程的程序, 但是**可以向远程站点发读/写请求**消息;
  - 规则2. 站点i 接收到来自不同站点j 的读/写请求**必须按时标顺序**, 即**每个站点必须按时标顺序发送读/写数据请求**，在传输中也不会改变这个顺序，以保证各站点能够**按时标顺序接收**来自不同站点的**全部读/写请求**。要实现这一规则，较好的方法是使每个事务做到**对同一数据先读后写**;
  - 规则3. **每个站点**都为其它站点发来的**读/写操作开辟一个缓冲区**, 把接收到的读/写操作分别保存在相应的缓冲区队列中。

4 分布式数据库系统并发控制的时标技术

4.3 保守时标法

例如：**站点i** 的缓冲区队列中有来自**所有站点**的读/写请求如下所示:

站点1	站点2	站点3	……	站点n
R11	R21	R31		Rn1
R12	R22	R32		
R13	R23			
	R24			
W11	W21	W31	……	Wn1
	W22	W32	…	Wn2
	W23			

4 分布式数据库系统并发控制的时标技术

4.3 保守时标法

- 3. 保守时标法的执行步骤：
  - 假定某个站点k上，其各个缓冲区队列都已不为空，即**每个站点都已向它至少发送了一个读和一个写操作**，就停止接收，处理在缓冲区中的操作。
  - 假定站点i 至少有一个缓冲的读和缓冲的写来自网络中其它站点，根据规则2，站点i 知道没有年老的请求来自其它站点（因为按序接收，所以不可能有比此更年老的请求到来，年老的比年轻的先到）

4 分布式数据库系统并发控制的时标技术

4.3 保守时标法

(1) 设RT=min(R_{ij}), WT= min(W_{ij})

(2) 按下列方法处理缓冲区队列中的R_{ij}和W_{ij}

a. 扫描**R队列**，若各队列中**存在(R_{ij}) < WT**的R_{ij}，则顺序执行这些R_{ij}，执行完从队列中把它们**删掉**；

b. 扫描**W队列**，若各队列中**存在(W_{ij}) < RT**的W_{ij}，则顺序执行这些W_{ij}，执行完从队列中把它们**删掉**；

(3) 更新 RT=min(R_{ij}),WT=min(W_{ij}), 此时R_{ij}和W_{ij}是队列中剩余的R_{ij}和W_{ij}；

(4) 重复上述(2)和(3), 直到没有满足条件的操作, 或者:

a. 若某个或某些R队列为空时, RT=0;

b. 若某个或某些W队列为空时, WT=0

继续接收各站点发送来的读/写请求操作。回到上述情形，若其各个缓冲区队列又都不空，仍按上面步骤继续。

4 分布式数据库系统并发控制的时标技术

4.3 保守时标法

4. 存在问题和解决方法

- **存在问题**：如果一个站点从来不向某个站点发送操作的话，那么执行过程中的假定就不符合，操作就无法进行。
- **解决办法**：要求对无实际读/写请求的每个站点，要**周期性的发送带有时标的空操作**；或由被阻断的站点请求无实际读/写请求的每个站点向它发送带时标的空操作。**空操作**是指**只传送时标信息**而不是真正的操作。
- **此方法要求网络上所有站点都连通**，这在大系统中很难办到。为避免不必要的通信，**可对无读写操作请求的站点，发送一个时标很大的空操作**。
- 此方法过分保守，一律按照时标顺序执行R和W，其中包括了不冲突的操作，也被缓冲起来同等处理。

5 分布式数据库系统并发控制的多版本技术

5.1 多版本概念和思想

- 多版本基本思想
  - 把保存了**已更新数据项的旧值**的并发控制协议称为**多版本控制协议**，因为它维护了一个数据项的多个版本(值)。
  - 当事务请求访问一个数据项时，如果可能，系统就会选择一个合适的版本，来维护当前执行调度的可串行性。
  - **它的思想**：通过**读取数据项的较老版本来维护可串行性**，使得系统可以**接受**在其他技术中被拒绝的一些**读操作**。当某事务写一个数据项时，它写入了一个新版本，但该数据项的老版本依然被保存。
- 多版本技术缺点
  - 需要**更多的存储**来维持数据库数据项的多个版本。
- 多版本并发控制模式分为两类：
  - 基于**时间戳排序**和基于**两阶段封锁**。

5 分布式数据库系统并发控制的多版本技术

5.2 基于时标排序的多版本技术

- **每个数据项X**都**保留了多版本**X_1, X_2, X_3,…, X_k。
- 对于每个版本，系统将保存**版本X_i的值**和以下**两种时间戳**：
  - **Read_TS(X_i):** X_i的**读时间戳**，它是所有成功读取版本X_i的**事务**的时间戳中**最大的一个**。
  - **Write_TS(X_i):** X_i的**写时间戳**，它是写入版本X_i值的**事务**的时间戳。

    只要允许一个事务执行write_item(X)操作，就会创建数据项X的一个新版本X_{k+1}，并且将Write_TS(X_{k+1})和Read_TS(X_{k+1})的值都置为**TS(T)**。

    相应地，当一个事务T被**允许读版本Xi 的值**时， **Read_TS(X_i)的值**被置为当前Read_TS(X_i)和TS(T)中**较大的一个**。

5 分布式数据库系统并发控制的多版本技术

5.2 基于时标的多版本技术

- 为**确保可串行性**，可以采用下面**两条多版本规则**：
  - 规则1. 如果事务**T发布一个write_item(X)操作**，并且X的版本X_i具有X所有版本中最高的write_TS(X_i)，同时 write_TS(X_i)<=TS(T)且read_TS(X_i)>TS(T)，即， **write_TS(X_i)≤TS(T)<read_TS(X_i)**，此时发生冲突，那么**撤销并回滚T**；否则，**创建X**的一个**新版本X_j**，并且令**read_TS(X_i)= write_TS(X_i)= TS(T)**。
  - 规则2. 如果事务**T发布一个read_item(X)操作**，并且X的版本X_i具有X所有版本中最高的write_TS(X_i),同时write_TS(X_i)<=TS(T)，那么把X_i的值返回给事务T，并且**将read_TS(X_i)的值**置为TS(T)和当前read_TS(X_i)中**较大的一个**。

5 分布式数据库系统并发控制的多版本技术

5.3 采用验证锁的多版本两阶段封锁

- 每个数据项都有三种锁方式：
  - 读，写，验证
- 对于一个**数据项X，LOCK(X)有四种锁状态**：
  - 读封锁（read_locked）
  - 写封锁（write_locked）
  - 验证封锁（certify_locked）
  - 未封锁（unlocked）
- 锁相容性
  - 标准模式锁相容性（写锁和读锁）
  - 验证模式锁相容性（写锁、读锁和验证锁）

5 分布式数据库系统并发控制的多版本技术

5.3 采用验证锁的多版本两阶段封锁

	读	写
读	是	否
写	否	否

(a) 读/写封锁模式的相容性表

	读	写	验证
读	是	是	否
写	是	否	否
验 证	否	否	否

(b) 读/写/验证封锁模式的相容性表

5 分布式数据库系统并发控制的多版本技术

5.3 采用验证锁的多版本两阶段封锁

- 多版本2PL的思想
  - 当只有一个**单独的事务T**持有**数据项上的写锁**时，**允许其他事务T'读该数据项X**，这是通过给予每个数据项X的**两个版本**来实现的：
    - 一个**版本X**是由一个**已提交的事务写入的**；
    - 另一个**版本X'**是每个事务T**获得**该数据项上**写锁时创建的**。
  - 当事务T持有这个写锁时，**其他事务可以继续读X的已提交版本**。
  - 事务T可以根据需要写X'的值，不影响X已提交版本的值。
  - 但是，一旦T准备提交，它**必须在能够提交之前，得到它目前持有写锁的所有数据项上的一个验证锁**。
  - 为得到验证锁，事务可能不得不延迟它的提交，直到所有被它加上写锁的数据项都被所有那些正在读它们的事务释放。一旦获得验证锁，数据项的已提交版本X被置为版本X'的值，版本X'被丢弃，验证锁被释放。

5 分布式数据库系统并发控制的多版本技术

5.3 采用验证锁的多版本两阶段封锁

- 多版本2PL的思想
  - 在这个多版本2PL模式中，**多个读操作**可以和**单独一个写操作并发地执行**，这种模式在标准的2PL模式中则是不允许的。
  - 其代价是，事务可能不得不延迟提交，直到它获得所有经过它更新的数据项上的排他的验证锁。
  - 这种模式避免了级联撤销，因为事务只允许读已提交事务写入的X版本。
  - 按上述规则处理事务的读/写请求的调度器能够保证产生可串行化的调度。为了节省空间，数据库的多个版本可以不时地被清除。清除工作在当分布式数据库管理系统确定不再接受需要访问被清除版本的事务时进行。

6 分布式数据库系统并发控制的乐观方法

6.1 基本思想和假设

- 并发控制乐观方法基本思想
  - 对于冲突操作不像悲观方法那样采取挂起或拒绝的方法，而是让**一个事务执行直到完成**。
- 乐观方法基于如下**假设**：
  - 冲突的事务是少数（查询为主的系统中，冲突少于5%）；
  - 大多数事务可以不受干扰地执行完毕。
- 因此在事务执行过程中，**事务都是先对欲操作的数据项的本地局部副本进行操作，执行完毕后再进行检验**。当不曾发生过冲突时，就把该本地局部副本全局化；若发生过冲突，则回退该事务并作为新事务而重新启动。

6 分布式数据库系统并发控制的乐观方法

6.2 执行阶段划分

- 在乐观方法中，改变通常每个“读集包含写集”事务执行的三个阶段的顺序：
  - **读/计算阶段**：事务从数据库读数据，进行计算，并为写集合中的数据项确定新值，但是这些新值暂不写入数据库中。**该阶段几乎包括了事务的整个执行内容。**
  - **验证阶段**：执行检测，检测事务对数据库的更新是否失去相容性，确保如果将事务的更新应用于数据库，不会违反可串行性。
  - **写阶段**：如验证阶段获得肯定结果，则把事务的更新应用于数据库，对数据进行更新(提交)；否则，忽略所有更新，并重新开始该事务。

这个算法有一个强制性假设：读集包含写集，即**不存在对一个数据项只写而不读**，而且**数据库采用完全冗余方式存储数据。**

6 分布式数据库系统并发控制的乐观方法

6.3 使用数据项和事务上的时标

- 读阶段末，事务产生一个**更新表u**，包括下列信息：
  - 读集的数据项，带有自身的时标；
  - 写集数据项的新值；
  - 该事务自身的时标。
- 验证阶段
  - 把**更新表发送到每一站点**，每个站点对是否确认该更新表进行**表决**，并把表决结果送回到该事务的源站点。
  - 如果源站点收到**多数肯定票**，则决定该**事务提交**并把决定通知所有其它站点；
  - 如果**多数表决是否定的**，则把放弃更新表的决定通知所有站点，然后**重新启动该事务**。

6 分布式数据库系统并发控制的乐观方法

6.3 使用数据项和事务上的时标

- 站点表决规则：
  - 比较**更新表读集中的每个数据项的时标**与它的**本地数据库**中存在的**对应数据项的时标**进行比较， 如果它们**全部相等**，则该站点投肯定票；不相等， 投反对票。 因为此时可能本地写入数据项之前或之后, 又有事务对数据项做了写操作。
- 表决结果
  - 若赞成票是大多数，则考虑Commit，并将结论发送给每个Site
  - 否则, 其更新表无效, 事务重启动

- 设有学生----课程数据库如下:

  Student (Sno, Sname, Ssex, Sage, Sdept)

  Course (Cno, Cname, Cpno, Ccredit)

  SC (Sno, Cno, Grade)

1 用SQL语句完成信息系(IS系)学生选修了的所有课程名称;

2 写出关系代数表达式;

3 画出用关系代数表达式表示的语法树以及优化后的语法树.

- 设有学生----课程数据库如下:

Student (Sno, Sname, Ssex, Sage, Sdept)

Course(Cno, Cname, Cpno, Ccredit)

SC(Sno, Cno, Grade)

1 用SQL语句完成信息系(IS系)学生选修了的所有课程名称;

```
Select Cname from Student, Course, SC
Where Student.Sno=SC.Sno and
        SC.Cno=Course.Cno and
        Student.Sdept='IS'
```