第八章 分布式并发控制

一、基本概念

- 简单地讲，并发就是多个事务的同时执行，并发能够提高系统的效率，但也可能带来三种错误
- 并发控制的主要目的是保证事务的一致性和隔离性，最终保证数据的一致性
- 当分布事务并发执行时，并发控制既要实现分布事务的可串行性，又要保持事务具有良好的并发度，以保证系统具有良好的性能

并发控制问题

- 丢失修改
- 不可重复读
- 读脏数据

并发控制就是利用正确的方式调度事务中所涉及的并发操作序列，避免造成数据的不一致性；防止一个事务的执行受到其它事务的干扰保证事务并发执行的可串行性。

二、并发控制的理论基础

无论在集中式数据库系统中，还是在分布式数据库系统中，并发调度都要解决并发事务对数据库的冲突操作问题，使**冲突操作串行执行，非冲突操作并发执行**

在分布式数据库系统中，事务是由分解为各个场地上的子事务的执行实现的。因此，分布式事务之间的冲突操作，就转化为了同一场地上的子事务之间的冲突操作，分布式事务的可串行性调度也转化为了子事务的可串行性调度问题

1.集中式数据库的可串行化问题

可串行化：在集中式数据库系统中的一个历程H，如果等价于一个串行历程，则称历程H是可串行化的。由可串行化历程H所决定的事务执行顺序，记为SR(H)

事务的执行顺序：分别属于两个事务的两个操作O_i和O_j，如果它们操作同一个数据项，且至少其中一个操作为写操作，则O_i和O_j这两个操作是冲突的例如：$R_1(x)$ 和$W_2(x)$，$W_1(x)$和 $W_2(x)$

在一个串行历程中，用符号“<”表示先于关系，对分别属于T_i和 T_j的两个冲突操作O_i和O_j，若存在$O_i<O_j$，则$T_i<T_j$

历程等价的判别方法

定理1：任意两个历程H_1和H_2等价的充要条件是

- 在H_1和H_2中，每个读操作读出的数据是由相同的写操作完成的
- 在H_1和H_2中，每个数据项上最后的写操作是相同的

定理1的引理：对于两个历程H_1和H_2，如果每一对冲突操作O_i和O_j，在H_1中有$O_i<O_j$，在H_2中也有$O_i<O_j$，则H_1和H_2是等价的

例 8.7　设有三个历程 H_1、H_2 和 H_3，分别为：

- H_1：$R_1(x)R_1(y)W_1(x)W_1(y)R_2(x)W_2(x)$
- H_2：$R_1(x)R_1(y)W_1(x)R_2(x)W_1(y)W_2(x)$
- H_3：$R_1(x)R_1(y)R_2(x)W_1(x)W_1(y)W_2(x)$

要求判断历程 H_2 和 H_3 是否为可串行化的历程。

从上面的历程 H_1 中可看出，事务 T_1 的所有操作都是在事务 T_2 的操作之前完成的，因此可判断历程 H_1 是串行历程。

下面根据两个历程的等价引理判断 H_2 和 H_3 是否与历程 H_1 等价，若等价，则该历程是可串行化的历程。判别如下：

1）首先找出历程 H_1、H_2和 H_3的冲突操作。

2）分别找出历程 H_2和 H_3中与历程 H_1等价的冲突操作，用“↔”表示。

则结果如下所示：

H_2的冲突操作	H_1的冲突操作	H_3的冲突操作
$R_1(x)<W_2(x)$ ↔	$R_1(x)<W_2(x)$ ↔	$R_1(x)<W_2(x)$
$W_1(x)<R_2(x)$ ↔	$W_1(x)<W_2(x)$ ↔	$R_2(x)<W_1(x)$
$W_1(x)<W_2(x)$ ↔	$W_1(x)<R_2(x)$	$W_1(x)<W_2(x)$

可知，历程 H_2 与历程 H_1 等价，因此历程 H_2 是可串行化的历程；历程 H_3 与历程 H_1 不等价，因此历程 H_3 不是可串行化的历程。

2.分布式事务的可串行化问题

局部历程：在分布式事务执行过程中，场地S_i上的所有子事务的操作的执行序列，称为局部历程，用$H(S_i)$表示

分布式事务可串行化判定定理：对于n个分布式事务T_1，T_2，…，T_n在m个场地上S_1，S_2，…，S_m上的并发执行序列，记为E。如果E是可串行化的，则必须满足以下条件：

- 每个场地S_i上的局部历程$H(S_i)$是可串行化的
- 存在E的一个总序，使得在总序中，如果有$T_i<T_j$，则在各局部历程中必须有$T_i<T_j$

分布式事务可串行化判定定理的引理：

- 设T_1，T_2，…，T_n是n个分布式事务，E是这组事务在m个场地上的并发执行序列，$H(S_1)$，$H(S_2)$，…，$H(S_m)$是在这些场地上事务的局部历程，如果E是可串行化的，则必须存在一个总序，使得T_i和T_j中的任意两个冲突操作O_i和O_j，如果在$H(S_1)$，$H(S_2)$，…，$H(S_m)$中有$O_i<O_j$，当且仅当在总序中也有$T_i<T_j$

三、基于锁的并发控制方法

1.基本思想

- 事务在对某一数据项操作之前，必须先申请对该数据项加锁，申请成功后才可以对该数据项进行操作
- 如果该数据项已被其它事务加了不相容的锁，那么后申请使用数据的事务必须等待，直到该数据项被解锁为止

2.锁分两种类型

- 排它锁(`exclusive`)
- 共享锁(`shared`)

3.锁的相容性

	读锁	写锁
读锁	共享	排斥
写锁	排斥	排斥

4.封锁规则

访问——加锁（申请不到：等待）

访问结束——释放

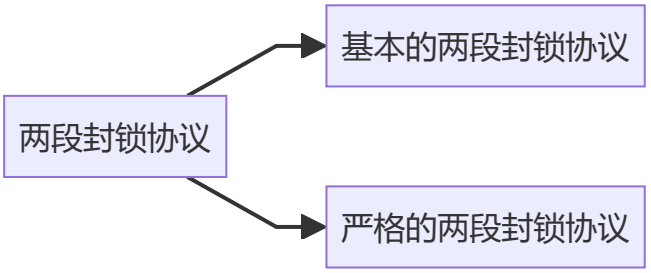
5.锁的粒度

- 锁的粒度
  - 封锁数据对象的单位称锁的粒度，指被封锁的数据对象的大小
    - 锁的粒度也称锁的大小
    - 系统根据自己的实际情况确定锁的粒度，锁的粒度可以是关系的属性（或字段）、关系的元组（或记录）、关系（也称文件）或整个数据库等
  - 系统的并发度与锁粒度成反比
    - 锁粒度越大，系统的开销越小，但降低了并发度
- 锁的种类
  - 显式锁，用户加封锁命令实现
  - 隐式锁，系统自动加锁实现

粒度	开销	并发度
小	大	高
大	小	低

四、两段封锁协议

- 两段封锁协议(`2PL`)是数据库系统中解决并发控制的重要方法之一，保证事务可串行性调度
- `2PL`的实现思想是将事务中的加锁操作和解锁操作分两阶段完成，要求并发执行的多个事务要在对数据操作之前进行加锁，且每个事务中的所有加锁操作要在解锁操作以前完成



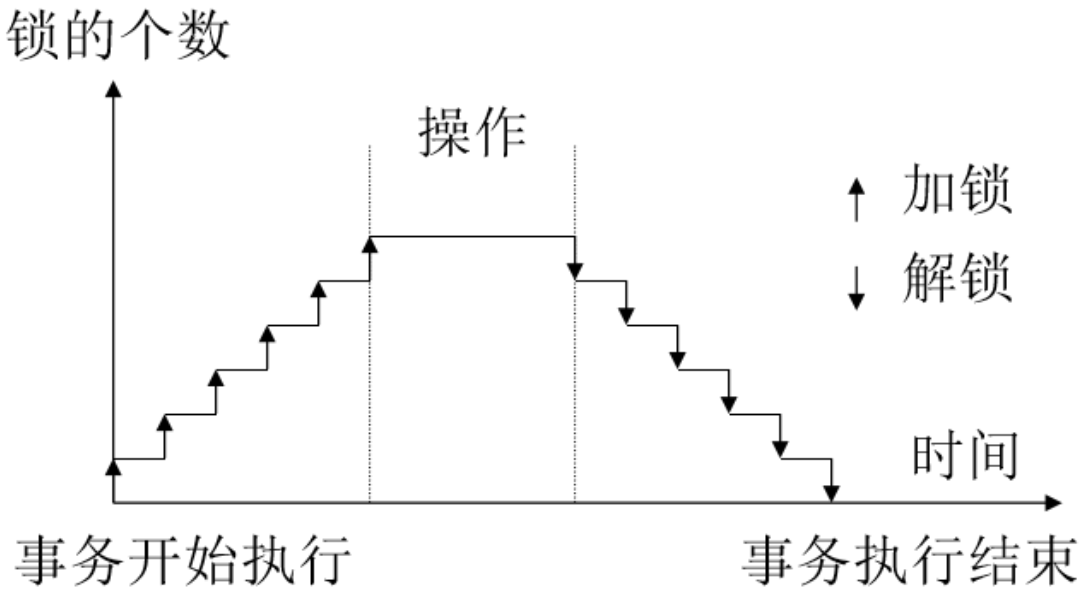
1.基本的两段封锁协议

阶段一：加锁阶段

- 事务在读一个数据项之前必须对其加锁
- 如果该数据项被其他使用者已加上不相容的锁。则必须等待

阶段二：解锁阶段

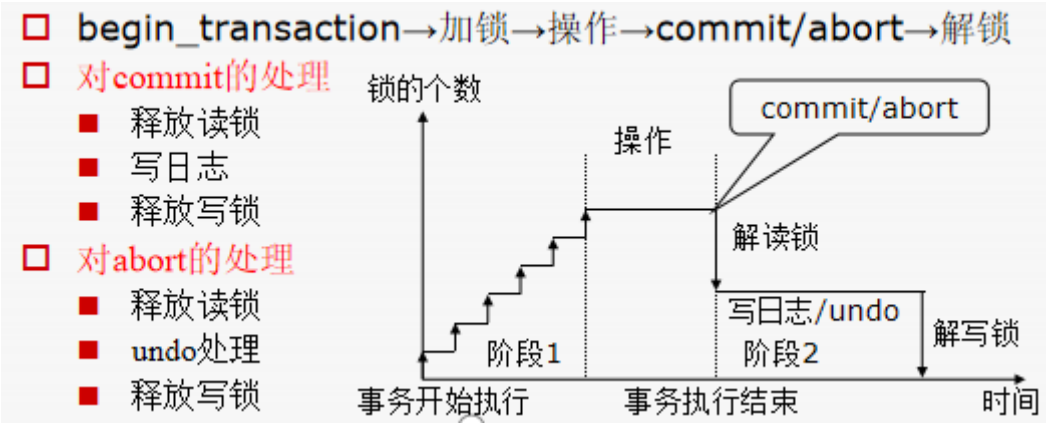
- 事务在释放锁之后，不允许再申请其他锁



- 丢失修改错误——通过加锁避免
- 不能重复读——通过加锁避免
- 读脏数据——

2.严格的两段封锁协议

- 严格的两段封锁协议与基本的两段封锁协议内容基本上是一致的，只是解锁时刻不同，是在事务结束时才启动解锁，保证了事务所更新数据的永久性
- 采用两段封锁协议（`2PL`）的事务执行过程为：



五、分布式数据库并发控制方法

六、分布式死锁管理

1.产生死锁的原因

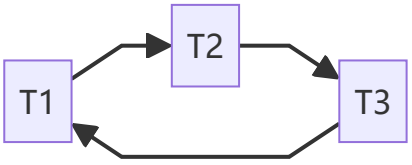
加不相容的锁——等待

多个彼此**相互等待**——死锁

2.死锁等待图

`a`指向`b`：`a`等待`b`

存在回路：出现死锁

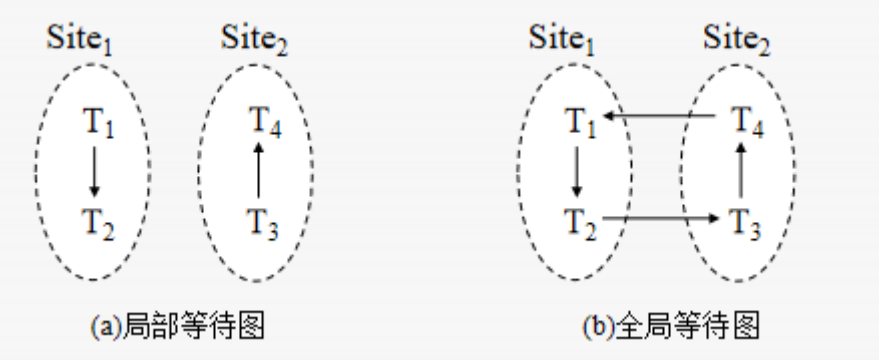


死锁等待图也可以用在分布式数据库系统中：

- 检测各场地事务的局部等待图(`LWFG`)是否存在回路
- 检测所有场地事务之间的全局等待图(`GWFG`)是否存在回路
- 全局等待图由局部等待图组成，首先要得到局部等待图，然后由所有的局部等待图联合而成全局等待图

导致死锁产生的两个事务很可能不在同一场地上，称为**全局死锁**

例：



先看局部：无回路，不死锁

再看全局：有回路，发生死锁

3.死锁的检测

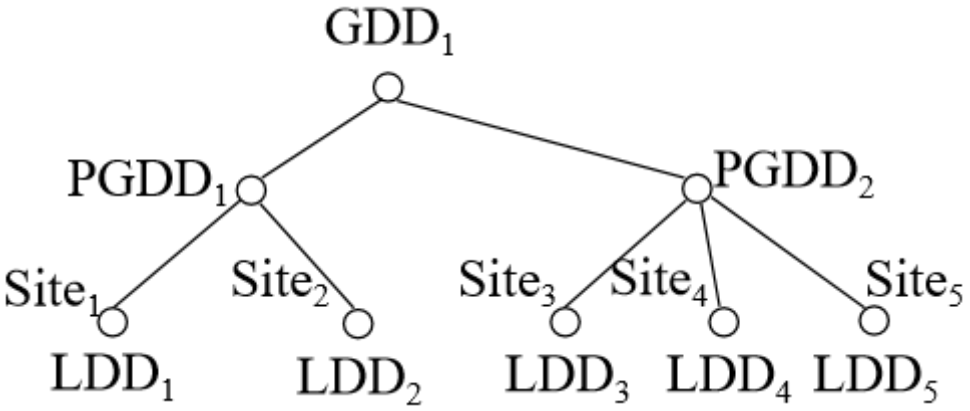
3.1 集中式死锁检测

- 步骤：
    1. 选择一个场地来做全局死锁检测
    2. 每个场地的锁管理器都向这个全局的死锁检测器传送局部等待图，从而逐渐构成全局等待图
    3. 检查全局等待图，看有么有回路
- 优点：简单
- 不足：
    - **通信代价大。**传送局部等待图带来大量的通信代价，因此，各个场地的锁管理器只需传输其等待图中变化部分的信息。
    - **单点失效**
    - 难于确定各个场地向检测器传输信息的时间间隔。间隔小，可减小死锁检测延迟，但增加数据传输代价；间隔大，可减少数据传输量，但影响死锁检测及时性

3.2 层次死锁检测

（改进集中式，目的是减少通信量）

集中式 `2PL` 并发控制算法系统中常选择的死锁检测方法



叶子节点：`LDD` 局部死锁检测程序

中间层节点：`PGDD` 部分全局死锁检测程序

根节点：`GDD` 全局死锁检测程序

方式：自底向上一层一层传，看是否死锁

特点：

- 层次死锁检测方法的运行性能与层次的选择至关重要，在进行场地组划分时，应考虑站点之间的物理距离以及站点上容纳的数据的相关度等
- `PGDD` 数目过大会增加运行成本，过少将失去其优越性

3.3 分布式死锁检测

- 由各个场地共同完成，每个场地对等，不区分全局死锁和局部死锁
- 局部 `WFG` 与其他

4.死锁的预防和避免

- 预防

  顺序封锁

- 解决

  使用事务的时间戳来解决死锁，基本思路是：杀死死锁等待图中的某个事务，破坏环路，典型的算法有两种

  - 等待死亡
  - 负伤等待

七、大数据库并发控制技术

- 多读少写，少有复杂事务需求

  单线程写，多线程读，写的时候写时拷贝

- 简化系统设计，有些摒弃事务的概念，提高性能

1.事务读写模式扩展

- 读事务模式扩展
  - 最新读：读最新记录
  - 快照读：读历史版本
  - 非一致性读：不考虑日志状态并且直接读取最后一个值
- 最新读和快照读都是读取单个实体组
- 写事务模式
- 封锁机制扩展
  - 建议性锁
  - 强制性锁
- 多版本扩展
  - 事务版本号分配方法
  - `MVCC`
    - 全局唯一的事务版本号标记，递增特性
    - `MVCC`可以避免事务重启的代价，同时可以满足快照读取的要求，保证能够读到任意一个有效历史时刻上的数据