Lecture 13: 并发控制

Database

Author: Forliage

Email: masterforliage@gmail.com

Date: June 9, 2025

College: 计算机科学与技术学院

Abstract

本讲笔记全面介绍了关系型数据库中的并发控制机制，首先从基于锁的协议入手，阐述了共享锁与排他锁的加锁规则、两阶段锁（2PL）及其严格化变种、锁转换与自动获取策略，以及锁管理器与锁表的实现原理。

随后，笔记深入多粒度锁设计，说明了在不同粒度层次（如数据库、表、页、行）上加锁的意向共享锁（IS）、意向排他锁（IX）与共享意向排他锁（SIX）模式，并给出了粒度锁兼容性与获取、释放规则。

在死锁方案部分，讲解了死锁的成因与表现，比较了死锁预防（如保守2PL、图协议、时间戳方案）、死锁检测（等待图环路检测）与死锁恢复（牺牲品选择、部分回滚）等方法，并分析了各自的优缺点。

最后，笔记讨论了插入与删除操作中的幻象问题，提出了通过全表锁或更细粒度的索引锁定协议解决幻象现象的思路，详细说明了事务在插入/删除时对相关索引桶进行共享或排他锁定的流程，以在保证可串行化的同时提高并发度

（该Abstract由ChatGPT-o4-mini-high生成）

Contents

1 基于锁的协议

1.1 基于锁的协议

可串行化调度是并发控制的基础。

数据锁可以两种模式加锁：

1. 排他(X)模式：数据项可以被读取和写入。使用lock-X指令请求X锁。
2. 共享(S)模式：数据项只能被读取。使用lock-S指令请求S锁。

锁请求发送给并发管理器。只有在请求被批准后，事务才能继续进行。

如果请求的锁与其它事务已持有的该项目上的锁兼容，则事务可能会被授予该项目的锁。

任意数量的事务可以在一个项目上持有共享锁。但是，如果任何事务持有该项目的排他锁，则其他事务不能在该项目上持有任何锁。

如果无法授予锁，则请求事务将被要求等待，直到其它事务持有的所有不兼容锁都被释放。然后再授予该锁。

加锁协议是所有事务在请求和释放锁时遵循的一组规则。加锁协议限制了可能的调度集合。

1.2 基于锁的协议的陷阱

考虑如下部分调度：

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

Figure 1

T_3和T_4都无法取得进展——执行锁$S(B)$会使T_4等待T_3释放其对B的锁，而执行锁$-X(A)$会使T_3等待T_4释放其对A的锁。这种情况称为死锁。

要处理死锁，必须回滚T_3或T_4中的一个并释放其锁。

大多数锁定协议都存在死锁的可能性，死锁是不可避免的问题。

如果并发控制管理器设计不当，也可能出现饥饿现象。例如：一个事务可能正在等待某个项目加排他锁，而其它一系列事务却在请求并授予对同一项目的共享锁。同一事务由于死锁而反复回滚。

可以设计并发控制管理器来防止饥饿现象。

1.3 两阶段锁协议

这是一种确保冲突可串行化调度的协议。

阶段1：增长阶段。事务可以获取锁，事务不能释放锁。

阶段2：收缩阶段。事务可能会释放锁，事务可能无法获取锁。

该协议确保可串行性。可以证明，事务可以按照其锁点（即事务获取其最后一个锁的点）的顺序进行串行化。

两阶段锁不能确保避免死锁。

在两阶段锁机制下，级联回滚是可能发生的。为避免这种情况，可采用一种改进的协议，即严格两阶段锁协议。在该协议中，事务必须持有其所有排他锁，直到提交或中止。

严格两阶段锁协议更为严格：在此协议中，所有锁都要持有到事务提交或中止。在这个协议中，事务可以按照提交的顺序进行系列化。

如果使用两阶段锁，可能会存在无法得到的冲突可串行化调度。

然而，在没有额外信息（例如，对数据的访问顺序）的情况下，两阶段锁在以下意义上是实现冲突可串行化所必须的：给定一个不遵循两阶段的事务T_i，我们可以找到一个使用两阶段锁的事务T_i，以及一个针对T_i 和T_j的非冲突可串行化调度。

1.4 锁转换

带有锁转换的两阶段锁：

- 第一阶段
  - 可以对项获取共享锁
  - 可以对项获取排他锁
  - 可以将S锁转化为X锁（升级）
- 第二阶段

- 可以释放S锁
- 可以释放X锁
- 可以将X锁转化为S锁（降级）

该协议确保可串行性，但仍依赖程序员插入各种加锁指令。

1.5 锁的自动获取

事务T_i发出标准的读/写指令，无需显式的加锁调用

操作read(D)按如下方式处理:

```
if Ti has a lock on D then
    read(D)
else begin
    if necessary wait until no other
    transaction has a lock-X on D then
        grant Ti a lock-S on D;
        read(D)
    end
```

write(D)按以下方式处理：

```
if Ti has a lock-X on D then
    write(D)
else begin
    if necessary wait until no other trans.has any lock on D,
        if Ti has a lock-S on D then
            upgrade lock on D to lock-X
        else
            grant Ti a lock-X on D
            write(D)
    end;
```

所有锁在提交或中止后释放。

1.6 锁的实现

锁管理器可以实现为一个单独的进程，事务向该进程发送加锁和解锁请求。

锁管理器通过发送锁授予消息（或者在发生死锁的情况下，发送一条要求事务回滚的消息）来响应加锁请求。

请求事务会一直等待，直到其请求得到响应。

锁管理器维护一个称为锁表的数据结构，用于记录已授予的锁和待处理的请求。

锁表通常实现为一个内存的哈希表，以被锁定的数据项的名称作为索引。

1.7 锁表

黑色矩形表示已授予的锁，白色矩形表示等待中的请求。

锁表还会记录已授予或请求的锁的类型。

新请求被添加到数据项请求队列的末尾，若与所有先前的锁兼容，则被批准。

解锁请求会导致该请求被删除，并检查后续请求是否现在可以被批准。

如果事务中止，则该事务的所有等待或已批准的请求都将被删除：锁管理器可以维护每个事务持有的锁列表，以高效实现此功能。

1.8 基于图的协议

基于图的协议是两阶段锁的一种替代方案。

对所有数据项的集合$D = \{d_1, d_2, ..., d_h\}$施加偏序$\rightarrow$。如果$d_i \rightarrow d_j$，则任何同时访问$d_i$和$d_j$的事务必须在访问$d_j$之前访问$d_i$。意味着集合$D$现在可以被视为一个有向无环图，称为数据库图。

树协议是一种简单的图协议。

1. 只允许使用X锁
2. T_i的第一个锁可以加在任何数据项上。随后，T_i只能在Q的父节点当前已被T_i锁定的情况下，才能锁定数据Q。
3. 数据项可以在任何时候解锁
4. 一个已被T_i锁定并解锁的数据项，随后不能再由T_i重新锁定。

优点：树形协议可确保冲突可串行化，且不会产生死锁。与两阶段锁协议相比，树形锁定协议中的解锁操作可能会更早发生：等待时间更短，并发度提高；协议无死锁，无需回滚。

缺点：协议不保证可恢复性或无级联性——需要引入提交依赖关系以确保可恢复性——事务可能需要锁定它们不访问的数据项。——增加了锁定开销和额外的等待时间——并发度可能降低。在两阶段锁定下不可能的调度在树协议下是可能的，反之亦然。

2 多粒度

2.1 多粒度

为方便起见，允许数据项根据需求以不同大小进行加锁——即多粒度。

定义一个数据粒度层次结构，其中小粒度嵌套在大粒度中，并且可以用图形表示为一棵树

当一个事务显式地对树中的一个节点加锁时，它会以相同的模式隐式地对该节点的所有后代节点加锁。

加锁粒度（进行加锁的树的层级）：细粒度：高并发，高加锁开销；粗粒度：低锁开销，低并发

2.2 意向锁

问题：T_1在X锁中锁定了r_{a_1}，T_2在S锁中锁定了F_b。现在T_3希望在S锁中锁定F_a. T_4希望在S锁中锁定整个DB。

在显式锁定一个节点之前，会对该节点的所有祖先节点设置意向锁。

意向锁允许在S或X模式下锁定更高级别的节点，而无需检查所有子节点。

2.3 意向锁模式

存在三种具有多种粒度的意向锁模式：

- 意向共享锁（IS）：表示在树的较低层级使用共享锁进行显式锁定（表明其后代存在S锁）
- 意向排他锁（IX）：表示在较低层级使用排他锁进行显式锁定（表明其后代存在X锁）
- 共享意向排他锁（SIX）：以该节点为根的子树以共享模式进行显式锁定，并且在较低层级使用排他模式锁进行显式锁定。

2.4 多粒度锁机制

事务T_i可以使用以下规则锁定节点Q：

1. 必须遵守锁兼容性矩阵
2. 必须先锁定树的根节点，并且可以以任何模式锁定。
3. 只有当节点Q的父节点当前被T_i以IX或IS模式锁定时，T_i才能以S或IS模式锁定节点Q
4. 只有当节点Q的父节点当前被T_i以IX或SIX模式锁定时，T_i才能以X，SIX或IX模式锁定节点Q
5. T_i仅在之前未解锁任何节点时才能锁定一个节点（即T_i是两个阶段的）
6. T_i仅在Q的任何子节点当前都未被T_i锁定时才能解锁节点Q

注意，锁时按照从根到叶的顺序获取的，而释放则是按照从叶到根的顺序进行的。（加锁自顶向下，解锁自下而上，且遵守2PL协议）

优点：增强并发性，降低加锁开销。

3 死锁方案

3.1 死锁处理

如果存在一组事务，使得该组中的每个事务都在等待该组中的另一个事务，则系统发生死锁。

如何处理？死锁预防/死锁检测与死锁恢复

3.2 死锁预防

死锁预防协议可确保系统永远不会进入死锁状态。一些预防策略：

1) 要求每个事务在开始执行前锁定其所有数据项（预先声明）——保守两阶段锁协议（要么全部锁定，要么都不锁定）。

2) 对所有数据项施加部分顺序，并要求事务只能按此顺序锁定数据项（基于图协议）——因此永远不会形成循环

在等待-死亡和伤口-等待方案中，回滚的事务都会使用其原始时间戳重新启动。因此，较旧的事务优先于较新的事务，从而避免了饥饿问题。

基于超时的方案：

- 事务仅在指定的时间内等待锁。之后，等待超时，事务回滚。
- 因此不可能发生死锁
- 实现简单；但可能会出现饥饿问题。此外，很难确定合适的超时时间间隔值。

以下方案仅为预防死锁而使用事务时间戳：

等待-死亡方案——非抢占式：较旧的事务可能会等待较新的事务释放数据项。较新的事务从不等待较旧的事务；相反，它们会被回滚。

伤害—等待方案——抢占式：较旧的事务会导致较新事务的旧事务伤口（强制回滚），而不是等待它。较新的事务可能会等待较旧事务。与等待-死亡方案相比，回滚次数可能更少，回来时的时间戳仍是之前的。

3.3 死锁检测

死锁可以用等待图来描述，该图由一对$G = (V, E)$组成，V是一组顶点(系统中的所有事务)，E是一组边；每个元素都是一个有序对$T_i \to T_j$ 如果$T_i \to T_j$在E中，那么从T_i到T_j存在一条有向边，这意味着T_i正在等待T_j释放一个数据项。

当T_i请求一个当前由T_j持有的数据项时，边T_iT_j会被插入到等待图中。只有当T_j不再持有T_i所需的数据项时，这条边才会被移除。

当且仅当等待图存在环时，系统处于死锁状态。必须定期调用死锁检测算法查找环。

3.4 死锁恢复

必须回滚某些事务（将其作为牺牲品）以打破死锁。选择成本最小的事务作为牺牲品。

回滚——确定将事务回滚多远。

- 完全回滚：中止事务，然后重新启动它
- 部分回滚：仅将事务回滚到打破死锁所需的程度更为有效

如果总是选择同一事务作为牺牲品，就会发生饥饿现象。将回滚次数纳入成本因素以避免饥饿。

4 插入和删除操作

使用两阶段锁：

- 仅当删除元组的事务对要删除的元组具有排他锁时，才可以执行删除操作。
- 项数据库中插入新元组的事务会获得该元组的X模式锁

插入和删除操作可能会导致幻影现象。

扫描关系的事务（例如，查找佩里里奇所有账户余额的总和）以及在关系中插入一个元组的事务（例如，在佩里里奇插入一个新账户）（从概念上讲）进官没有共同访问任何元组，但仍会发生冲突。

如果仅使用元组锁，则可能会产生不可串行化的调度。例如，扫描事务看不到新账户，但会读取更新事务写入的其它一些元组。

扫描该关系的事务正在读取指示该关系包含哪些元组的信息，而插入元组的事务会更新相同的信息。该信息应被加锁。

一种解决方案：将一个数据项与该关系关联起来，以表示关于该关系包含哪些元组的信息；扫描该关系的事务在该数据项上获取共享锁。插入或删除元组的事务在该数据项上获取排他锁（注意：数据项上的锁与单个元组上的锁不冲突）

上述协议在插入/删除操作方面提供的并发度非常低。

索引锁定协议通过对某些索引桶加锁，在防止幻象现象的同时提供了更高的并发度。

索引锁定协议

每个关系必须至少一个索引。对关系的访问必须仅通过该关系的索引之一进行。

执行查找操作的事务T_i必须以共享(S)模式锁定其访问的所有索引桶。

事务T_i在未更新关系r的所有索引情况下，不得将元组t_i插入到关系r中。

T_i必须对每个索引执行查找操作，以找到所有可能包含指向元组t_i的指针的索引桶(假设该元组已经存在)，并以X模式锁定这些索引桶。T_i还必须以X模式锁定其修改的所有索引桶。

必须遵守两阶段锁定协议的规则。