并发控制理论(Concurrency Control Theory)

> 为了使得多个事务在并发执行的时候也能保留隔离性。数据库系统必须要对并发执行进行控制。其控制所用的规则称为并发控制协议。并发控制协议有多种，每种都有不同的最适场景。

---

1. 事务(Transaction)

事务是在一个共享数据库中为完成一个高级函数而执行的一个包含若干读写操作的操作序列。事务是对改变数据库的基本单元。其一般是由高级DML(e.g SQL)或者编程语言(e.g. C++, Java)通过JDBC或ODBC提供的嵌入式数据库访问来初始化的。

---

2. 正确性保证原则 - ACID(Correctness Ensurance Criteria ACID)

2.1. 原子性(Atomic)

数据库管理系统保证事务要么被完全执行，要么完全不执行。通过以下两种方法来保证：

- 日志(Logging)：记录所有的动作以便于重新执行。
- 影子页(Shadow Paging)：拷贝事务涉及到的页，在拷贝页上执行操作，等待事务提交时，拷贝页对事务外的部分可见。

2.2. 一致性(Consistency)

数据库存储的数据在逻辑上是正确的，数据库返回的数据也是逻辑上正确的。例如，一个数据项Q的初始值为1，事务T将Q值更新为2。那么在事务T提交后，所有的事务在访问数据项Q时，都应该看到Q的值为2。

- 数据库一致性：数据库准确地对真实世界建模并且满足所有数据的一致性限制。这是数据库管理系统的责任。
- 事务一致性：在事务提交前后，数据库都满足一致性。这是应用的责任。

2.3. 隔离性(Isolation)

数据库系统给事务提供一个它们独自运行于整个系统的假象。如同让事务以序列化的顺序独占系统地执行一样，但为了提供更好的并发性从而得到更好性能，数据库系统需要交叉执行不同事务的操作。

并发控制协议 规定数据库管理系统如何在运行时正确地交叉执行多个事务的操作。

- 乐观的：数据库管理系统认为事务之间的冲突是罕见的，所以数据库管理系统在事务提交后处理可能的冲突。
- 悲观的：数据库管理系统认为事务会出现冲突，所以一开始就阻止冲突出现。

数据库管理系统执行操作的顺序称为执行调度 。有以下三种执行调度：

- 顺序调度：在执行任何一个事务时，没有穿插着其他事务的操作的执行调度。
- 等价调度：两个效果一样的不同调度。
- 可串行化的调度：可等价为顺序调度的执行调度。

如果两个来自于不同事务的操作同时作用于一个对象上，可能会产生冲突。 冲突的情形包括：读-写，写-读，写-写。

如果两个调度包含的事务是一样的并且所有的冲突对的顺序也是一样的，那么这两个调度是冲突等价*(Confilict Equivalent)* 的。 如果一个调度冲突等价与一个串行化的调度，那么这个调度是冲突可串行化的*(Conflict Serializable)* 的。简单来说，冲突可串行化的调度保证每个事务执行的结果就像其单独执行一样。检测冲突是否可串行化的的方法包括: 1)交换(swapping)；2)依赖图(dependency graph)，其中每个节点是一个事务，一个有向边Ti→Tj表示Ti中的操作Oi与Tj中的操作Oj冲突。

如果调度S和S'对于一个事务T满足：1）T在S和S'中都读取数据项Q的初始值；2）T在S中读取Q的值由T'产生，那么在S中也如此；3）在S中，T执行最后的Write(Q)，那么在S'中，T也应该执行最后的Write(Q)。那么这两个调度是视图等价*(View Equivalent)* 的。如果一个调度视图等价于一个串行化的调度，那么其是视图可串行化的*(View Serializable)* 的。

通过选择合适的隔离等级，我们能够在并行度与异常容忍度之间做出平衡。在DBMS并发执行中存在的异常(Anomalies)包括：脏读(Dirty Read)，不可重复读(Unrepeatable Read)，幻读(Phantom Read)。SQL标准制定了以下的隔离等级：

等级	脏读	不可重复读	幻读
可串行化的(Serializable)	无	无	无
可重复读(Repeatable Read)	无	无	有
读提交(Read Commited)	无	有	有
读未提交(Read Uncommited)	有	有	有

2.4. 持久性(Durability)

所有提交的事务造成的改变必须是持久的。通过日志和影子页可以保证数据库系统的持久性。

---

3. 基于锁的协议(Lock-Based Protocols)

基于锁的协议通过实现当一个事务在访问一个数据项时，另一事务不能修改该数据项这一目标来控制并发。锁协议指明事务应该什么时候上锁与开锁。

3.1. 锁(Locks)

在DBMS中，锁由锁管理器*(Lock Manager)* 来管理。通过使用锁管理器，DBMS可以实现死锁预防与阻止、支持更多的锁的类型以及提供不同等级的隔离性。合法的调度 指的是满足锁协议指定规则的执行调度。锁协议确保它所有的合法调度都是冲突可串行化的。

在使用锁的过程中可能出现两种问题，一个是在最后一次访问数据后就立刻开锁而造成的不一致性，另一个是多事务的上锁开锁顺序不当而造成的死锁问题。并发控制管理器可以通过回滚一个事务从而开锁来解决后一个问题。

锁管理器在授予锁时需要避免饿死 现象。通过该规则可以避免饿死现象：只有在没有其他的事务正在持有冲突模式的该锁并且没有其他的事物正在等待该锁的情况下，锁管理器才能授予事务锁。

3.1.1. 锁的实现(Implementation of Locking)

事务通过与锁管理器进程*(Lock Manager Process)* 进行交互来对完成对数据项的上锁与开锁。锁管理器进程通过回复授予信息或者请求回滚信息来响应事务的上锁请求信息，通过回复通知信息来响应事务的开锁请求(可能会发送授予锁信息给其他事务)。锁管理器进程维护着一个锁表*(Lock Table)*，每个锁表都是若干个数据项的相关的事务链。具体的算法见18.1.4。

3.1.2. 多粒度锁(Multiple Granularity Locking)

将数据库的所有数据分级得到一个粒度层级树，自底向上可以是记录、文件、区域和数据库。通过给不同粒度的数据加不同的锁，系统可以提高锁的粒度，进而获得更高的并发性。通过引入意向锁*(intention lock)*，数据库系统实现了为不同的粒度数据加锁并且避免判断能否上锁时需要搜索整棵树的情况。在一个结点被显示地上锁前，其所有的祖先都先被上意向锁。多粒度锁策略*(multiple-granularity locking protocol)* 通过使用这些锁来确保串行化。

意向锁引入三种新的锁模式：意向共享模式*(intention-shared mode)*, 意向排斥模式*(intention-exclusive mode)* 和 共享意向排斥模式*(shared intention-exclusive mode)*。

多粒度锁策略在上锁时自底向上，开锁时自顶向下。由于该策略的上锁与开锁是两阶段的，所以避免不了死锁。

当需要上锁相当多的细粒度锁时，可能需要考虑锁升级*(lock escalation)* 来减少开锁过多的代价。

总的来说，DBMS锁一般有五种类型：共享锁(S)、排斥锁(X)、意向共享锁(IS)、意向排斥锁(IX)和共享意向排斥锁(SIX)。结点处于意向锁模式意味着其子结点中存在被显示上排斥锁或共享锁的结点。这样能够减少上锁父结点时的检查。共享互斥锁指的是当前节点获得共享锁，并且子节点有获得互斥锁。理解共享互斥锁的使用情况可以考虑将所有不及格的学生成绩设为及格的SQL语句`UPDATE grades SET grade=60 WHERE grade<60`。在该语句中，需要扫描表的所有行，因此需要对表grades上共享锁，又因为需要改一些行，因此需要对该表上意向排斥锁。这里不存在IX和S的冲突。

3.2. 两阶段锁协议(Two-Phase Locking Protocol)

两阶段锁协议保证调度的冲突可串行化，但不保证不出现死锁。它要求事务在两个阶段发出上锁与开锁的请求：

- 增长阶段(Growing Phase)：事务只有上锁请求而不开锁请求。上锁点*(Lock Point)* 指的是该阶段的最后一次上锁的位置。
- 收缩阶段(Shrinking Phase)：事务只有开锁请求而无上锁请求。

严格两阶段锁协议*(Strict Two-Phase Locking Protocol)* 指定事务的所有排斥锁直到提交前必须一直持有。这能够避免级联回滚调度。强两阶段锁协议*(Rigorous Two-Phase Lokcing Protocol)* 指定事务的所有锁直到提交前必须一直持有。

锁转换*(Lock Conversion)* 指的是共享锁在增长阶段升级为排斥锁，排斥锁在收缩阶段降级为共享锁。

自动锁策略*(Automatic Lock Scheme)* 指的是数据库系统自动地为事务的读写请求上适当的锁，并且在事务提交后开锁。

3.3. 基于图的锁协议(Graph-Based Locking Protocol)

数据库图*(DataBase Graph)* 描述了锁的依赖关系。该图需要数据库系统对数据具有先验的访问顺序知识。具体的算法见18.1.5 。它能够从源头上避免死锁问题以及实现冲突可串行化。但不能处理级联回滚问题，添加提交依赖*(Commit Dependency)* 后能够保证恢复。

3.4. 死锁处理(Deadlock Handling)

处理死锁的方法有死锁预防和死锁发现与恢复两种。前者在死锁出现概率较高的负载中表现较好，而后者在死锁出现频率较低的负载中效率较高。

3.4.1. 死锁预防(Deadlock Prevention)

有两种方法来避免死锁预防。一种是通过组织上锁请求来保证没有循环等待从而避免死锁，另一种是在可能出现死锁情况的时候回滚事务。

- 锁顺序(Lock ordering)

事务遵循预定义的锁层次体系来获取锁，当获取低层次的锁后，不能获取更高层次的锁。这种方法需要精心设计所层次体系。

- 事务时效性(Transaction Aging)

当事务在等待锁时，根据其分配的时间戳选择等待或终止。在wait-die策略中，如果请求事务比持有事务有更小的时间戳，其等待，否则终止。在wound-die策略中，如果请求事务比持有事务有更小的时间戳，持有事务终止并释放锁，否则请求事务等待。

另外的死锁预防策略是基于上锁超时*(lock timeouts)*。

3.4.2. 死锁发现与恢复(Deadlock Detection and Recovery)

死锁发现 系统通过维护一个等待图*(wait-for graph)* 并周期性的检测该图是否有环来确定是否有死锁。等待图的每个结点表示一个事务，Ti→Tj表示Ti等待Tj释放锁。该算法需要确定什么时候对图进行一次搜索。这需要考虑两个因素：死锁的发生频率、死锁影响的事务多少。

从死锁中恢复 系统在从死锁中恢复时需要考虑三个因素：

- 如何选择牺牲者？通常选择回滚代价最小的事务来作为牺牲者，代价最小的衡量指标包括事务已经执行了多久以及大致需要多久执行完、事务已经请求了多少个数据项以及还要多少个数据项、多少个事务与该事务相关。
- 如何回滚？回滚的策略包括完全回滚(回滚一个事务并重启它)和部分回滚(回滚一个事务的一部分使得死锁问题被解决即可)，前者简单而后者高效。
- 如何避免饿死？选择牺牲者的时候把事务回滚次数考虑在内。

---

4. 基于时间戳的协议(Timestamp-Based Protocol)

4.1. 时间戳(Timestamp)

数据库系统使用系统时钟或者逻辑计数器在事务到达时为其分配时间戳。时间戳有R-Timestamp(Q) 和 W-Timestamp(Q)两种，其中Q是数据项。

4.2. 基础的时间戳顺序协议(Basic Timestamp-Order Protocol)

事务的读写请求是否被准许的判断标准涉及到的信息是请求事务的时间戳、数据项的读写时间戳。一旦事务请求不被准许，事务回滚，随后时间戳重新设置并重启。相反，如果事务请求被准许，数据项的时间戳更新。

该协议能够保证冲突可串行化以及避免死锁，但存在事务饥饿的可能、不具有可恢复性以及拷贝数据到私有空间和更新时间戳带来的不小开销。

为了保证可恢复，有三种解决方法：1）事务的所有写在事务末尾执行；2）读Q的请求等待写Q请求的事务提交后被准许；3）读Q事务等待所有写Q事务提交后才能提交来实现。其中，前两种方法还可以解决串联回滚的问题。

4.3. 托马斯写规则(Thomas' Write Rules)

与基础的时间戳顺序协议不同的只是当事务T提交对数据项Q的写请求时，如果Q的写时间戳大于T的时间戳，那么就忽略该请求。

这一规则提高了潜在的并发性，但不能保证冲突可串行化，只能保证视图可串行化。

4.4. 乐观并发控制(Optimistic Concurrency Control,OCC)

对于冲突出现概率很小的工作负载，数据库管理系统采用乐观并发控制来保证事务的隔离性可以获得更小的并发控制开销。乐观并发控制是基于验证的协议(Validation-Based Protocols)。其一共

有三个阶段：

- 读阶段(Read Phase)：事务将需要访问的所有数据都拷贝到自己的私有工作区，对数据的所有更改都先写入该工作区。此时事务的时间戳是无穷大。
- 验证阶段(Validate Phase)：当事务提交时，数据库系统检查它是否与其它事务冲突。此时事务的时间戳被分配。
- 写阶段(Write Phase)：如果事务通过验证，那么其在私有工作区所做的更改将在数据库中生效。

该协议具有以下缺点：

- 读阶段的拷贝造成的开销可能比较大。
- 验证/写阶段造成的开销可能会成为瓶颈。
- 事务重启的代价会比悲观并发控制的大，因为重启在事务已经执行完后发生。
- 时间戳的分配开销。

---

5. 多版本控制协议(Multi-Version Control Protocol)

每个数据项都有若干个版本，多版本控制协议确保选择读的版本能够保证串行化。出于性能考虑，版本的选择还要快而简单。多版本控制协议有四个设计考虑：并发控制协议、版本存储、垃圾回收和索引管理。

5.1. 并发控制协议比较(Concurrency Control Comparisons)

5.1.1. 多版本时间戳顺序(Multiversion Timestamp Ordering)

每个数据项都有若干个版本，每个版本含有三个字段：内容、读时间戳和写时间戳。

读写规则是：1）事务读最近写过的版本。2）如果事务写得过晚，那么就回滚事务。

有效区间：对于一个数据项Q，其版本的时间戳有效区间要么是[x,y)，要么是[x,∞)。这两个有效区间分别对应于该版本在该数据项的版本列表中的非末版本和末版本。

版本删除规则：如果一个数据项的两个版本的写时间戳都小于系统中存在的最旧事务，那么删除这两个版本的更旧版本。

多版本时间戳顺序的好处是没有读请求需要等待，在读请求占比在一定程度上多于写请求的系统中能够取得更好的性能。其坏处是：1）写请求如果被拒绝，只能回滚，代价可能不小；2）每个读请求都需要有两次磁盘访问。

5.1.2. 多版本两阶段锁(Multiversion Two-Phase Locking)

在该协议中，事务被分为只读事务*(read-only transactions)* 和更新事务*(update transactions)* 。可读事务与多版本时间戳顺序一致。在更新事务中，实施强两阶段锁协议，其具体执行是：假定读和写数据项Q

```
Read(Q):
    lock-s(Q)
    read-latest-version(Q)
    unlock-s(Q)

Write(Q):
    lock-x(Q)
    create-new-version(Q)
    write-last-version(Q)
    set-last-version-timestamp(Q, TS-Counter+1)
    inc(TS-Counter)
    unlock-x(Q)
    commit
```

5.1.3. 快照隔离(Snapshot Isolation)

每个事务都有两个时间戳：开始时间戳和提交时间戳，开始时间戳一般在进入验证阶段时得到分配。快照隔离是基于多版本的，每个事务更新一个数据项时都创建一个该数据项的新版本，一个版本只有时间戳。

快照隔离通过使用先提交者赢*(first commiter wins)* 或先更新者先赢*(first commiter wins)* 来避免事务并发执行时的丢失更新*(lost update)* 问题。

快照隔离可以通过在提交阶段检查一致性限制来解决一致性限制问题。为了解决快照隔离的不保证串行化的问题，三种办法可以考虑：

- 使用可串行化的快照隔离，通过追踪事务的读写冲突，维护事务的依赖图来实现。
- 使用不同等级的隔离，例如为只读事务提供快照隔离，为更新事务提供可串行化的隔离等级。
- 为SQL程序员提供的*for update* 从句来制造冲突，从而让数据库系统采用更严格的策略保证串行化。

5.2. 版本存储(Version Storage)

5.2.1. 方法一：仅追加存储(Append-Only Storage)

所有的数据项的所有版本存储与一个表中，其中元组的格式为(Value, Pointer,ValidInterval).

5.2.2. 方法二：时间遍历存储(Time Traverse Storage)

为版本存储维护两个表，一个主表，一个时间遍历表。每次更新数据项，都把主表上该数据项对应的版本元组拷贝到时间遍历表中。

5.2.3. 方法三：变化量存储(Delta Storage)

为版本维护两个表，一个主表，一个变化量表。每次更新数据项，只拷贝有变化的值的变化量到变化量表中，并覆写主表的版本。

5.3. 垃圾回收(Garbage Collection)

一个数据项的一个版本能够被清理时，其意味着其有效时间戳区间最大值小于当前所有事务中最小时间戳的事务。可以从元组层和事务层上进行垃圾回收。

5.3.1. 元组层(Tuple Level)

直接找到可以回收的版本。有后台清理*(Background Vacumming)* 和协作清理*(Cooperative Cleaning)* 两种方法。

5.3.2. 事务层(Transaction Level)

事务追踪使用过的数据项的版本，然后在提交阶段告知数据库管理系统，然后数据库管理系统来决定是否需要进行垃圾回收。这中做法可以让数据库管理系统无需扫描版本即可进行垃圾回收。

5.4. 索引管理(Index Management)

主键的索引直接指向数据项版本号列表的头节点，而辅助索引的版本号通过索引指向一个固定的标识符来实现，例如指向主键。

---

6. 实践中的弱一致性(Weak Levels of Consistency in Practice)

6.1. 二度一致性(Degree-Two Consistency)

使用两种模式的锁来保护数据，共享锁可以在任何地方上锁与开锁，而排斥锁必须在事务提交或者事务终止时开锁。串行化得不到保证。

6.2. 游标稳定性(Cursor Stability)

游标稳定性是一种二度一致性。游标访问元组时上锁，访问后开锁。而排斥锁在事务提交或终止时开锁。

6.3. 用户交互间的并发控制(Concurrency Control Across User Interactions)

为避免不同用户在更新同一数据而引起的冲突，使用版本号来实现并发控制。