LECTURE 12 DISTRIBUTED XACTION

1. 前置知识

- 事务：通常将**并发控制**和**原子提交** 组合起来称为事务
- 并发控制（concurrency ctl）的常用办法：
  - pessimistic 悲观方法：锁。锁的开销和代价很大
  - optimistic 乐观方法 （occ， optimistic concurrency ctl）：事务会持续性地推进，在事务执行结束时，检查事务执行时是否有别的事务干预
- 结果：指数据库实际的 记录的改变（例如增删改元组） 或者是数据库的输出（例如printf）

2. 9.1.5 Before or After Atomicity

- Sequence Coordination 序列协调：事件A先于B发生。要求事件B必须在A完成之后才开始

  通常，可以假定序列协调，并且认为编程者知道并发操作的身份

- Before-or-after Atomicity： 任意两个满足本性质的操作A与B，要么A**完全**在B之前发生，要么A**完全**在B之后发生

  与Sequence coordination 不同，before-or-after atomicity 并不知道其他使用共享变量的并发操作的身份。这导致了，在多线程环境之下，很难通过一些显式的编程步骤来协调多线程的行为，因此需要一些隐式的、自动的处理共享变量的机制。

  若所有操作都是before-or-after atomicity的，则这些操作必定是可串行化的！

3. 9.1.6 正确性与串行化

- 事务的ACID特性：
  - Atomic：在没有故障的情况下，所有操作要么都做，要么都不做。也即**all-or-none atomicity**
  - Consistency：本Lecture不重要
  - Isolation：事务之间不应该相互影响。也就是意味着事务操作室可线性化的
  - Durable：在事务提交之后，内容的改变和写入是持久的，不会被擦除

- 可串行化：并发事务的执行结果是可串行化的，如果：

  存在一个由这些事务组成的串行化序列， 事务按照此串行序列执行结果与并发执行的结果一致

- 正确性：并发行为之间的协调是正确的，当：

  每个结果都可以由一个串行执行的结果得到。 当所有的操作都是before-or-after atomicity时，则一定能满足上面的条件！

- 新状态是正确的，当：

  - 旧状态是正确的
  - 行为（action）本身能将一个正确的状态转变为另一个正确的状态

- 并不要求状态的转换沿着某一个特定的路径进行，只要求最终状态是任意一个可接受的状态即可（即串行化执行得到的状态）。同时，有很多路径可以到达可接受的新状态，但是要从这些路径之中找到性能最好的路径是相当困难的。

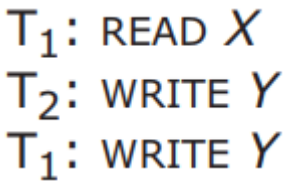
4. 9.5.2 简单封锁

- 规则：
  - 在读写 每一个共享数据之前 必须要获得该共享数据的锁
  - 共享数据的锁应该被释放，仅当：
    - 事务完成了最后的更新并且已经提交（commit），或
    - 数据均已被恢复且事务终止（abort）
- 封锁点（lock point）：获得所有的锁的第一个时刻
- 锁集合 （lock set）：到达 封锁点 时已经获得的锁。因此一个数据对象不可能同时出现在两个事务的锁集合中。
- 如何保证 简单封锁：
  - 通过锁管理器实现（lock manager）
  - 锁管理器要求：锁集合被传递给**begin_xaction** 操作作为参数。**begin_xaction** 操作会获取锁集合的中所有的锁。当无法获取这些锁时，则会等待。
- 锁管理器的作用：

    - 保证简单封锁（将锁集合传递给begin_transaction）
    - 对于读数据和改变日志的调用，会检查这些调用所指向的共享变量是否在锁集合中
    - 当事务调用commit或abort时，自动释放所有的锁
- 简单封锁协议的优缺点：
    - 优点：当事务本身很简单时，简单封锁协议很高效
    - 缺点：当事务可能读取的数据对象的集合比事务实际读取的数据对象的集合更大时，可能会影响并发程度

5. 9.5.3 两段封锁

- 规则：
    - 在读写 每一个共享数据之前 必须要获得该共享数据的锁
    - 事务可以释放锁，仅当：
        - 事务到达了封锁点
        - 共享数据是只读的，并且共享数据后续不会再读取（即使是abort）
- 两段封锁可以划分为：锁的数量单调上升（第一阶段）+ 锁的数量单调下降（第二阶段）。封锁点作为分隔
- 锁管理器如何实现两段封锁：
    - 拦截 所有 读数据/写数据的调用
    - 对于第一次使用的共享变量 尝试获得锁。
    - 当锁管理器 拦截commit/abort 调用或是事务END记录的调用 时，会释放所有的锁
- 两段锁协议保证了before-or-after atomicity。但两段锁协议仍有可能降低并发程度。如下图中，并发是正确的，但是两段锁协议会避免下面的这种情况发生：



- 事务的中止（abort）与锁：事务的锁仅当：最后的更新已完成且已提交/修改已回滚且中止后才会释放锁。这个策略对于事务的中止很重要。若在完成中止之前已经释放了锁，则别的事务可能会修改本事务修改的数据，因此不可能再回滚本事务的修改。可以将事务的中止看做是没有发生任何变化的提交了的事务。
- 崩溃恢复与锁：
  - 此处崩溃恢复是使用日志的方式进行
  - 锁本身被存在易失性介质中（可以存在易失性介质中的原因：在崩溃的瞬间，各个未完成的事务之间的锁集合必定是不相交的（锁集合的定义），因此在恢复时可以不需要锁，但是需要阻止新的事务的到来。）
  - 在崩溃恢复结束之前，不允许接受新的事务
  - 当崩溃恢复结束时，不能再持有锁（锁仅被用于协调未完成的事务）
  - 锁机制为事务创建了一个串行化的序列，而崩溃恢复时，也是在重复这样的串行化序列，因此能够保证恢复之后的串行化序列和事务崩溃之前的串行化序列一致。

6. 9.6.3 多站点（site）分布式2PC

- 结构：
  - TC：transaction coordinator，需要处理整个事务，由于数据分布在不同的work site（不包含TC的所有）上，因此需要向work site发送消息，这些work site保存了事务中涉及的数据。每个事务可以分成若干个组件事务（component xaction）。
  - work site维护的数据结构：
    - 每个事务都有一个transaction ID/TID，当事务开始时，会由coordinator指定TID
    - 维护一张表格，维护了各个事务执行的状态
- 前提与假设：每个站点自己都能实现本地的事务
- 正确性：当所有的站点都提交了组件事务或中止组件事务时，则整个事务时正确的；若部分站点提交组件事务且部分站点中止组件事务，则事务出错
- 过程：
  - **协调器（coordinator，设为Alice）** 为整个事务创建顶层 **结果记录（outcome record）**。而后协调器向别的worker站点（设为Bob） 通过RPC发出**Prepare消息**

From:Alice
To: Bob
Re: my transaction 271

Please do X as part of my transaction.

- worker收到RPC之后，会检查是否重复，创建一个自己的事务（该事务被嵌套在Alice的事务之内，会收到上层事务的监督和控制）。worker随后将新的数据写入自己的log（用于恢复！当worker发送了prepared消息后，整个事务可能可以提交，若没有log，则在当前worker崩溃之后不能保证all-or-none，因为至少本机的组件事务无法进行），再向Alice报告自己的事务已经可以提交：**这一消息称为prepared消息！**
  - 若coordinator长时间未收到某个worker site的prepared消息，则可以重新发送prepare消息，或者abort
- Alice收到 **所有** worker的预提交之后，向所有的worker发送RPC，内容为：各个worker的事务可以提交/中止。当coordinator在这一阶段崩溃时，可能导致一些worker site 得到了commit message，另一些worker site 没有收到。因此coordinator在发送commit message之前，必须要先使用log记录事务的状态， 崩溃恢复时根据状态来重发commit或者abort消息。**因此有些worker site可能会重复收到commit message！**
- worker 将新数据 通过日志写入存储器中的数据库，而后删除日志，释放锁，向Alice发送ACK消息

- 2PC缺点：
  - 太慢，消息太多了
  - 在worker site 发出了prepared消息后，收到commit消息之前，worker site必须等待！因为此时worker site并不知道整个事务应该commit 还是abort，需要等待coordinator的决定！