Mysql 技术内幕：InnoDB 学习（7）

jask

2024-08-11

Mysql 技术内幕

事务

InnoDB 存储引擎中的事务完全符合 ACID 的特性。ACID 是以下 4 个词的缩写：

原子性（atomicity）

一致性（consistency）

隔离性（isolation）

持久性（durability）

分类

从事务理论的角度来说，可以把事务分为以下几种类型：

扁平事务（Flat Transactions）

带有保存点的扁平事务（Flat Transactions with Savepoints）

链事务（Chained Transactions）

嵌套事务（Nested Transactions）

分布式事务（Distributed Transactions）

扁平事务（Flat Transaction）是事务类型中最简单的一种，但在实际生产环境中，这可能是使用最为频繁的事务。在扁平事务中，所有操作都处于同一层次，其由 BEGIN WORK 开始，由 COMMIT WORK 或 ROLLBACK WORK 结束，其间的操作是原子的，要么都执行，要么都回滚。因此扁平事务是应用程序成为原子操作的基本组成模块。

扁平事务的主要限制是不能提交或者回滚事务的某一部分，或分几个步骤提交。

带有保存点的扁平事务（Flat Transactions with Savepoint），除了支持扁平事务支持的操作外，允许在事务执行过程中回滚到同一事务中较早的一个状态。这是因为某些事务可能在执行过程中出现的错误并不会导致所有的操作都无效，放弃整个事务不合乎要求，开销

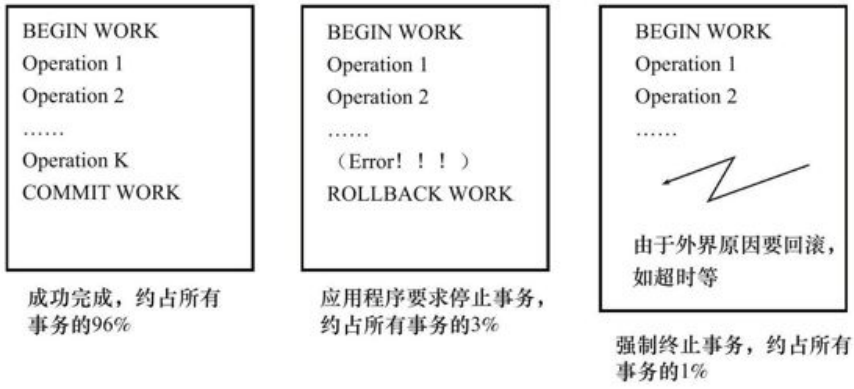

Figure 1： 三种结果

也太大。保存点（Savepoint）用来通知系统应该记住事务当前的状态，以便当之后发生错误时，事务能回到保存点当时的状态。

链事务（Chained Transaction）可视为保存点模式的一种变种。带有保存点的扁平事务，当发生系统崩溃时，所有的保存点都将消失，因为其保存点是易失的（volatile），而非持久的（persistent）。这意味着当进行恢复时，事务需要从开始处重新执行，而不能从最近的一个保存点继续执行。

链事务的思想是：在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要开始的事务。注意，提交事务操作和开始下一个事务操作将合并为一个原子操作。这意味着下一个事务将看到上一个事务的结果，就好像在一个事务中进行的一样。

链事务与带有保存点的扁平事务不同的是，带有保存点的扁平事务能回滚到任意正确的保存点。而链事务中的回滚仅限于当前事务，即只能恢复到最近一个的保存点。对于锁的处理，两者也不相同。链事务在执行 COMMIT 后即释放了当前事务所持有的锁，而带有保存点的扁平事务不影响迄今为止所持有的锁。

嵌套事务（Nested Transaction）是一个层次结构框架。由一个顶层事务（top-level transaction）控制着各个层次的事务。顶层事务之下嵌套的事务被称为子事务（sub-transaction），其控制每一个局部的变换。

下面给出 Moss 对嵌套事务的定义：

1）嵌套事务是由若干事务组成的一棵树，子树既可以是嵌套事务，也可以是扁平事务。

2）处在叶节点的事务是扁平事务。但是每个子事务从根到叶节点的距离可以是不同的。

3）位于根节点的事务称为顶层事务，其他事务称为子事务。事务的前驱称（predecessor）为父事务（parent），事务的下一层称为儿子事务（child）。

4）子事务既可以提交也可以回滚。但是它的提交操作并不马上生效，除非其父事务已经提交。因此可以推论出，任何子事物都在顶层事务提交后才真正的提交。

5）树中的任意一个事务的回滚会引起它的所有子事务一同回滚，故子事务仅保留 A、C、I 特性，不具有 D 的特性。

原子性、一致性、持久性通过数据库的 redo log 和 undo log 来完成。redo log 称为重做日志，用来保证事务的原子性和持久性。undo log 用来保证事务的一致性。

当通过保存点技术来模拟嵌套事务时，用户无法选择哪些锁需要被子事务继承，哪些需要被父事务保留。这就是说，无论有多少个保存点，所有被锁住的对象都可以被得到和访问。而在嵌套查询中，不同的子事务在数据库对象上持有的锁是不同的。例如有一个父事务 P 1 ，其持有对象 X 和 Y 的排他锁，现在要开始一个调用子事务 P 11 ，那么父事务 P 1 可以不传递锁，也可以传递所有的锁，也可以只传递一个排他锁。如果子事务 P 11 中还要持有对象 Z 的排他锁，那么通过反向继承（counter-inherited），父事务 P 1 将持有 3 个对象 X、Y、Z 的排他锁。如果这时又再次调用了一个子事务 P 12 ，那么它可以选择传递那里已经持有的锁。

如果系统支持在嵌套事务中并行地执行各个子事务，在这种情况下，采用保存点的扁平事务来模拟嵌套事务就不切实际了。这从另一个方面反映出，想要实现事务间的并行性，需要真正支持的嵌套事务。

日志

redo 重做日志用来实现事务的持久性，即事务 ACID 中的 D。其由两部分组成：一是内存中的重做日志缓冲（redo log buffer），其是易失的；二是重做日志文件（redo log file），其是持久的。

InnoDB 是事务的存储引擎，其通过 Force Log at Commit 机制实现事务的持久性，即当事务提交（COMMIT）时，必须先将该事务的所有日志写入到重做日志文件进行持久化，待事务的 COMMIT 操作完成才算完成。这里的日志是指重做日志，在 InnoDB 存储引擎中，由两部分组成，即 redo log 和 undo log。redo log 用来保证事务的持久性，undo log 用来帮助事务回滚及 MVCC 的功能。redo log 基本上都是顺序写的，在数据库运行时不需要对 redo log 的文件进行读取操作。而 undo log 是需要进行随机读写的。

为了确保每次日志都写入重做日志文件，在每次将重做日志缓冲写入重做日志文件后，InnoDB 存储引擎都需要调用一次 fsync 操作。由于重做日志文件打开并没有使用 O_DIRECT 选项，因此重做日志缓冲先写入文件系统缓存。为了确保重做日志写入磁盘，必须进行一次 fsync 操作。由于 fsync 的效率取决于磁盘的性能，因此磁盘的性能决定了事务提交的性能，也就是数据库的性能。

InnoDB 存储引擎允许用户手工设置非持久性的情况发生，以此提高数据库的性能。即当事务提交时，日志不写入重做日志文件，而是等待一个时间周期后再执行 fsync 操作。由于并非强制在事务提交时进行一次 fsync 操作，显然这可以显著提高数据库的性能。但是当数据库发生宕机时，由于部分日志未刷新到磁盘，因此会丢失最后一段时间的事务。

在 MySQL 数据库中还有一种二进制日志（binlog），其用来进行 POINT-IN-TIME（PIT）的恢复及主从复制（Replication）环境的建立。从表面上看其和重做日志非常相似，都是记录了对于数据库操作的日志。然而，从本质上来看，两者有着非常大的不同。

首先，重做日志是在 InnoDB 存储引擎层产生，而二进制日志是在 MySQL 数据库的上层产生的，并且二进制日志不仅仅针对于 InnoDB 存储引擎，MySQL 数据库中的任何存储引擎对于数据库的更改都会产生二进制日志。

其次，两种日志记录的内容形式不同。MySQL 数据库上层的二进制日志是一种逻辑日志，其记录的是对应的 SQL 语句。而 InnoDB 存储引擎层面的重做日志是物理格式日志，其记录的是对于每个页的修改。

二进制日志只在事务提交完成后进行一次写入。而 InnoDB 存储引擎的重做日志在事务进行中不断地被写入，这表现为日志并不是随事务提交的顺序进行写入的。

Log Block 若一个页中产生的重做日志数量大于 512 字节，那么需要分割为多个重做日志块进行存储。此外，由于重做日志块的大小和磁盘扇区大小一样，都是 512 字节，因此重做日志的写入可以保证原子性，不需要 doublewrite 技术。

Log Group log group 为重做日志组，其中有多个重做日志文件。虽然源码中已支持 log group 的镜像功能，但是在 ha_innobase.cc 文件中禁止了该功能。因此 InnoDB 存储引擎实际只有一个 log group。

log group 是一个逻辑上的概念，并没有一个实际存储的物理文件来表示 log group 信息。log group 由多个重做日志文件组成，每个 log group 中的日志文件大小是相同的，且在 InnoDB 1.2 版本之前，重做日志文件的总大小要小于 4GB（不能等于 4GB）。从 InnoDB 1.2 版本开始重做日志文件总大小的限制提高为了 512GB。InnoSQL 版本的 InnoDB 存储引擎在 1.1 版本就支持大于 4GB 的重做日志。

在 InnoDB 存储引擎运行过程中，log buffer 根据一定的规则将内存中的 log block 刷新到磁盘。这个规则具体是：

事务提交时

当 log buffer 中有一半的内存空间已经被使用时

log checkpoint 时

对于 log block 的写入追加（append）在 redo log file 的最后部分，当一个 redo log file 被写满时，会接着写入下一个 redo log file，其使用方式为 round-robin。

重做日志格式 不同的数据库操作会有对应的重做日志格式。此外，由于 InnoDB 存储引擎的存储管理是基于页的，故其重做日志格式也是基于页的。虽然有着不同的重做日志格式，但是它们有着通用的头部格式。

redo_log_type：重做日志的类型。

space：表空间的 ID。

page_no：页的偏移量。

LSN LSN 是 Log Sequence Number 的缩写，其代表的是日志序列号。在 InnoDB 存储引擎中，LSN 占用 8 字节，并且单调递增。LSN 表示的含义有：重做日志写入的总量

checkpoint 的位置

页的版本

LSN 表示事务写入重做日志的字节的总量。例如当前重做日志的 LSN 为 1 000，有一个事务 T1 写入了 100 字节的重做日志，那么 LSN 就变为了 1100，若又有事务 T2 写入了 200 字节的重做日志，那么 LSN 就变为了 1 300。可见 LSN 记录的是重做日志的总量，其单位为字节。

LSN 不仅记录在重做日志中,还存在于每个页中。在每个页的头部,有一个值 FIL_PAGE_LSN，记录了该页的 LSN。在页中，LSN 表示该页最后刷新时 LSN 的大小。因为重做日志记录的是每个页的日志，因此页中的 LSN 用来判断页是否需要进行恢复操作。例如，页 P1 的 LSN 为 10 000，而数据库启动时，InnoDB 检测到写入重做日志中的 LSN 为 13 000，并且该事务已经提交，那么数据库需要进行恢复操作，将重做日志应用到 P1 页中。同样的，对于重做日志中 LSN 小于 P1 页的 LSN，不需要进行重做，因为 P1 页中的 LSN 表示页已经被刷新到该位置。

恢复 InnoDB 存储引擎在启动时不管上次数据库运行时是否正常关闭，都会尝试进行恢复操作。因为重做日志记录的是物理日志，因此恢复的速度比逻辑日志，如二进制日志，要快很多。与此同时，InnoDB 存储引擎自身也对恢复进行了一定程度的优化，如顺序读取及并行应用重做日志，这样可以进一步地提高数据库恢复的速度。

InnoDB 存储引擎在启动时不管上次数据库运行时是否正常关闭，都会尝试进行恢复操作。因为重做日志记录的是物理日志，因此恢复的速度比逻辑日志，如二进制日志，要快很多。与此同时，InnoDB 存储引擎自身也对恢复进行了一定程度的优化，如顺序读取及并行应用重做日志，这样可以进一步地提高数据库恢复的速度。

可以看到记录的是页的物理修改操作，若插入涉及 B+ 树的 split，可能会有更多的页需要记录日志。此外，由于重做日志是物理日志，因此其是幂等的。

undo redo 存放在重做日志文件中，与 redo 不同，undo 存放在数据库内部的一个特殊段（segment）中，这个段称为 undo 段（undo segment）。undo 段位于共享表空间内。可以通过 py_innodb_page_info.py 工具来查看当前共享表空间中 undo 的数量。

用户通常对 undo 有这样的误解：undo 用于将数据库物理地恢复到执行语句或事务之前的样子——但事实并非如此。undo 是逻辑日志，因此只是将数据库逻辑地恢复到原来的样子。所有修改都被逻辑地取消了，但是数据结构和页本身在回滚之后可能大不相同。这是因为在多用户并发系统中，可能会有数十、数百甚至数千个并发事务。数据库的主要任务就是协调对数据记录的并发访问。比如，一个事务在修改当前一个页中某几条记录，同时还有别的事务在对同一个页中另几条记录进行修改。因此，不能将一个页回滚到事务开始的样子，因为这样会影响其他事务正在进行的工作。

当 InnoDB 存储引擎回滚时，它实际上做的是与先前相反的工作。对于每个 INSERT，InnoDB 存储引擎会完成一个 DELETE；对于每个 DELETE，InnoDB 存储引擎会执行一个 INSERT；对于每个 UPDATE，InnoDB 存储引擎会执行一个相反的 UPDATE，将修改前的行放回去。

最后也是最为重要的一点是，undo log 会产生 redo log，也就是 undo log 的产生会伴随着 redo log 的产生，这是因为 undo log 也需要持久性的保护。

undo 存储管理 InnoDB 存储引擎对 undo 的管理同样采用段的方式。但是这个段和之前介绍的段有所不同。首先 InnoDB 存储引擎有 rollback segment，每个回滚段种记录了 1024 个 undo log segment，而在每个 undo log segment 段中进行 undo 页的申请。共享表空间偏移量为 5 的页（0，5）记录了所有 rollback segment header 所在的页，这个页的类型为 FIL_PAGE_TYPE_SYS。

在 InnoDB 存储引擎中，undo log 分为：

insert undo log

update undo log

insert undo log 是指在 insert 操作中产生的 undo log。因为 insert 操作的记录，只对事务本身可见，对其他事务不可见（这是事务隔离性的要求），故该 undo log 可以在事务提交后直接删除。不需要进行 Purge 操作。

update undo log 记录的是对 delete 和 update 操作产生的 undo log。该 undo log 可能需要提供 MVCC 机制，因此不能在事务提交时就进行删除。提交时放入 undo log 链表，等待 purge 线程进行最后的删除。

Purge delete 和 update 操作可能并不直接删除原有的数据。

purge 用于最终完成 delete 和 update 操作。这样设计是因为 InnoDB 存储引擎支持 MVCC，所以记录不能在事务提交时立即进行处理。这时其他事物可能正在引用这行，故 InnoDB 存储引擎需要保存记录之前的版本。而是否可以删除该条记录通过 purge 来进行判断。若该行记录已不被任何其他事务引用，那么就可以进行真正的 delete 操作。可见，purge 操作是清理之前的 delete 和 update 操作，将上述操作“最终”完成。而实际执行的操作为 delete 操作，清理之前行记录的版本。

group Commit 若事务为非只读事务，则每次事务提交时需要进行一次 fsync 操作，以此保证重做日志都已经写入磁盘。当数据库发生宕机时，可以通过重做日志进行恢复。虽然固态硬盘的出现提高了磁盘的性能，然而磁盘的 fsync 性能是有限的。为了提高磁盘 fsync 的效率，当前数据库都提供了 group commit 的功能，即一次 fsync 可以刷新确保多个事务日志被写入文件。对于 InnoDB 存储引擎来说，事务提交时会进行两个阶段的操作：

1）修改内存中事务对应的信息，并且将日志写入重做日志缓冲。

2）调用 fsync 将确保日志都从重做日志缓冲写入磁盘。

然而在 InnoDB1.2 版本之前，在开启二进制日志后，InnoDB 存储引擎的 group commit 功能会失效，从而导致性能的下降。并且在线环境多使用 replication 环境，因此二进制日志的选项基本都为开启状态，因此这个问题尤为显著。

导致这个问题的原因是在开启二进制日志后，为了保证存储引擎层中的事务和二进制日志的一致性，二者之间使用了两阶段事务，其步骤如下：

1）当事务提交时 InnoDB 存储引擎进行 prepare 操作。

2）MySQL 数据库上层写入二进制日志。

3）InnoDB 存储引擎层将日志写入重做日志文件。

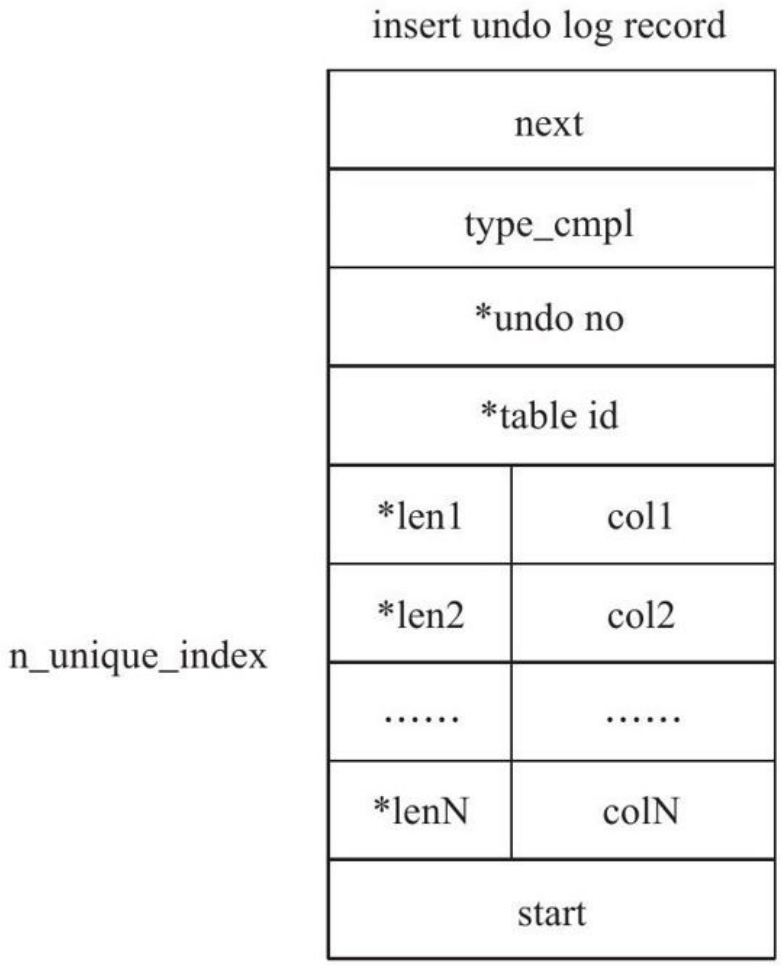

图 7-14 insert undo log的格式

Figure 2: Insert Undo Log 格式

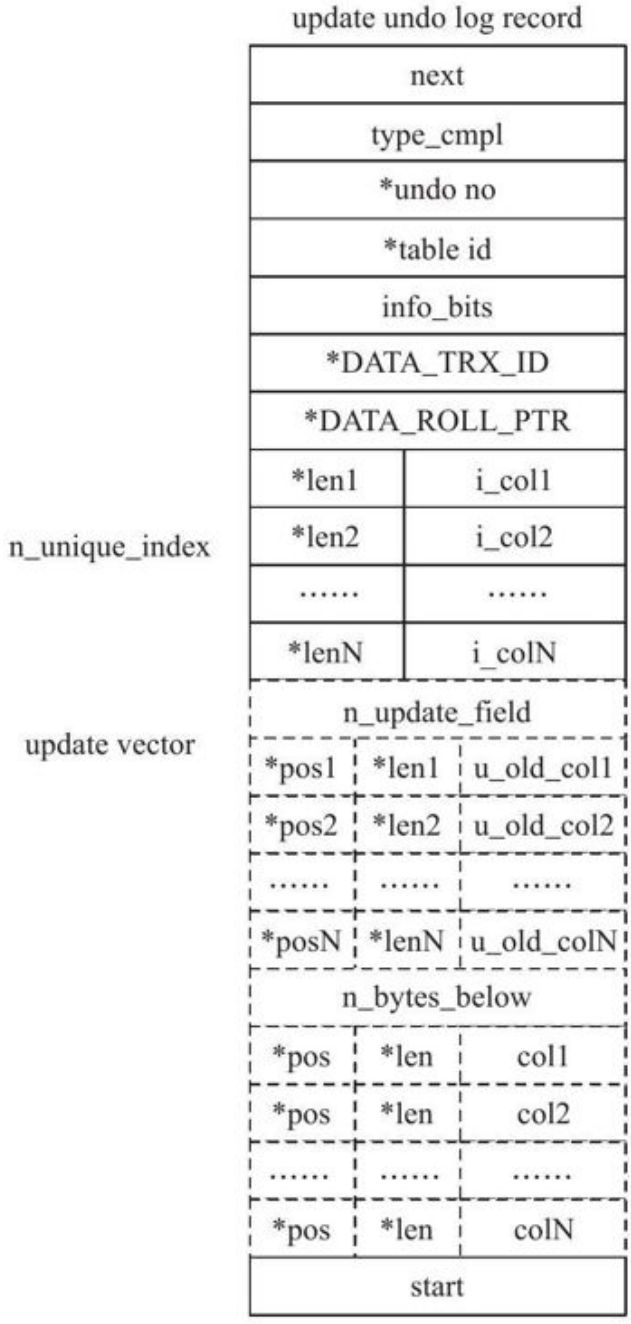

图 7-15 update undo log格式

Figure 3: Update Undo Log 格式

a）修改内存中事务对应的信息，并且将日志写入重做日志缓冲。

b）调用 fsync 将确保日志都从重做日志缓冲写入磁盘。

分布式事务

InnoDB 存储引擎提供了对 XA 事务的支持，并通过 XA 事务来支持分布式事务的实现。分布式事务指的是允许多个独立的事务资源（transactional resources）参与到一个全局的事务中。事务资源通常是关系型数据库系统，但也可以是其他类型的资源。全局事务要求在其中的所有参与的事务要么都提交，要么都回滚，这对于事务原有的 ACID 要求又有了提高。另外，在使用分布式事务时，InnoDB 存储引擎的事务隔离级别必须设置为 SERIALIZABLE。

XA 事务允许不同数据库之间的分布式事务，如一台服务器是 MySQL 数据库的，另一台是 Oracle 数据库的，又可能还有一台服务器是 SQL Server 数据库的，只要参与在全局事务中的每个节点都支持 XA 事务。

XA 事务由一个或多个资源管理器(Resource Managers)、一个事务管理器(Transaction Manager）以及一个应用程序（Application Program）组成。

资源管理器：提供访问事务资源的方法。通常一个数据库就是一个资源管理器。

事务管理器：协调参与全局事务中的各个事务。需要和参与全局事务的所有资源管理器进行通信。

应用程序：定义事务的边界，指定全局事务中的操作。

分布式事务使用两段式提交（two-phase commit）的方式。在第一阶段，所有参与全局事务的节点都开始准备（PREPARE），告诉事务管理器它们准备好提交了。在第二阶段，事务管理器告诉资源管理器执行 ROLLBACK 还是 COMMIT。如果任何一个节点显示不能提交，则所有的节点都被告知需要回滚。可见与本地事务不同的是，分布式事务需要多一次的 PREPARE 操作，待收到所有节点的同意信息后，再进行 COMMIT 或是 ROLLBACK 操作。

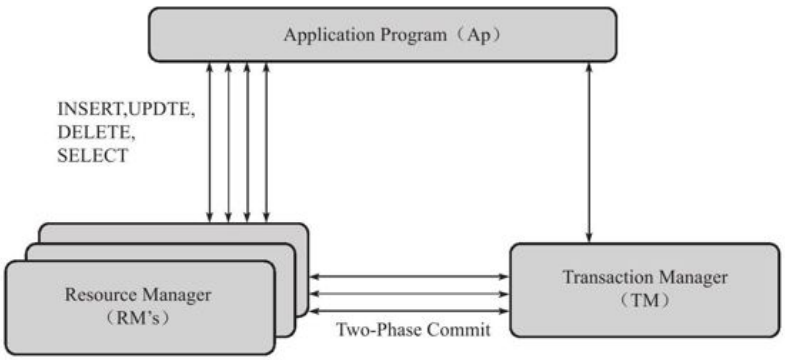

图 7-22 分布式事务模型

Figure 4: 分布式事务模型

内部 XA 事务　在 MySQL 数据库中还存在另外一种分布式事务，其在存储引擎与插件之间，又或者在存储引擎与存储引擎之间，称之为内部 XA 事务。