

(26)

第七章 事务管理

7.1 引言

事务是 DBMS 中的基本执行单位。

在任何情况下，DBMS 都应保证事务的 ACID 性质。

— 系统正常运行时

— 系统发生故障时

— 单个应用运行时

— 多个应用并行运行时

恢复技术：保证事务在故障发生时满足 ACID 性质的措施

并发控制技术：保证多个事务在并行执行时满足 ACID 的措施

事务管理：恢复和并发控制是保证事务正确运行的两项基本技术，它们被合称为事务管理。

7.2 数据库恢复概述

故障：→ 对于不同的故障，还原的方式也是不同的

硬件的故障、软件的错误、操作员的失误、误以及恶意的破坏

恢复：

数据库管理系统（恢复子系统）把数据库从错误状态恢复到某一已知的正确状态（亦称为一致状态或完整状态）的功能。

7.3 故障的类型

1. 事务内部的故障

事务内部的故障有的可以通过事务程序本身发现，有的是非预期的，不能由事务程序处理。

例如：银行转账业务，把金额从账户 A 转给账户 B。

Begin transaction

读取账户 A 的余额 balance；

事务内部故障：可能出现转

Balance = Balance - Amount；

账金额不够的情况

if (Balance < 0) then { → }

这个例子中可以通过程序来解决

打印“金额不足，不能转账”；

loose leaf: 7.8mm x 27 lines * gimen

Rollback; {
else {

读取账户B的余额 Balance 1;

Balance 1 = Balance 1 + Amount;

写回 Balance 1;

Commit; }

~~2. 事务故障~~ → 程序无法判断 ^{发生} → 事务内部故障

事务内部更多的故障是非预期的，是不能由应用程序处理的。如溢出、并发事务发生死锁而被选中撤销该事务，违反了某些完整性限制等。以后，事务故障仅指这类非预期的故障。→ 程序无法判断的故障

事务故障意味着事务没有达到预期的终点（commit 或者显示的 rollback），因此，数据库可能处于不正确状态。

恢复程序在不破坏其它事务运行的情况下，强行回滚（Rollback）该事务，即撤消该事务已经作出的任何对数据库的修改，使得该事务好像根本没有启动一样。这类恢复操作称为事务撤消（UNDO）。

2. 系统故障

指造成系统停止运转的任何事件，使得系统需要重新启动。例如，特定类型的硬件错误（CPU 故障）、操作系统故障、DBMS 代码错误、突然停电等。

这类故障影响正在运行的所有事务，但不破坏数据库。这时内存内容，尤其是数据库缓冲区（在内存）中的内容可能会丢失，所有运行事务都必须重启动。

发生系统故障时，一些尚未完成的事务的结果可能已进入物理数据库。有些已完成的事务可能有一部分甚至全部留在缓冲区，尚未写回到磁盘的物理数据库中，从而造成数据库可能处于不正确的状态。

3. 介质故障

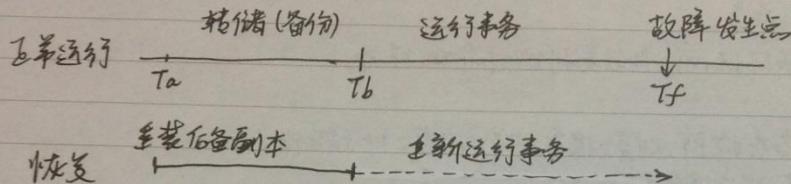
系统故障简称软故障（Soft Crash），介质故障称为硬故障（Hard Crash）。

硬故障指外存故障，如磁盘损坏、磁头碰撞、瞬时强磁场干扰等。

- 这类故障将破坏数据库或部分数据库，并影响正在存取这部分数据的所有事务。 \rightarrow 无法恢复
 - 这类故障比前两类故障发生的可能性小得多，但破坏性最大。
 - 计算机病毒
 - 具有破坏性，可以自我复制的计算机程序。
 - 总结各类故障，对数据库的影响有2种可能性。
 - 一是数据库本身被破坏。
 - 二是数据库没有破坏，但数据可能不正确。这是因为事务的运行被非正常终止造成的。
 - 恢复的原理十分简单，可以用一个词来概括：冗余。这就是说，数据库中任何一部分被破坏的或不正确的数据可以根据存储在系统别处的冗余数据来重建。
 - 恢复的基本原理简单但实现技术的细节却相当复杂。
- ## 7.4 恢复的实现技术
- 1. 概述
 - 恢复机制涉及的2个关键问题是：
 - ①如何建立冗余数据
 - ②如何利用这些冗余实现数据库恢复。
 - 建立冗余数据最常用的技术是数据备份和事务日志文件通常在一个数据库系统中，这两种方法是一起使用的。
 - 2. 数据库备份
 - 所谓备份即DBA定期将整个数据库复制到磁带或另一个磁盘上保存起来的过程。这些备用的数据文件称为后备副本或恢复副本。
 - 当数据库遭到破坏后可以将后备副本重新装入，但通常后备副本只能将数据库恢复到备份时的状态，要想恢复到故障发生时的状态，必须重新

运行自备份以后的所有更新事务。事务

例如在下图中，系统在 T_a 时刻停止进行数据库备份，在 T_b 时刻备份完毕，得到 T_b 时刻的数据库一致性副本。



系统运行到了 T_f 时刻发生故障。^{为恢复数据库}首先由 DBA 重装数据库
到最近的备份副本，将数据库恢复到 T_b 时刻的状态，然后重新运行自 T_b 时刻至 T_f
时刻的所有更新事务（使用日志文件），这样就把数据库恢复到故障发生
前的一致状态。
日志文件：记录更新事务的。

备份十分耗费时间和资源，不能频繁进行。DBA 应根据数据库的使用
情况确定一个适当的备份周期。

• 备份按照是否联机分为：

— 脱机备份 (off line) → 数据库停止向外提供服务，只备份

— 联机备份 (on line) → ——一边备份，一边向外提供服务。

• 备份按照是否全部备份分为：

— 海量备份：每次备份全部数据库（全库备份）

— 增量备份：每次只备份上一次备份后更新过的数据。

结合起来使用 → 前提是使用了海量备份。

3. 数据库日志

1) 日志文件的格式和内容

主要记录 insert, update, delete 操作

• 日志文件是用来记录事务对数据库的更新操作的文件。

• 不同数据库系统采用的日志文件格式并不完全一样，概括起来主要有
2 种格式：

— 以记录为单位的日志文件

— 以数据块为单位的日志文件。

- 以记录为单位的日志文件包括:
 - 各事务的开始 (begin transaction) 标记
 - — — — 事务 (commit 或 rollback) 标记
 - — — — 所有更新操作
 - 这里每个事务开始的标记、每个事务的结束标记和每个更新操作均作为日志文件中的一个日志记录 (log record)
 - 每个日志记录的内容主要包括:
 - 事务标识 TID (标明是哪个事务)
 - 操作的类型 (插入、删除或修改)
 - 操作对象 (记录内部标识)
 - 更新前数据的旧值 (对 insert 而言, 此项为空值)
 - 更新后数据的新值 (对 delete 而言, 此项为空值)

2) 日志文件的作用

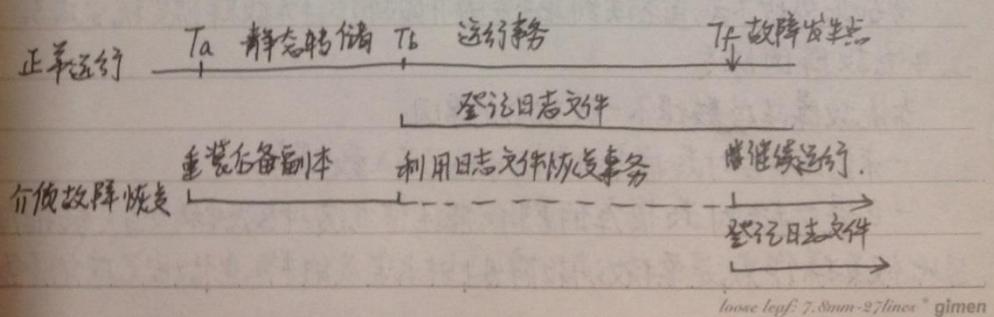
日志文件在数据恢复中起着非常重要的作用。可以用来进行事务故障恢复和系统故障恢复，并协助后备副本进行介质故障恢复。

具体地讲，事务故障恢复和系统故障必须用日志文件。

- 在动态备份方式中必须建立日志文件，代理副本和日志文件综合起来才能有效地恢复数据库。

在竞争态准备方式中，也可以建立日志文件。

当数据库毁坏后可重新装入后援副本把数据库恢复到备份结束时刻的正确状态,然后利用日志文件,把已完成的事务进行重做处理,对故障发生时尚未完成的事务进行撤销处理。



3) 登记日志文件 (logging)

为保证数据库是可恢复的，登记日志文件必须遵循 2 条原则：

1. 登记的次序要和各原子并发事务执行的时间次序

2. 必须先写日志文件，后写数据库

如果先写日志，但没有修改数据库，按日志文件恢复时只不过是多执行了一次不必要的 ROLLBACK 操作，并不会影响数据库的正确性。所以为了安全，一定要先写日志文件，即首先将日志记录写到日志文件中，然后写数据库的修改。

这就是所谓“提前写日志”的原则。

7.5 恢复的策略

1. 事务故障的恢复

事务故障是指事务在运行至正常终止前被中断。这时恢复子系统应利用日志文件撤销操作 (UNDO) 此事务已对数据库进行的修改。

事务故障的恢复是由系统自动完成的，对用户是透明的。系统的恢复步骤是：(4步)

用户无法干涉

1) 仅向扫描文件日志 (即从最后向前扫描日志文件)，查找该事务的更新操作。
操作。从后往前查找。

2) 对该事务的更新操作执行逆操作。即将日志记录中“更新前的值”写入数据库，这样，如果记录中是插入操作，则相当于做删除操作 (因为此时“更新前的值”为空)。若记录中是删除操作，则做插入操作，若是修改操作，则相当于用修改前值代替修改后值。

3) 继续仅向扫描日志文件，查找该事务的其它更新操作，并做同样处理。

4) 如此处理下去，直至读到此事务的开始标记，事务故障恢复就完成了。

2. 系统故障的恢复

系统故障造成数据不一致状态的原因：

一 未完成事务对数据库的更新可能已写入数据库；

二 已提交事务对数据库的更新可能还留在缓冲区没来得及写入数据库。

因此恢复操作就是要取消故障发生时未完成的事务，重做已完成的事务。

- 系统故障的恢复是由系统在重新启动时自动完成的，不需要用户干预。
- 系统的恢复步骤（4步）
- 1) 正向扫描日志文件（即从头扫描日志文件），找出故障发生前已经提交的事务（这些事务既有 begin transaction 记录，也有 commit 记录），将其事务标识放入重做（REDO）队列。
 - 2) 同时找出故障发生时尚未完成的事务（这些事务只有 begin transaction 记录，无相应的 commit 记录），将其事务标识放入撤销（UNDO）队列。
 - 3) 对撤销队列中的各个事务进行撤销（UNDO）处理。
注：进行 UNDO 处理的方法是：反向扫描日志文件，对每个 UNDO 事务的更新操作执行逆操作，即将日志记录中“更新前的值”写入数据库。→ 因为这个需要“恢复”，所以用反向扫描
 - 4) 对重做队列中的各个事务进行重做（REDO）处理。
注：进行 REDO 处理的方法是：正向扫描日志文件，对每个 REDO 事务更新执行日志文件登记的操作。即得日志记录中“更新后的值”写入数据
- 介质故障的恢复
- 发生介质故障后，磁盘上的物理数据和日志文件被破坏，这是最严重的一种故障，恢复方法是重建数据库，然后重做已完成的事务。
- 具体步骤（2步）：
1) 装入最新的数据库后备副本（即故障发生时刻最近的备份副本），使数据库恢复到最近一次备份时的一致性状态。最近状态与日志文件有关。
2) 装入相应的日志文件副本。（备份结束时刻的日志文件副本），重做已完成的事务。即：
- 1 首先扫描日志文件，找出故障发生时已提交的事务的标识，将其放入重做（REDO）队列。
 - 2 然后正向扫描日志文件，对重做队列中的所有事务进行重做处理。即将日志记录中“更新后的值”写入数据库。
 - 3 将数据库恢复至故障前某一时刻的一致性状态。
- 介质故障 > 系统恢复； 介质故障 → DBA 介入（副本、日志文件的恢复）

Date

利用日志技术进行数据库恢复时，恢复子系统必须搜索常日志，确定哪些事务需要重做(REDO)，哪些事务需要UNDO。

一般来说，我们需要检查所有日志记录。这样做具有2个问题：

1) 搜索整个日志将耗费大量的时间

2) 很多需要REDO的事务实际上已经将它们的更新操作结果写到数据库中了，然而恢复子系统又重新执行了这些操作，浪费了大量时间。

为了解决这些问题，又发展了具有检查点的恢复技术。

检查点技术(check point)就是在日志文件中增加一类新的记录—检查点记录(check point)，增加一个重新开始文件，并让恢复子系统在登陆日志文件期间动态地维护日志。
↓
用于记录检查点的。

• 检查点记录的内容包括：

1) 建立检查点时刻所有正在执行的事务清单

2) 这些事务最近一个日志记录的地址。

3) 重新开始文件用来记录各个检查点记录在日志文件中的地址。

动态维护日志文件的方法是：周期性地建立检查点，保存数据库状态。具体步骤是：
(1小时, 2小时)

1) 将当前日志缓冲中的所有日志记录写入磁盘的日志文件上。

2) 在日志文件中写入一个检查点记录。

3) 将当前数据缓冲中的所有数据记录写入磁盘的数据库中。

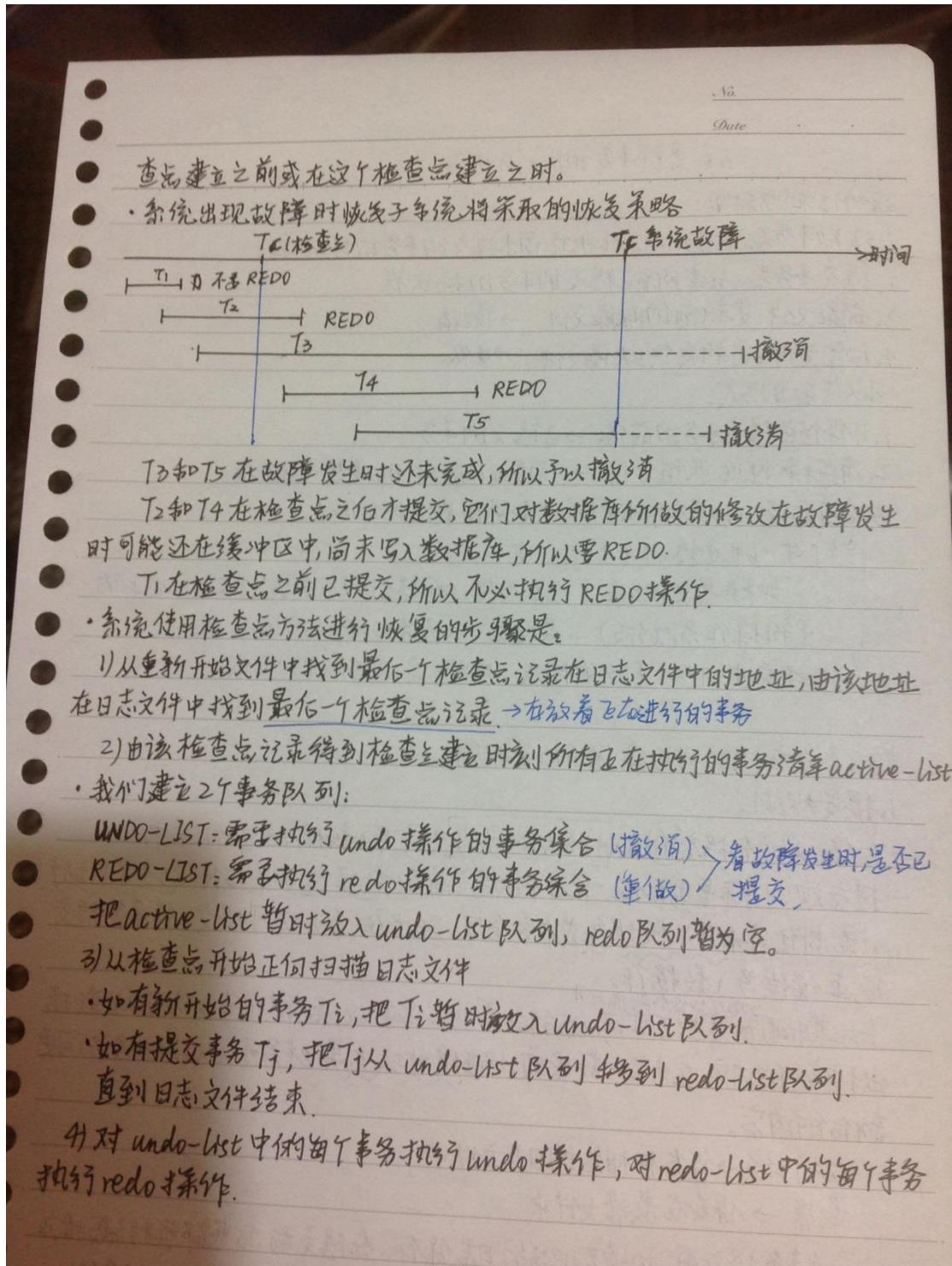
4) 把检查点记录在日志文件中的地址写入一个重新开始文件。

恢复时，只要做检查点之后的数据就可以了，因为之前的数据已提交。

恢复子系统可以定期或不定期地建立检查点并保存数据库状态。

检查点可以按照预定的一个时间间隔建立，如每隔一小时建立一个检查点；也可以按照某种规则建立检查点，如日志文件已写满一半建立一个检查点。

使用检查点方法可以改善恢复效率。当事务T在一个检查点之前提交，T对数据库所做的修改一定都已写入数据库，写入时间是在这个检



7.6 更新事务的执行与恢复

运行记录的基本：

1. 活动事务表：记录所有正在执行、尚未提交的事务的标识符。

2. 提交事务表：记录所有已提交的事务的标识符。

3. 前像文件：更新前的映像文件 → 撤销

4. 后像文件：更新后的映像文件 → 重做

可以采取的方式：

1. 不保留已提交事务的前像 → 已提交的事务

2. 有选择性地保留后像 → 目的是在 REDO 时，用后像替代已有数据。

3. 合并后像（如果有多次更新，只要保留最近的后像）

↓ 针对 上午更新 下午更新 晚上更新

如果数据出现差异，需恢复数据，对已提交的事务重做（上午、下午、晚上的

更新操作都重做）→ 最佳是只保留晚上更新的数据

↓ 针对需进行多次更新的情况，此时只需保留最近的后像

更新事务在执行时应遵循的 3 条规则：

1. 提交规则。

- 后像必须在提交前写入非易失存储器中，即写入数据库或运行记录中。

- 提交规则并不要求后像一定在事务提交前写入数据库，如果后像已经写入运行记录，即使还未写入数据库或未完全写入数据库，事务仍可提交。待事务提交后，再继续写入数据库。

- 在此期间，^{提交和保存数据之间}如发生故障，可以运行记录的后像重做；若有其他事务访问这些数据，由于更新的内容在未写入数据前，仍在缓冲块中，其他事务可以从缓冲块访问更新后的内容。

运行记录 → 保存在相应的日志文件中

数据 → 保存在数据文件中

一个事务提交前，必须先把运行日志保存，在提交前，后像都可以被数据库的保存。如果在此期间发生故障，可根据运行记录的后项对事务进行重做。

2. 先远远后写规则
- 如果后像在事务提交前写入数据库，则必须把前像首先写入运行记录。
 - 事务提交前，都有可能失败。事务失败后，必须撤消事务对数据库所作的一切更新。
 - 必须在改动数据库前，先将前像注入运行记录。
- 在更新事务时，按照后像写入数据库的时间的不同，又有3种方案。
→具体什么时候保存后备信息，对恢复有很大影响
1. 后像在事务提交前完全写入数据库
- (1) TID → ATL 事务的TID号写入活动事务表(事务正在运行还未提交)
 - (2) BI → LOG 事务的前像写到相应的日志中
 - (3) AI → DB 后像写入数据库中
 - (4) TID → CTL 事务TID号写入已提交的事务表中(标志着事务已经结束)
 - (5) 从ALT删除TID → 事务之后才表示事务完全提交
- 恢复措施：没有提交的事务，其恢复策略都是撤销。如果BI没写入log，则无需操作
ALT CTL / 事务所处状态 恢复措施 ② 从ALT删除TID
- | | | | |
|---|---|-------------------|-------------------------------|
| 有 | - | (1) 已完成, 但(4)尚未完成 | ① 若BI已写入log, 则undo; 否则无需 undo |
| 有 | 有 | (4) 执行完 | 从ALT中删除TID |
| - | 有 | (5) 执行完 | 无需处理。 |
2. 后像在事务提交后才写入数据库
- (1) TID → ATL
 - (2) BI → LOG
 - (3) TID → CTL 先提交，之后才把后像写到数据库中
 - (4) AI → DB
 - (5) 从ALT删除TID
- 恢复措施：提交, 没有数据 → redo
没提交 → undo
- 恢复措施：事务没提交, 只做了半
ALT CTL / 事务所处状态 恢复措施 成为做操作
- | | | | |
|---|---|--------------------------------|--------------------|
| 有 | - | (1) 已完成, (3)未完成 | 从ALT删除TID |
| 有 | 有 | (3)已完成, (5)未完成 → 已经提交, 但后项不需写入 | ① redo ② 从ALT删除TID |
| - | 有 | (5)执行完 | 无需处理 |

3. 之后在事务提交前和提交后分别写入数据库中

(1) TID → ATL

(2) A], B] → LOG

(3) A] → DB (部分写入) → 不保存所有数据

(4) TID → CTL

(5) A] → DB (继续完成)

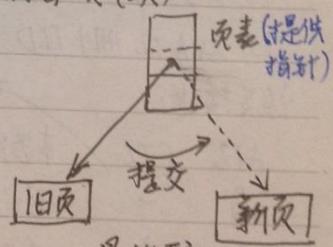
从 ATL 中除 TID

恢复措施:

ATL	CTL	事务何处状态	恢复措施
有	一	(1) 已完成, (4) 未完成 → 部分信息已写入 DB, 但可能有部分没写入 DB.	① 若 B] 已写入 log, 则 undo; 否则无需 undo ② 从 ATL 中除 TID
有	有	(4) 执行完, 但 (6) 未完成	① redo; ② 从 ATL 中除 TID
一	有	(6) 执行完	无需处理

三种方案的比较:

方案	redo	undo	A]	B]	执行的并发度	使用情况
1 先写再插入	✓	—	✓	差	用得较少	
2 先插入再写入	—	✓	—	中	用得较多	
3	✓	✓	✓	好	用得较多	数据的位置不变 ↑ 内容改变



7.7 并发控制概述

1. 引言

1. 引言
数据库是一种共享资源，可供多个用户使用。允许多个用户同时使用的数据库系统称为多用户数据库系统。

系统称为多用户数据库系统。
当多个用户并发地存取数据库时就可能会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会存取和存储不正确的数据，破坏数

（三）数据在竣工验收阶段提供，并发控制机制。

并发控制机制是衡量一个数据库管理系统性能的重要标志之一。

2. 数据库系统中的并发

如果数据库中的事务必须顺序执行，即一个事务完全结束之后，另一个事务才开始，则称这种执行方式为串行访问。(serial access)

如果DBMS可以同时接受多个事务，事务在时间上重叠执行，则称这种执行方式为并发访问。(concurrent access).

在单CPU系统中，同一时间只能有一个事务占用CPU，即各个事务需要交叉使用CPU，称这种执行方式为交叉并发。(interleaved concurrency)

在多CPU系统中，可以允许多个任务同时占用CPU，称这种执行方式为同时并发。*(simultaneous concurrency)* → 目前很多服务器提供多CPU
并发不是并行。并发可能是交替的，也可能是并行的。

并发不是并行。并发可能是交替的，也可能是并行的。

3. 事务并发执行可能引起的问题

例：飞机订票系统

1) 甲信零空(甲未备)读出某航班的机票余额A,设 A=16

2)乙 --- 2 - - - 同 - - - - A 也为 16

3) 甲卖出一张票,修改金额 $A \leftarrow A - 1$, : A 为 15, 把 A 写回数据库

4)已卖出—— — — — — — — — —

间隔：填写朋友寄出几张机票，航班集中购票会更优惠。

向公局不明确突出。纵机事，数据库中机票余额却只减少1张。

并发操作带来的数据不一致性包括3类：

1)丢失修改 (lost update) → 因为2个事务都在写 ⇒ 写和写之间有冲突

两个事务T₁和T₂读入同一数据并修改，T₂提交的结果破坏了T₁提交的结果，导致T₁的修改丢失。飞机订票例子就属此类。2个事务都读 ⇒ 无影响

2)不可重复读 (non-repeatable read) → 2个事务一个在读一个在写 ⇒ 读和写之间有冲突
指事务T₁读取数据后，T₂执行更新操作，使T₁无法再现前一次读取结果。具体的讲，不可重复读包括3种情况：

(i) 事务T₁读取某一数据后，事务T₂对其实做了修改，当T₁再次读取该数据时，

得到与前一次不同的值。例如T₁读取B=100，并，T₂读取B，修改后把B=200写回数据库。T₁为了对该值进行重读，B已为200，与第一次读取值不一致。

(ii) T₁按一定条件从数据库中读取了某些数据记录后，T₂删除了某部分记录，当T₁再次按相同条件读取数据时，发现某些记录神秘地消失了。

(iii) T₁按一定条件从数据库中读取某些数据记录后，T₂插入了一些记录，当T₁再次按相同条件读取数据时，发现多了一些记录。

3)读“脏”数据 (dirty read) 但T₂已经读取了这个无效数据 而T₁的修改是无效的

指T₁修改某一数据，并将其写回磁盘，T₂读取同一数据后，由于某种原因被撤销，这时T₁已修改过的数据恢复原值，T₂读到的数据就与数据库中的数据不一致，则T₂读到的数据就为“脏”数据，即不正确的数据。

↓原因：读、写冲突。

4. 并发控制的目标

产生上述3类数据不一致性的主要原因是并发操作破坏了事务的隔离性。

并发控制就是要用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性。

此外，并发操作还能提高系统资源利用率改善短事务的响应时间。

并发控制的正确性准则

1. 基本定义

计算机系统对并发事务中并发操作的调度是随机的。而不同的调度可能会产生不同的结果。

定义1:

设数据库系统中参与并发执行的事务集为 $\{T_1, T_2, \dots, T_n\}$, 一个调度(schedule)

S 就是对 n 个事务中的所有操作执行次序的一个安排。

在同一个调度中, 不同事务的操作可以交叉执行, 但必须保持同一个事务中各个操作的次序。

定义2:

设数据库系统中参与并发执行的事务集为 $\{T_1, T_2, \dots, T_n\}$, 其中两个调度 S_1 和 S_2 , 在数据库的任何初始状态下, 所有读出的数据都是一样的, 留给数据库的最终状态也是一样的, 则称 S_1 和 S_2 是目标等价的。(view equivalence)

定义3:

设数据库系统中参与并发执行的事务集为 $\{T_1, T_2, \dots, T_n\}$, 如果其中的两个操作 OP_1 和 OP_2 满足下列条件:

(1) 它们属于不同的事务

(2) 其中至少有一个写(write)操作

则称 OP_1 和 OP_2 是冲突操作

冲突操作包括读-写冲突和写-写冲突两种。

存在“写”会导致冲突

定义4:

设数据库系统中参与并发执行的事务集为 $\{T_1, T_2, \dots, T_n\}$, S 为一个调度, 我们将那些通过转换 S 中不冲突的操作所得到的新调度称为 S 的冲突等价调度。(conflict equivalence)

显然两个调度是冲突等价, 一定是目标等价的; 反之,未必正确。
如果

多个事务的并发执行是正确的，当且仅当其结果与按某一次序平行地执行它们时的结果相同。称这种调度策略为可串行化(Serializable)的调度。

定义5：(目标)

如果 S 能与一个串行调度冲突等价，则称 S 是冲突(目标)可串行的。

可串行性(Serializability)是并发事务正确性的准则。

按这个准则规定，一个给定的并发调度，当且仅当它是可串行化的，才认为是正确的调度。

一个调度 S 是否可串行，可用其前趋图(precedence graph)来判定。

2. 可串行性的判别方法

前趋图是一个有向图 $G = (V, E)$ ，其中 V 是顶点的集合， E 是边的集合。 V 包含所有参与调度的事务。边可以通过分析事务之间的冲突操作获得。

规则如下：

如果两个事务 T_i 和 T_j 之间有一对冲突操作 OP_1 和 OP_2 ，在 S 中 OP_1 被安排在 OP_2 之前，则在 E 中加入边： $T_i \rightarrow T_j$

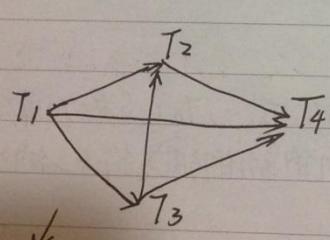
· 可以证明：不能按照有向图的方向写出串行调度。

(1) 如果前趋图中存在回路，则 S 不可能冲突等价于任何串行调度；

(2) 如果前趋图中不存在回路，则可以用拓扑排序得到 S 的一个等价的串行调度。

例1：设有事务集 $\{T_1, T_2, T_3, T_4\}$ 的一个调度。 x, y, z 是数据

$$S = W_3(y) R_1(x) R_2(y) W_3(x) W_2(z) W_3(z) R_4(z) W_4(x)$$



去掉 T_1 : $T_2 \rightarrow T_4$ 队列: T_1

去掉 T_3 : $T_2 \rightarrow T_4$ 队列: T_1, T_3

去掉 T_4 : 空。队列: T_1, T_3, T_2, T_4

这个图没有回路，可串行化。

去掉 T_2 : T_4

队列: T_1, T_3, T_2

例12：现有2个事务，分别包含下列操作：

事务T₁: R₁(B); A=B+1; W₁(A)

事务T₂: R₂(A); B=A+1; R₂(B)

假设A,B的初值均为2。

按T₂, T₁顺序执行结果为A=3, B=4

- - T₂, T₁ - - - - B=3, A=4.

下面给出对这2个事务的4种不同的调度策略：

(a) R₁(B); A=B+1; W₁(A); R₂(A); B=A+1; R₂(B)

(b) R₂(A); B=A+1; R₂(B); R₁(B); A=B+1; W₁(A)

(c) R₁(B); A=B+1; R₂(A); W₁(A); B=A+1; R₂(B)

(d) R₂(A); B=A+1; R₁(B); R₂(B); A=B+1; W₁(A)

(a), (b)为2种不同的串行调度策略，虽然执行结果不同，但它们都是正确的调度。

(c)中两个事务是交错执行的，由于其执行结果与(a), (b)的结果都不同，所以是错误的调度。

(d)中两事务也是交错执行的，其执行结果与串行调度(b)的执行结果相同，所以是正确的调度。

为了保证并发操作的正确性，DBMS的并发性控制机制必须提供一定的手段来保证调度是可串行化的。

目前DBMS普遍采用加锁方法实现并发操作调度的可串行性，从而保证调度的正确性。

两阶段锁(Two-Phase Locking, 简称2PL)协议就是保证并发调度可串行性的加锁协议。

除此之外还有其他一些方法，如时标方法、乐观方法等来保证调度的正确性。

7.8 加锁协议

1. 概述

加锁是实现并发控制的一个非常重要的技术。

所谓加锁就是事务T在对某个数据对象如表、记录等操作之前，先向系统发出请求，对其加锁。加锁后事务T就对该数据对象有了一定的控制，在事务T释放它的锁之前，其他事务不能更新此数据对象。

· 基本加锁类型有3种：

一排他锁 (Exclusive locks, 简记为X锁)

读操作可共享

一共享锁 (Share locks, 简记为S锁)

写操作不可共享

一U锁

2. 排他锁 (X锁)

又称为写锁。

若事务T对数据对象A加上X锁，则只允许T读取和修改A，其他任何事务不能再对A加任何类型的锁，直到T释放A上的锁。这就保证了其它事务在T未释放A上的锁之前不能再读取和修改A。

其他事务已拥有的锁

锁请求		其他事务已拥有的锁		
		NL	X	Y(允许) N(不允许)
X	NL	→没加任何锁	→已经加上了锁	
	Y	(允许)	(不允许)	

X锁兼容矩阵

3. 共享锁 (S锁)

共享锁 又称为读锁。若事务T对数据对象A加上S锁，则事务T对数据对象A加S锁，刚事务T可以读A但不能修改A，其他事务只能对A加S锁，而不能加X锁，直到T释放A上的S锁。这就保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

其他事务已拥有的锁

锁请求		NL	S	X
S	NL	Y	Y	N
	S	Y	N	N

(S, X) 锁兼容矩阵

4. U锁 → 针对数据的更新的。

事务在进行更新时，一般需先读出旧的内容，在内存中修改后，再将修改后的内容写回。在此过程中，除了最后写回阶段，被更新的数据对象仍然可能被其他事务访问。

事务在需要（或有可能）更新一个数据对象时，首先申请对它的U锁，数据对象加了U锁后，仍允许其他事务对其加S锁。

待最后需要写入时，事务再申请将U锁升级为X锁。由于不必在事务执行的全过程中加X锁，所以进一步提高了系统的并发度。

其他事务已拥有的锁

	N	L	S	U	X
锁请求 S	Y	Y	Y	N	
U	Y	Y	Y	N	N
X	Y	N	N	N	N

(S, U, X) 锁兼容矩阵

5. 加锁协议

在运用X锁、S锁和U锁这3种基本加锁，对数据对象加锁时，还需要约定一些规则，例如应何时申请X锁或S锁、持锁时间、何时释放等。

我们称这些规则为加锁协议（Locking Protocol）。对加锁方式规定不同的规则，就形成了各种不同的加锁协议。

三级加锁协议

1) 一级加锁协议

对“读”操作不加锁，只对“写”操作加锁

事务T在修改数据R之前必须先对其加X锁，直到事务结束（EOT）才释放。事务结束包括正常结束（commit）和非正常结束（rollback）。

一级加锁协议可防止丢失修改，并保证事务T是可恢复的。

在一级加锁协议中，如果仅仅是读数据而不对其进行修改，则不需要加锁，所以它不能保证可重复读及“脏”数据。

读值不可重现 (不能重读)

	T ₁	T ₂	
时间	↓	:	
	lock (D)	:	T ₂ 对数据做了更改, 但T ₁ 不知道, 因仍然
	R ₁ (D)	:	读出数据, 此时情况为: 读“脏”数据
	unlock (D)	:	
	↓	lock (D)	
	lock (D)	W ₂ (D)	
	lock (D)	unlock (D)	
	R ₁ (D)	:	
	unlock (D)	:	
	↓		

2) 2级加锁协议

1级加锁协议加上事务T在读取数据A之前必须先对其加S锁, 读完后可释放S锁。

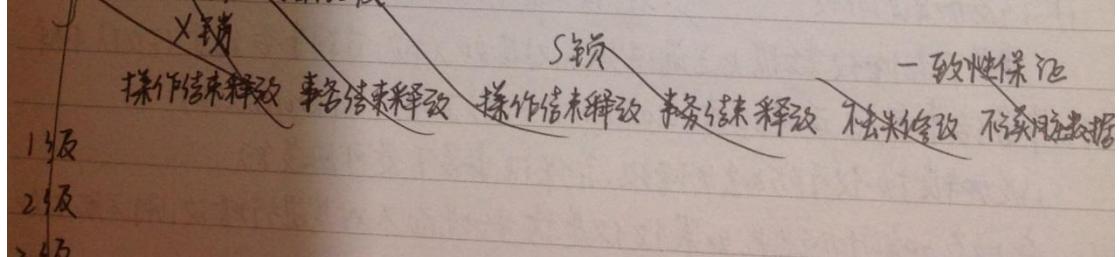
2级加锁协议除了可以防止丢失修改, 还可进一步防止读“脏”数据。

3) 3级加锁协议 → 并发度不是很好,

1级加锁协议加上事务T在读取数据A之前必须先对其加S锁, 直到事务结束才释放。
2次读之间不能写。

3级加锁协议除防止了丢失修改和不读“脏”数据外, 还进一步防止了不可重复读。有时可以放松加锁协议的条件。(采取宽松策略)

3级加锁协议的比较



4) 3级及加锁协议的比较

	X 锁		Y 锁		一致性保证
	操作得 事名得 来释放	操作得 事名得 来释放	不丢失 修改 数据	不读“脏” 数据	可重 复读
1级	✓		✓		
2级	✓	✓	✓	✓	
3级	✓		✓	✓	✓

6. 两段锁协议 (2PL)

两段锁协议 (two-phase locking protocol, 简称 2PL 协议):

指所有事务必须分两个阶段对数据项加锁和解锁:

1) 在对任何数据进行读、写操作之前, 首先要申请并获得对该数据的封锁, 而且

2) 在释放一个封锁之后, 事务不再申请和获得任何其他封锁。

所谓“两段”锁的含义是, 事务分为两个阶段, 第一阶段是获得封锁, 也称为扩展阶段。在这阶段, 事务可以申请获得任何数据项上的任何类型的锁, 但不能释放任何锁。^{因为锁会越来越多}

第二阶段是释放封锁, 也称为收缩阶段。在这阶段, 事务可以释放任何数据项上的任何类型的锁, 但是不能再申请任何锁。

例如事务 T1 遵守两段锁协议, 其封锁序列是: (A, B, C) 的 lock, unlock

↓
lock A ... lock B ... lock C ... unlock B ... unlock A ... unlock C
^{顺序可以不同}

事务 T2 不遵守两段锁协议, 其封锁序列是:

lock A ... unlock A ... lock B ... unlock B ... lock C ... unlock C ... unlock B;

两段锁协议保证事务的串行性。

合式事务 (well-formed)

一个事务如果遵守先加锁后操作的原则, 则此事务称为合式事务。

如果所有事务都是合式、两段事务, 则它们的任何调度都是可串行化的。

注意：2PL 协议是调度可串行化的充分条件，但不是调度可串行化的必要条件。如：

$$S = R_2(x) W_3(x) R_1(y) W_2(y)$$

T_2 执行完 $R_2(x)$ 后，释放 x 上的锁而让 T_3 加锁；

T_1 执行完 $R_1(y)$ 后，释放 y 上的锁而让 T_2 加锁；

由于 T_2 释放之后又加锁，因而不是两段事务。但 S 是可串行化的，即 $T_1 \rightarrow T_2 \rightarrow T_3$

7. 两段锁协议与一次封锁法的异同 → 不是非可行的。^{事务操作前知道事务需要的数据}

一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行，因此一次封锁法遵守两段锁协议。^{条件要求更高，并发性非常低}

两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁。^{加锁时间是阶段性的，每次对一个数据加锁}

先加锁后解锁，但先对哪个加锁或先对哪个解锁顺序是不固定的

8. 活锁

如果事务 T_1 封锁了数据 R ，事务 T_2 又请求封锁 R ，于是 T_2 等待。 T_3 也请求封锁 R ，当 T_1 释放了 R 上的封锁之后系统首先批准了 T_3 的请求， T_2 仍然等待。然后 T_4 又请求封锁 R ，当 T_3 释放了 R 上的封锁之后系统又批准了 T_4 的请求，…， T_2 有可能永远等待，这就是活锁。^{还是有可能获得锁的。}

避免活锁的简单方法是采用“先来先服务”的策略。

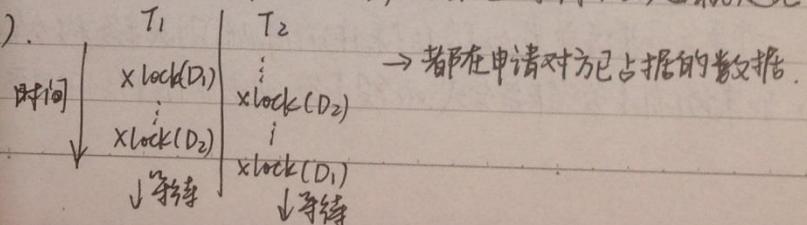
7.9 死锁的检测、处理和预防

1. 什么是死锁？

和操作系统一样，基于封锁的并发控制方法可能引起死锁

一个事务如果申请锁而没有获准，则必须等待其他事务释放锁。这就形成事务间的等待关系。

当事务中出现循环等待时，如果不加以干涉，则会一直等待下去，这就是死锁。(dead lock).



对于死锁有2种方法：

- ▷ 防止死锁的出现（悲观方法） → 一般是一般是2种方法结合起来使用
- ▷ 由系统检测死锁，一旦发现则对其进行处理（乐观方法）

2. 死锁的预防

在数据库中产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求已为其他事务封锁的数据对象加锁，从而出现死等待。

防止死锁的发生其实就是要破坏产生死锁的条件。

1) 一次封锁法

要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行。

一次就将以后要用到的全部数据加锁，势必扩大锁的范围，从而降低了系统的并发度。

2) 顺序封锁法 → 采用不是很多

预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁。这种方法可以有效地防止死锁，但也同样存在问题。

事务的封锁请求往往随着事务的执行而动态地决定，很难事先确定每一个事务要封锁哪些对象，因此也就很难按规定的顺序对数据对象加锁。

3) 一种比较实用的死锁预防方法

当事务申请锁而未获准时，不是一律等待，而是让某些事务卷回重试（retry），以避免死锁。

为区别事务开始执行的先后，每个事务开始时被赋予一个唯一的并随时间递增的整数，称为时间标记（time stamp, ts）

设有2个事务TA和TB，如 $ts(TA) < ts(TB)$ ，则表示TA早于TB，也称TA比TB“年老”，或者说TB比TA“年轻”。

· 事务重试的2种策略：

1) 等待-死亡（wait-die）策略

在这种策略中，设TB已持有某数据对象R的锁，TA申请同一数据对象的锁而发生冲突时，则按如下规则处理：

loose leaf: 7.8mm-27lines * gimp

Date

```
if ts(TA) < ts(TB) then TA waits  
else {
```

```
    rollback TA; /* Die */  
    restart TA with the same ts(TA);
```

}

在这种策略中，总是年老的事务等待年轻的事物。

2) 击伤-等待 (wound-wait) 策略

这种策略按另一种规则处理冲突：

```
if ts(TA) > ts(TB) then
```

```
    TA waits;
```

```
else {
```

```
    rollback TB; /* Die */
```

```
    restart TB with the same ts(TB);
```

}

在这种策略中，总是年轻的事物等待年老的事物。

2. 死锁的检测与解除

1) 超时法 → 不合乎情理，因为各个事务执行时间差异非常大

如果一个事务的等待时间超过了规定的时限，就认为发生了死锁。

超时法实现简单，但不足也很明显。

一是有可能误判死锁，事务因为其他原因使等待时间超过时限，系统会误认为发生了死锁。

二是时限若设置得太长，死锁发生后不能及时发现。

2) 等待图法 → 代价非常大！事务多 → 有向图大 → 难判断回路 → 耗时长

事务等待图是一个有向图 $G = (T, U)$ 。 T 为结点的集合，每个结点表示正在运行的事务， U 为边的集合，每条边表示事务等待的情况。若 T_1 等待 T_2 ，则 T_1, T_2 之间画一条有向边，从 T_1 指向 T_2 。

事务等待图动态地反映了所有事务的等待情况。并发控制子系统周期性地（比如每隔1分钟）检测事务等待图，如果发现图中存在回路，则表

示系统中出现了死锁。

3) 死锁的解除

死锁发生后，靠事务自身无法解决解除，必须由DBMS干预。

通常采用的方法是选择一个处理死锁代价最小的事务，将其撤锁、释放此事务持有的所有的锁，使其他事务得以继续运行下去。

当然，对撤销的事务所执行的数据修改操作必须加以恢复。

7.10 多粒度封锁

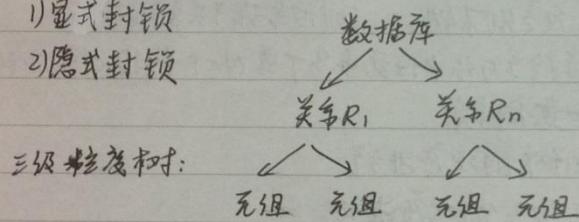
1. 多粒度锁

多粒度封锁协议允许多粒度树中的每个结点被独立地加锁。

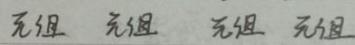
对一个结点加锁意味着这个结点的所有低层结点也被加以同样类型的锁。因此，在多粒度封锁中一个数据对象可能以2种方式封锁：

1) 显式封锁

2) 隐式封锁



三级粒度树：



显式封锁：事务在执行过程中涉及到某些对象、数据，就对这些数据进行加锁。

隐式封锁：在对某一个关系加锁时，那么意味着对所有高加上同样的锁。

(隐含的关系，不是明确指定的，但是还是存在的)

2. 意向锁

含义是如果对一个结点加意向锁，则说明该结点的下层结点将要被加锁；对任一结点加锁时，必须先对它的上层结点加意向锁。

例：对任一元组加锁时，必须先对它所在的关系加意向锁。

事务T要对关系R₁加X锁时，系统只要检查根结点数据库和关系R₁是否已加了不兼容的锁，而不再需要搜索和检查R₁中的每一个元组是否加了X锁。

体现了层次关系。

三种常用的意向锁：

1) 意向共享锁 (intent share lock) - IS 锁

表示它的后裔结点拟（意向）加 S 锁。例如，要对某个元组加 S 锁，则要对关系和数据库加 IS 锁。

2) 意向排他锁 (intent exclusive lock) - IX 锁

表示它的后裔结点拟（意向）加 X 锁。例如，要对某个元组加 X 锁，则要先对关系和数据库加 IX 锁。

3) 共享意向排他锁 (share intent exclusive lock) - SIX 锁

如果对一个数据对象加 SIX 锁，表示对它加 S 锁，再加 IX 锁，即 $SIX = S + IX$ 。例如对某个表加 SIX 锁，则表示该事务要读整个表（所以要对该表加 S 锁），同时会更新个别元组。（所以要对该表加 IX 锁）

所谓锁的强度是指它对其他锁的排斥程度。一个事务在申请封锁时以强锁代替弱锁是安全的，反之则不然。（排他享强于共享锁）

具有意向锁的多粒度封锁方法中任意事务只要对一个数据对象加锁，必须首先对它的上层结点加意向锁。

申请封锁时应按自上而下的次序进行

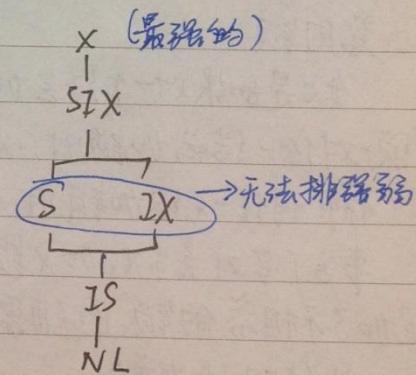
释放封锁时应按自下而上的次序进行。

具有意向锁的多粒度封锁方法提高了系统的并发度，减少了加锁和解锁的开销，它已经在实际的数据库管理系统产品中得到广泛的应用，DBMS 中就采用了这种封锁方法。

其他事务已拥有锁

	NL	IS	IX	S	SIX	X
锁请求	IS	Y	Y	Y	Y	Y
IX		Y	Y	Y	N	N
S		Y	Y	N	Y	N
SIX		Y	Y	N	N	N
X		Y	N	N	N	N

意向锁的相容矩阵



表示锁的强度的偏序关系

7.11 事务的并发控制

· 事务并发控制

✓ 数据库中的数据

✓ 数据的索引

索引的加锁方式与数据类似。(对索引加上“锁”来进行控制)

7.12 幻影及其防止

幻影(phantom): 细步读封锁的缺陷所导致的一种现象

例: 关系 enrollment(选课) 属性: name, SNO, course, 而且 course 上有

B+树索引。因为有索引, 所以 T₁ 不会将整个关系封锁, 而是用 database 作为索引关键

事务 T₁: read1: select name from enrollment

where course = 'database';

read2: 先于 read1

事务 T₂: insert: insert into enrollment values ('张三', '01141039', 'database')

注意: 如果 T₁ 只是把所有现有的 course = 'database' 的元组加锁, 那么 T₂ 就

可以在 read1 之后插入元组, 然后 read2 再执行时就会发现 (张三, 01141039) 像幻影一样出现了。

7.13 事务的隔离等级

隔离等级	可能出现的后果			加锁需求
	读脏数据	读值不可见	幽灵	
Read uncommitted	✓	✓	✓	读不加锁, 写加 X 锁 保持到 EOT
Read committed	-	✓	✓	读加 S 锁, 然后释放 写加 X 锁, 保持到 EOT
Repeatable Read	-	-	✓	严格级的 2PL 协议
Serializable	-	-	-	严格的 2PL 协议, 叶节点上加锁 保持到 EOT

loose leaf: 7.8mm x 9.7inches " ginen

7.14 基于时间标记的并发控制技术

$ts(T)$: 事务的时间标记, 每个事务在被启动时赋予当前的时间标记。

tr : 读时间标记, 在读过该数据的所有事务的时间标记中, 取最大的作为 tr 。

tw : 写时间标记, 在写过该数据的所有事务的时间标记中, 取其中最大的作为 tw 。

tr, tw 会随着数据的运行, 不断地变化。

1. 事务T读数据D

`read(D);`

`if $ts(T) \geq tr$`

`then /* 符合时间标记协议, read(D)即为读取值 */`

`$tr = \max(ts(T), tr);$`

`else /* 已有比T年轻的事务写入D, 违反时间标记协议, T rollback */`

`rollback T and reset it with a new $ts(T)$;`

2. 事务T写数据D

`if $ts(T) \geq tr \text{ AND } ts(T) \geq tw$`

`then /* 符合时间标记协议 */`

`{ write(D); }`

`$tw = ts(T)$ }`

`else /* 违反时间标记协议 */`

`roll back T and restart it with a new $ts(T)$;`

7.15 乐观并发控制技术

并发控制方法

- 悲观法 (pessimistic method)

· 封锁法 \downarrow 会先判断一定的条件 (在执行之间)

· 时间标记法

- 乐观法 (optimistic method) \rightarrow 先进行操作, 遇到问题 \downarrow rollback 或进行其他控制

