第8章 事务管理

- 8.1 事务的概念
- 8.2 数据库故障
- 8.3 数据库恢复技术
- 8.4 数据库的并发控制
- 8.5 基于锁的并发控制
- 8.6 其他并发控制技术*

8.1事务的概念

- 事务的定义
- 事务的性质
- 事务的操作
- 事务的状态
- SQL的事务管理

事务的定义

- 事务(Transaction)是数据库环境下由一组数据库操作序列组成的逻辑工作单元(Unit of Work/UOW)。
- 这些操作应是一个不可分割的整体，要么全做要么全不做(all or nothing)。
- 事务的执行应保证将数据库从一个一致状态转变成另一个一致状态，而在此过程中不须保证数据不一致。
- 如果发生事务中某些操作执行，而另一些操作未执行的情况，则会使数据库处于不一致的状态。

事务的定义

- 事务概念的引入，源于数据库的某些操作必须完整的执行，才能保证正确性。
- 例：银行转帐事务，这个事务把一笔金额从一个帐户A转给另一个帐户B：

```
BEGIN TRANSACTION
 读帐户A的余额BALANCEA;
 BALANCEA = BALANCEA—AMOUNT；/AMOUNT 为转帐金额)
  写回BALANCEA
 IF(BALANCEA<0 ) THEN  {
        打印'金额不足，不能转帐';
        ROLLBACK；(撤消刚才的修改，恢复事务)}
 ELSE{
        读帐户B的余额BALANCEB;
        BALANCEB = BALANCEB+AMOUNT;
       写回BALANCEB;
       COMMIT；}
  }
```

事务的定义

- 上面这个例子所包括的两个更新操作要么全部完成要么全部不做，否则就会使数据库处于不一致状态，例如只把帐户甲的余额减少了而没有把帐户乙的余额增加。

事务的定义

- 事务通常是以BEGIN TRANSACTION开始，以COMMIT或ROLLBACK结束。
- COMMIT表示提交，即提交事务的所有操作。具体地说就是将事务中所有对数据库的更新写回到磁盘上的物理数据库中去，事务正常结束。
- ROLLBACK表示回滚，即在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤消(undo)，回滚到事务开始时的状态。
- 这里的操作指对数据库的更新。

事务的宏观性和微观性

事务的宏观性(应用程序员看到的事务): 一个存取或改变数据库内容的程序的一次执行，或者说一条或多条SQL语句的一次执行被看作一个事务。

- 事务一般是由应用程序员提出，因此有开始和结束, 结束前需要提交或撤消。

```
Begin  Transaction
    exec sql …
    …
    exec sql …
    exec sql  commit work  |  exec sql rollback work
End Transaction
```

- 在嵌入式SQL程序中，任何一条数据库操纵语句(如exec sql select等)都会引发一个新事务的开始，只要该程序当前没有正在处理的事务。
- 而事务的结束是需要应用程序员通过commit或rollback确认。

事务的宏观性和微观性

- 一个事务可处理一个数据或一条记录，如下例：

  “Update T1 Set val = :pgmval2 where uniqueid = B;”

- 复杂一些的事务也可能处理一批数据或一批记录，如下例：

  “Update T1 Set val = 1.15 * val where uniqueid between :low and :high;”

- 一段程序语句，可能会循环执行。执行中，由SQL语句引出事务，至Commit/RollBack结束事务，每次重复执行都将产生一个事务。

事务开始

事务结束

```
while((prompt(cid_prompt,1,cust_id,4)) >=0) {
    exec sql select cname,discnt
             into :cust_name, :cust_discnt
             from customers where cid=:cust_id;
    exec sql commit work;
    printf("Customer's name is %s and discount
             is %5.1f\n",cust_name,cust_discnt);
    continue;
notfound: printf("Can't find customer %s,
             continuing\n",cust_id); }
exec sql commit release;
```

循环多少次，则将产生多少个事务

- 更为复杂的事务由多条SQL语句构成。

事务的宏观性和微观性

事务的微观性(DBMS看到的事务): 对数据库的一系列基本操作(读、写)的一个整体性执行。

T：
```
read(A);
A := A – 5000;
write(A);
read(B);
B := B + 5000;
write(B);
```

事务的并发执行：多个事务从宏观上看是并行执行的，但其微观上的基本操作(读、写)则可以是交叉执行的。

事务的性质

- 事务具有四个特性：
  - ✓ 原子性(Atomicity)
  - ✓ 一致性(Consistency)
  - ✓ 隔离性(Isolation)和
  - ✓ 持续性(Durability)
- 这个四个特性简称为事务的ACID特性。

事务的性质

1. 原子性

事务是数据库的逻辑工作单位，事务中包括的诸操作要么都做，要么都不做。

2. 一致性

事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。

3. 隔离性

一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

事务的性质

4．持续性

持续性也称永久性或耐久性(Permanence)，指一个事务一旦提交，它对数据库中数据的改变就应该是永久的，接下来的其他操作或故障不应该对其执行结果有任何影响。

事务的性质

- 在任何情况下，DBMS都应保证事务的ACID性质。
- 不但在系统正常运行时应保证事务的ACID性质；而且在系统发生故障时，也应保证事务的ACID性质。
- 不但在单个应用运行时应保证事务的ACID性质；而且在多个应用并行运行时，也应保证事务的ACID性质。

事务的性质

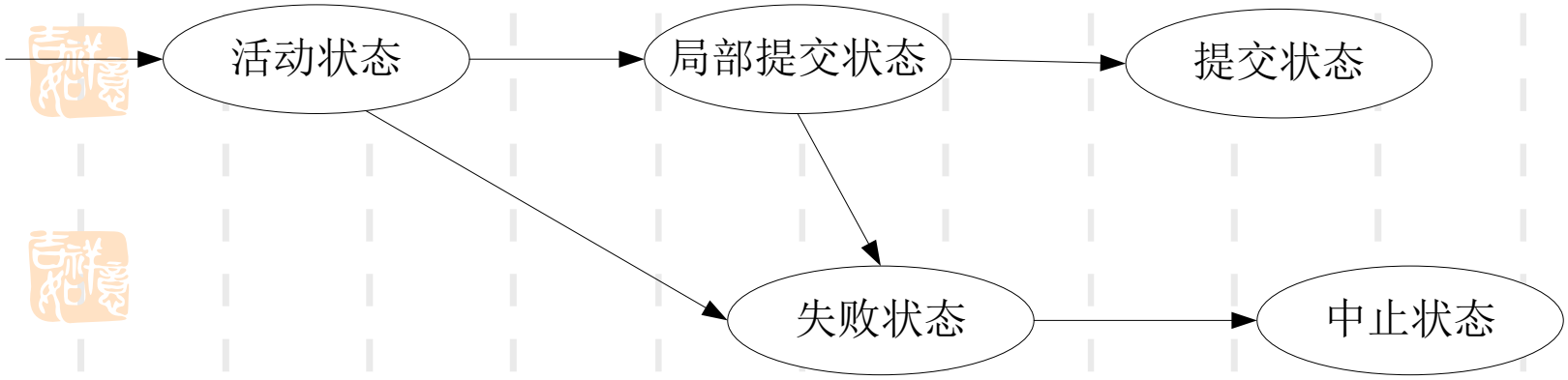
- 保证事务在故障发生时满足ACID性质的措施称为恢复技术。
- 保证多个事务在并行执行时满足ACID性质的措施称为并发控制技术。
- 恢复和并发控制是保证事务正确运行的两项基本技术，它们被合称为事务管理。

事务的操作

- 基于简化的表达方式，可把事务涉及的基本存取操作分为以下两种：
  - **Read(X)**：从数据库中将数据对象**X**读入到执行**read**操作的事务的一个局部缓冲区中。
  - **Write(X)**：从执行**write**的事务的局部缓冲区把数据对象**X**写入数据库。
- **Write**操作的结果可以暂时存储在内存中，**DBMS**的恢复子系统和底层操作系统协同控制，以决定写回磁盘的时机。

事务的状态

- 为了在发生故障时方便恢复，系统需要对事务所处的状态进行跟踪记录，包括事务的起始、提交、撤销或终止等信息。
- 下图所示是事务的状态转移图



事务的状态

- 活动状态：事务开始执行后就处于这个状态，此时事务可以执行读、写操作。
- 部分提交状态：事务结束时进入此状态。
- 失败状态：如果事务执行时检查出故障或在活动状态期间被撤销，那么事务就将进入失败状态。
- 提交状态：只有在事务进入提交状态后，事务已经成功结束，才说事务已提交。
- 中止状态：数据库被恢复到事务开始执行前的状态后，事务就进入了中止状态。

SQL的事务管理

- 在SQL中，事务的定义和前面提到的事务的概念类似，即它是一个逻辑工作单元，并且保持原子性。
- 事务显式的结束语句可以是COMMIT或ROLLBACK，书写形式如下列SQL语句：
  - COMMIT [WORK]：提交当前事务，开始执行一个新事务。
  - ROLLBACK [WORK]：中止当前事务，撤销事务的影响。

SQL的事务管理

- SQL中每个事务都有属于它的特性，包括存取模式、隔离级别等。
- 由SET TRANSACTION语句设置，语法格式如下：

```
SET TRANSACTION [{READ WRITE |READ ONLY }]……
[ISOLATION LEVEL
  {READ UNCOMMITTED | READ COMMITTED
    | REPEATABLE READ| SERIALIZABLE }]……
```

8.2 数据库系统故障

- 尽管数据库系统中采取了各种保护措施来防止数据库的安全性和完整性被破坏，保证并发事务的正确执行，但是计算机系统中硬件的故障、软件的错误、操作员的失误以及恶意的破坏仍是不可避免的。
- 我们将使得数据库系统无法正常运行的任何事件称为故障。

故障分类

- 根据故障的影响范围，一般将数据库系统中的故障分为以下类型:

1. 事务故障

2. 系统故障

3. 介质故障

事务故障

- 事务故障是指影响单个事务正常运行的任何事件。有两类错误可能引起事务故障，迫使事务执行失败：
  - 逻辑错误：事务由于内部逻辑上的问题而无法继续正常执行。
  - 系统错误：系统进入非正常状态，中断了事务的执行。

事务故障

- 事务故障有的是可以通过事务程序本身发现并处理；而有的则是非预期的，不能由事务程序自身处理。
- 一般而言，事务故障更多是非预期的，不能由应用程序自身处理。
- 如：运算溢出、并发事务发生死锁而被选中撤消该事务、违反了某些完整性限制等。
- 以后，事务故障一般是指这类非预期的故障。

事务故障

- 事务故障意味着事务没有达到预期的终点(COMMIT或者显式的ROLLBACK)，因此，数据库可能处于不正确状态。
- 恢复程序要在不影响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，使得该事务好象根本没有启动一样。
- 这类恢复操作称为事务撤消(UNDO)。

系统故障

- 系统故障是指造成系统停止运转的任何事件，使得系统要重新启动。
- 例如，特定类型的硬件错误(如CPU故障)、操作系统故障、DBMS代码错误、突然停电等等。
- 系统故障将影响所有正在运行的事务，虽然不破坏数据库,但是主存内容，尤其是数据库缓冲区中的内容都被丢失，导致所有运行事务都非正常终止。

系统故障

- 发生系统故障时，一些尚未完成的事务的结果可能已送入物理数据库，有些已完成的事务可能有一部分甚至全部留在缓冲区，尚未写回到磁盘上的物理数据库中，从而造成数据库可能处于不正确的状态。
- 为保证数据一致性，恢复子系统必须在系统重新启动时让所有非正常终止的事务回滚，强行撤消(UNDO)所有未完成事务，并且要重做(Redo)所有已提交的事务，以将数据库恢复到一致状态。

介质故障

- 介质故障主要是指外存故障，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。
- 介质故障比前两类故障发生的可能性小得多，但破坏性最大。
- 在对介质故障恢复时，可利用其它磁盘上的数据备份或档案数据库进行重构，然后利用事务日志(log)对最近后备副本以后提交的所有事务进行重做(redo)。

计算机病毒

- 计算机病毒是一种可以自我复制并具有破坏性的计算机程序。
- 计算机病毒已成为计算机系统的主要威胁，自然也是数据库系统的主要威胁。
- 因此，一旦数据库由于计算机计算机病毒遭到破坏，仍要用恢复技术对数据库加以恢复。

8.3 数据库恢复技术

- 数据库系统故障轻则造成运行事务非正常中断，影响数据库中数据的正确性；重则破坏数据库，使数据库中全部或部分数据丢失。
- 因此数据库管理系统(即恢复子系统)必须具有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，这就是数据库的恢复。

数据库恢复的基本原理

- 总结各类故障对数据库的影响有两种可能性
  - 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
- 故障恢复的基本原理十分简单，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”。
- 尽管恢复的基本原理很简单但实现技术的细节却相当复杂。

恢复机制

- 故障恢复机制涉及两个关键问题是：
  - ✓第一，如何建立冗余数据
  - ✓第二，如何利用这些冗余数据实施数据库恢复
- 建立冗余数据最常用的技术是建立数据备份和事务日志文件。
- 在数据库系统中，这两种方法通常是一起使用。

数据库备份

- 所谓备份即DBA定期地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。这些备用的数据文本称为后备副本或后援副本。
- 当数据库遭到破坏后可以将后备副本重新装入，但重装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。

数据库备份

- 例如，在下图中，系统在**Ta**时刻停止运行事务进行数据库备份，在**Tb**时刻备份完毕，得到**Tb0**时刻的数据库一致性副本。

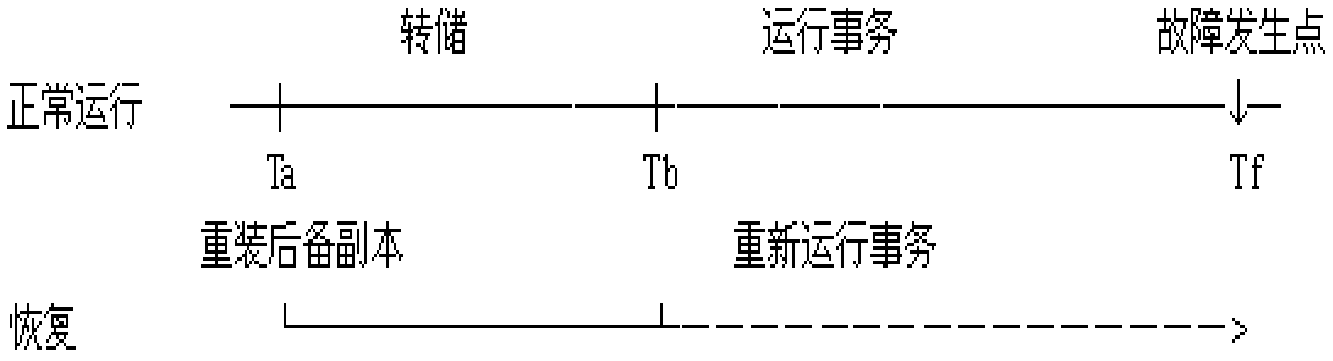


图 7.1 转储和恢复

数据库备份

- 系统运行到Tf时刻发生故障，为恢复数据库，首先由DBA重装数据库后备副本，将数据库恢复至Tb时刻的状态，然后重新运行自Tb时刻至Tf时刻的所有更新事务，这样就把数据库恢复到故障发生前的一致状态。

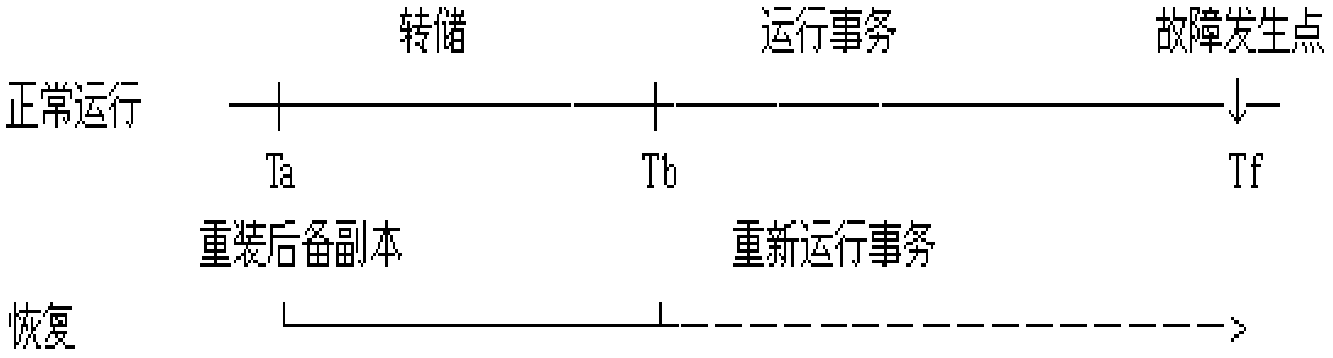


图 7.1 转储和恢复

数据库备份

- 由于备份十分耗费时间和资源，因此不能频繁进行。
- DBA应该根据数据库的使用情况确定一个适当的备份周期。
- 备份可分为脱机(off line)备份和联机(on line)备份两种方式。

数据库备份

- 脱机备份是在备份过程中数据库系统处于脱机状态，即在备份过程中数据库系统不再接受应用程序的访问，因此，脱机备份又称为静态备份。
- 联机备份是在备份过程中数据库系统处于联机状态，即在进行备份的同时数据库系统还可接受应用程序的访问，因此，联机备份又称为动态备份。

数据库备份

- 备份又可以分为全备份，部分备份和增量备份3种方式。
- 全备份是指每次备份全部数据库。
- 增量备份是指每次只备份上一次备份后更新过的数据。
- 从恢复角度看，使用全备份得到的后备副本进行恢复一般说来会更方便些。
- 但是，如果数据库很大，事务处理又十分频繁，则增量备份方式更实用更有效。

数据库日志

1. 日志文件的作用

- 日志文件在数据库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。
- 事务故障恢复和系统故障恢复必须用日志文件。
- 在动态备份方式中必须建立日志文件，后援副本和日志文件综合起来才能有效地恢复数据库。

数据库日志

1. 日志文件的作用

- 在静态备份方式中，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消处理。
- 这样不必重新运行那些已完成的事务程序就可把数据库恢复到故障前某一时刻的正确状态,如下图所示：

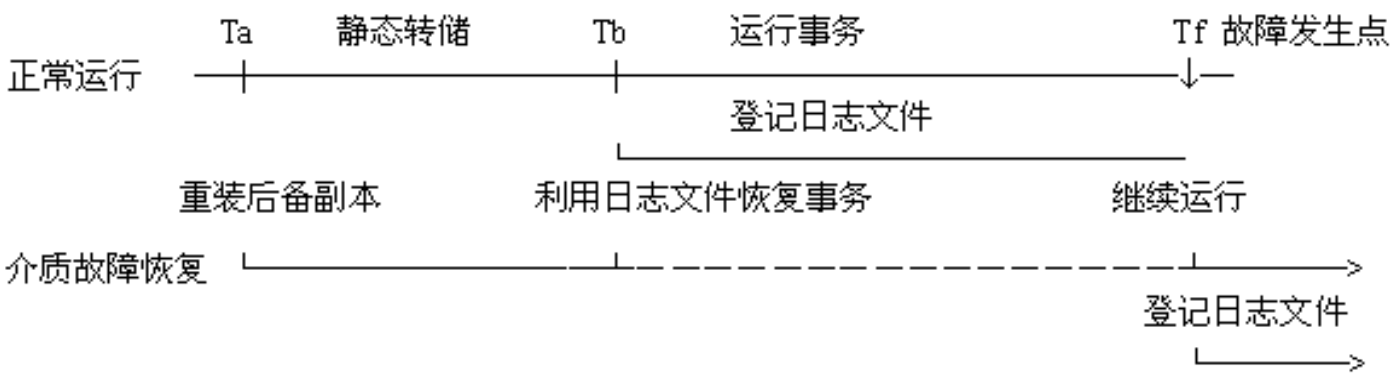


图 7.2　利用日志文件恢复

数据库日志

2. 日志文件的格式和内容

- 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。
- 概括起来日志文件主要有两种格式：
  - 以记录为单位的日志文件
  - 以数据块为单位的日志文件

数据库日志

2. 日志文件的格式和内容

- 对于以记录为单位的日志文件，日志文件中需要登记的内容包括：
  - 各个事务的开始(BEGIN TRANSACTION)标记
  - 各个事务的结束(COMMIT或ROLL BACK)标记
  - 各个事务的所有更新操作
- 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) 。

数据库日志

- 日志记录的内容主要包括：
- 事务标识TID(标明是那个事务)
- 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(对插入操作而言，此项为空值)
- 更新后数据的新值(对删除操作而言，此项为空值)

数据库日志

4. 登记日志文件(logging)

- 为保证数据库是可恢复的，登记日志文件时必须遵循两条原则：

1. 登记的次序必须严格按并发事务执行的时间次序
2. 必须先记日志文件，后写数据库

数据库日志
2022-4-10

5. 登记日志文件(logging)

- 把对数据的修改写到数据库中和把写表示这个修改的日志记录写到日志文件中是两个不同的操作，因此，有可能在这两个操作之间发生故障，即这两个写操作只完成了一个。
- 因此，如果先写了数据库修改，而在运行记录中没有登记下这个修改，则以后就无法恢复这个修改了。
- 而如果先写日志，但没有修改数据库，按日志文件恢复时只不过是多执行一次不必要的**REDO**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，即首先把日志记录写到日志文件中，然后写数据库的修改。
- 这就是所谓的“提前写日志”原则。

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点前被中止（“有头无尾”），这时恢复子系统应利用日志文件撤消(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。

恢复的策略

1. 事务故障的恢复

- 恢复步骤是：

① 反向扫描文件日志(即从最后向前扫描日志文件)，查找该事务的更新操作；

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 继续反向扫描日志文件，查找该事务的其他更新操作，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，事务故障恢复完成。

恢复的策略

2. 系统故障的恢复

- 系统故障造成数据库不一致状态的原因有两个，一是未完成事务对数据库的更新可能已写入数据库；二是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，恢复操作就是要撤消(undo)故障发生时未完成的事务，并且重做(redo)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，一般不需要用户干预。

恢复的策略

- 系统故障的恢复步骤：

① 正向扫描日志文件(即从头扫描日志文件)，找出在故障发生前已经提交事务(这些事务既有BEGIN TRANSACTION记录，也有COMMIT记录)，将其事务标识记入重做(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)，将其事务标识记入撤消(UNDO)队列

③ 对撤消队列中的各个事务进行撤消(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

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事务重新执行日志文件登记的操作。即将日志记录中“更新后的值”写入数据库。

恢复的策略

3. 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，恢复方法是重装数据库，然后重做已完成的事务。
- 介质故障的恢复需要DBA的介入。
- 但DBA只需要重装最近备份的数据库副本和有关的各日志文件副本，然后执行系统提供的恢复命令即可，具体的恢复操作仍由DBMS完成。

恢复的策略

3. 介质故障的恢复步骤：

① 装入最新的数据库后备副本(离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。

② 对于联机备份的数据库副本，还须同时装入备份开始时刻的日志文件副本，利用恢复系统故障的方法(即REDO+UNDO)，才能将数据库恢复到一致性状态。

③ 装入相应的日志文件副本(即备份结束时刻的日志文件副本)，重做已完成的事务。即：

- 首先扫描日志文件，找出故障发生时已提交的事务的标识，将其记入重做(REDO)队列。
- 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。

采用检查点的恢复技术

- 利用日志技术进行数据库恢复时，恢复子系统必须搜索日志，确定哪些事务需要REDO，哪些事务需要UNDO。
- 一般来说，系统需要检查所有日志记录，但这样做具有两个问题:
  - 一是搜索整个日志将耗费大量的时间。
  - 二是很多需要REDO处理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。

采用检查点的恢复技术

- 检查点技术就是在日志文件中增加一类新的记录--检查点记录(checkpoint)，增加一个重新开始(Restar)文件，并让恢复子系统在登录日志文件期间动态地维护日志。
- 检查点记录的内容包括：
  - 建立检查点时刻所有正在执行的事务清单。
  - 这些事务最近一个日志记录的地址。
  - 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。

采用检查点的恢复技术

- 动态维护日志文件的方法是，周期性地执行如下操作：建立检查点，保存数据库状态，
- 具体步骤是：

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

采用检查点的恢复技术

- 恢复子系统可以定期或不定期地建立检查点保存数据库状态。
- 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。
- 例如，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查点建立之前或在这个检查点建立之时，这样，在进行恢复处理时，没有必要对事务T执行REDO操作。

采用检查点的恢复技术

- 系统出现故障时恢复子系统将根据事务的不同状态采取不同的恢复策略。
- 如下图所示：

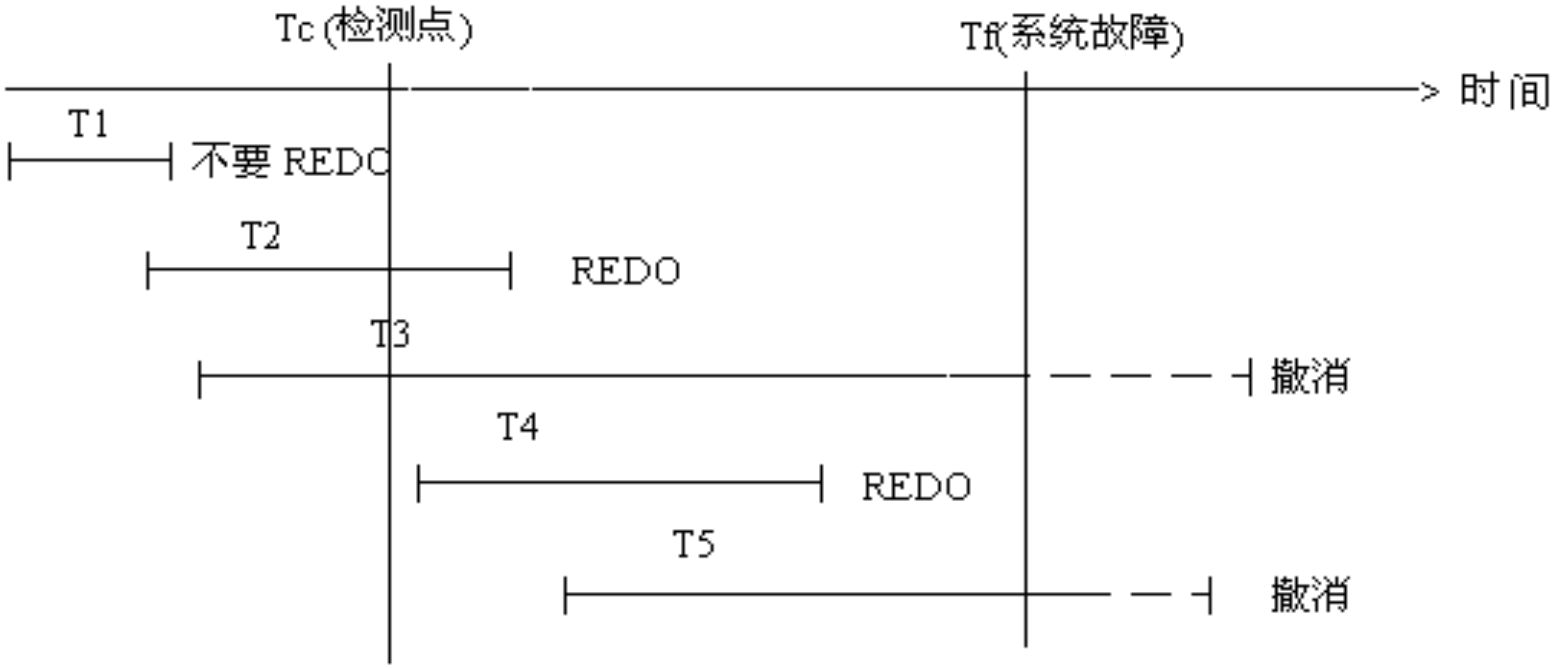


图 7.4

采用检查点的恢复技术

- T1：在检查点之前提交
- T2：在检查点之前开始执行，在检查点之后故障点之前提交
- T3：在检查点之前开始执行，在故障点时还未完成
- T4：在检查点之后开始执行，在故障点之前提交
- T5：在检查点之后开始执行，在故障点时还未完成
- T3和T5在故障发生时还未完成，所以予以撤消
- T2和T4在检查点之后才提交，它们对数据库所做的修改在故障发生时可能还在缓冲区中，尚未写入数据库，所以要REDO
- T1在检查点之前已提交，所以不必执行REDO操作

采用检查点的恢复技术

- 系统使用检查点进行恢复的步骤：

①从重新开始文件中找到最后一个检查点记录在日志文件中的地址，由该地址在日志文件中找到最后一个检查点记录

②由该检查点记录得到检查点建立时刻所有正在执行的事务清单**ACTIVE-LIST**

③建立两个事务队列:

- **UNDO-LIST:** 需要执行**undo**操作的事务集合
- **REDO-LIST:** 需要执行**redo**操作的事务集合
- 把**ACTIVE-LIST**暂时放入**UNDO-LIST**队列
- **REDO-LIST**队列暂为空

采用检查点的恢复技术

④ 从检查点开始正向扫描日志文件

- 如有新开始的事务Ti，把Ti暂时放入UNDO-LIST队列；
- 如有提交的事务Tj，把Tj从UNDO-LIST队列移到REDO-LIST队列；

直到日志文件结束。

⑤ 对UNDO-LIST中的每个事务执行UNDO操作，对REDO-LIST中的每个事务执行REDO操作

作业

- P303
- 10.1
- 10.4
- 10.7

8.5 数据库并发控制

- 数据库是一种共享资源，可以供多个用户使用，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。
- 当多个用户并发地存取数据库时就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，破坏数据库的一致性。
- 所以数据库管理系统必须提供并发控制机制。
- 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。

数据库系统中的并发

- 如果数据库中的事务顺序执行，即一个事务完全结束后，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，则称这种执行方式为交叉并发(interleaved concurrency)。
- 在多CPU系统中，可以允许多个事务同时占用CPU，我们称这种执行方式为同时并发(simultaneous concurrency)。

事务并发执行可能引起的问题

- 下面我们先来看一个例子，说明并发操作可能带来的数据的不一致性问题。
- 考虑飞机订票系统中的一个活动序列:

① 甲售票点(甲事务)读出某航班的机票余额A,设A=18

② 乙售票点(乙事务)读出同一航班的机票余额A,也为18

③ 甲售票点卖出一张机票,修改余额A←A-1.所以A为17,把A写回数据库.

④ 乙售票点也卖出一张机票,修改余额A←A-1.所以A为17,把A写回数据库

- 问题是，结果明明卖出两张机票，数据库中机票余额却只减少1张！！！

事务并发执行可能引起的问题

1. 丢失修改(lost update)

- 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的修改被丢失
- 上面飞机订票例子就属此类

事务并发执行可能引起的问题

2. 读“脏”数据(dirty read)

- 读“脏”数据是指事务T1修改某一数据，并将其写回磁盘，事务T2读取同一数据后，T1由于某种原因被撤消，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

事务并发执行可能引起的问题

3. 不可重复读(non-repeatable read)

- 不可重复读是指事务T1读取数据后，事务T2执行更新操作，使T1无法再现前一次读取结果。具体地讲，不可重复读包括三种情况:

i. 事务T1读取某一数据后，事务T2对其做了修改，当事务1再次读该数据时，得到与前一次不同的值。例如T1读取B=100进行运算，T2读取同一数据B，对其进行修改后将B=200写回数据库。T1为了对读取值校对重读B，B已为200，与第一次读取值不一致。

ii. 事务T1按一定条件从数据库中读取了某些数据记录后，事务T2删除了其中部分记录，当T1再次按相同条件读取数据时，发现某些记录神密地消失了。

iii. 事务T1按一定条件从数据库中读取某些数据记录后，事务T2插入了一些记录，当T1再次按相同条件读取数据时，发现神密地多了一些记录,这种现象又称为“读幻像行”。

并发控制的目标

- 产生上述三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它事务的干扰，从而避免造成数据的不一致性。
- 此外，并发控制还应能提高系统资源利用率并改善短事务的响应时间。

并发控制的正确性准则

- 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，那么哪个结果是正确的，哪个是不正确的呢？
- 如果一个事务运行过程中没有其他事务同时运行，也就是说它没有受到其他事务的干扰，那么就可以认为该事务的运行结果是正常的或者预想的。
- 因此将所有事务串行起来的调度策略一定是正确的调度策略。
- 虽然以不同的顺序串行执行事务可能会产生不同的结果，但由于不会将数据库置于不一致状态，所以都是正确的。

并发控制的正确性准则

- 定义1：设数据库系统中参与并发 执行的事务集为{T1,T2,…,Tn},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{T1,T2,…,Tn}，如果其中2个调度S1和S2，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S1和S2是目标等价(view equivalence)的。

并发控制的正确性准则

- 定义3：设数据库系统中参与并发 执行的事务集为{T1,T2,…,Tn}，如果其中的两个操作op1和op2满足下列条件：

  1) 它们属于不同的事务；

  2) 它们针对同一个数据单元；

  3) 其中至少有一个写(Write)操作；

  则称op1和op2是冲突操作。
- 冲突操作包括读-写冲突和写-写冲突两种
- 定义4：设数据库系统中参与并发 执行的事务集为{T1,T2,…,Tn}，S为一个调度，我们将那些通过调换S中不冲突的操作所得到的新的调度称为S的冲突等价(conflict equivalence)调度。
- 显然，如果两个调度是冲突等价，一定是目标等价的；反之，未必正确。

并发控制的正确性准则

- 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串行地执行它们时的结果相同。
- 我们称这种调度策略为可串行化(Serializable)的调度。
- 定义5：设数据库系统中参与并发 执行的事务集为{T1,T2,…,Tn}， S为一个调度，如果S能与一个串行调度冲突(目标)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。

可串行性的判定方法

- 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包含所有参与调度事务。边可以通过分析事务之间的冲突操作获得，其规则如下:
  - 如果2个事务Ti和Tj之间有一对冲突操作op1和op2，在S中op1被安排在op2之前，则在E中加入边：Ti → Tj
- 可以证明:

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。

可串行性的判定方法

- 例:设有下面4个事务:

T1: R11(x)R12(y)R13(z)W14(x)W15(y)W16(z)
T2: R21(x)W22(x)R23(y)W24(y)R25(z)W26(z)
T3: R31(x)R32(y)R33(z)W34(x)
T4: R41(x)R42(y)R43(z)W44(y)

试判断下列调度:

S1: R21(x)W22(x) R31(x)R23(y)W24(y) R32(y) R25(z)W26(z)R33(z)W34(x)R11(x)R12(y) R13(z)W14(x) R41(x)W15(y) R42(y)W16(z) R43(z)W44(y)

S2:R11(x)R21(x)W22(x)R23(y)W24(y)R12(y)R25(z) W26(z) R13(z)R41(x)W14(x)W15(y)W16(z) R42(y)R31(x)R32(y)R43(z)W44(y) R33(z)W34(x)

是否为冲突可串行的调度?

可串行性的实现方法

- 为了保证并发操作的正确性，DBMS的并发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。

8.6 基于锁的并发控制

- 锁是实现并发控制的重要的技术。
- 所谓加锁就是事务T在对某个数据对象例如表、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，其它的事务不能更新此数据对象。

锁的类型

- 基本的加锁类型:
  - 排它锁(Exclusive locks 简记为X锁)
  - 共享锁(Share locks 简记为S锁)及
  - 更新锁(Update lock简记为U锁)

锁的类型

1.排它锁(X锁)

- 排它锁(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，其它任何事务都不能再对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释放A上的锁之前不能再读取和修改A。

2.共享锁(S锁)

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

锁的类型

3. 更新锁(U锁)

- 除X锁、S锁两种所之外，又增加了U锁。事务在进行更新时，一般都需要先读出旧的内容，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被其他事务访问。
- U锁就是为此目的设置的。
- 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。

锁的相容性

	S	U	X
S	Y	Y	N
U	Y	N	N
X	N	N	N

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。
- 下面介绍三级加锁协议。对并发操作的不正确调度可能会带来丢失修改、不可重复读和读“脏”数据等不一致性问题，三级加锁协议分别在不同程度上解决了这一问题。为并发操作的正确调度提供一定的保证。
- 应注意，不同级别的加锁协议达到的系统一致性级别是不同的。

加锁协议

- 1级加锁协议：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)。
- 1级加锁协议可防止丢失修改，并保证事务T是可恢复的。
- 在1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不读“脏”数据。

加锁协议

- 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，读完后即可释放S锁。
- 2级加锁协议除了可以防止了丢失修改，还可进一步防止读“脏”数据，但2级加锁协议不能防止不可重复读。

加锁协议

- 3级加锁协议：2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(EOT)才释放。
- 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。

加锁协议

- 3级加锁协议的比较

表8.1 不同级别的封锁协议

	X 锁		S 锁		一致性保证		
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不丢失修改	不读'脏'数据	可重复读
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

SQL中一致性级别的定义

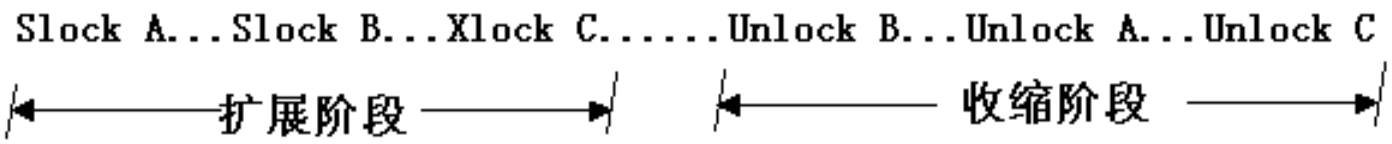
- **serializable：** 一个调度的执行必须等价于一个串行调度的结果。
- **repeatable read：** 只允许读取已提交的记录，并要求一个事务对同一记录的两次读取之间，其它事务不能对该记录进行更新。
- **read committed：** 只允许读取已提交的记录，但不要求可重复读。
- **read uncommitted：** 允许读取未提交的记录。

两段锁协议(2PL)

- 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。这在阶段，事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁。
- 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琐，但是不能再申请任何锁。

两段锁协议(2PL)

- 例如事务T1遵守两段锁协议，其封锁序列是：



- 又如事务T2不遵守两段锁协议，其封锁序列是：
  Slock A … Unlock A … Slock B … Xlock C … Unlock C … Unlock B;
- 可以证明，若所有并发执行的事务均遵守两段锁协议，则对这些事务的任何并发调度都是可串行化的。

两段锁协议(2PL)

- 另外要注意两段锁协议和防止死锁的一次封锁法的异同。
- 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。
- 但是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。

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，则其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，如果不加干涉，则会一直等待下去，这就是死锁(dead lock)。
- 对于死锁有两种方法：一是防止死锁的出现；二是由系统检测死锁，一旦发现则对其处理。

死锁的检测处理和预防

- 现实中的“死锁”！

死锁的检测处理和预防

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，然后又都请求对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。

死锁的检测处理和预防

⑩ 预防死锁通常有以下几种方法：

1) 一次封锁法

- 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。
- 一次封锁法虽然可以有效地防止死锁的发生，但也存在问题，一次就将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度。

死锁的检测处理和预防

2) 顺序封锁法

- 顺序封锁法是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。
- 顺序封锁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封。

死锁的检测处理和预防

2022-4-13

3) 比较实用的死锁预防方法

- 在操作系统中广为采用的预防死锁的策略并不很适合数据库的特点，因此下面介绍一种在DBMS中比较实用的死锁的预防方法。
- 在这种方法中，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执(retry)，以避免循环等待。
- 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T_A和T_B，如$ts(T_A)<ts(T_B)$，则T_A表示早于T_B。也称T_A比T_B“年老”，或者说T_B比T_A“年轻”。

死锁的检测处理和预防

- 事务重执一般有2种策略

1) 等待-死亡(wait-die)策略

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

```
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务。

死锁的检测处理和预防

2) 击伤-等待(wound-wait)策略

这种策略按另一规则处理冲突：

if ts(T_A) > ts(T_B) then T_A waits;

```
else{
    rollback TB;   /* Die */
    restart TB with the same ts(TB);
}
```

- ⑩ 在这种策略中，总是年轻的事务等待年老的事务。

死锁的检测处理和预防

2. 死锁的检测与解除

1) 超时法

- 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，但其不足也很明显。
- 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。

死锁的检测处理和预防

2) 等待图法

- 事务等待图是一个有向图G=(T,U), 其中：T为结点的集合，每个结点表示正运行的事务。U为边的集合，每条边表示事务等待的情况。
- 若T1等待T2,则T1、T2之间划一条有向边，从T1指向T2。
- 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。

死锁的检测处理和预防

3) 死锁的解除

- 死锁发生后，靠事务自身无法解除，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，将其撤消，释放此事务持有的所有的锁，使其它事务得以继续运行下去。
- 当然，对撤消的事务所执行的数据修改操作必须加以恢复。

活锁

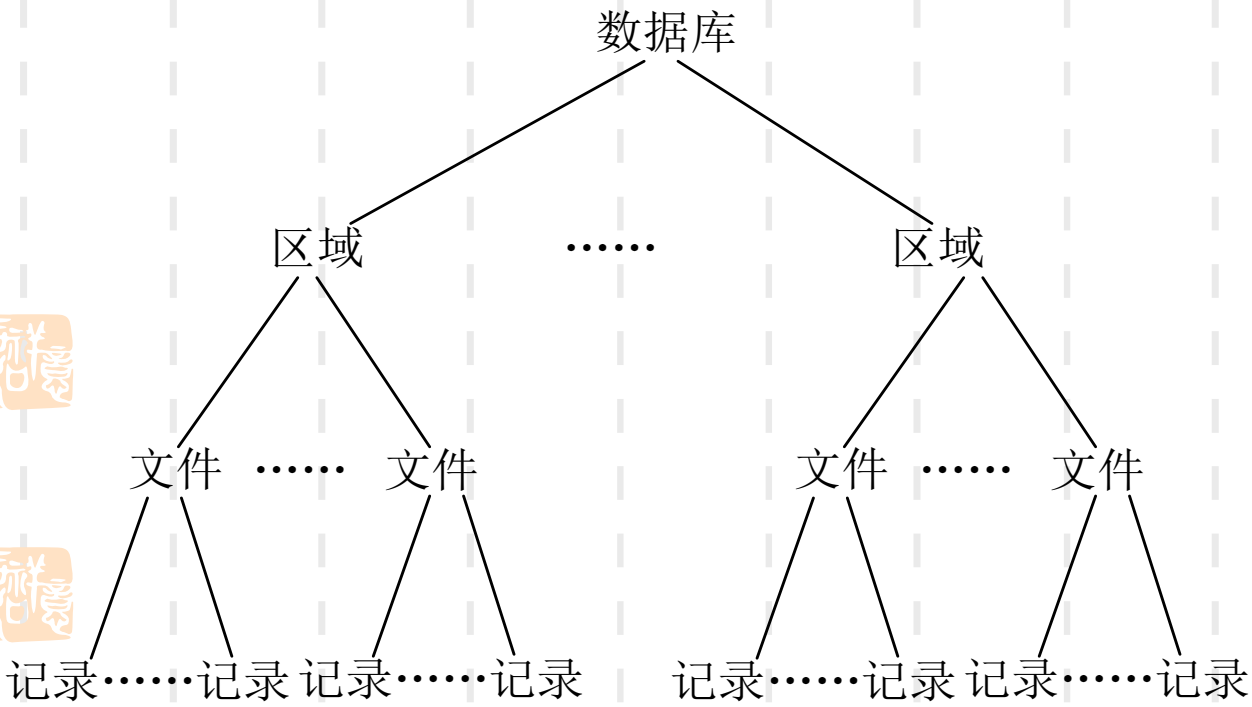
- 如果事务T1封锁了数据R，事务T2又请求封锁R，于是T2等待；T3也请求封锁R，当T1释放了R上的封锁之后系统首先批准了T3的请求，T2仍然等待；然后T4又请求封锁R，当T3释放了R上的封锁之后系统又批准了T4的请求，...，T2有可能永远等待，这就是活锁。
- 避免活锁的简单方法是采用“先来先服务”的策略。

多粒度封锁

- 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后裔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两种方式封锁：

1) 显式封锁
2) 隐式封锁

多粒度封锁



多粒度封锁

- 意向锁
  - 一般地，对某个数据对象加锁，系统要检查该数据对象上有无显式封锁与之冲突；不仅要检查其所有上级结点，看本事务的显式封锁是否与该数据对象上的隐式封锁(即由于上级结点已加的封锁造成的)冲突；而且还要检查其所有下级结点，看上面的显式封锁是否与本事务的隐式封锁(将加到下级结点的封锁)冲突。
  - 显然，这样的检查方法效率很低，为此引进了意向锁(intention lock)。

多粒度封锁

⑩意向锁

- 意向锁的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。
- 例如，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是否已加了不相容的锁，而不再需要搜索和检查R1中的每一个元组是否加了X锁。

多粒度封锁

1) 意向共享锁，简记为IS锁

- 如果对一个数据对象加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，则要首先对关系和数据库加IS锁。

2) 意向排它锁(intent exclusive lock)，简记为IX锁

- 如果对一个数据对象加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，则要首先对关系和数据库加IX锁。

3) 共享意向排它锁(share intent exclusive lock)，简记为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)。

多粒度封锁

- 图8.9(A)给出了这些锁的相容矩阵，从中可以发现这五种锁的强度有图8.9(B)所示的偏序关系。
- 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。
- 具有意向锁的多粒度封锁方法中任意事务T要对一个数据对象加锁，必须先对它的上层结点加意向锁。

多粒度封锁

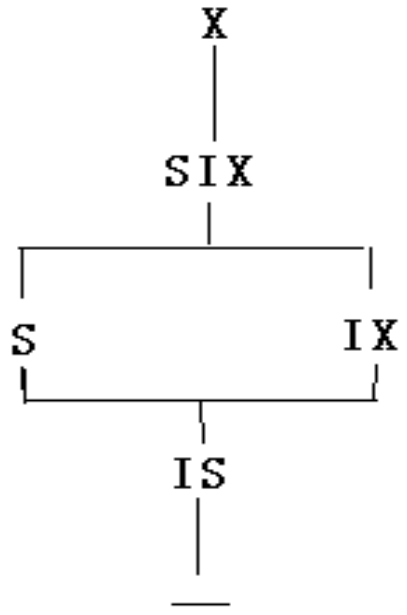
- 申请封锁时应该按自上而下的次序进行。
- 释放封锁时则应该按自下而上的次序进行。
- 具有意向锁的多粒度封锁方法提高了系统的并发度，减少了加锁和解锁的开销，它已经在实际的数据库管理系统产品中得到广泛应用。
- 例如，IBM DB2 UDB DBMS中就采用了这种封锁方法。

多粒度封锁

T1 \ T2	S	X	IS	IX	SIX	—
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
—	Y	Y	Y	Y	Y	Y

Y=Yes，表示相容的请求　　N=No，表示不相容的请求

（a）数据锁的相容矩阵



（b）锁的强度的偏序关系

图8.9 锁的相容矩阵

插入与删除操作

- 读和写操作：针对数据库中已经存在的数据。
- 删除操作：删除数据库中的数据项。
- 插入操作：插入新的数据项到数据库。
- 冲突情况：(分析删除操作 I 与其它操作J的冲突)

  I = delete (Q)

  J = read (Q): 如果I在J之前，出现逻辑错误

  J = write (Q): 如果I在J之前，出现逻辑错误

  J = delete (Q): 出现逻辑错误

  J = insert(Q): 如果Q存在，J在I之前，出现逻辑错误；如果Q不存在， I在J之前，出现逻辑错误
- 插入操作情况类似，和其他操作也冲突。

插入与删除操作

如果使用两阶段锁:

- 仅当删除元组的事务对删除元组具有排他锁时, **delete** 操作才可以被执行。
- 向数据库插入一条新元组的事务对该元组获得X锁(排他锁)。
- 插入和删除可能导致幻影现象
- 扫描关系的事务和向关系中插入元组的事务尽管不存取任何共同的元组也可能冲突。
- 如果只使用元组锁, 可导致非串行化调度。

插入与删除操作

一种解决方法:

- 将关系与一个数据项相关联, 该数据项表示“关系中包含什么元组”的信息。
- 扫描关系的事务获得该数据项上的共享锁。
- 插入或删除元组的事务获得该数据项上的排他锁。
- 注: 该数据项上的锁与各个元组上的锁不冲突。
- 上述协议对插入/删除操作提供了很低的并发性。
- 索引封锁协议提供了较高的并发性同时防止了幻影现象, 此协议要求对某些索引桶加锁 。

8.6 其他并发控制技术*

1.基于时间标记的并发控制技术

- 事务T的时间标记： ts(T)
- 数据D的时间标记：
  - 读时间标记tr(D)
  - 写时间标记tw(D)
- 事务T读数据read(D);

```
if ts(T) ≥tw(D)
then        /*符合时间标记协议，即
  read(D)     为读取值*/
  tr (D):= 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卷回*/
  rollback T and resart it with a new ts(T);
```

8.6 其他并发控制技术*

- 事务T写数据D：

```
if  ts(T) ≥tr(D) AND ts(T) ≥tw(D)
then        /*符合时间标记协议*/{
     write(D);
     tw(D) := ts(T);
     }
else        /*违反时间标记协议，T应卷回*/
   rollback T and resart it with a new ts(T);
```

8.6 其他并发控制技术*

2. 乐观并发控制技术

- 事务*Ti* 的执行分成三个阶段：
  1. 读与执行阶段: 事务*Ti* 的write操作只写到临时局部变量。
  2. 有效性检查阶段: 事务*Ti* 执行“有效性检查”来决定局部变量的值是否可以写到数据库而不违反可串行化。
  3. 写阶段: 若*Ti* 通过有效性检查, 则更新数据库; 否则*Ti* 回滚。
- 并发执行的各事务的三个阶段可以交叉, 但是每个事务必须按顺序通过三个阶段。
- 也称为乐观并发控制，因为事务执行时完全寄希望于在有效性检查时一切都好。

8.6 其他并发控制技术*

- 每个事务Ti 有三个时间戳
  - Start(Ti): Ti 开始执行的时间
  - Validation(Ti): Ti 进入有效性检查阶段的时间
  - Finish(Ti): Ti 完成写阶段的时间
- 为了增加并发性，可串行化次序由有效性检查时的时间戳次序决定。
- 因此TS(Ti) 被赋予 Validation(Ti)的值。
- 如果冲突的概率较小, 本协议很有用, 能带来更大程度的并发性，这是因为可串行化次序不是预先决定的, 相对较少的事务会被回滚。

8.6 其他并发控制技术*

- 若对所有满足TS (T_i) < TS (T_j)的T_i 都有下列任一条件成立:

1) finish(T_i) < start(T_j)

2) start(T_j) < finish(T_i) < validation(T_j) 并且T_i 所写的数 据项集合与T_j 所读的数据项集合不相交。

- 则通过有效性检查, T_j 可以提交. 否则有效性检查失败, T_j 夭折。
- 正确性说明: 要么满足第一个条件, 没有重叠的执行; 要么满足第二个条件, 并且:

1. T_j 的写不影响T_i 的读, 因为它们发生在T_i 完成读操作之后.
2. T_i 的写不影响T_j 的读, 因为T_j 不读T_i 所写的任何数据项.

作业

- P323
- 11.2
- 11.3
- 11.6
- 11.9
- 11.10