第16章
Database System Concepts, 6E

恢复系统

计算机系统与其他任何设备一样易发生故障。故障的原因多种多样，包括磁盘故障、电源故障、软件错误、机房失火，甚至人为破坏。一旦有任何故障发生，就可能会丢失信息。因此，数据库系统必须预先采取措施，以保证即使发生故障，也可以保持第14章所讲的事务的原子性和持久性。**恢复机制**(recovery scheme)是数据库系统必不可少的组成部分，它负责将数据库恢复到故障发生前的一致的状态。恢复机制还必须提供**高可用性**(high availability)，即：它必须将数据库崩溃后不能使用的时间缩减到最短。

16.1 故障分类

系统可能发生的故障有很多种，每种故障需要不同的方法来处理。在本章中，我们将只考虑如下类型的故障。

- **事务故障**(transaction failure)。有两种错误可能造成事务执行失败：
  - □ **逻辑错误**(logical error)。事务由于某些内部条件而无法继续正常执行，这样的内部条件如非法输入、找不到数据、溢出或超出资源限制。
  - □ **系统错误**(system error)。系统进入一种不良状态(如死锁)，结果事务无法继续正常执行。但该事务可以在以后的某个时间重新执行。
- **系统崩溃**(system crash)。硬件故障，或者是数据库软件或操作系统的漏洞，导致易失性存储器内容的丢失，并使得事务处理停止。而非易失性存储器仍完好无损。

  硬件错误和软件漏洞致使系统终止，而不破坏非易失性存储器内容的假设称为**故障－停止假设**(fail-stop assumption)。设计良好的系统在硬件和软件层有大量的内部检查，一旦有错误发生就会将系统停止。因此，故障－停止假设是合理的。
- **磁盘故障**(disk failure)。在数据传送操作过程中由于磁头损坏或故障造成磁盘块上的内容丢失。其他磁盘上的数据拷贝，或三级介质(如DVD或磁带)上的归档备份可用于从这种故障中恢复。 721

要确定系统如何从故障中恢复，我们首先需要确定用于存储数据的设备的故障方式。其次，我们必须考虑这些故障方式对数据库内容有什么影响。然后我们可以提出在故障发生后仍保证数据库一致性以及事务的原子性的算法。这些算法称为恢复算法，由两部分组成：

1. 在正常事务处理时采取措施，保证有足够的信息可用于故障恢复。
2. 故障发生后采取措施，将数据库内容恢复到某个保证数据库一致性、事务原子性及持久性的状态。

16.2 存储器

正如我们在第10章中所看到的，数据库中的各种数据项可在多种不同存储介质上存储并访问。在14.3节中，我们看到存储介质可以按照它们相对的速度、容量和顺应故障的能力来划分。我们把存储器分为以下三类：

- **易失性存储器**(volatile storage)
- **非易失性存储器**(nonvolatile storage)
- **稳定存储器**(stable storage)

稳定存储器，或更准确地说是接近稳定的存储器，在恢复算法中起到至关重要的作用。

16.2.1 稳定存储器的实现

要实现稳定存储器，我们需要在多个非易失性存储介质（通常是磁盘）上以独立的故障模式复制所
722 需信息，并且以受控的方式更新信息，以保证数据传送过程中发生的故障不会破坏所需信息。

前面（第10章）讲到RAID系统保证了单个磁盘的故障（即使发生在数据传送过程中）不会导致数据丢失。最简单并且最快的RAID形式是磁盘镜像，即在不同的磁盘上为每个磁盘块保存两个拷贝。RAID的其他形式代价低一些，但性能也差一些。

但是，RAID系统不能防止由于灾难（如火灾或洪水）而导致的数据丢失。许多系统通过将归档备份存储在磁带上并转移到其他地方来防止这种灾难。但是，由于磁带不能被连续不断地移至其他地方，最后一次磁带被移至其他地方以后所做的更新可能会在这样的灾难中丢失。更安全的系统远程为稳定存储器的每一个块保存一份拷贝，除在本地磁盘系统进行块存储外，还通过计算机网络写到远程去。由于在往本地存储器输出块的同时也要输出到远程系统，一旦输出操作完成，即使发生火灾或洪水这样的灾难，输出结果也不会丢失。我们在16.9节学习这种*远程备份系统*。

本节剩余的部分将讨论如何在数据传送过程中保护存储介质不受故障损害。在内存和磁盘存储器间进行块传送有以下几种可能结果：

- **成功完成**（successful completion）。传送的信息安全地到达目的地。
- **部分失败**（partial failure）。传送过程中发生故障，目标块有不正确信息。
- **完全失败**（total failure）。传送过程中故障发生得足够早，目标块仍完好无缺。

我们要求，如果**数据传送故障**（data-transfer failure）发生，系统能检测到并且调用恢复过程将块恢复成为一致的状态。为达到这个要求，系统必须为每个逻辑数据库块维护两个物理块；若是镜像磁盘，则两个块在同一个地点；若是远程备份，则一个块在本地，另一个在远程节点。输出操作的执行如下：

1. 将信息写入第一个物理块。
2. 当第一次写成功完成时，将相同信息写入第二个物理块。
3. 只有第二次写成功完成时，输出才算完成。

如果在对块进行写的过程中系统发生故障，有可能一个块的两个拷贝互相不一致。在恢复过程中，对于每一个块，系统需要检查它的两个拷贝。如果它们相同并且没有检测到错误存在，则不需要采取进一步动作。（前面讲到，磁盘块中的某些错误，如部分写块，可由存储在每个块中的校验和检测
723 到。）如果系统检测到一个块中有错误，则可以用另一个块的内容替换这一块的内容。如果两个块都没有检测出错误，但它们的内容不一致，则我们用第二块的值替换第一块的内容，或者用第一块的值替换第二块的内容。不管用哪个方法，恢复过程都保证，对稳定存储器的写要么完全成功（即更新所有拷贝），要么没有任何改变。

在恢复过程中要求比较每一对相应块的开销太大。通过使用少量非易失性RAM，跟踪正在进行的对块的写操作，我们可以大大降低开销。在恢复时，只须比较正在写的块。

将块写到远程节点的协议类似于将块写到镜像磁盘系统的协议，我们在第10章讨论过了，特别是实践练习10.3中。

我们可以将这个过程很容易地推广为允许为稳定存储器的每一个块使用任意多的拷贝。尽管使用大量拷贝比使用两个拷贝发生故障的可能性要低，但通常只用两个拷贝模拟稳定存储器是合理的。

16.2.2 数据访问

正如第10章中所看到的，数据库系统常驻于非易失性存储器（通常为磁盘），在任何时间都只有数据库的部分内容在主存中。[⊖] 数据库分成称为**块**（block）的定长存储单位。块是磁盘数据传送的单位，

⊖ 有一类特殊的数据库系统，称作*主存数据库系统*，它的整个数据库可以一次全部载入内存。26.4节讨论这样的系统。

可能包含多个数据项。我们假设没有数据项跨两个或多个块。这个假设对于大多数数据处理应用，例如银行或大学，都是正确的。

事务由磁盘向主存输入信息，然后再将信息输出回磁盘。输入和输出操作以块为单位完成。位于磁盘上的块称为**物理块**(physical block)，临时位于主存的块称为**缓冲块**(buffer block)。内存中用于临时存放块的区域称为**磁盘缓冲区**(disk buffer)。

磁盘和主存间的块移动是由下面两个操作引发的：

1. input(B)传送物理块 B 至主存。
2. output(B)传送缓冲块 B 至磁盘，并替换磁盘上相应的物理块。

这一机制如图 16-1 所示。

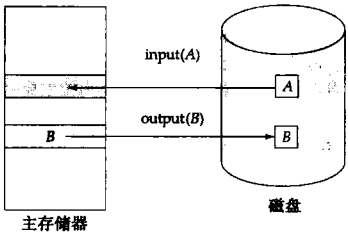

图 16-1 块存储操作

在概念上，每个事务 T_i 有一个私有工作区，用于保存 T_i 所访问及更新的所有数据项的拷贝。该工作区在事务初始化时由系统创建；在事务提交或中止时由系统删除。事务 T_i 的工作区中保存的每一个数据项 X 记为 x_i。事务 T_i 通过在其工作区和系统缓冲区之间传送数据，与数据库系统进行交互。我们使用下面两个操作来传送数据：

1. read(X)将数据项 X 的值赋予局部变量 x_i。该操作执行如下：
   a. 若 X 所在块 B_X 不在主存中，则发指令执行 input(B_X)。
   b. 将缓冲块中 X 的值赋予 x_i。
2. write(X)将局部变量 x_i 的值赋予缓冲块中的数据项 X。该操作执行如下：
   a. 若 X 所在块 B_X 不在主存中，则发指令执行 input(B_X)。
   b. 将 x_i 的值赋予缓冲块 B_X 中的 X。

注意这两个操作都可能需要将块从磁盘传送到主存。但是，它们都没有特别指明需要将块从主存传送到磁盘。

缓冲块最终写到磁盘，要么是因为缓冲区管理器出于其他用途需要内存空间，要么是因为数据库系统希望将 B 的变化反映到磁盘上。如果数据库系统发指令执行 output(B)，则我们称数据库系统对缓冲块 B 进行**强制输出**(force-output)。

当事务第一次需要访问数据项 X 时，它必须执行 read(X)。该事务然后对 X 的所有更新都作用于 x_i。在一个事务执行中的任何时间点，事务都可以执行 write(X)，以在数据库中反映 X 的变化；在对 X 进行最后的写之后，当然必须做 write(X)。 724 ↑ 725

对 X 所在的缓冲块 B_X 的 output(B_X)操作不需要在 write(X)执行后立即执行，因为块 B_X 可能包含其他仍在被访问的数据项。因此，可能一段时间以后才真正执行输出。注意，如果在 write(X) 操作执行后但在 output(B_X)操作执行前系统崩溃，X 的新值并未写入磁盘，于是就丢失了 X 的新值。正如我们很快就会看到的，数据库系统执行额外的动作来保证，即使发生了系统崩溃，由提交的事务所做的更新也不会丢失。

16.3 恢复与原子性

再来考虑简化的银行系统和事务 T_i，它将 \$50 从账户 A 转到账户 B，A 和 B 的初始值分别为 \$1000 和 \$2000。假设在 T_i 执行过程中，在 output(B_A)之后，output(B_B)之前，发生了系统崩溃，其中 B_A 和 B_B 表示 A 和 B 所在的缓冲块。由于内存的内容丢失，因此我们无法知道事务的结局。

当系统重新启动时，A 的值会是 \$950，而 B 的值是 \$2000，这显然和事务 T_i 的原子性需求不一致。遗憾的是，没有办法通过检查数据库状态来找出在系统崩溃发生前哪些块已经输出，哪些块还没有。有可能事务已经完成了，对稳定存储器中的数据库初始状态 A 和 B 的值分别为 \$1000 和 \$1950 进行了更新；也可能事务没有对稳定存储器产生任何影响，A 和 B 的值初始就是 \$950 和 \$2000；或者更新后的 B 已经输出了，而更新后的 A 还没有输出，或更新后的 A 已经输出了，而更新后的 B 还

没有输出。

我们的目标是要么执行 T_i 对数据库的所有修改，要么都不执行。但是，若 T_i 执行多处数据库修改，就可能需要多个输出操作，并且故障可能发生于某些修改完成后而全部修改完成前。

为达到保持原子性的目标，我们必须在修改数据库本身之前，首先向稳定存储器输出信息，描述要做的修改。我们将看到，这种信息能帮助我们确保已提交事务所做的所有修改都反映到数据库中(或者在故障后的恢复过程中反映到数据库中)。这种信息还能帮助我们确保中止事务所做的任何修改都不会持久存在于数据库中。

16.3.1 日志记录

使用最为广泛的记录数据库修改的结构就是**日志**(log)。日志是**日志记录**(log record)的序列，它记录数据库中的所有更新活动。

726 日志记录有几种。**更新日志记录**(update log record)描述一次数据库写操作，它具有如下几个字段：

- **事务标识**(transaction identifier)，是执行 **write** 操作的事务的唯一标识。
- **数据项标识**(data-item identifier)，是所写数据项的唯一标识。通常是数据项在磁盘上的位置，包括数据项所驻留的块的块标识和块内偏移量。
- **旧值**(old value)，是数据项的写前值。
- **新值**(new value)，是数据项的写后值。

我们将一个更新日志记录表示为 $< T_i, X_j, V_1, V_2 >$，表明事务 T_i 对数据项 X_j 执行了一个写操作，写操作前 X_j 的值是 V_1，写操作后 X_j 的值是 V_2。其他专门的日志记录用于记录事务处理过程中的重要事件，如事务的开始以及事务的提交或中止。如下是一些日志记录类型：

- $< T_i$ start $>$。事务 T_i 开始。
- $< T_i$ commit $>$。事务 T_i 提交。
- $< T_i$ abort $>$。事务 T_i 中止。

后面将介绍几种其他的日志记录类型。

每次事务执行写操作时，必须在数据库修改前建立该次写操作的日志记录并把它加到日志中。一旦日志记录已存在，就可以根据需要将修改输出到数据库中。并且，我们有能力撤销已经输出到数据库中的修改，这是利用日志记录中的旧值字段来做的。

为了从系统故障和磁盘故障中恢复时能使用日志记录，日志必须存放在稳定存储器中。现在我们假设每一个日志记录创建后立即写入稳定存储器中的日志的尾部，在 16.5 节中我们将看到什么时候可以放宽这个要求，以减少写日志带来的开销。由于日志包含所有的数据库活动的完整记录，因此日志中存储的数据量会变得非常大，在 16.3.6 节我们将看到什么时候可以安全删除日志信息。

影子拷贝和影子页面

在影子拷贝(shadow-copy)模式下，想要更新数据库的事务应首先创建数据库的一个完整拷贝。所有的更新在数据库的这个新拷贝上进行，而不去动那个原来的拷贝，影子拷贝(shadow copy)。如果在任何一个点上事务需要中止，系统仅仅删除这个新拷贝。数据库的旧拷贝没有受到影响。数据库的当前拷贝由一个指针来标识，称作数据库指针，它存放在磁盘上。

如果事务部分提交(即，执行它的最后一条语句)，那么它如下进行提交：首先，要求操作系统确保数据库的新拷贝的所有页面都写到磁盘上。(为此目的，UNIX 系统使用 fsync 命令。)在操作系统将所有页面都写到磁盘上之后，数据库系统更新数据库指针，让它指向数据库的新拷贝；然后新拷贝变成数据库的当前拷贝。然后删除掉数据库的旧拷贝。在更新后的数据库指针写到磁盘上这一时间点，我们说该事务已经提交了。

影子拷贝的实现实际上依赖于对数据库指针的写是原子的；即，或者它的所有字节全部写出，或者没有任何字节写出。磁盘系统提供对整个块的原子更新，或至少是对一个磁盘扇区的。换句话说，磁盘系统保证它会原子地更新数据库指针，只要我们确保数据库指针完全处于单个扇区中，而这是我们通过将数据库指针存放在块的开头所能够保证的。

影子拷贝模式普遍用于正文编辑器(保存文件等价于事务提交，不保存文件就退出等价于事务中止)。影子拷贝可以用于小的数据库，但拷贝一个大型数据库会是极其昂贵的。影子拷贝的一个变种，称作**影子页面**(shadow-paging)，它采用如下方式来减少拷贝工作量：此种模式使用一个包含指向所有页面的指针的页表；页表自身和所有更新的页面被拷贝到一个新的位置。事务没有更新的任何页面都不拷贝，而新的页表只存储一个指向原来页面的指针。当提交事务时，它原子地更新指向页表(页表的作用和数据库指针相同)的指针，以指向新的拷贝。

遗憾的是，影子页面对于并发事务不能很好地工作，在数据库中它没有广泛使用。

16.3.2 数据库修改

正如我们前面已经注意到的，事务在对数据库进行修改前创建了一个日志记录。日志记录使得系统在事务必须中止的情况下能够对事务所做的修改进行撤销；并且在事务已经提交但在修改已存放到磁盘上的数据库中之前系统崩溃的情况下能够对事务所做的修改进行重做。为了使我们能够理解恢复过程中日志记录的作用，我们需要考虑事务在进行数据项修改中所采取的步骤：

1. 事务在主存中自己私有的部分执行某些计算。
2. 事务修改主存的磁盘缓冲区中包含该数据项的数据块。
3. 数据库系统执行 **output** 操作，将数据块写到磁盘中。

727
728

如果一个事务执行了对磁盘缓冲区或磁盘自身的更新，我们说这个事务修改了数据库；而对事务在主存中自己私有的部分进行的更新不算数据库修改。如果一个事务直到它提交时都没有修改数据库，我们就说它采用了**延迟修改**(deferred-modification)技术。如果数据库修改在事务仍然活跃时发生，我们就说它采用了**立即修改**(immediate-modification)技术。延迟修改所付出的开销是，事务需要创建更新过的所有的数据项的本地拷贝；而且如果一个事务读它更新过的数据项，它必须从自己的本地拷贝中读。

本章描述的恢复算法支持立即修改。正如所描述的，即使对于延迟修改，它们也能正确工作，但是当与延迟修改一起使用时可以进行优化，以减少开销；我们将细节留作练习。

恢复算法必须考虑多种因素，包括：

- 有可能一个事务已经提交了，虽然它所做的某些数据库修改还仅仅存在于主存的磁盘缓冲区中，而不在磁盘上的数据库中。
- 有可能处于活动状态的一个事务已经修改了数据库，而作为后来发生的故障的结果，这个事务需要中止。

由于所有的数据库修改之前必须建立日志记录，因此系统有数据项修改前的旧值和要写给数据项的新值可以用。这就使得系统能执行适当的 *undo* 和 *redo* 操作。

- undo 使用一个日志记录，将该日志记录中指明的数据项设置为旧值。
- redo 使用一个日志记录，将该日志记录中指明的数据项设置为新值。

16.3.3 并发控制和恢复

如果并发控制模式允许一个事务 T_1 修改过的数据项 X 在 T_1 提交前进一步地由另一个事务 T_2 修改，那么通过将 X 重置为它的旧值(T_1 更新 X 之前的值)来撤销 T_1 的影响同时也会撤销 T_2 的影响。为避免这样的情形发生，恢复算法通常要求如果一个数据项被一个事务修改了，那么在该事务提交或中止前不允许其他事务修改该数据项。

这一要求可以通过对更新的数据项获取排他锁，并且持有该锁直至事务提交来保证；换句话说，
729 通过使用严格两阶段封锁。快照隔离性和基于有效性验证的并发控制技术在有效性验证时，在修改数据项之前，也要获取数据项上的排他锁，直至事务提交；其结果是，即使通过这些并发控制协议，上述要求也能得到满足。

后面在 16.7 节讨论，在一定的情况下可以如何放松上述要求。

在采用快照隔离性或有效性验证进行并发控制时，事务所做的数据库更新(从概念上)是延迟到事务部分提交时；延迟修改技术与这些并发控制模式自然吻合。然而，值得注意的是，快照隔离性的某些实现采用了立即修改技术，而根据需要提供了一个逻辑快照：当事务需要读被并发的事务更新的一个数据项时，就生成该数据项(已经更新)的一个拷贝，在这个拷贝上，并发事务所做的更新回滚。类似地，数据库的立即修改与两阶段封锁自然吻合，但延迟修改也可以和两阶段封锁一起使用。

16.3.4 事务提交

当一个事务的 commit 日志记录——这是该事务的最后一个日志记录——输出到稳定存储器后，我们就说这个事务**提交**(commit)了；这时所有更早的日志记录都已经输出到稳定存储器中。于是，在日志中就有足够的信息来保证，即使发生系统崩溃，事务所做的更新也可以重做。如果系统崩溃发生在日志记录 $<T_i$ commit$>$ 输出到稳定存储器之前，事务 T_i 将回滚。这样，包含 commit 日志记录的块的输出是单个原子动作，它导致一个事务的提交。⊖

对于大多数基于日志的恢复技术，包括本章描述的技术，不是在一个事务提交时必须将包含该事务修改的数据项的块输出到稳定存储器中，可以在以后的某个时间再输出。16.5.2 节进一步讨论这个问题。

16.3.5 使用日志来重做和撤销事务

我们现在提供一个关于如何使用日志来从系统崩溃中进行恢复以及在正常操作中对事务进行回滚的概览。但是，我们将故障恢复和回滚过程的细节推迟到 16.4 节。

730 考虑简化的银行系统。令 T_0 是一个事务，它将 50 美元从账户 A 转到账户 B。

```
T0 : read(A);
     A := A - 50;
     write(A);
     read(B);
     B := B + 50;
     write(B).
```

令 T_1 是一个事务，它从账户 C 中取出 100 美元。

```
T1 : read(C);
     C := C - 100;
     write(C).
```

日志包含与这两个事务相关信息的部分如图 16-2 所示。

```
<T0 start>
<T0, A, 1000, 950>
<T0, B, 2000, 2050>
<T0 commit>
<T1 start>
<T1, C, 700, 600>
<T1 commit>
```

图 16-2 系统日志中与 T_0 和 T_1 相应的部分

图 16-3 显示了一个可能的顺序，在这个顺序中，作为 T_0 和 T_1 的执行结果，对于数据库系统和日志的实际的输出都发生了。⊜

使用日志，系统可以对付任何故障，只要它不导致非易失性存储器中信息的丢失。恢复系统使用两个恢复过程，它们都利用日志来找到每个事务 T_i 更新过的数据项的集合，以及它们各自的旧值和新值。

- redo(T_i) 将事务 T_i 更新过的所有数据项的值都设置成新值。

⊖ 一个块的输出可以通过对付数据传输故障的技术而做成原子的，正如 16.2.1 节所描述的。

⊜ 请注意，如果使用延迟修改技术，就不能得到这个顺序，因为在 T_0 提交之前数据库不会修改，对于 T_1 也一样。

通过 redo 来执行更新的顺序是非常重要的；当从系统崩溃中恢复时，如果对一个特定数据项的多个更新的执行顺序不同于它们原来的执行顺序，那么该数据项的最终状态将是一个错误的值。大多数的恢复算法，包括 16.4 节描述的算法，都没有把每个事务的重做分别执行，而是对日志进行一次扫描，在扫描过程中每遇到一个 redo 日志记录就执行 redo 动作。这种方法能确保保持更新的顺序，并且效率更高，因为仅需要整体读一遍日志，而不是对每个事务读一遍日志。

日志	数据
$<T_0$ start>	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0$ commit>	
$<T_1$ start>	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1$ commit>	

图 16-3 与 T_0 和 T_1 相应的系统日志和数据库状态

- undo(T_i) 将事务 T_i 更新过的所有数据项的值都恢复成旧值。在 16.4 节中描述的恢复机制中：
  - □ undo 操作不仅将数据项恢复成它的旧值，而且作为撤销过程的一个部分，还写日志记录来记下所执行的更新。这些日志记录是特殊的 **redo-only** 日志记录，因为它们不需要包含所更新的数据项的旧值。

    与重做过程一样，执行更新的顺序是非常重要的；我们还是将细节推迟到 16.4 节中。
  - □ 当对于事务 T_i 的 undo 操作完成后，它写一个 $<T_i$ abort > 日志记录，表明撤销完成了。

    正如我们将在 16.4 节中看到的，对于每一个事务，uedo(T_i) 只执行一次，如果在正常的处理中该事务回滚，或者在系统崩溃后的恢复中既没有发现事务 T_i 的 **commit** 记录，也没有发现事务 T_i 的 **abort** 记录。其结果是，在日志中每一个事务最终或者有一条 **commit** 记录，或者有一条 **abort** 记录。

发生系统崩溃之后，系统查阅日志以确定为保证原子性需要对哪些事务进行重做，对哪些事务进行撤销。

- 如果日志包括 $<T_i$ start > 记录，但既不包括 $<T_i$ **commit** >，也不包括 $<T_i$ **abort** > 记录，则需要对事务 T_i 进行撤销。
- 如果日志包括 $<T_i$ **start** > 记录，以及 $<T_i$ **commit** > 或 $<T_i$ **abort** > 记录，需要对事务 T_i 进行重做。如果日志包括 $<T_i$ **abort** > 记录还要进行重做，看来比较奇怪。要明白这是为什么，请注意如果在日志中有 $<T_i$ **abort** > 记录，日志中也会有 undo 操作所写的那些 redo-only 日志记录。于是，这种情况下最终结果将是对 T_i 所做的修改进行撤销。这一轻微的冗余简化了恢复算法，并使得整个恢复过程变得更快。

作为一个描述，让我们回到银行的例子，有事务 T_0 和 T_1，按照 T_1 跟在 T_0 后面的顺序执行。假定在事务完成之前系统崩溃。我们将要考虑三种情况。在图 16-4 中显示了各种情况下的日志。

首先，我们假定崩溃恰好发生在事务 T_0 的

write(B)

步骤的日志记录已经写到稳定存储器之后(见图 16-4a)。当系统重新启动时，它在日志中找到记录 $<T_0$ **start** >，但是没有相应的 $<T_0$ **commit** > 或 $<T_0$ **abort** > 记录。这样，事务 T_0 必须撤销，于是执行 undo(T_0)。其结果是，(磁盘上)账户 A 和账户 B 的值分别恢复成 $1000 和 $2000。

其次，我们假定崩溃恰好发生在事务 T_1 的

write(C)

步骤的日志记录已经写到稳定存储器之后(见图 16-4b)。当系统重新启动时，需要采取两个恢复动作。因为 $<T_1$ **start** > 记录出现在日志中，但是没有 $<T_1$ **commit** > 或 $<T_1$ **abort** > 记录，所以必须执行 **undo**(T_1)。因为日志中既包括 $<T_0$ **start** > 记录，又包括 $<T_0$ **commit** > 记录，所以必须执行 **redo**(T_0)。在整个恢复过程结束时，账户 A、B、和 C 的值分别为 $950、$2050 和 $700。

最后，我们假定崩溃恰好发生在事务 T_1 的日志记录：

$<T_1$ **commit** >

已经写到稳定存储器之后(见图 16-4c)。当系统重新启动时，因为 $<T_0$ **start** > 记录和 $<T_0$ **commit** > 记录都在日志中，并且 $<T_1$ **start** > 记录和 $<T_1$ **commit** > 记录也都在日志中，所以 T_0 和 T_1 都必须重做。

731
↓
733

在系统执行 **redo**(T_0) 和 **redo**(T_1)过程后，账户 A、B 和 C 的值分别为 $950、$2050 和 $600。

a)	b)	c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

图 16-4 在三个不同时间显示的同一个日志

16.3.6 检查点

当系统故障发生时，我们必须检查日志，决定哪些事务需要重做，哪些需要撤销。原则上，我们需要搜索整个日志来确定该信息。这样做有两个主要的困难：

1. 搜索过程太耗时。

2. 根据我们的算法，大多数需要重做的事务已把其更新写入数据库中。尽管对它们重做不会造成不良后果，但会使恢复过程变得更长。

为降低这种开销，引入检查点。

下面描述一个简单的检查点，它(a)在执行检查点操作的过程中不允许执行任何更新，(b)在执行检查点的过程中将所有更新过的缓冲块都输出到磁盘。后面会讨论如何通过放松这两条要求来修改检查点和恢复过程，以提供更高的灵活性。

检查点的执行过程如下：

1. 将当前位于主存的所有日志记录输出到稳定存储器。

2. 将所有修改的缓冲块输出到磁盘。

3. 将一个日志记录 < **checkpoint** L > 输出到稳定存储器，其中 L 是执行检查点时正活跃的事务的列表。

在检查点执行过程中，不允许事务执行任何更新动作，如往缓冲块中写入或写日志记录。16.5.2 节会讨论如何强制满足这个要求。

在日志中加入 < **checkpoint** L > 记录使得系统提高恢复过程的效率。考虑在检查点前完成的事务 T_i。对于这样的事务，< T_i **commit** > 记录(或 < T_i **abort** > 记录)在日志中出现在 < **checkpoint** > 记录之前。T_i 所做的任何数据库修改都必然已在检查点前或作为检查点本身的一部分写入数据库。因此，在恢复时就不必再对 T_i 执行 **redo** 操作了。

734 在系统崩溃发生之后，系统检查日志以找到最后一条 < **checkpoint** L > 记录(这可以通过从尾端开始反向搜索日志来进行，直至遇到第一条 < **checkpoint** L > 记录)。

只需要对 L 中的事务，以及 < **checkpoint** L > 记录写到日志中之后才开始执行的事务进行 undo 或 redo 操作。让我们把这个事务集合记为 T。

- 对 T 中所有事务 T_k，若日志中既没有 < T_k **commit** > 记录，也没有 < T_k **abort** > 记录，则执行 undo(T_k)。
- 对 T 中所有事务 T_k，若日志中有 < T_k **commit** > 记录或 < T_k **abort** > 记录，则执行 redo(T_k)。

请注意，要找出事务集合 T，和确定 T 中的每个事务是否有 commit 或 abort 记录出现在日志中，我们只需要检查日志中从最后一条 checkpoint 日志记录开始的部分。

作为一个例子，考虑事务集合 $\{T_0, T_1, \cdots, T_{100}\}$。假设最近的检查点发生在事务 T_{67} 和 T_{69} 执行的过程中，而 T_{68} 和下标小于 67 的所以事务在检查点之前都已完成。于是，在恢复机制中只需要考虑事务 T_{67}，T_{69}，$\cdots$，T_{100}。其中已完成(即已提交或已终止)的需要重做；否则就是未完成的，需要撤销。

考虑检查点日志记录中的事务集合 L。对于 L 中的每一个事务 T_i，如果它没有提交，那么为了对该事务进行撤销，可能需要该事务发生在检查点日志记录之前的所有日志记录。无论如何，一旦检查点完成了，我们就不再需要位于最先出现的日志记录 < T_i **start** > (这儿的 T_i 是 L 中的任何事务)之前的所有日志记录了。当数据库系统需要回收这些记录占用的空间时，就可以清掉这些日志记录。

在检查点过程中不允许事务对缓冲块或日志进行任何更新，这一要求会引起相当的麻烦，因为这样一来在检查点进行的过程中事务处理就必须停顿下来。**模糊检查点**(fuzzy checkpoint)是这样的检查点，它即使在缓冲块正在写出时也允许事务执行更新。16.5.4 节对模糊检查点模式进行描述。然后16.8 节再描述一个检查点模式，它不仅是模糊的，而且甚至不要求在检查点时刻将所有修改过的缓冲块输出到磁盘中。

16.4 恢复算法

到目前为止，在对故障恢复的讨论中，我们确定了需要对哪些事务进行重做和需要对哪些事务进行撤销，但是我们没有给出进行这些动作的详细算法。现在我们来给出使用日志记录从事务故障中恢复的完整恢复算法，以及将最近的检查点和日志记录结合起来以从系统崩溃中进行恢复的算法。 735

本节描述的恢复算法要求未提交的事务更新过的数据项不能被任何其他事务修改，直至更新它的事务或者提交或者中止。这一限制在16.3.3 节曾经讨论过。

16.4.1 事务回滚

首先考虑正常操作时(即不是从系统崩溃中恢复时)的事务回滚。事务 Ti 的回滚如下执行。

1. 从后往前扫描日志，对于所发现的 Ti 的每一个形如 $< T_i, X_j, V_1, V_2 >$ 的日志记录：
   a. 值 V_1 被写到数据项 X_j 中，并且
   b. 往日志中写一个特殊的只读日志记录 $< T_i, X_j, V_1 >$，其中 V_1 是在本次回滚中数据项 X_j 恢复成的值。有时这种日志记录称作补偿日志记录(compensation log record)。这样的日志记录不需要 undo 信息，因为我们绝不会需要撤销这样的 undo 操作。后面会解释如何使用这些日志记录。
2. 一旦发现了 $< T_i,$ **start** $>$ 日志记录，就停止从后往前的扫描，并往日志中写一个 $< Ti,$ **abort** $>$ 日志记录。

请注意，现在事务所做的或者我们为事务做的每一个更新动作，包括将数据项恢复成其旧值的动作，都记录到日志中了。在16.4.2 节中我们将看到为什么这是个好办法。

16.4.2 系统崩溃后的恢复

崩溃发生后当数据库系统重启时，恢复动作分两阶段进行：

1. **在重做阶段**，系统通过从最后一个检查点开始正向地扫描日志来重放所有事务的更新。重放的日志记录包括在系统崩溃之前已回滚的事务的日志记录，以及在系统崩溃发生时还没有提交的事务的日志记录。这个阶段还确定在系统崩溃发生时未完成，因此必须回滚事务。这样的未完成事务或者在检查点时是活跃的，因此会出现在检查点记录的事务列表中，或者是在检查点之后开始的；而且，这样的未完成事务在日志中既没有 $< Ti,$ **abort** $>$ 记录，也没有 $< Ti,$ **commit** $>$ 记录。

   扫描日志的过程中所采取的具体步骤如下：
   a. 将要回滚的事务的列表 undo-list 初始设定为 **<checkpoint** L **>** 日志记录中的 L 列表。 736
   b. 一旦遇到形为 $< T_i, X_j, V_1, V_2 >$ 的正常日志记录或形为 $< T_i, X_j, V_2 >$ 的 redo-only 日志记录，就重做这个操作；也就是说，将值 V_2 写给数据项 X_j。
   c. 一旦发现形为 $< T_i,$ **start** $>$ 的日志记录，就把 T_i 加到 undo-list 中。
   d. 一旦发现形为 $< T_i,$ **abort** $>$ 或 $< T_i,$ **commit** $>$ 的日志记录，就把 T_i 从 undo-list 中去掉。

   在 redo 阶段的末尾，undo-list 包括在系统崩溃之前尚未完成的所有事务，即，既没有提交也没有完成回滚的那些事务。
2. **在撤销阶段**，系统回滚 undo-list 中的所有事务。它通过从尾端开始反向扫描日志来执行回滚。
   a. 一旦发现属于 undo-list 中的事务的日志记录，就执行 undo 操作，就像在一个失败事务的回滚过程中发现了该日志记录一样。

b. 当系统发现 undo-list 中事务 T_i 的 $<T_i$, **start**$>$ 日志记录，它就往日志中写一个 $<T_i$, **abort**$>$ 日志记录，并且把 T_i 从 undo-list 中去掉。

c. 一旦 undo-list 变为空表，即系统已经找到了开始时位于 undo-list 中的所有事务的 $<T_i$, **start**$>$ 日志记录，则撤销阶段结束。

当恢复过程的撤销阶段结束之后，就可以重新开始正常的事务处理了。

请注意，在重做阶段从最近的检查点记录开始重放每一个日志记录。换句话说，重启动恢复的这个阶段重复检查点之后执行并且其日志记录已经到达稳定存储器中的日志的所有更新动作。这些动作包括未完成事务的动作和为回滚失败的事务所执行的动作。这些动作按照它们原先执行的次序重复；因此，这一过程称作**重复历史**(repeating history)。虽然这看起来是浪费，但即使对失败事务也重复，这样的做法简化了恢复过程。

图 16-5 显示了在正常操作中由日志记录下的动作，以及在故障恢复中执行的动作的一个例子。在图 16-1 中显示的日志中，在系统崩溃之前，事务 T_1 已提交，事务 T_0 已完全回滚。请注意在 T_0 的回滚中如何恢复数据项 B 的值。还请注意检查点记录，它的活跃事务列表包含 T_0 和 T_1。

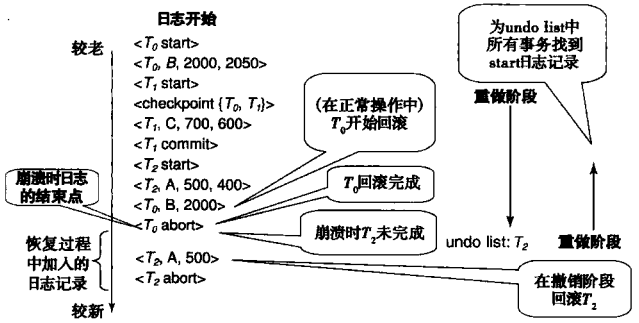

图 16-5 记录在日志中的动作和恢复中的动作的例子

当从崩溃中恢复时，在重做阶段，系统对最后一个检查点记录之后的所有操作执行 redo。在这个阶段中，undo-list 初始时包含 T_0 和 T_1；当 T_1 的 **commit** 日志记录被发现时，T_1 先被从表中去掉，而当 T_2 的 **start** 日志记录被发现时，T_2 被加到表中。当事务 T_0 的 **abort** 日志记录被发现时，T_0 被从 undo-list 中去掉，只剩 T_2 在 undo-list 中。撤销阶段从尾端开始反向扫描日志，当发现 T_2 更新 A 的日志记录时，将 A 恢复成旧值，并往日志中写一个 redo-only 日志记录。当发现 T_2 开始的日志记录时，就为 T_2 添加一条 **abort** 记录。由于 undo-list 不再包含任何事务了，因此撤销阶段终止，恢复完成。

16.5 缓冲区管理

在本节中，我们考虑几个微妙的细节，它们对实现保证数据一致性且只增加少量与数据库交互的开销的故障恢复机制非常重要。

16.5.1 日志记录缓冲

迄今为止，我们假设每个日志记录在创建时都输出到稳定存储器。该假设增加了大量系统执行的开销。其原因是：通常向稳定存储器的输出是以块为单位进行的。在大多数情况下，一个日志记录比一个块要小得多，因此，每个日志记录的输出转化成在物理上大得多的输出。另外，正如 16.2.1 节中所看到的，向稳定存储器输出一块在物理层上可能涉及几个输出操作。

将一个块输出到稳定存储器的开销非常高，因此最好是一次输出多个日志记录。为了达到这个目的，我们将日志记录写到主存的日志缓冲区中，日志记录在输出至稳定存储器以前临时保存在那儿。可以集中多个日志记录在日志缓冲区中，然后再用一次输出操作输出到稳定存储器中。稳定存储器中

的日志记录顺序必须与写入日志缓冲区的顺序完全一样。

由于使用了日志缓冲区，日志记录在输出到稳定存储器前可能有一段时间只存在于主存(易失性存储器)中。由于系统发生崩溃时这种日志记录会丢失，我们必须对恢复技术增加一些要求以保证事务的原子性：

- 在日志记录 < Ti **commit** > 输出到稳定存储器后，事务 Ti 进入提交状态。
- 在日志记录 < Ti **commit** > 输出到稳定存储器前，与事务 Ti 有关的所有日志记录必须已经输出到稳定存储器。
- 在主存中的数据块输出到数据库(非易失性存储器)前，所有与该数据块中数据有关的日志记录必须已经输出到稳定存储器。

  这一条规则称为**先写日志**(Write-Ahead Logging，WAL)规则。(严格地说，WAL 规则只要求日志中的 undo 信息已经输出到稳定存储器中，而 redo 信息允许以后再写。这一区别与 undo 信息和 redo 信息分别存储在不同的日志记录中的系统是相关的。)

这三条规则表明，在某些情况下某些日志记录必须已经输出到稳定存储器中。而提前输出日志记录不会造成任何问题。因此，当系统发现需要将一个日志记录输出到稳定存储器时，如果主存中有足够的日志记录可以填满整个日志记录块，就将其整个输出。如果没有足够的日志记录填入该块，那么就将主存中的所有日志记录填入一个部分填充的块，并输出到稳定存储器。

将缓冲的日志写到磁盘有时称为**强制日志**(log force)。

16.5.2 数据库缓冲

16.2.2 节描述了两层存储层次结构的使用。系统将数据库存储在非易失性存储器(磁盘)中，并且在需要时将数据块调入主存。由于主存通常比整个数据库小得多，因此可能在需要将块 B_2 调入内存的时候覆盖主存中的块 B_1。如果 B_1 已经修改过，那么 B_1 必须在输入 B_2 前就输出。与 10.8.1 节讨论的一样，这个存储层次结构类似于操作系统中标准的*虚拟内存*概念。

人们可能期望事务在提交时强制地将修改过的所有的块都输出到磁盘。这样的策略称作强制(force)策略。另一种方法是**非强制**(no-force)策略，即使一个事务修改了某些还没有写回到磁盘的块，也允许它提交。本章描述的所有恢复算法即使在非强制策略的情况下也能正确工作。非强制策略使得事务能更快速地提交；而且它使得在一个块输出到稳定存储器之前可以将多个更新积聚在一起，这可 739
以大大减小频繁更新的块的输出操作的数目。其结果是，大多数系统所采用的标准的方法是非强制策略。

类似地，人们可能期望一个仍然活跃的事务修改过的块都不应该写出到磁盘。这一策略称作**非窃取**(no-steal)策略。另一种方法是**窃取**(steal)策略，允许系统将修改过的块写到磁盘，即使做这些修改的事务还没有全部提交。只要遵守先写日志规则，本章研究的所有的恢复算法都能正确工作，即使采用窃取策略。而且，非窃取策略不适合于执行大量更新的事务，因为缓冲区可能被已更新过但不能逐出到磁盘的页面占满，进而使得事务不能进展下去。其结果是，大多数系统所采用的标准方法是窃取策略。

为阐明先写日志要求的必要性，我们考虑银行例子中的事务 T_0 和 T_1。假设日志的状态是

$$< T_0 \text{ start} >$$
$$< T_0, A, 1000, 950 >$$

并假设事务 T_0 发指令执行 **read**(B)。假设 B 所在的块不在主存中，并且主存已满。假设选择 A 所在的块输出到磁盘上。如果将该块输出到磁盘上后系统崩溃，数据库中账户 A、B 和 C 的值分别就是 $950、$2000 和 $700，这是不一致的数据库状态。但是，由于有 WAL 要求，因此日志记录

$$< T_0, A, 1000, 950 >$$

必须在输出 A 所在块之前输出到稳定存储器中。系统可以在恢复时使用该日志记录将数据库恢复到一致状态。

当要将块 B_1 输出到磁盘时，所有与块 B_1 中的数据相关的日志记录必须在块 B_1 输出之前先输出到稳定存储器中。重要的是，在块 B_1 正在输出时，不能有往块 B_1 中的写正在进行，因为这样的写会违反先写日志规则。我们可以通过使用特殊的封锁方法来保证没有正在进行的写：

- 在事务对一个数据项执行写操作之前，它要获得数据项所在的块的排他锁。当更新执行完后立即释放该锁。
- 740 当一个块要输出时，采取以下动作序列：
  - □ 获取该块上的排他锁，以确保没有任何事务正在对该块执行写操作。
  - □ 将日志记录输出到稳定存储器，直至与块 B_1 相关的所有日志记录都输出了。
  - □ 将块 B_1 输出到磁盘。
  - □ 一旦块输出完成，就释放锁。

缓冲块上的锁与用于事务并发控制的锁无关，按照非两阶段的方式释放这样的锁对于事务可串行性没有任何影响。这样的锁以及其他类似的短期持有的锁，通常称作**闩锁**(latch)。

缓冲块上的锁还可以用来保证在检查点进行的过程中缓冲块不更新，而且没有新的日志记录产生。可以通过要求在检查点操作开始执行之前必须获得在所有的缓冲块上的排他锁以及对日志的排他锁来实施这一限制。一旦检查点操作完成就可以释放这些锁。

数据库系统通常有一个不断地在缓冲块间循环，将修改过的缓冲块输出到磁盘的过程。在输出缓冲块时当然必须遵循上述的封锁协议。作为不断地输出修改过的缓冲块的结果，缓冲区中**脏块**(即缓冲区中修改过，但还没有输出的块)的数目极大地减小了。于是，在检查点过程中需要输出的块的数目减小了；而且，当需要从缓冲区中逐出一个块时，很可能就有不脏的块可以被逐出，使得输入马上可以进行，而不必等待输出的完成。

16.5.3 操作系统在缓冲区管理中的作用

我们可以用下面两种方法之一来管理数据库缓冲区：

1. 数据库系统保留部分主存作为缓冲区，并对它进行管理，而不是让操作系统来管理。数据库系统按照 16.5.2 节讨论的那些要求管理数据块的传送。

这种方法的缺点是限制了主存使用的灵活性。缓冲区必须足够小，使其他应用有足够的主存满足需要。但是，即使其他应用并未运行，数据库系统也不能利用所有可用内存。同样地，非数据库应用也不能使用为数据库缓冲区保留的那部分主存，即使数据库缓冲区的一些页并未使用。

2. 数据库系统在操作系统提供的虚拟内存中实现其缓冲区。因为操作系统知道系统中的所有进程的内存需求，所以最好由它决定哪个缓冲块在什么时候必须强制输出到磁盘中。但是，为保证 16.5.1
741 节讲到的先写日志要求，不应由操作系统自己写数据库缓冲页，而应由数据库系统强制输出缓冲块。数据库系统在将相关日志记录写入稳定存储器后，强制输出缓冲块至数据库中。

遗憾的是，几乎所有当前的操作系统都完全控制虚拟内存。操作系统保留磁盘空间以存储当前不在主存的那些虚拟内存页，这些空间称为**交换区**(swap space)。如果操作系统决定输出一个块 B_X，该块就输出到磁盘交换区中，并且没有办法让数据库系统控制缓冲块的输出。

因此，如果数据库缓冲区在虚拟内存中，数据库文件和虚拟内存中的缓冲区之间的数据传送必须由数据库系统管理，它可以实现前面提到的先写日志的要求。

这种方法可能导致额外的数据到磁盘的输出。如果块 B_X 由操作系统输出，则那个块不是输出到数据库中，而是输出到为操作系统的虚拟内存准备的交换区中。当数据库系统需要输出 B_X，操作系统可能需要先从它的交换区输入 B_X。因此，这里可能不止一次 B_X 输出，而可能需要两次 B_X 输出(一次是由操作系统进行，一次是由数据库系统进行)和一次 B_X 输入。

尽管两种方法都有一些缺点，但总要选择其一，除非操作系统设计成支持数据库日志的要求。

16.5.4 模糊检查点

16.3.6 节描述的检查点技术要求对数据库的所有更新在检查点进行过程中暂缓执行。若缓冲区中页的数量很大，则完成一个检查点的时间会很长，这会导致事务处理中不可接受的中断。

为避免这种中断，可以修改检查点技术，使之允许在 checkpoint 记录写入日志后，但在修改过的缓冲块写到磁盘前开始做更新。这样产生的检查点称为**模糊检查点**(fuzzy checkpoint)。

由于只有在写入 checkpoint 记录之后页面才输出到磁盘，因此系统有可能在所有页面写完之前崩溃。这样，磁盘上的检查点可能是不完全的。一种处理不完全检查点的方法是：将最后一个完全检查点记录在日志中的位置存在磁盘上固定的位置 last_checkpoint 上。系统在写入 checkpoint 记录时不更新该信息，而是在写 checkpoint 记录前，创建所有修改过的缓冲块的列表，只有在该列表中的所有缓冲块都输出到磁盘上以后，last_checkpoint 信息才会更新。 742

即使使用模糊检查点，正在输出到磁盘的缓冲块也不能更新，虽然其他缓冲块可以并发地更新。必须遵守先写日志协议，使得与一个块有关的(undo)日志记录在该块输出前已写到稳定存储器中。

16.6 非易失性存储器数据丢失的故障

到目前为止，我们只考虑了一种情况，就是故障导致了易失性存储器中的信息丢失，而非易失性存储器中的内容完整无损的情况。尽管导致非易失性存储器中内容丢失的故障极少，我们仍然需要准备好对这种类型的故障加以处理。本节只讨论磁盘存储器。这些讨论也适用于其他类型的非易失性存储器。

基本的方法是周期性地将整个数据库的内容**转储**(dump)到稳定存储器中，比如一天一次。例如，我们可以将数据库转储到一盘或多盘磁带。如果发生了一个导致物理数据库块丢失的故障，系统就可以用最近的一次转储将数据库复原到前面的一致状态。一旦复原完成，系统再利用日志将数据库系统恢复到最近的一致状态。

数据库转储的一种方法要求在转储过程中不能有事务处于活跃状态，并且必须执行一个类似于检查点的过程：

1. 将当前位于主存的所有日志记录输出到稳定存储器中。
2. 将所有缓冲块输出到磁盘中。
3. 将数据库的内容拷贝到稳定存储器中。
4. 将日志记录 < **dump** > 输出到稳定存储器中。

第 1、2 和 4 步对应于 16.3.6 节中检查点的那三个步骤。

为从非易失性存储器数据丢失中恢复，系统利用最近一次转储将数据库复原到磁盘中。然后，根据日志，重做最近一次转储后所做的所有动作。注意，这里不必执行任何 undo 操作。

在非易失性存储器部分故障的情况，例如单个块或少数块故障，只需要对那些块进行复原，并只对那些块进行重做。

数据库内容的转储也称为**归档转储**(archival dump)，因为我们可以将转储归档，以后可以用它们来查看数据库的旧状态。数据库转储和缓冲区的检查点机制很类似。 743

大多数的数据库系统还支持 **SQL 转储**(SQL dump)，它将 SQL DDL 语句和 SQL insert 语句写到文件中，这些语句可以重执行，以重新创建数据库。当将数据移植到数据库的另一个实例中或数据库软件的另一个版本中时，这样的转储非常有用，因为在另外的数据库实例或数据库软件版本中，物理位置或布局可能是不同的。

这里描述的简单转储过程开销较大，原因有以下两个。首先，整个数据库都必须拷贝到稳定存储器中，这导致大量的数据传送。其次，由于在转储过程中要中止事务处理，这样就浪费了 CPU 周期。**模糊转储**(fuzzy dump)机制对此作了改进，它允许转储过程中事务仍是活跃的。这类似于模糊检查点机制；详细描述可参见文献注解。

16.7 锁的提前释放和逻辑 undo 操作

在事务处理中使用到的任何索引，例如 B^+ 树，都可以当作一般的数据来对待，但是为了提高并发度，我们可以采用 15.10 节描述的 B^+ 树并发控制算法，允许按非两阶段的方式提前释放锁。作为提前释放锁的结果，有可能一个 B^+ 树结点的值被一个事务 T_1 更新过，插入项(V_1，R_1)，然后又被另一个

事务 T_2 更新，在同一个结点中插入项(V_2，R_2)，甚至在 T_1 完成执行之前就移动项(V_1，R_1)。[⊖] 在这一点上，我们不能通过用 T_1 执行插入前的旧值代替结点的内容来撤销事务 T_1，因为这也会撤销事务 T_2 所执行的插入；事务 T_2 可能仍然会提交(或者可能已经提交了)。在这个例子中，对插入(V_1，R_1)的影响进行撤销的唯一办法是执行一个对应的删除操作。

本节其余部分讨论如何扩充 16.4 节讲的恢复算法，来支持锁的提前释放。

16.7.1 逻辑操作

插入和删除操作是一类操作的例子，它们需要逻辑 undo 操作，因为它们提前释放锁；我们把这样的操作称作**逻辑操作**(logical operation)。这类提前的锁释放不仅对索引很重要，而且对于其他频繁存取和更新的系统数据结构上的操作也很重要；这样的例子包括追踪包含一个关系中诸记录的块、一个块中的空闲空间、一个数据库中的空闲块的数据结构。如果在这样的数据结构上执行操作后不提前释放锁，事务就趋向于是顺序执行的，影响系统性能。

744 冲突可串行化理论向操作做了扩展，基于什么操作与什么其他操作有冲突。例如，如果 B⁺树的两个插入操作插入不同的键值，则它们互相不冲突，即使它们都更新相同的索引页中互相重叠的区域。但是，如果使用相同的键值，则插入和删除操作与其他的插入和删除操作冲突，也与读操作冲突。关于这一话题，请阅读文献注解以获取更多的信息。

操作在执行时需要获得*低级别的锁*，操作完成就释放锁；然而相应的事务必须按两阶段方式保持*高级别的锁*，以防止并发事务执行冲突动作。例如，当在一个 B⁺树页面上执行插入操作时，就获得在该页面上的一个短时间的锁，从而允许该页中的项在插入过程中在页内移动；一旦页面更新完成，就释放这个短时间的锁。这样的提前释放锁使得第二次插入可以在同一页面上执行。但是，每个事务必须获得在插入或删除的键值上的锁，并按两阶段方式持有锁，以防止并发的事务在相同的键值上执行冲突的读、插入、或删除操作。

一旦释放了低级别的锁，就不能用更新的数据项的旧值来对操作进行撤销，而必须通过执行一个补偿操作来撤销；这样的操作称作**逻辑 undo 操作**(logical undo operation)。重要的是，在操作中获得的低级别的锁要足以执行后来对操作的逻辑 undo，理由在 16.7.4 节中讲。

16.7.2 逻辑 undo 日志记录

要允许操作的逻辑 undo，在执行修改索引的操作之前，事务创建一个 < T_i，O_j，operation-begin > 日志记录，其中 O_j 是该操作实例的唯一标识。[⊜] 当系统执行这个操作时，它为这个操作所做的所有更新按正常方式创建更新日志记录。于是，对于该操作所做的所有更新，通常的旧值和新值信息照常写出；在该操作完成之前事务需要回滚的情况下，需要旧值信息。当操作结束时，它写一个形如 < T_i，O_j，operation-end，U > 的 **operation-end** 日志记录，其中 U 表示 undo 信息。

例如，如果操作往 B⁺树中插入一个项，则 undo 信息 U 会指出要执行删除操作，并指明是哪一棵 B⁺树，以及从树中删除哪个项。记关于操作的这类信息的日志称作**逻辑日志**(logical logging)。与此相反，记关于旧值和新值信息的日志称作**物理日志**(physical logging)，对应的日志记录称作**物理日志记录**(physical logging record)。

请注意，在上述机制中，逻辑日志仅用于撤销，不用于重做；redo 操作全部使用物理日志记录来
745 执行。这是因为系统故障之后数据库状态可能反映一个操作的某些更新，而不反映其他操作的更新，这依赖于在故障之前哪些缓冲块已写到磁盘。像 B⁺树这样的数据结构会处于不一致的状态，逻辑 redo 和逻辑 undo 操作都不能在不一致状态的数据结构上执行。要执行逻辑 redo 或 undo，磁盘上的数据库状态必须是**操作一致**(operation consistent)的，即不应该有任何操作的部分影响。然而，正如我们将要看到的，在逻辑 undo 操作执行之前，恢复机制的重做阶段的物理 redo 处理以及使用物理日志记录的 undo

---

⊖ 请回忆，一个项包括一个键值和一个记录标识，或者在 B⁺树文件结构树叶层次的情况下包括一个键值和一个记录。

⊜ **operation-begin** 日志记录在日志中的位置可以用作唯一标识。

处理确保被一个逻辑 undo 操作存取的数据库中的部分处于一个操作一致的状态。

如果一个操作在一行上执行多次与执行一次结果相同，则称该操作是**幂等的**(idempoent)。往 B⁺ 树中插入一个项这样的操作可能不是幂等的，因此恢复算法必须保证已经执行过的操作不会再执行。与此相反，物理日志记录是幂等的，因为无论所记录的更新执行一次或多次，对应的数据项都会是同样的值。

16.7.3 有逻辑 undo 的事务回滚

当回滚事务 T_i 时，从后往前扫描日志，对事务 T_i 的日志记录做如下处理：

1. 在扫描中遇到的物理日志记录像以前描述的那样处理，除了下面马上要简短描述的需跳过的那些记录。使用由操作产生的物理日志记录对不完全的逻辑操作进行撤销。

2. 由 **operation-end** 标记的已完成的逻辑操作的回滚与此不同。一旦系统发现一个 $< T_i, O_j,$ **operation-end**, $U >$ 日志记录，就做以下特殊动作：

a. 它通过使用日志记录中的 undo 信息 U 来回滚该操作。在对操作的回滚中，它将所执行的更新记入日志，就像操作首次执行时进行的更新一样。

在操作回滚的最后，数据库系统不产生 $< T_i, O_j,$ **operation-end**, $U >$ 日志记录，而是产生 $< T_i, O_j,$ **operation-abort** > 日志记录。

b. 随着对日志的反向扫描的继续进行，系统跳过事务 T_i 的所有日志记录，直至遇到 $< T_i, O_j,$ **operation-begin** > 日志记录。

请注意，系统在回滚中将所执行的更新的物理 undo 信息记入日志，而不是使用一个只读的补偿日志记录。这是因为在逻辑 undo 进行的过程中可能发生系统崩溃，当恢复时系统必须完成逻辑 undo；要做到这一点，重启恢复将使用物理 undo 信息撤销早先的 undo 的部分影响，然后再重新执行逻辑 undo。 746

还请注意，在回滚中当遇到 **operation-end** 日志记录时跳过物理日志记录能保证，一旦操作完成，物理日志记录中的旧值就不会用来进行回滚。

3. 如果系统遇到一个 $< T_i, O_j,$ **operation-abort** > 日志记录，它就跳过前面所有的记录(包括 O_j 的 **operation-end** 记录)，直至它找到 $< T_i, O_j,$ **operation-begin** > 日志记录。

仅在要回滚的事务事先已经部分回滚的情况下才会遇到 **operation-abort** 日志记录。回忆一下，逻辑操作可能不是幂等的，因此逻辑 undo 操作不能多次执行。在前面的回滚过程中发生崩溃，事务已经部分回滚的情况下，前面的日志记录必须跳过，以防止同一操作的多次回滚。

4. 与前面一样，当遇到 $< T_i,$ start > 日志记录时，事务回滚就完成了，系统往日志中添加一个 $< T_i,$ abort > 日志记录。

如果在一个逻辑操作进行的过程中发生故障，则当事务回滚时不会找到该操作的 operation-end 日志记录。但是，对于该操作所执行的每一个更新，在日志中都有其 undo 信息，是以物理日志记录中的旧值的形式出现的。物理日志记录将用来回滚未完成的操作。

现在假设当系统崩溃发生时一个 undo 操作正在进行中，如果崩溃发生时一个事务正在回滚，就可能发生这样的事情。于是就会发现在 undo 操作中所写的物理日志记录，使用这些物理日志记录就能撤销此部分的 undo 操作自身。继续进行日志的反向扫描，会遇到原来操作的 operation-end 日志记录，于是会再次执行 undo 操作。使用物理日志记录回滚早先的 undo 操作的部分影响使数据库进入一个一致状态，从而使得逻辑 undo 操作可以再次执行。

图 16-6 显示了由两个事务产生的日志的例子，这些事务对一个数据项的值进行增加或减少。事务 T_0 在操作 O_1 完成后对数据项 C 上锁的提前释放使得事务 T_1 能够用 O_2 更新该数据项，甚至不用等到事务 T_0 完成，但这就使逻辑 undo 成为必要。逻辑 undo 需要对数据项增加或减少一个值，而不是恢复数据项的旧值。

图 16-6 中的注释说明，在操作完成之前，回滚可以执行物理 undo；在操作完成并释放低级别锁之后，回滚必须通过减少或增加一个值来执行，而不是恢复旧值。在图 16-6 中的例子中，T_0 通过往 C 中增加 100 来回滚操作 O_1；与此相反，对于数据项 B(它没有涉及锁的提前释放)，进行了物理的 undo。请注意，T_1 对 C 执行了一个更新并提交了，它的更新 O_2 在 O_1 的撤销之前执行，往 C 中增加了 200，

这个更新保持下来，尽管 O_1 撤销了。

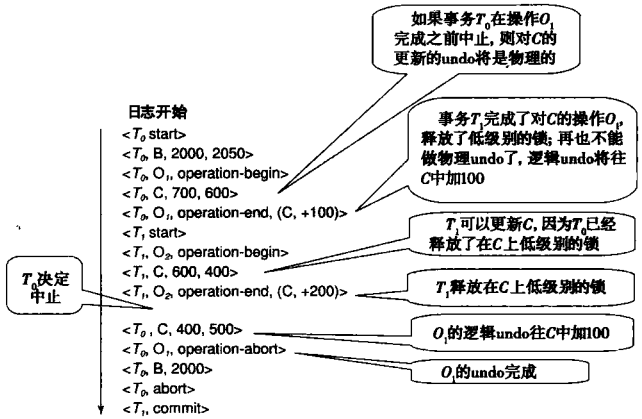

图 16-6 有逻辑 undo 操作的事务回滚

图 16-7 显示了一个使用逻辑 undo 日志从系统崩溃中恢复的例子。在这个例子中，在检查点时，事务 T_1 是活跃的，正在执行操作 O_4。在重做阶段，O_4 在检查点日志记录之后的动作重做。当系统崩溃时，T_2 正在执行操作 O_5，但是操作是不完全的。在重做阶段结束时，undo-list 包含 T_1 和 T_2。在撤销阶段，使用物理日志记录中的旧值对操作 O_5 进行撤销，将 C 置成 400；使用 redo-only 日志记录将这一操作记到日志中。然后遇到 T_2 的 **start** 记录，导致将 < T_2, **abort** > 记到日志中，并将 T_2 从 undo-list 中去掉。

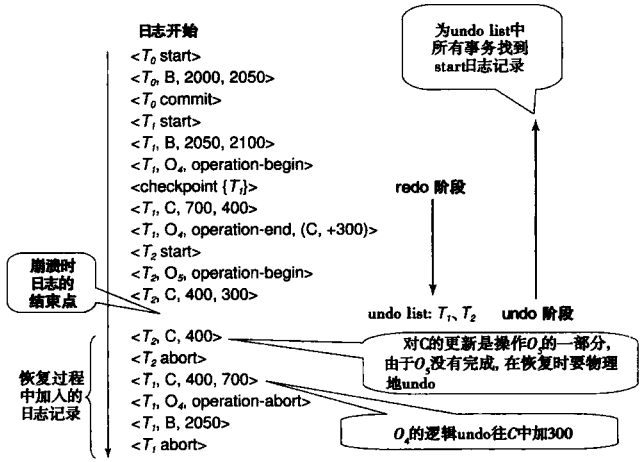

图 16-7 有逻辑 undo 操作的故障恢复动作

扫描中遇到的下一个日志记录是 O_4 的 operation-end 记录；通过给 C 的值增加 300 来执行这个操作的逻辑 undo，（这件事是物理地记了日志的），然后增加一个 O_4 的 operation-abort 日志记录。跳过作为 O_4 的一部分的物理日志记录，直至遇到 O_4 的 operation-begin 日志记录。在这个例子中，没有夹在中间的其他日志记录，但一般说来，在我们到达 operation-begin 日志记录之前可能遇到其他事务的日志记

录；当然这样的日志记录不应该跳过(除非它们是相应事务已完成操作的一部分，并且算法跳过这些记录)。在遇到 O_4 的 operation-begin 日志记录之后，又遇到 T_1 的一个物理日志记录，对它物理地回滚。最后遇到 T_1 的 **start** 日志记录；这导致往日志中添加一个 $< T_1$, **abort** > 日志记录，并且 T_1 被从 undo-list 中删掉。这时候 undo-list 为空，撤销阶段完成了。

16.7.4 逻辑 undo 中的并发问题

正如前面已经提到过的，操作中获取的低级别的锁要足以支持后来对该操作的逻辑 undo 的执行，这一点很重要；否则在正常处理中的并发操作可能导致撤销阶段中的问题。例如，假设事务 T_1 的操作 O_1 的逻辑 undo 与并发执行的事务 T_2 的操作 O_2 会在数据项层次上冲突，O_2 没完成时 O_1 完成了。还假设系统崩溃时两个事务都没提交。O_2 的物理更新日志记录可能出现在 O_1 的 operation-end 记录之前或之后，在恢复过程中 O_1 的逻辑 undo 中所做的更新可能会完全地或部分地被 O_2 的物理 undo 中写的旧值所覆盖。如果 O_1 已获得了 O_1 逻辑 undo 所需的所有低级别锁，上述问题就不会发生，因为这样的话，就不会有此类并发执行的 O_2 了。 747 / 749

如果原来的操作和它的逻辑 undo 操作都访问同一个页面(这样的操作称作物理逻辑操作，在 16.8 节讨论)，则上述封锁要求容易满足。否则，在决定需要获得什么样的低级别锁时，要考虑具体操作的细节。例如，B^+ 树上的更新操作可以获得一个在根结点上的短时间锁，以确保操作的串行执行。关于采用逻辑 undo 日志的 B^+ 树并发控制和恢复，请参看文献注解。在文献注解中还可以看到一个另外的方法，称作多级恢复，该方法放松对锁的要求。

16.8 ARIES**

ARIES 恢复方法是恢复方法的技术发展水平的最佳例子。16.4 节讨论的恢复技术和 16.7 节描述的逻辑 undo 日志技术是模仿 ARIES 设计的，但为了突出关键概念和便于理解，它已大大简化。不同的是，ARIES 使用了一些减少恢复时间，以及减少检查点开销的技术。特别地，ARIES 能够避免重做许多已重做过的日志中记录的操作，并减少日志信息量。其代价是大大增加了复杂度，但物有所值。

与我们先前给出的恢复算法的主要区别是，ARIES：

1. 使用一个**日志顺序号**(Log Sequence Number，LSN)来标识日志记录，并将 LSN 存储在数据库页中，来标识哪些操作已经在一个数据库页上实施过了。

2. 支持**物理逻辑** redo(physiological redo)操作，它是物理的，因为受影响的页从物理上标识出来，但在页内它可能是逻辑的。

例如，如果采用分槽的页结构(见 10.5.2 节)，从页中删除一条记录会导致该页中的许多记录被调整。采用物理 redo 日志，必须为该页中受调整影响的每个字节记录日志，而采用物理逻辑日志，可以记录下删除操作，其结果是日志记录会小得多。重做该删除操作将删除那条记录并根据需要调整其他记录。

3. 使用**脏页表**(dirty page table)来最大限度地减少恢复时不必要的重做。正如前面所说过的，脏页是那些在内存中已更新，而磁盘上的版本还未更新的页。

4. 使用模糊检查点机制，只记录脏页信息和相关的信息，甚至不要求将脏页写到磁盘。它不是在 750
检查点时将脏页写入磁盘，而是连续地在后台刷新脏页面。

本节剩下部分将给出 ARIES 概览，ARIES 的完全描述请参见文献注解。

16.8.1 数据结构

ARIES 中每个日志记录都有一个唯一标识该记录的日志顺序号(LSN)，该标号概念上只是一个逻辑标识，日志中记录产生得越晚，其标号数字越大。实际上，LSN 是以一种可以用来定位磁盘上的日志记录的方式产生的。典型地，ARIES 将一个日志分为多个日志文件，其中每个文件有一个文件号。当一个日志文件增长到某个限度，ARIES 将后面的日志记录添加到一个新的日志文件中；该新日志文件的文件号比前个记录一文件的大 1。这样 LSN 由一个文件号以及在该文件中的偏移量组成。

每一页也维护一个叫页日志顺序号(PageLSN)的标识，每当一个更新操作(无论物理的还是物理逻辑的)发生在某页上时，该操作将其日志记录的 LSN 存储在该页的 PageLSN 域中。在恢复的撤销阶段，

LSN 值小于或等于该页的 PageLSN 值的日志记录将不在该页上执行，因为它的动作已经反映在该页上了。稍后我们会讲到，通过结合将记录 PageLSN 作为检查点过程的一部分的机制，ARIES 甚至能够避免读其日志记录的操作已经在磁盘上反映的许多页。这样，恢复时间大量减少了。

要在物理逻辑 redo 操作时保证幂等性，PageLSN 是必不可少的，因为对一已实施物理逻辑 redo 操作的页重新实施会造成对页的错误更改。

在更新正在执行时，页不应往磁盘上写，因为在磁盘上页的部分更新状态下，物理逻辑操作不能重做。因此，ARIES 在缓冲页上加闩锁以阻止它们在正更新时被写往磁盘。只有在更新完成，以及更新的日志记录已写入日志之后，才释放该缓冲页闩锁。

每个日志记录也包含同一事务的前一日志记录的 LSN，该值存放在 PrevLSN 字段中，它使得一个事务的日志记录能够由后往前提取，而不必读整个日志。在事务回滚过程中会产生一些特殊的 redo-only 日志记录，在 ARIES 中称为补偿日志记录(Compensation Log Record，CLR)，它和我们前面讲的恢复机制中的 redo-only 日志记录的作用相同。而且 CLR 还起到该机制中 operation-abort 日志记录的作用。CLR 有一个额外的字段，叫做 UndoNextLSN，记录当事务被回滚时，日志中下一个需要 undo 的日志的 LSN。该字段和我们早先讲的恢复机制中的 operation-abort 日志记录中的操作标识作用相同，它有助于跳过那些已经回滚的日志记录。

脏页表(DirtyPageTable)包含一个在数据库缓冲区中已更新的页的列表，它为每一页保存其 PageLSN 和一个称为 RecLSN 的字段，其中 RecLSN 用于标识已经实施于该页的磁盘上的版本的日志记录。当一页插入到脏页表(当它首次在缓冲池中修改)时，RecLSN 的值被设置成日志的当前末尾。只要页被写入磁盘，就从脏页表中移除该页。

检查点日志记录(checkpoint log record)包含脏页表和活动事务的列表。检查点日志记录也为每个事务记录其 LastLSN，即该事务所写的最后一个日志记录的 LSN。磁盘上一个固定的位置记录最后一个(完整的)检查点日志记录的 LSN。

图 16-8 描述了 ARIES 中使用的一些数据结构。图 16-8 中显示的日志记录前面加上了其 LSN 作为前缀；在实际实现中，这可能并不显式地存储，而是通过在日志中的位置就能推断出来。日志记录中的数据项标识分两部分显示，例如 4894.1；第一部分标明页码，第二部分标明页中的一条记录(假定采用分槽的页结构)。请注意对日志的显示是最新的记录在顶部，磁盘上较老的日志记录在图 16-8 中较低的位置显示。

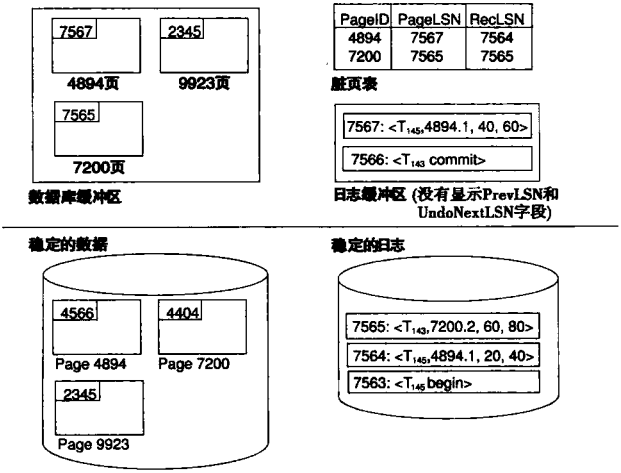

图 16-8 ARIES 中使用的数据结构

每个页面(无论在缓冲区中还是在磁盘中)有一个 PageLSN 字段。你可以验证，最后一个日志记录是对页面 4894 进行更新，它的 LSN 是 7567。通过将缓冲区中页面的 PageLSN 与稳定存储器中对应页面的 PageLSN 进行比较，你可以看到脏页表包含了关于缓冲区中自从从稳定存储器中取来后已修改过的所有页面的条目。脏页表中的 RecLSN 条目反映当页面被加到脏页表中时日志末端的 LSN，它应该大于或等于稳定存储器中该页的 PageLSN。

16.8.2 恢复算法

ARIES 从系统崩溃中恢复的过程经历三个阶段。

- **分析阶段**(analysis pass)：这一阶段决定哪些事务要撤销，哪些页在崩溃时是脏的，以及重做阶段应从哪个 LSN 开始。
- **redo 阶段**(redo pass)：这一阶段从分析阶段决定的位置开始，执行重做，重复历史，将数据库恢复到发生崩溃前的状态。
- **undo 阶段**(undo pass)：这一阶段回滚在发生崩溃时那些不完全的事务。

16.8.2.1 分析阶段

分析阶段找到最后的完整检查点日志记录，并从该记录读入脏页表。它然后将 RedoLSN 设置为脏页表中页的 RecLSN 的最小值，如果没有脏页，它就将 RedoLSN 设置为检查点日志记录的 LSN。重做阶段从 RedoLSN 开始扫描日志，该点之前的日志记录已经反映到磁盘上的数据库页中。分析阶段将要撤销的事务列表 undo-list 初始设置为检查点日志记录中的事务列表，分析阶段也为 undo-list 中的每一个事务从检查点日志记录中读取其最后一个日志记录的 LSN。

分析阶段从检查点继续正向扫描，只要找到一个不在 undo-list 中的事务的日志记录，就把该事务添加到 undo-list 中。只要找到一个事务的 end 日志记录，就把该事务从 undo-list 中删除。到分析阶段结束时，所有留在 undo-list 中的事务将在后面的撤销阶段中回滚。分析阶段也记录 undo-list 中每一个事务的最后一个记录，它在撤销阶段中有用。

一旦分析阶段发现一个在页上更新的日志记录，它还更新脏页表。如果该页不在脏页表中，分析阶段就将它添加进脏页表，并设置该页的 RecLSN 为该日志记录的 LSN。 751 753

16.8.2.2 重做阶段

重做阶段通过重演所有没有在磁盘页中反映的动作来重复历史。重做阶段从 RedoLSN 开始正向扫描日志，只要它找到一个更新日志记录，它就执行如下动作：

1. 如果该页不在脏页表中，或者该更新日志记录的 LSN 小于脏页表中该页的 RecLSN，重做阶段就跳过该日志记录。

2. 否则重做阶段就从磁盘调出该页，如果其 PageLSN 小于该日志记录的 LSN，就重做该日志记录。

注意如果上述两个测试中的任何一个是否定的，那么该日志记录的作用已经反映到页面中；否则日志记录的作用就还没有反映到页面中。由于 ARIES 允许非幂等性的物理逻辑日志记录，因此如果一个日志记录的影响已经反映到页面中，该日志记录就不应该重做。如果第一个测试是否定的，那么甚至不必从磁盘取出该页去检查它的 PageLSN。

16.8.2.3 撤销阶段和事务回滚

撤销阶段比较直截了当。它对日志进行一遍反向扫描，对 undo-list 中的所有事务进行撤销。撤销阶段仅检查 undo-list 中事务的日志记录；用分析阶段所记录的最后一个 LSN 来找到 undo-list 中每个日志的最后一个日志记录。

每当找到一个更新日志记录，就用它来执行一个 undo(无论是在正常处理过程的事务回滚中，还是在重启动的撤销阶段中)。撤销阶段产生一个包含 undo 执行动作(必须是物理逻辑的)的 CLR，并将该 CLR 的 UndoNextLSN 设置为该更新日志记录的 PrevLSN 值。

如果遇到一个 CLR，则它的 UndoNextLSN 值指明了该事务需要 undo 的下一个日志记录的 LSN；该事务的在其后面的日志记录已经回滚了。除 CLR 之外的那些日志记录，PrevLSN 字段指明该事务需要 undo 的下一个日志记录的 LSN。在撤销阶段中的每一站中，要执行的下一个日志记录是 undo-list 中所有事务的下一个日志记录 LSN 中最大的一个。

图 16-9 描述了 ARIES 基于一个样例日志所执行的恢复动作。假设磁盘上最后一个完整的检查点指针指向 LSN 为 7568 的检查点日志记录。在图 16-9 中用箭头指示日志记录中的 PrevLSN 值，用虚线箭头指示那个 LSN 为 7565 的补偿日志记录中的 UndoNextLSN 值。分析阶段从 LSN 7568 开始，当分析阶段完成时，RedoLSN 为 7564。于是，重做阶段必须从 LSN 为 7564 的日志记录开始。请注意，该 LSN 小于检查点日志记录的 LSN，因为 ARIES 检查点算法不将修改过的页面刷新到稳定存储器。在分析阶段结束时脏页表会包含检查点日志记录中的页面 4894 和 7200，以及页面 2390，它是被 LSN 为 7570 的日志记录修改过的。在本例中分析阶段结束时，需要撤销的事务列表仅包括 T_{145}。

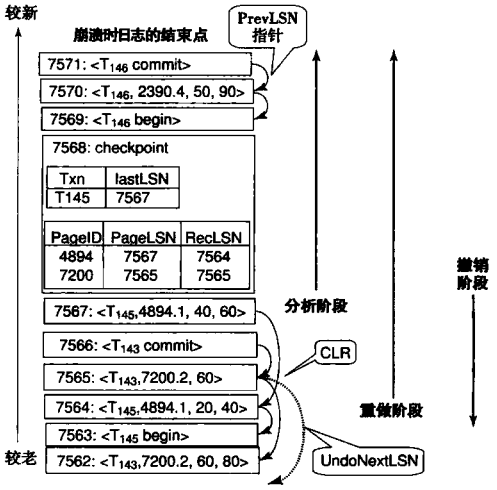

图 16-9 ARIES 中的恢复动作

上述例子的重做阶段从 LSN 7564 开始，对其页面出现在脏页表中的日志记录进行重做。撤销阶段仅需要对事务 T_{145} 进行撤销，因此从它的 LastLSN 值 7567 开始，反向扫描，直至在 LSN 7563 处遇到 < T_{145} **start** > 记录。

16.8.3 其他特性

ARIES 提供的其他关键特性包括以下几方面。

- **嵌套的顶层动作**(nested top action)：ARIES 允许对即使事务回滚也不应该撤销的那些操作记日志；例如，如果一个事务分配一个页面给一个关系，那么即使该事务回滚，此页面分配也不能撤销，因为其他事务可能已经存储记录在这个页面上了。这样的不应该回滚的操作称作嵌套的顶层动作。这样的操作可以模拟为其 undo 动作什么都不做的操作。在 ARIES 中，这样的操作通过创建一个虚拟的 CLR 来实现，设置其 UndoNextLSN，使得事务回滚跳过由此操作产生的日志记录。

754
|
755

- **恢复的独立性**(recovery independence)：有些页能够独立于其他页进行恢复，以便它们甚至能够在别的页正在恢复时使用。如果一张磁盘的某些页出错，它们无须停止其他页上的事务处理就能恢复。
- **保存点**(savepoint)：事务能够记录保存点，并能部分回滚到一个保存点。这对于死锁处理特别有用，因为事务能够回滚到某个点来允许释放必要的锁，然后从那个点重新开始。

  程序员还可以使用保存点来部分地撤销一个事务，然后继续执行；对于处理在事务执行中发现的某些类型的错误，这个方法会是很有用的。

- **细粒度的封锁**(fine-grained locking)：ARIES恢复算法可以和允许在索引中元组级封锁而不是页级封锁的索引并发控制算法一起使用。这大大地提高了并发性。
- **恢复最优化**(recovery optimization)：脏页表能用于在重做时预先提取页，而不是只在系统找到一个要应用到页的日志记录时才提取该页。重做也可能是不按顺序的：当页正在被从磁盘提取时，重做可以推迟，在页被取出来后，重做再执行。同时，其他日志记录能够继续处理。

总之，ARIES算法代表恢复算法的最新技术发展水平，它综合运用了为提高并发度、减少日志开销和缩短恢复时间所设计的各种优化技术。

16.9 远程备份系统

传统的事务处理系统是集中式或客户/服务器模式的系统。这样的系统易受自然灾难(如火灾、洪水和地震)的攻击。要求事务处理系统无论系统故障还是自然灾难都能运行的需求日益高涨。这种系统必须提供**高可用性**(high availability)；即系统不能使用的时间必须非常地短。

我们可以这样来获得高可用性，在一个站点执行事务处理，称为**主站点**(primary site)，使用一个**远程备份**(remote backup)站点，这里有主站点所有数据的备份。远程备份站点有时也叫**辅助站点**(secondary site)，随着更新在主站点上执行，远程站点必须保持与主站点同步。我们通过发送主站点的所有日志记录到远程备份站点来达到同步。远程备份站点必须物理地与主站点分离——例如，我们可以将它放在一个不同的州，这样发生在主站点的灾难不会破坏远程备份站点。图16-10显示了远程备份系统的体系结构。 756

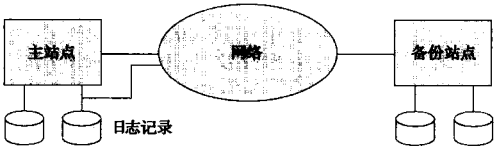

图16-10 远程备份系统的体系结构

当主站点发生故障，远程备份站点就接管处理。但它首先使用源于主站点的(也许已过时的)数据拷贝，以及收到的来自主站点的日志记录执行恢复。事实上，远程备份站点执行的恢复动作就是主站点要恢复时需执行的恢复动作。对于标准的恢复算法稍加修改，就可用于远程备份站点的恢复。一旦恢复执行完成，远程备份站点就开始处理事务。

即使主站点的数据全部丢失，系统也能恢复，因此，相对于单站点系统而言，系统的可用性大大地提高了。

在设计一个远程备份系统时有几个问题必须考虑。

- **故障检测**(detection of failure)。对于远程备份系统而言，检测什么时候主站点发生故障是很重要的。通信线路故障会使远程备份站点误以为主站点已发生故障。为避免这个问题，我们在主站点和备份站点之间维持几条具有独立故障模式的通信线路，例如，可以使用几种互相独立的网络连接，或许包括通过电话线路的调制解调器连接。这些连接可能由那些能通过电话系统进行通信的操作人员的手工干预来提供支持。
- **控制权的移交**(transfer of control)。当主站点发生故障时，备份站点就接管处理并成为新的主站点。当原来的主站点恢复后，它可以作为远程备份站点工作，抑或再次接管并作为主站点。在任意情况下，原主站点都必须收到一份在它故障期间备份站点上所执行更新的日志。

  移交控制权最简单的办法是原主站点从原备份站点收到redo日志，并将它们应用到本地以赶上更新。然后原主站点就可以作为远程备份站点工作，如果控制权必须回传，原备份站点可以假装发生故障，导致原主站点重新接管。 757
- **恢复时间**(time to recovery)。如果远程备份站点上的日志增长到很大，恢复就会花很长时间。远程备份站点可以周期性地处理它收到的redo日志，并执行一个检查点，从而日志中早期的部分

可以删除。这样，远程备份站点接管的延迟显著缩短。

采用**热备份**(hot-spare)配置可使备份站点几乎能在一瞬间接管，在该配置中，远程备份站点不断地处理到达的 redo 日志记录，在本地进行更新。一旦检测到主站点发生故障，备份站点就通过回滚未完成的事务来完成恢复；然后就做好处理新事务的准备。

- **提交时间**(time to commit)。为保证已提交事务的更新是持久的，只有在其日志记录到达备份站点之后才能宣称该事务已提交。该延迟会导致等待事务提交的时间变长，因此某些系统允许较低程度的持久性。持久性的程度可以按如下分类：
  - □ **一方保险**(one-safe)。事务的提交日志记录一写入主站点的稳定存储器，事务就提交。

    这种机制的问题是，当备份站点接管处理时，已提交事务的更新可能还没有在备份站点执行，这样，该更新好像丢失了。当主站点恢复后，丢失的更新不能直接并入，因为它可能与后来在备份站点上执行的更新相冲突。因此，可能需要人工干预来使数据库回到一致状态。
  - □ **两方强保险**(two-very-safe)。事务的提交日志记录一写入主站点和备份站点的稳定存储器，事务就提交。

    这种机制的问题是，如果主站点或备份站点其中的一个停工，事务处理就无法进行。因此，虽然丢失数据的可能性很小，但其可用性实际上比单站点的情况还低。
  - □ **两方保险**(two-safe)。如果主站点和备份站点都是活跃的，该机制与两方强保险机制相同。如果只有主站点是活跃的，事务的提交日志记录一写入主站点的稳定存储器事务，就允许它提交。

    这一机制提供了比两方强保险机制更好的可用性，同时避免了一方保险机制面临的事务丢失问题。它导致比一方保险机制较慢的提交，但总的来说利多于弊。

一些商用共享磁盘系统提供一定级别的容错，成为集中式系统和远程备份系统的折中。在这些商
758 用系统中，CPU 故障不会导致系统故障，而是由其他 CPU 接管并执行恢复。恢复动作包括回滚正在故障 CPU 上执行的事务以及恢复这些事务所持有的锁。由于数据在共享磁盘上，因此不需要日志记录的传送。但我们应该保护数据，防止磁盘故障造成的丢失，例如，使用 RAID 磁盘组织。

另一种达到高可用性的可选方法是使用分布式数据库，将数据复制到不止一个站点。此时事务更新任何一个数据项，都要求更新其所有副本。第 19 章将讨论分布式数据库，包括复制。

16.10 总结

- 计算机系统与其他机械或电子设备一样容易发生故障。造成故障的原因有很多，包括磁盘故障、电源故障和软件错误。这些情况造成了数据库系统信息的丢失。
- 除系统故障外，事务也可能因各种原因造成失败，例如破坏了完整性约束或发生死锁。
- 数据库系统的一个重要组成部分就是恢复机制，它负责检测故障以及将数据库恢复至故障发生前的某一状态。
- 计算机中的各种存储器类型有易失性存储器、非易失性存储器和稳定存储器。易失性存储器(如 RAM)中的数据在计算机发生故障时会丢失。非易失性存储器(如磁盘)中的数据在计算机发生故障时一般不丢失，只是偶尔由于某些故障如磁盘故障才会丢失。稳定存储器中的数据从不丢失。
- 必须能联机访问的稳定存储器用镜像磁盘或 RAID 的其他形式模拟，它们提供冗余数据存储。脱机或归档稳定存储器可能是数据的多个磁带备份，并存放在物理上安全的地方。
- 一旦故障发生，数据库系统的状态可能不再一致，即它不能反映数据库试图保存的现实世界的状态。为保持一致性，我们要求每个事务都必须是原子的。恢复机制的责任就是要保证原子性和持久性。
- 在基于日志的机制中，所有的更新都记入日志，并存放在稳定存储器中。当事务的最后一个日志记录，即该事务的 commit 日志记录，输出到稳定存储器时，就认为这个事务已提交。
- 日志记录包括所有更新过的数据项的旧值和新值。当系统崩溃后需要对更新进行重做时，就使用新
759 值。如果在正常操作中事务中止，回滚事务所做的更新时需要用到旧值；在事务提交之前发生系统崩溃的情况下，回滚事务所做的更新也需要用到旧值。

- 在延迟修改机制中，事务执行时所有 **write** 操作要延迟到事务提交时才执行，那时，系统在执行延迟写中会用到日志中与该事务有关的信息。在延迟修改机制中，日志记录不需要包含已更新的数据项的旧值。
- 为减少搜索日志和重做事务的开销，我们可以使用检查点技术。
- 当前的恢复算法基于重复历史的概念，在恢复的重做阶段重演（自最后一个已完成的检查点以来）正常操作中所做的所有动作。重复历史的做法将系统状态恢复到系统崩溃之前最后一个日志记录输出到稳定存储器时的系统状态。然后从这个状态开始执行一个撤销阶段，反向处理未完成事务的日志记录。
- 不完全事务的撤销写出特殊的 redo-only 日志记录和一个 **abort** 日志记录。然后，就可以认为该事务已完成，不必再对它进行撤销。
- 在事务处理所基于的存储模型中，主存储器中有一个日志缓冲区、一个数据库缓冲区和一个系统缓冲区。系统缓冲区中有系统目标码页面和事务的局部工作区域。
- 恢复机制的高效实现需要尽可能减少向数据库和稳定存储器写出的数目。日志记录在开始时可以保存在易失性的日志缓冲区中，但是当下述情况之一发生时必须写到稳定存储器中：
  - 在 $< T_i$, **commit** > 日志记录可以输出到稳定存储器之前，与事务 T_i 相关的所有日志记录必须已经输出到稳定存储器中。
  - 在主存中的一个数据块输出到（非易失性存储器中的）数据库之前，与该块中的数据相关的所有日志记录必须已经输出到稳定存储器中。
- 当前的恢复技术支持高并发性封锁技术，例如用于 B^+ 树并发控制的封锁技术。这些技术允许提前释放通过插入或删除这样的操作获得的低级别的锁，低级别的锁允许别的事务其他的这类操作可以执行。低级别的锁被释放之后，不能进行物理 undo，而需要进行逻辑 undo，例如，用删除来对插入做 undo。事务保持高级别的锁以确保并发的事务不会执行这样的动作，它可能导 760
致一个操作的逻辑 undo 是不可能的。
- 为从造成非易失性存储器中数据丢失的故障中恢复，我们必须周期性地将整个数据库的内容转储到稳定存储器中——例如每天一次。如果发生了导致物理数据库块丢失的故障，我们使用最近一次转储将数据库恢复至前面的某个一致状态。一旦完成该恢复，我们再用日志将数据库系统恢复至最当前的一致状态。
- ARIES 恢复机制是最新技术水平的机制，它支持一些提供更大并发性，削减日志开销和最小化恢复时间的特性。它也是基于重复历史的，并允许逻辑 undo 操作。该机制连续不断地清洗页，从而不需要在检查点时清洗所有页。它使用日志顺序号（LSN）来实现各种优化从而减少恢复所花时间。
- 远程备份系统提供了很高程度的可用性，允许事务处理即使在主站点遭受火灾、洪水或地震的破坏时也能继续。主站点上的数据和日志记录连续不断地备份到远程备份站点。如果主站点发生故障，远程备份站点就执行一定的恢复动作，然后接管事务处理。

术语回顾

- 恢复机制
- 故障分类
  - 事务故障
  - 逻辑错误
  - 系统错误
  - 系统崩溃
  - 数据传输故障
- 故障—停止假设
- 磁盘故障
- 存储器类型
  - 易失性存储器
  - 非易失性存储器
  - 稳定存储器
- 块
  - 物理块
  - 缓冲块
- 磁盘缓冲区
- 强制输出
- 基于日志的恢复
- 日志
- 日志记录
- 更新日志记录
- 延迟的修改
- 立即的修改
- 未提交的修改
- 检查点
- 恢复算法

- 重启动恢复
- 事务回滚
- 物理 undo
- 物理日志
- 事务回滚
- 检查点
- 重启动恢复
- 重做阶段
- 撤销阶段
- 重复历史
- 缓冲区管理
- 日志记录缓冲
- 先写日志(WAL)
- 强制日志
- 数据库缓冲
- 闩锁
- 操作系统与缓冲区管理
- 模糊检查点
- 锁的提前释放
- 逻辑操作
- 逻辑日志
- 逻辑 undo
- 非易失性存储器数据丢失
- 文档转储
- 模糊转储
- ARIES
  - □ 日志顺序号(LSN)
  - □ 页面顺序号(PageLSN)
  - □ 物理逻辑 redo
  - □ 补偿日志记录(CLR)
  - □ 脏页表
  - □ 检查点日志记录
  - □ 分析阶段
  - □ 重做阶段
  - □ 撤销阶段
- 高可用性
- 远程备份系统
  - □ 主站点
  - □ 远程备份站点
  - □ 辅助站点
- 故障检测
- 控制权移交
- 恢复时间
- 热备份配置
- 提交时间
  - □ 一方保险
  - □ 两方强保险
  - □ 两方保险

实践习题

16.1 请解释为什么 undo-list 中事务的日志记录必须由后往前进行处理，而 redo-list 中事务的日志记录则必须由前往后进行处理。

16.2 请解释检查点机制的目的。应该间隔多长时间做一次检查点？执行检查点的频率对以下各项有何影响？

- 无故障发生时的系统性能如何？
- 从系统崩溃中恢复所需的时间多长？
- 从介质(磁盘)故障中恢复所需的时间多长？

761 / 762

16.3 某些数据库系统允许系统管理员在*正常日志*(用于从系统崩溃中恢复)和*归档日志*(用于从介质(磁盘)故障中恢复)这两种日志形式间进行选择。采用 16.4 节的恢复算法，对于这两种情况的每一种，一个日志记录在什么时候可以删除？

16.4 请描述如何对 16.4 节的恢复算法进行修改，以实现保存点和执行对保存点的回滚。(保存点在 16.8.3 节描述。)

16.5 假设在数据库中使用延迟的修改技术。

a. 更新日志记录中的旧值还需要吗？为什么？

b. 如果没有将旧值存放在更新日志记录中，则显然无法对事务进行撤销。其结果是需要对恢复的重做阶段做什么样的修改？

c. 通过将更新过的数据项保存在事务的局部存储中，并且直接从数据库缓冲区中读没有更新过的数据项，可以实行延迟的修改。请给出如何高效地实现数据项的读，要保证事务看到它自己的更新。

d. 如果事务进行大量的更新，那么上述技术会有什么问题？

16.6 影子页技术需要对页表进行拷贝。假设页表表示成 B^+ 树形式。

a. 请说明如何在 B^+ 树的新拷贝和影子拷贝之间共享尽可能多的结点，假设更新仅对叶结点的项进行，并且没有插入和删除。

b. 即使做了上述优化，对于进行少量更新的事务，日志的方法仍然比影子拷贝的方法开销小得多。请解释为什么。

16.7 假设我们(错误地)修改了 16.4 节的恢复算法，对于事务回滚中执行的动作不记日志。当从系统崩溃中恢复时，于是早先已经回滚了的事务就会被包括在 undo-list 中，并且被再次回滚。请给出一个例子，说明在恢复的撤销阶段执行的动作如何会导致一个不正确的数据库状态。(提示：考虑已中止事务更新过，然后又被一个提交的事务更新的一个数据项。)

16.8 作为一个事务的结果分配给一个文件的磁盘空间不应该释放，即使该事务回滚了。请解释这是为什么，并解释 ARIES 如何保证这样的动作不回滚。 763

16.9 假设一个事务删除了一条记录，甚至在这个事务提交之前，所产生的自由空间就被分配给另一个事务插入的一条记录。

a. 如果第一个事务需要回滚的话，会产生什么问题？

b. 如果使用页级别的封锁，而不是元组级别的封锁，那么这还是个问题吗？

c. 如果支持元组级别的封锁，请解释如何通过在特殊的日志记录中记下提交后的动作，并在提交后执行它们，来解决这个问题。要确保在你给出的机制中这样的动作恰好执行一次。

16.10 请解释为什么交互式事务的恢复比批处理事务的恢复更难处理。是否有简单的方法处理该困难？（提示：考虑用于提取现金的自动取款机事务。）

16.11 有时事务在完成提交之后不得不撤销，因为它错误地执行了，比如银行出纳员的错误输入。

a. 举例说明采用通常的事务撤销机制来撤销这样一个事务会导致不一致状态。

b. 一个处理这种状况的途径是使整个数据库回到该错误事务提交前的某一状态（称为及时点（point-in-time）恢复），该点之后提交的事务回滚其影响。

请对 16.4 节的恢复算法加以修改，使用数据库转储来实现及时点恢复。

c. 及时点之后的无错误事务可以从逻辑上重新执行，但不能使用它们的日志记录重新执行，为什么？

习题

16.12 从 I/O 开销的角度解释易失性存储器、非易失性存储器和稳定存储器三类存储器的区别。

16.13 稳定存储器是不可能实现的。

a. 请解释为什么。

b. 请解释数据库系统如何对待这个问题。 764

16.14 如果与某块有关的某些日志记录没有在该块输出到磁盘前输出到稳定存储器中，请解释数据库可能会如何地变得不一致。

16.15 请概述非窃取的和强制的缓冲区管理策略的缺点。

16.16 物理逻辑 redo 日志可以显著减小日志开销，特别是对于分槽的页记录组织。请解释这是为什么。

16.17 请解释为什么逻辑 undo 日志广泛使用，而逻辑 redo 日志（除了物理逻辑 redo 日志）很少使用。

16.18 考虑图 16-5 中的日志。假设恰好在 $< T_0$ abort > 日志记录写出之前系统崩溃。请解释在系统恢复时会发生什么。

16.19 假设有一个事务已运行了很长时间，但仅做了很少的更新。

a. 如果使用 16.4 节的恢复算法，该事务对恢复时间的影响是什么？如果用 ARIES 恢复算法呢？

b. 该事务对于删除老的日志记录的影响是什么？

16.20 考虑图 16-6 中的日志。假设在恢复中发生了崩溃，发生的时间恰好在操作 O_1 的 operation abort 日志记录写出之前。请解释系统再次恢复时会发生什么。

16.21 请对于下述情况，从开销的角度，对基于日志的恢复和影子拷贝机制进行比较：数据要加入到新分配的磁盘页中（换句话说，如果事务中止的话，没有旧值需要恢复）。

16.22 在 ARIES 恢复算法中：

- 如果在分析阶段开始时，某个页面不在检查点脏页表中，我们需要将任何 redo 记录应用于该页吗？为什么？
- RecLSN 是什么，如何应用它来尽可能减少不必要的重做？

16.23 请解释系统崩溃和“灾难”的区别。

16.24 为以下各项需求，指出在远程备份系统中，持久性程度最合适的选择：

a. 必须避免数据丢失，但失去一些可用性可以接受。

b. 事务提交必须迅速完成，甚至可以冒在灾难发生时丢失一些已提交事务的风险。

c. 要求高可用性和持久性，但事务提交协议的较长运行时间是可以接受的。 765

16.25 Oracle 数据库系统使用 undo 日志记录来提供快照隔离模式下的数据库的一种快照视图。事务 T_i 所看

到的快照视图反映了当 T_i 开始时所有已提交事务的更新，以及 T_i 的更新；所有其他事务的更新对 T_i 是不可见的。

请描述一种缓冲处理机制，它能向事务提供缓冲区的页面的一个快照。说明如何使用日志来生成快照视图的细节。你可以假设操作及其 undo 动作只影响一个单独的页面。

文献注解

Gray 和 Reuter[1993]是一本优秀的教科书，提供了关于恢复的信息资料，包括有趣的实现和历史细节。Bernstein 和 Goodman[1981]是早期关于并发控制和恢复的信息资料教材。

System R 关于恢复机制的概述由 Gray 等[1981]给出。关于数据库系统的各种恢复技术的指导性和概述性文章包括 Gray[1978]、Lindsay 等[1980]和 Verhofstad[1978]。模糊检查点和模糊转储的概念是在 Lindsay 等[1980]中阐述的。Haerder 和 Reuter[1983]提供了恢复原理的全面阐述。

恢复方法的最新技术水平在 ARIES 方法中得到了最好的体现，在 Mohan 等[1992]和 Mohan[1990b]中描述。Mohan 和 Levine[1992]给出了 ARIES 的扩展 ARIES IM，使用逻辑 undo 日志来优化 B^+ 树并发控制和恢复。ARIES 和它的几个变种用在好几个数据库产品中，包括 IBM DB2 和 Microsoft SQL Server。Oracle 的恢复在 Lahiri 等[2001]中描述。

索引结构的特殊恢复技术在 Mohan 和 Levine[1992]，以及 Mohan[1993]中作了阐述，Mohan 和 Narang[1994]描述了客户—服务器架构的恢复技术，Mohan 和 Narang[1992]描述了并行数据库体系架构的恢复技术。

Weikum[1991]描述了可串行性理论的一般化版本和操作进行中的短时间低级别锁，以及与长时间高级别锁的结合。在 16.7.3 节中，我们看到这样的需求：一个操作应该获得该操作的逻辑 undo 可能需要的所有低级别锁。可以通过在执行任何逻辑 undo 操作之前首先执行所有的物理 undo 操作来放宽这一需求。在 Weikum 等[1990]中给出这一想法的一个一般化版本，称作多级恢复，它允许多个级别的逻辑操作，在恢复时逐级地进行撤销阶段。

766 灾难恢复的远程备份算法在 King 等[1991]和 Polyzois 和 Garcia-Molina[1994]中给出。