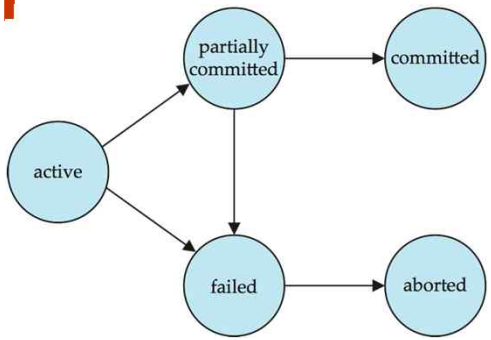
Chapter 16: 회복 시스템 (Recovery system)

Database System Concepts, 6th Ed.

Chapter 16: 회복 시스템

1. 고장의 종류
2. 저장 장치 구조
3. 회복과 원자성
4. 로그 기반 회복

1. 고장의 종류



- 트랜잭션 실패(Transaction failure)
  - 논리적 에러: 어떤 내부적인 에러 조건으로 인해 트랜잭션이 정상 종료할 수 없다.
  - 시스템 에러: 에러 조건(예, 교착상태)으로 인해 데이터 베이스 시스템은 활동 트랜잭션을 중단시켜야 한다

- 시스템 이상(system crash): 정전 또는 기타 하드웨어 및 소프트웨어 고장은 시스템 이상을 야기 한다. 비 휘발성 저장 장치의 내용은 손상되지 않는다고 가정한다

- 디스크 고장: 헤드 손상 또는 비슷한 고장이 디스크 저장 장치의 전부 또는 일부를 파괴한다.

회복 알고리즘

- 회복 알고리즘 2 가지 부분
  1. 정상적인 트랜잭션 수행 중에 고장이 발생해도 회복시킬 수 있도록 정보를 저장함.
  2. 고장 발생 후 데이터베이스가 일관성을 유지하도록 후속 조치를 수행함.

2. 저장 장치 구조

- 휘발성 저장 장치:
  - 시스템 이상시 정보가 유실된다
  - 예: 메인 메모리 , 캐쉬 메모리
- 비 휘발성 저장 장치
  - 시스템 이상시 정보가 유실되지 않는다
  - 예: 디스크, 테이프, 플래쉬 메모리, 비 휘발성 (배터리 백업) RAM
- 안정 저장 장치:
  - 어떠한 고장에도 정보가 유실되지 않는 가공의 저장 장치 유형
  - 서로 다른 비 휘발성 매체에 여러 개의 사본을 둠으로써 구현한다.

안정 저장 장치 구현

- 각 블록의 여러 사본을 별도의 디스크에 유지 한다; 화재 또는 홍수와 같은 자연 재해에 대비하여 사본을 원격 사이트에 둘 수 있다
- 데이터 전송중의 고장으로 사본들의 불일치를 초래할 수 있다
- 데이터 전송중의 고장으로부터 저장 매체를 보호하려면
  - 다음과 같이 출력 연산을 실행한다 (각 블록에 두개의 사본을 가정):
    1. 첫번째 물리 블록에 정보를 기록한다
    2. 첫번째 쓰기가 성공적으로 끝나면, 같은 정보를 두번째 물리 블록에 기록한다
    3. 두번째 기록이 성공적으로 끝난 후에만 출력은 완료된다

안정 저장 장치 구현 (계속)

- 데이터 전송중의 고장으로부터 저장 매체를 보호하려면 (계속):
  - 출력 연산중의 고장으로 인해 블록의 사본이 다를 수 있다. 고장으로부터 회복하려면:

    1. 먼저 불일치 블록을 찾는다:

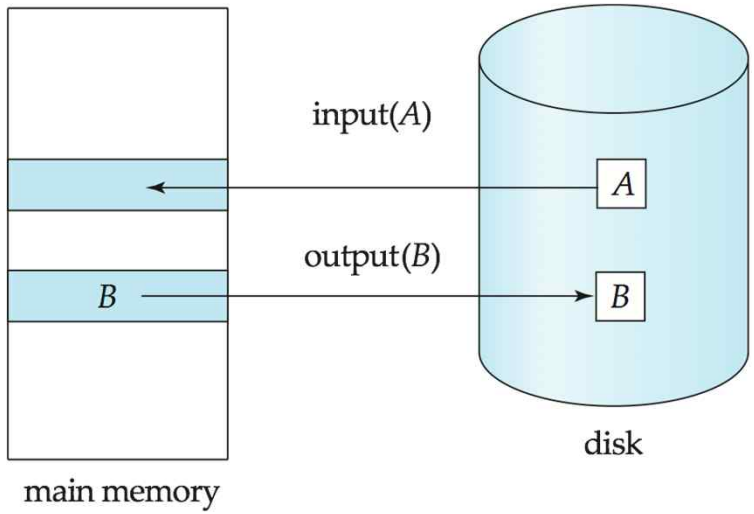
    (a) 비용이 많이 드는 해결책: 모든 디스크 블록의 쌍을 비교한다

    (b) 보다 나은 해결책: 비 휘발성 저장 장치에 진행중인 디스크 쓰기(in-progress disk writes)를 기록한다. 회복 중 이 정보를 사용해 일치하지 않는 블록을 찾아 이들 사본만을 비교한다.

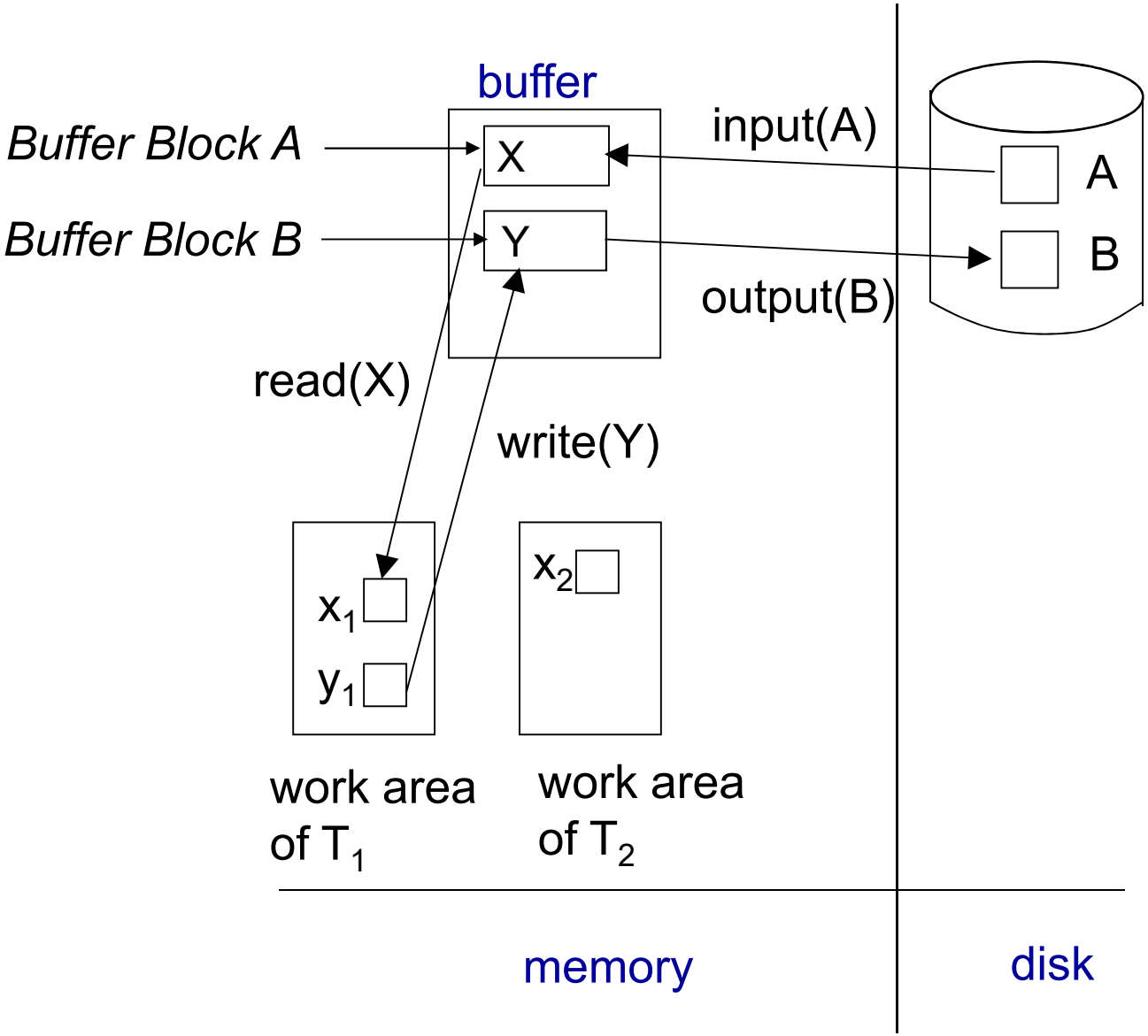
    2. 불일치 블록의 한쪽 사본에서 에러(bad checksum)가 검출되면, 다른 사본으로 그것을 대치한다. 양쪽 모두에 에러는 없는데 내용이 다르면, 첫 번째 블록의 내용으로 두 번째 블록의 값을 변경시킨다.

데이터 액세스

- 물리 블록(**Physical block**)은 디스크 상의 블록이다.
- 버퍼 블록(**Buffer block**)은 메인 메모리에 임시로 존재하는 블록이다.
- 디스크와 메인 메모리 간의 블록 이동은 다음 두 연산으로 기동된다:
  - input(A)는 물리 블록 A를 메인 메모리로 전송한다.
  - output(*B*)는 버퍼 블록 *B*를 디스크로 전송하고, 디스크 상의 적절한 물리 블록을 대치한다.
- 설명의 편의를 위해 각 데이터 항목은 하나의 블록에 들어 간다고 가정한다.

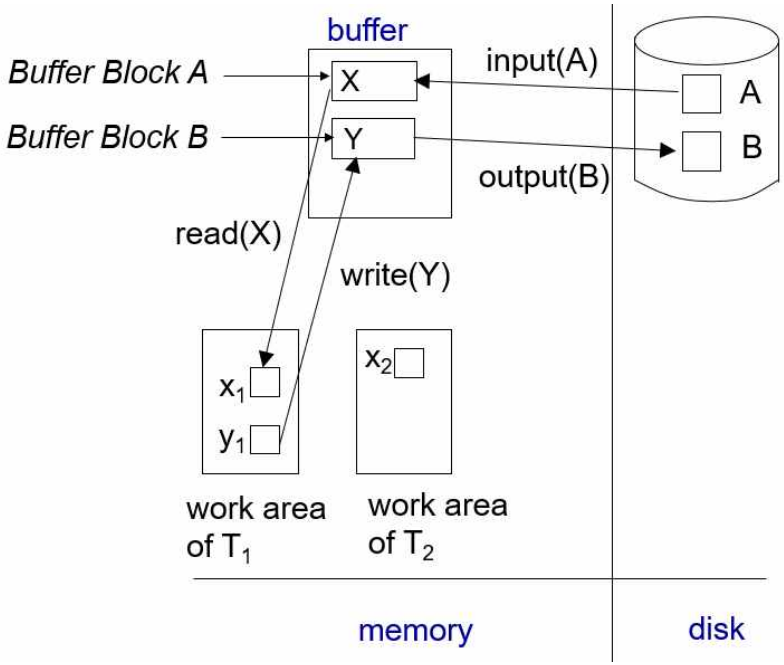


데이터 액세스의 예



데이터 액세스(계속)

- 각 트랜잭션 T_i는 사적인 작업 영역을 가지고, 액세스하고 갱신한 모든 데이터 항목의 사본을 보관한다.
  - T_i에 의해 관리되는 데이터 아이템 X 의 지역 변수를 x_i라 하자.
- 트랙잭션은 다음 연산을 사용해 시스템 버퍼 블록과 그의 작업 영역 간에 데이터 항목을 전송한다:
  - **read**(X) 는 데이터 항목 X의 값을 지역 변수 x_i에 할당한다.
  - **write**(X) 는 지역 변수 x_i의 값을 버퍼 블록내 데이터 항목 X에 할당한다.
  - **Note:** Output(B_x)는 write(X) 이후 즉시 이루어질 필요는 없다. 시스템은 적당하다고 생각할 때 output 연산을 수행할 수 있다.
- 트랜잭션은
  - 처음 X를 액세스할 때 read(X)를 수행한다; 계속되는 모든 액세스는 지역 사본에 이루어진다.
  - write(X)는 트랜잭션 commit 연산 전 아무 때나 실행될 수 있다.



3. 회복과 원자성

- 고장에도 불구하고 원자성을 보장하기 위해, 데이터베이스 자체를 수정하지 않고 수정 사항을 설명하고 있는 정보를 먼저 안정 저장 장치에 출력한다.

> 원자성(Atomicity): 트랜잭션의 작업이 부분적으로실행되거나 중단되지 않는 것을 보장하는 것을 말한다. 즉, 작업 단위를 일부분만 실행하지 않는다는 것을 의미한다. 즉 도중에 트랜잭션이 실패하면, 시스템은 그 시점까지의 갱신이 데이터베이스에 반영되지 않도록 해야 그렇지 않으면 불일치가 발생한다

- 회복 기법
  - 로그 기반 회복 기법 (**log-based recovery mechanisms**)
  - 그림자 페이징 (**shadow-paging**): 현재는 잘 쓰이지 않음.

4. 로그 기반 회복 (Log-Based Recovery)

- 로그는 안정 저장 장치에 저장된다.
- 로그는 로그 레코드로 이루어지며, 로그 레코드는 데이터베이스에 대한 갱신 행위를 기록한 것이다.
  - 트랜잭션 T_i가 시작할 때, <T_i **start**> 로그 레코드를 기록하여 등록한다.
  - T_i 가 write(X) 를 실행하기 전에, 로그 레코드 < T_i, X, V_1, V_2>를 기록한다 (V_1은 갱신하기 전 X의 값이고, V_2 는 갱신후 X의 값이다).
  - T_i 가 마지막 문장을 마치면, 로그 레코드 < T_i **commit**>를 기록한다.
- 지금은 로그 레코드가 안정 저장 장치에 직접 기록된다고 가정한다(즉, 버퍼에 기록되지 않는다).
- 로그 기반의 두 가지 알고리즘
  - 지연 데이터베이스 수정 (Deferred database modification)
  - 즉시 데이터베이스 수정 (Immediate database modification)

즉시 데이터베이스 수정 (Immediate database modification)

- 이 기법은 미완료 트랜잭션의 데이터베이스 갱신이 write가 제기되는 즉시 이루어지도록 한다 ;
- 취소가 필요하기 때문에, 갱신 로그에는 이전 값과 새로운 값 모두를 가져야 한다.
- 데이터베이스 항목이 기록(write)되기 전에 갱신 로그 레코드가 기록되어야 한다.
- 갱신된 버퍼 블록의 출력 연산(output)은 트랜잭션 완료 전 또는 후 어느 시점에서도 발생할 수 있다.
- 버퍼 블록이 디스크에 출력(output)되는 순서는 그들이 기록(write)되는 순서와는 다를 수 있다.

- 지연 데이터베이스 수정 (**deferred-modification)** 기법은 모든 write를 부분 완료 이후로 지연시킨다.
  - 회복 기법이 간단하다
  - 그러나 지역 복사본을 저장하여야 하는 오버헤드가 발생한다.

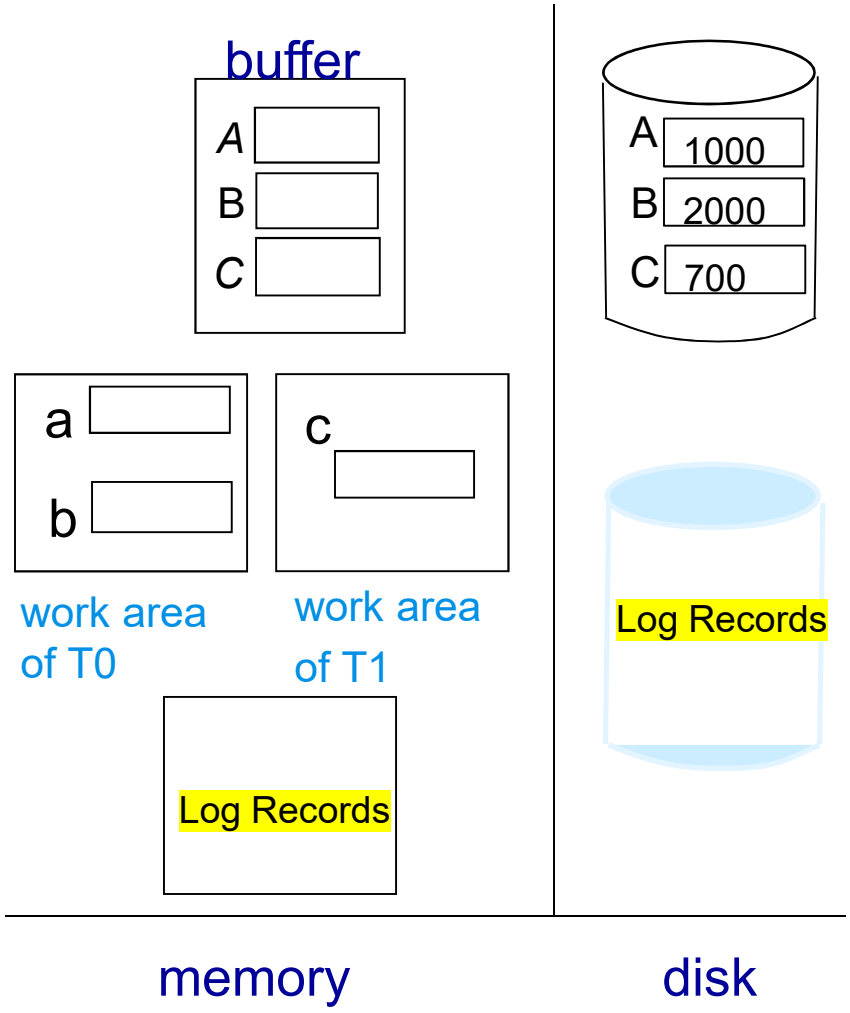
트랜잭션 완료 (Transaction Commit)

트랜잭션 *T* 의 **< *T* commit>** 로그 레코드가 안전 저장장치에 출력된 후 그 트랜잭션은 완료되었다고 말한다.

당연히 그 전의 모든 로그 레코드는 이미 출력되었다고 가정한다.

트랜잭션이 완료되어도 그 트랜잭션에 의한 Write 연산 결과는 버퍼에 그대로 남아 있어 이후에 출력될 수 있다.

		Log Records
T0		$<T_0$ start$>$
	read(A)	
	A := A – 50;	
	write(A);	$<T_0$, A, 1000, 950$>$
	read(B);	
	B := B + 50;	
	write(B).	$<T_o$, B, 2000, 2050$>$ $<T_0$ commit$>$
T1		$<T_1$ start$>$
	read(C);	
	C:=C-100;	
	Write(C).	$<T_1$, C, 700, 600$>$ $<T_1$ commit$>$



즉시 데이터베이스 수정 예제

Log	Write	Output

<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

B_B , B_C

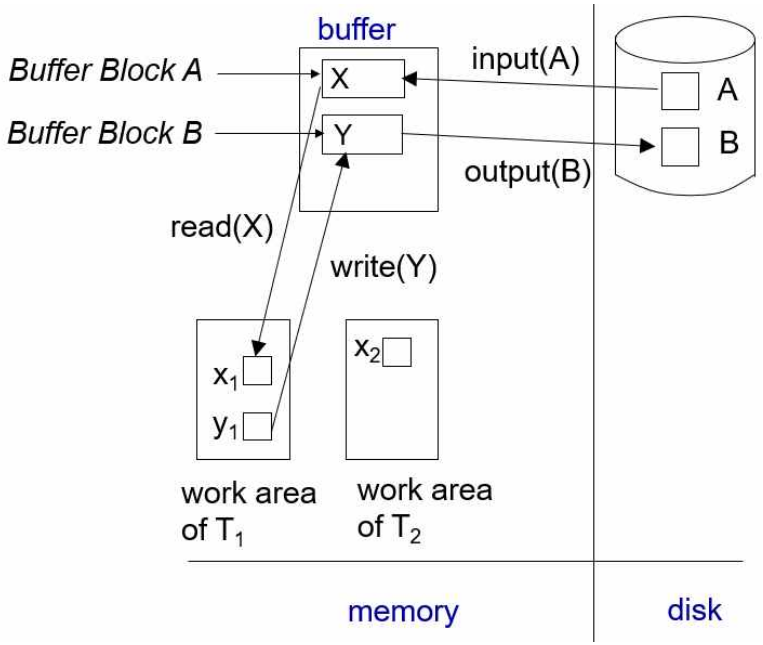
B$_C$ output before T$_1$ commits

<T_1 **commit**>

B_A

B$_A$ output after T$_0$ commits

Note: B_X denotes block containing X.



Undo와 Redo 연산

<T_i, X, V_1, V_2> 로그 레코드에 대한 **Undo** 연산: V_1 을 X에 write 한다.

<T_i, X, V_1, V_2> 로그 레코드에 대한 **Redo** 연산: V_2 을 X에 write 한다.

트랜잭션에 대한 Undo와 Redo 연산

undo(T_i)는 T_i 에 대한 마지막 로그 레코드로부터 역으로 진행하면서 T_i 가 갱신한 모든 데이터 항목의 값을 이전 값으로 복원한다.

- 데이터 항목 X 가 이전 값 V 로 복원될 때, <T_i , X, V> 의 *새로운* 로그 레코드를 기록한다.
- undo 연산이 완료되면 <T_i **abort**> 를 기록한다.

redo(T_i)는 T_i 에 대한 첫번째 로그 레코드로부터 앞으로 진행하면서 T_i 가 갱신한 모든 데이터 항목의 값을 새로운 값으로 설정한다.

- Logging을 하지 않는다.

고장 이후의 Undo, Redo 연산

고장 이후 회복할 때:

트랜잭션 T_i 에 대하여 다음 경우 undo 연산을 수행할 필요가 있다.

- 로그에 < T_i **start**> 레코드를 포함하지만
- < T_i **commit**> 혹은 <T_i **abort**> 레코드를 포함하지 않는다

트랜잭션 T_i 에 대하여 다음 경우 redo 연산을 수행할 필요가 있다.

- 로그에 < T_i **start**> 레코드를 포함하고
- < T_i **commit**> 혹은 <T_i **abort**> 레코드를 포함한다

주의 사항: 만약 트랜잭션 T_i 에 대하여 이전에 undo 연산이 수행되어 그 결과 로그 레코드 <T_i **abort**> 가 저장되어 있고, 그 후 다시 고장이 발생한 경우에는 트랜잭션 T_i 에 대하여 다시 redo 연산이 수행된다.

필요없는 redo 연산이 반복적으로 수행된다고 볼 수 있다.

그러나 전체적인 회복 절차가 매우 간단해진다.

즉시 DB 수정 회복 예제

3가지 시점에서 나타나는 로그의 예를 보이고 있다.

(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>

각 경우의 회복 행위는 다음과 같다:

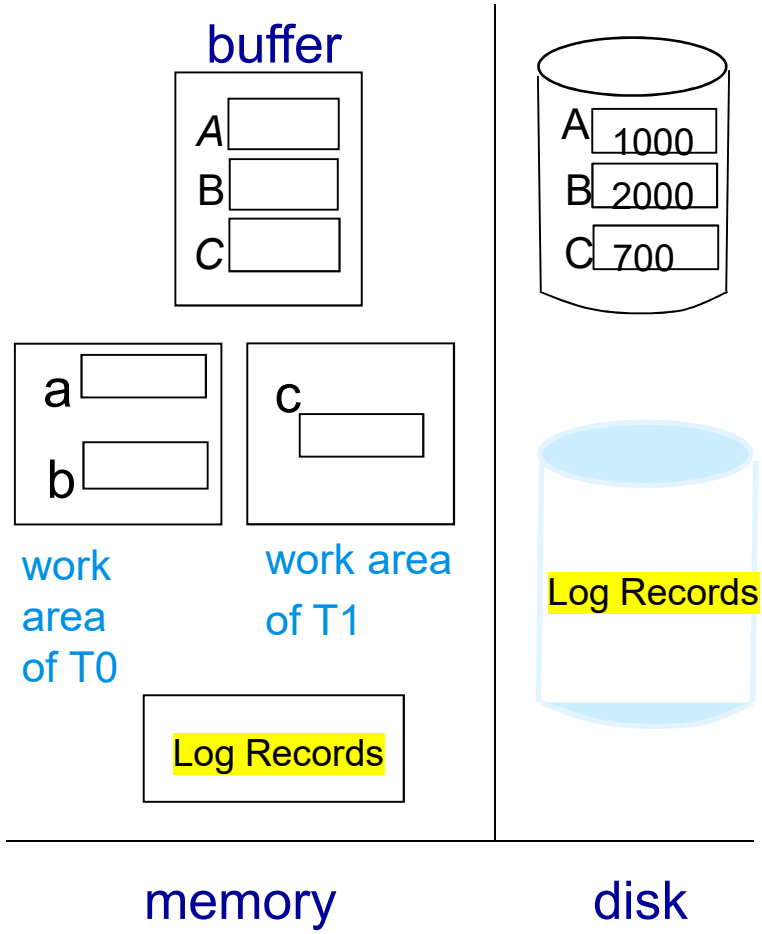
(a) undo (T_0): B는 2000 으로 A 는 1000으로 복귀되고, 로그 레코드 <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> 가 출력된다.

(b) redo (T_0), undo (T_1): *A* 와 *B* 가 950, 2050으로 설정되고, C 는 700으로 복귀되고, 로그 레코드 <T_1, C, 700>, <T_1, **abort**> 가 출력된다.

(c) redo (T_0), redo (T_1): A 와 B 가 950, 2050으로 설정되고, *C* 는 600으로 설정된다.

(a)의 경우

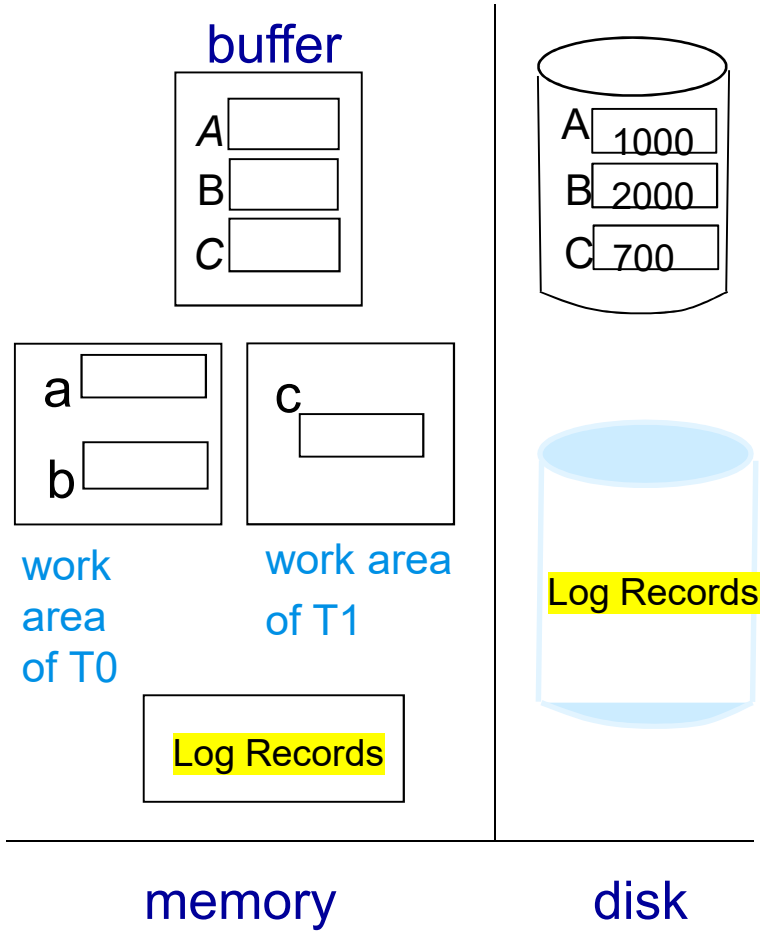
		Log Records
T0		<T_0 start>
	read(A)	
	A := A – 50;	
	write(A);	<T_0, A, 1000, 950>
	read(B);	
	B := B + 50;	
	write(B).	<T_0, B, 2000, 2050>
T1		
	read(C);	
	C:=C-100;	
	Write(C).	



회복 행위:
undo (T_0): B는 2000 으로 A 는 1000으로 복귀되고, 로그 레코드
<T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> 가 출력된다.

(b)의 경우

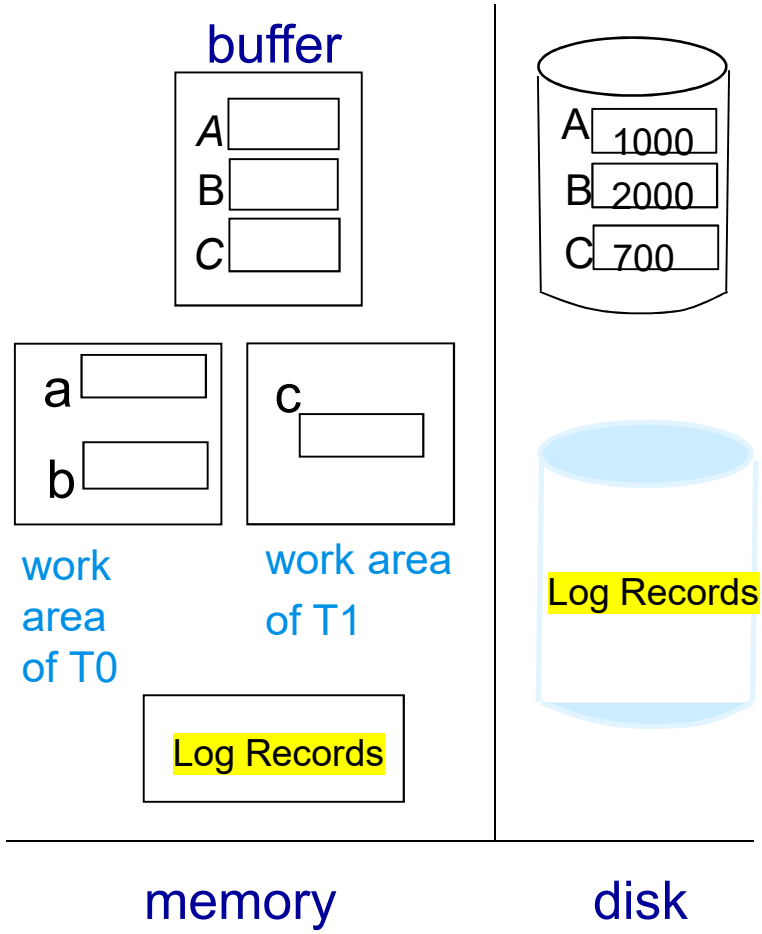
		Log Records
T0		<T_0 start>
	read(A)	
	A := A – 50;	
	write(A);	<T_0, A, 1000, 950>
	read(B);	
	B := B + 50;	
	write(B).	<T_0, B, 2000, 2050> <T_0 commit>
T1		<T_1 start>
	read(C);	
	C:=C-100;	
	Write(C).	<T_1, C, 700, 600>



회복 행위:
(b) redo (T_0), undo (T_1): A 와 B 가 950, 2050으로 설정되고, C 는 700으로 복귀되고, 로그 레코드 <T_1, C, 700>, <T_1, **abort**> 가 출력된다.

(c)의 경우

		Log Records
T0		<T_0 start>
	read(A)	
	A := A – 50;	
	write(A);	<T_0, A, 1000, 950>
	read(B);	
	B := B + 50;	
	write(B).	<T_0, B, 2000, 2050> <T_0 commit>
T1		<T_1 start>
	read(C);	
	C:=C-100;	
	Write(C).	<T_1, C, 700, 600> <T_1 commit>



회복 행위:

(c) redo (T_0), redo (T_1): A 와 B 가 950, 2050으로 설정되고, C 는 600으로 설정된다.

검사점 (Checkpoints)

앞에서 설명한 회복 절차의 문제점

1. 로그 전체를 검색하므로 시간 낭비이다.
2. 갱신을 데이터베이스에 이미 출력한 트랜잭션을 재실행하는 것은 불필요하다.

주기적으로 검사점을 수행하는 능률적 회복절차

1. 현재 메인 메모리에 존재하는 모든 로그 레코드를 안정 저장 장치에 출력한다.
2. 수정된 모든 버퍼 블록을 디스크에 출력한다.
3. 로그 레코드 <checkpoint>를 안정 저장 장치에 기록한다.

검사점 (계속)

회복시는 검사점 이전에 시작한 가장 최근의 트랜잭션 T_i와 T_i이후 시작한 트랜잭션들 만을 고려하면 된다.

로그의 끝에서부터 역방향으로 읽어 가장 최근의 <checkpoint> 레코드를 찾는다.

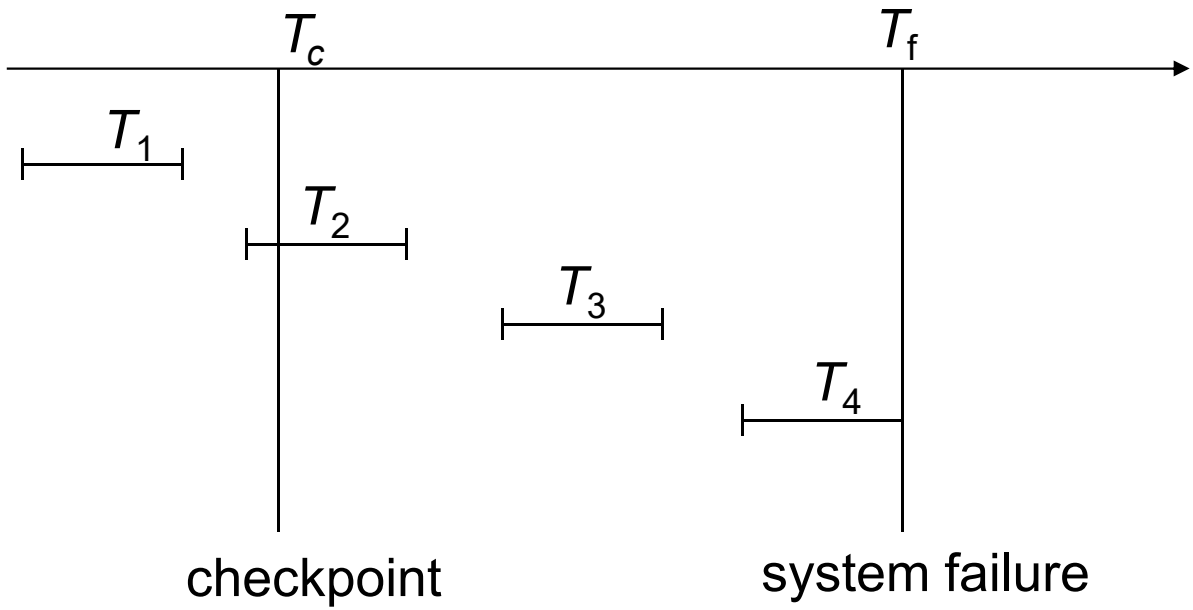
그 곳에서 < T_i **start**> 레코드를 찾을 때까지 계속 역방향으로 읽는다.

위에서 찾은 **start** 레코드 이후의 로그 부분만을 고려하면 된다. 로그의 이전 부분은 회복시 무시할 수 있고, 원하면 삭제할 수 있다.

로그를 앞으로 읽어 나가면서 < T_i **commit**>를 가진 T_i 로 시작하거나 그 이후의 모든 트랜잭션에 대해, redo(T_i)를 실행한다.

<T_i **commit**> 이 없는 모든 트랜잭션(T_i 로 시작하거나 또는 그 이후)에 대해, undo(T_i)를 실행한다.

검사점의 예제

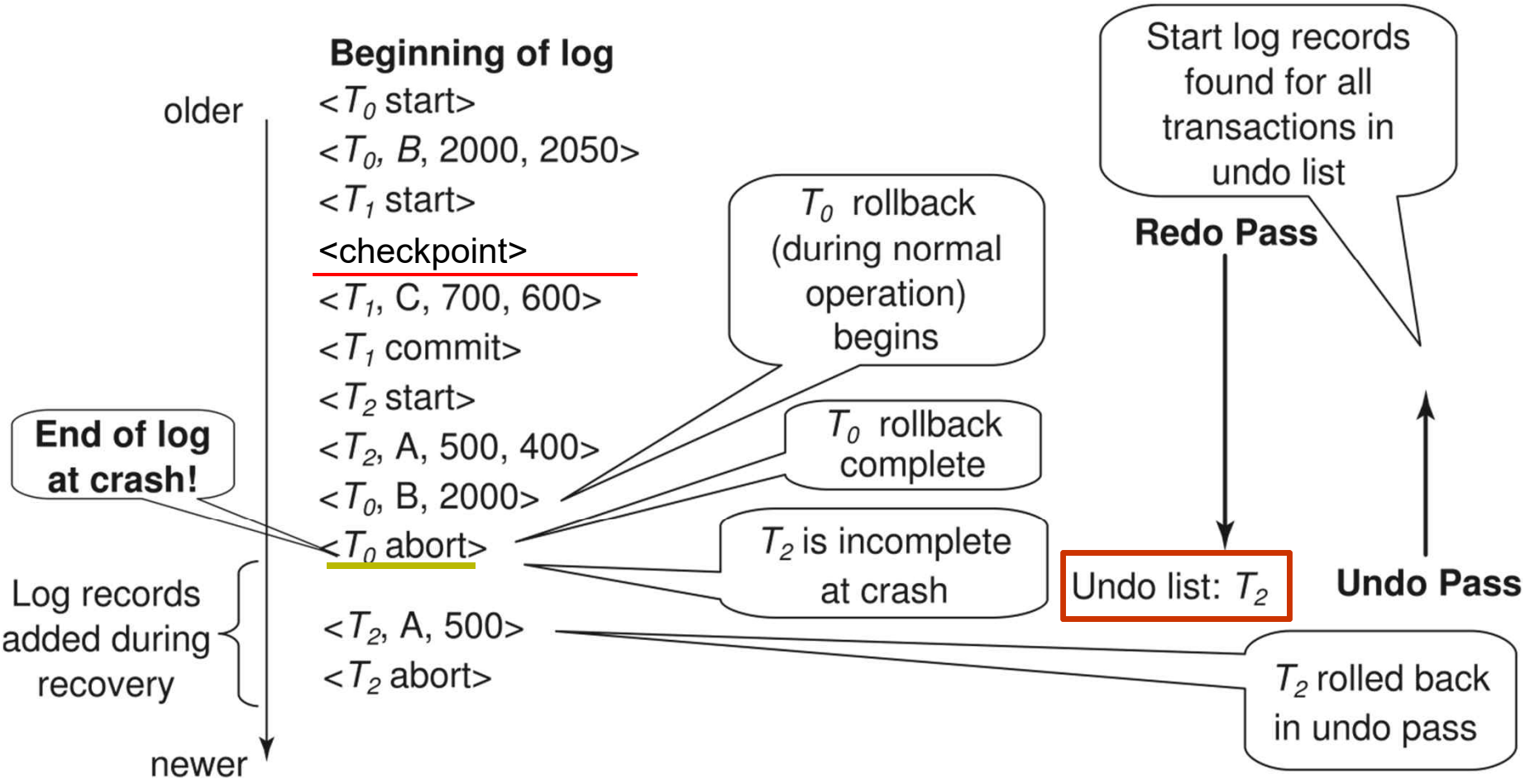


T_1은 무시할 수 있다(검사점으로 인해 갱신은 이미 디스크에 출력되었다.)

T_2 와 T_3 는 재실행된다.

T_4 는 취소된다.

Example of Recovery



redo (T_1), redo(T_0), undo (T_2): C 는 600으로, B는 2000으로 설정되며, *A* 가 500으로 복귀되고, 로그 레코드 <T_2, A, 500>, <T_2, **abort**> 가 로그 파일에 출력된다.

로그 레코드 버퍼링

로그 레코드 버퍼링 : 로그 레코드는 안정 저장 장치에 직접 출력되는 대신, 메인 메모리에서 버퍼링된다.

로그 레코드는 버퍼내의 로그 레크드 블록이 꽉 차거나 로그 강제 출력 (**log force**) 연산이 실행될 때 안정 저장 장치에 출력된다.

- 트랜잭션을 완료 시키기 위해 트랜잭션의 모든 로그 레코드(commit 레코드 포함)를 안정 저장 장치로 강제 출력시키는 것을 로그 강제 출력이라고 한다.

따라서, 하나의 output 연산을 사용해 여러 개의 로그 레코드가 출력될 수 있어 I/O 비용을 줄인다.

로그 레코드 버퍼링(계속)

로그 레코드가 버퍼링 되면 아래의 규칙을 따라야 한다:

로그 레코드는 생성된 순서대로 안정 저장 장치에 출력된다.

로그 레코드 <T_i **commit**> 가 안정 저장 장치에 출력된 이후에는 트랜잭션 T_i 가 완료 상태에 돌입한다.

메인 메모리 내 데이터 블록이 데이터 베이스에 출력되기 전, 그 블록 내 데이터에 속하는 모든 로그 레코드가 안정 저장 장치에 출력되어야 한다(이 규칙을 쓰기 전 로깅 또는 **WAL** (**write-ahead logging**)규칙이라 한다)

End of Chapter 16

Database System Concepts, 6th Ed.