데이터베이스: 강의노트 10

A. Silberschatz, H. Korth, S. Sudarshan
Database System Concepts,
Fourth Edition, McGraw-Hill, 2002.

Part V. Data Storage and Querying

15 트랜잭션

15.1 트랜잭션 개념

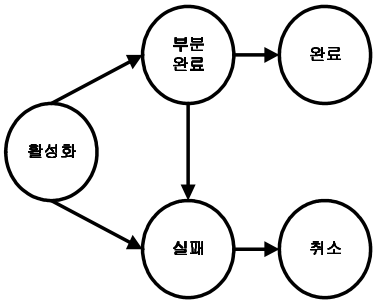
- **트랜잭션(transaction)**은 여러 데이터 항을 접근하고 갱신하는 프로그램 실행 단위이다.
- 트랜잭션은 begin transaction과 end transaction 사이에 실행되는 모든 연산으로 구성된다.
- 데이터의 무결성을 보장하기 위해 트랜잭션은 다음 네 가지 특성을 만족해야 한다.
  - **원자성(atomicity)**: 트랜잭션의 모든 연산이 실행되어 데이터베이스에 제대로 반영되던가 아니면 트랜잭션의 어떤 연산도 수행되지 않아야 한다.
  - **일관성(consistency)**: 트랜잭션을 홀로 수행하였을 때 그 결과가 데이터베이스의 일관성을 유지해야 한다.
  - **고립성(isolation)**: 여러 트랜잭션을 병행으로 수행하여도 그 결과는 이들 트랜잭션을 순차적으로 실행한 결과와 같아야 한다.
  - **지속성(duration)**: 트랜잭션이 성공적으로 완료되면 그 후에 시스템 실패가 발생하여도 트랜잭션이 변경한 내용은 계속 보존되어야 한다.
- 이 네 가지 특성을 **ACID** 특성이라 한다.
- 트랜잭션은 다음 두 가지 연산을 사용하여 데이터베이스에 접근한다고 가정한다.
  - read(X): 데이터 항 X를 데이터베이스로부터 지역 버퍼로 옮긴다.
  - write(X): 지역 버퍼에 있는 데이터 항 X를 데이터베이스에 쓴다.

  가정. 보통 데이터 항의 쓰기는 즉시 이루어지지 않는다. 하지만 여기서는 즉시 이루어진다고 가정한다.
- 예) 계좌 A에서 계좌 B로 5000원을 계좌 이체하는 트랜잭션 T_i을 생각하여 보자.

$$
\begin{array}{ll}
T_i & \mathsf{read}(A);\\
 & A := A - 5000;\\
 & \mathsf{write}(A);\\
 & \mathsf{read}(B);\\
 & B := B + 5000;\\
 & \mathsf{write}(B);
\end{array}
$$

- T_i에 대해 각 ACID 특성의 의미를 살펴보자.
  - **일관성**: 트랜잭션을 실행한 후에 A와 B의 합이 실행 전과 같아야 한다. 일관성을 유지하는 것은 응용 프로그래머의 몫이다.
  - **원자성**: 트랜잭션이 실행되는 도중에 시스템이 실패하여 중단될 수 있다. 이 경우에는 데이터베이스가 일관성 상태가 아닐 수 있다. 원자성이 보장되면 트랜잭션이 실행 도중에 중단하여도 데이터베이스가 일관성 상태를 유지한다. 이를 위해 트랜잭션이 갱신하는 데이터의 원래 값을 보관한 후에 중간에 중단되면 이 값을 이용하여 데이터베이스를 복원한다. 원자성의 보장은 데이터베이스의 **트랜잭션 관리 구성요소(transaction-management component)**가 담당한다. 참고로 트랜잭션이 실행되는 도중에 시스템이 비일관성 상태에 있을 수 있다.
  - **지속성**: 트랜잭션이 성공적으로 완료되어 그 사실이 트랜잭션을 실행한 사용자에게 통보되면 그 결과는 시스템이 차후에 실패하더라도 계속 유지되어야 한다. 즉, 계좌이체된 내용이 그 후에 손실되는 경우가 없어야 한다. 컴퓨터 시스템이 실패하면 주기억장치에 있는 내용은 손실되지만 디스크에 있는 내용은 손실되지 않는다고 가정한다. 이 경우에 지속성은 다음과 같은 방법으로 보장된다.
    - 방법 1. 트랜잭션에 의해 수행되는 갱신은 트랜잭션이 완료되기 전에 디스크에 기록되어야 한다.
    - 방법 2. 트랜잭션에 의해 수행된 갱신에 관한 정보를 보관하여 시스템이 실패한 후에 다시 데이터베이스가 가동되면 갱신을 다시 한다.

    지속성의 보장은 **회복 관리 구성요소(recovery-management component)**가 담당한다.
  - **고립성**: 트랜잭션의 일관성과 원자성이 보장되더라도 여러 트랜잭션을 병행으로 수행하면 그들의 연산들이 바람직하지 않은 방법으로 번갈아 수행될 수 있다. 이 문제를 해결하기 위해 트랜잭션들을 순차적으로 실행할 수 있다. 그러나 병행으로 수행하는 것이 성능측면에서 많은 이점이 있다. 고립성은 데이터베이스의 **병행제어 구성요소(concurrency-control component)**가 담당한다.



<그림 15.1> 트랜잭션 상태도

15.2 트랜잭션 상태

- 상태의 종류: 그림 15.1 참조
  - 활성화 상태(active): 실행 중인 상태
  - 부분 완료: 트랜잭션의 마지막 명령어가 실행된 후
  - 실패(failed): 정상적인 실행이 더 이상 불가능하다는 것을 발견한 후
  - **취소(aborted)**: 트랜잭션 또는 시스템이 실패하여 트랜잭션이 **롤백(rollback)**되어 트랜잭션이 시작하기 전 상태로 데이터베이스가 복원된 후
  - **완료(committed)**: 정상적으로 완료된 후
- 트랜잭션이 실패하면 원자성을 보장하기 위해 롤백되어야 한다. 롤백한다는 것은 취소된 트랜잭션에 의해 이루어진 변경을 이전 값으로 복구하는 것을 말한다.
- 트랜잭션이 완료되면 취소하여 그 트랜잭션이 행한 변경을 복구할 수 없다. 실제 복구할 필요가 있으면 완료 트랜잭션을 보상해주는 **보상 트랜잭션(compensating transaction)**을 실행해야 한다.
- 트랜잭션이 완료되거나 취소되면 **종료(terminated)**되었다고 한다.
- 트랜잭션의 마지막 명령어의 실행이 끝나면 트랜잭션은 부분 완료 상태가 된다. 실제 출력은 주기억장치에 일시적으로 저장되어 있을 수 있으므로 하드웨어 고장 등의 이유로 취소될 수 있다. 데이터베이스는 트랜잭션이 부분 완료되면 실패를 하더라도 트랜잭션에 의해 이루어진 변경을 다시 만들 수 있도록 충분한 정보를 디스크에 기록한다. 이 기록이 끝나면 트랜잭션은 완료 상태가 된다.
- 트랜잭션이 취소되면 시스템은 다음 중 하나를 선택한다.
  - 트랜잭션을 재시작(restart): 트랜잭션의 내부 논리 오류가 아닌 하드웨어 또는 소트트웨어 오류에 의해 취소된 경우에만 재시작할 수 있다.
  - 트랜잭션을 제거(kill): 트랜잭션의 내부 논리 오류에 의해 취소된 경우에는 응용프로그램을 다시 작성해야 문제를 해결할 수 있다. 이 경우에는 트랜잭션을 제거한다.

15.3 원자성과 지속성의 구현

- 그림자 복사본(shadow copy) 방식
  - 이 방식은 그림자 복사본이라고 하는 데이터베이스의 복사본을 유지한다.
  - 이 방식은 한 번에 하나의 트랜잭션만 실행된다고 가정하며, 데이터베이스 파일에 대한 포인터를 하나 유지한다.
  - 데이터베이스를 갱신하는 트랜잭션은 먼저 데이터베이스의 완전한 복사본을 만든 다음에 갱신은 원본이 아닌 이 복사본에 이루어진다.
  - 트랜잭션이 취소되면 복사본만 삭제된다.
  - 트랜잭션이 완료되면 새 복사본의 모든 페이지가 디스크에 기록되었는지 확인한다. 새 복사본의 모든 페이지가 디스크에 기록되면 포인터가 복사본을 가리키도록 한 다음에 기존 원본은 삭제된다.
  - 트랜잭션이 실패하는 경우: 포인터가 갱신되기 전에 트랜잭션이 실패하면 원본이 있으므로 새 복사본만 삭제하여 트랜잭션을 취소할 수 있다.
  - 시스템이 실패하는 경우: 포인터가 갱신되기 전에 시스템이 실패하면 트랜잭션이 실패한 경우와 마찬가지로 원본이 있으므로 시스템이 재가동되면 트랜잭션을 취소할 수 있다.
  - 많은 문서 편집기 소프트웨어가 이런 방식을 사용한다.
  - 문제점: 데이터베이스의 크기가 크면 데이터베이스 전체의 복사본을 만드는 것은 비효율적이다. 또한 트랜잭션의 병행 수행을 허용하지 않는다.

15.4 병행 수행

- 여러 트랜잭션의 병행 수행을 제공하는 이유
  - 처리율과 자원 사용 효율 향상
  - 대기 시간 감소
- 예) T_1은 계좌 A에서 B로 5000원을 이체하는 트랜잭션이고, T_2는 계좌 A의 10%을 계좌 B로 이체하는 트랜잭션이다.

$$
\begin{aligned}
T_1: \quad & \mathsf{read}(A);\\
& A := A - 5000;\\
& \mathsf{write}(A);\\
& \mathsf{read}(B);\\
& B := B + 5000;\\
& \mathsf{write}(B);
\end{aligned}
$$

T_2 : read(A);
temp $:= A * 0.1$;
$A := A -$ *temp*;
write(A);
read(B);
$B := B +$ *temp*;
write(B);

두 트랜잭션을 실행하기 전에 A와 B의 잔액이 각각 만원과 이만원이었다고 하자. 두 트랜잭션을 T_1, T_2 순으로 순차적으로 실행하면 결과는 A는 4500원이되고 B는 25,500원이 된다. 반대로 T_2, T_1 순으로 순차적으로 실행하면 결과는 A는 4000원이되고 B는 26,000원이 된다.

- 트랜잭션이 실행된 순서를 **스케줄(schedule)**이라 한다. 스케줄은 트랜잭션의 각 명령어가 실행된 순서를 나타내며, 스케줄은 트랜잭션의 내부 순서를 유지해야 한다.

- T_1과 T_2과 다음과 같이 병행 수행되었다고 하자.

T_1	T_2
read(A);	
$A := A - 5000$;	
write(A);	
	read(A);
	temp $:= A * 0.1$;
	$A := A -$ *temp*;
	write(A);
read(B);	
$B := B + 5000$;	
write(B);	
	read(B);
	$B := B +$ *temp*;
	write(B);

이 스케줄 대로 실행되면 T_1, T_2 순으로 순차적으로 실행하였을 때와 결과가 같다. 따라서 고립성을 만족한다. 하지만 다음과 같이 병행 수행되면 고립성을 만족하지 않는다.

T_1	T_2
read(A);	
$A := A - 5000$;	
	read(A);
	temp $:= A * 0.1$;
	$A := A -$ *temp*;
	write(A);
	read(B);
write(A);	
read(B);	
$B := B + 5000$;	
write(B);	
	$B := B +$ *temp*;
	write(B);

이와 같이 스케줄되지 않도록 보장하는 것은 병행 제어 구성요소의 몫이다.

15.4.1 직렬성

- 여러 트랜잭션을 병행 수행한 결과가 이들을 순차적으로 수행한 결과와 같아야 고립성이 만족된다.

- 고립성의 만족 여부를 고려할 때에는 트랜잭션의 read와 write 연산만 고려한다.

T_1	T_2
read(A);	
write(A);	
	read(A);
	write(A);
read(B);	
write(B);	
	read(B);
	write(B);

15.4.2 충돌 직렬성

- 스케줄 S에서 두 트랜잭션 T_i와 T_j의 명령어 I_i와 I_j가 연속해서 등장한다고 하자.
  - 이 두 명령어가 서로 다른 데이터를 접근하면 실행 순서를 바꾸어도 결과는 바뀌지 않는다.
  - 그러나 두 명령어가 같은 데이터를 접근하면 실행 순서가 결과에 영향을 미칠 수 있다.

- 두 명령어가 같은 데이터를 접근할 경우
  - 경우 1. $I_i =$ read(Q), $I_j =$ read(Q): 순서가 상관이 없음.
  - 경우 2. $I_i =$ read(Q), $I_j =$ write(Q): 순서가 서로에게 영향을 줌.
  - 경우 3. $I_i =$ write(Q), $I_j =$ read(Q): 순서가 서로에게 영향을 줌.
  - 경우 4. $I_i =$ write(Q), $I_j =$ write(Q): 순서가 서로에게 영향을 주지 않지만 다음에 오는 명령어에 영향을 줌.

- 두 명령어가 같은 데이터를 조작하고 둘 중 하나가 write이면 두 명령어는 서로 **충돌(conflict)**한다고 한다.

- 서로 충돌하지 않는 명령어는 위치를 바꾸어 실행할 수 있다.

- S'이 스케줄 S에서 충돌하지 않는 명령어들의 위치를 바꾸어 만든 스케줄이면 두 스케줄은 **충돌 일치(conflict equivalent)**한다고 한다.

- 스케줄 S가 순차 스케줄과 충돌 일치하면 **충돌 직렬성(conflict serializability)**을 만족한다고 한다.

- 예1) 다음 스케줄은 충돌 직렬성을 만족하지 않는다.

T_3	T_4
read(Q);	
	write(Q);
write(Q);	

- 예2) 다음 스케줄은 충돌 직렬성을 보장하지 않지만 실행 결과가 순차 실행한 것과 같다. 따라서 단순히 읽기와 쓰기 연산뿐만 아니라 트랜잭션에 의해 수행되는 계산까지 고려해야 한다. 그러나 계산까지 고려하는 것은 복잡한 분석이 필요하므로 구현하기가 어렵다.

T_1	T_5
read(A);	
$A := A - 5000$;	
write(A);	
	read(B);
	$B := B - 1000$;
	write(B);
read(B);	
$B := B + 5000$;	
write(B);	
	read(A);
	$A := A + 1000$;
	write(A);

15.4.3 뷰 직렬성

- 두 스케줄 S와 S'이 다음 세 가지 조건을 충족하면 **뷰 일치(view equivalent)**한다고 한다.
  - 조건 1. 각 데이터 항 Q에 대해 T_i가 스케줄 S에서 Q의 초기 값을 읽는다면 스케줄 S'에서도 T_i가 Q의 초기 값을 읽어야 한다.
  - 조건 2. 각 데이터 항 Q에 대해 T_i가 스케줄 S에서 read(Q)를 수행하고 이 읽기가 T_j의 write(Q)에 의해 쓴 값을 읽는다면 스케줄 S'에서도 T_i의 read(Q)는 S와 같은 T_j의 write(Q)에 의해 쓴 값을 읽어야 한다.
  - 조건 3. 각 데이터 항 Q에 대해 스케줄 S에서 write(Q)를 최종으로 수행한 트랜잭션이 스케줄 S'에서도 최종으로 write(Q)를 수행해야 한다.

- 스케줄 S가 순차 스케줄과 뷰 일치하면 **뷰 직렬성(view serializability)**을 만족한다고 한다.

- 예) 다음 스케줄은 뷰 직렬성을 만족하지만 충돌 직렬성은 만족하지 않는다.

T_3	T_4	T_6
read(Q);		
	write(Q);	
write(Q);		
		write(Q);

15.5 회복성

- 여러 트랜잭션이 병행으로 수행되고 있을 때, 하나의 트랜잭션이 취소되면 그것과 의존 관계에 있는 다른 모든 트랜잭션도 취소되어야 한다.

15.5.1 회복가능한 스케줄

- 예) 다음과 같은 스케줄을 고려하여 보자.

T_8	T_9
read(A);	
write(A);	
	read(A);
read(B);	

T_9는 read를 수행한 후에 바로 완료되었다고 하자. 즉, T_8보다 먼저 완료하였다고 하자. 이 때 T_8이 완료하기 직전에 취소되면 T_9는 T_8과 의존 관계가 있으므로 같이 취소되어야 한다. 그러나 이미 완료하였으므로 취소할 수가 없다.

- 이 처럼 회복할 수 없는 스케줄을 회복불가능 스케줄이라 한다.

- **회복가능 스케줄(recoverable schedule)**이란 스케줄에 있는 모든 트랜잭션 쌍 T_i와 T_j에 대해 T_j가 T_i에 의해 기록된 데이터를 읽으면 T_i의 완료 연산(commit operation)이 T_j보다 앞서 있는 스케줄을 말한다.

15.6 비연쇄 스케줄

- 예) 다음과 같은 스케줄을 고려하여 보자.

T_{10}	T_{11}	T_{12}
read(A);		
read(B);		
write(A);		
	read(A);	
	write(A);	
		read(A);

이 때 T_{10}이 취소되면 T_{11}과 T_{12} 모두 취소되어야 한다. 이런 현상을 **연쇄 롤백(cascading rollback)**이라 한다.

- 연쇄 롤백은 바람직하지 않은 현상으로 발생하지 않아야 한다.

- 연쇄 롤백이 일어나지 않는 스케줄을 **빈연쇄 스케줄(cascadeless schedule)**이라 한다.

15.7 고립성의 구현

- 스케줄이 충돌 직렬성 또는 뷰 직렬성을 만족하고 비연쇄 스케줄이면 가장 바람직하다.

- 병행으로 수행되는 트랜잭션들이 직렬성을 만족하고 빈연쇄 스케줄이 되도록 보장해주는 다양한 병행 제어 기법이 있다.

16 병행 제어

16.1 잠금 기반 프로토콜

16.1.1 잠금

- 잠금 모드
  - 공유 잠금: T_i가 Q에 대한 공유 잠금을 획득하면 Q에 대해 읽기만 가능하다.
  - 배타 잠금: T_i가 Q에 대한 배타 잠금을 획득하면 Q에 대해 읽기와 쓰기가 모두 가능하다.
- 공유 잠금은 S로 나타내고, 배타 잠금은 X로 나타낸다.
- 트랜잭션은 각 데이터를 접근하기 전에 그 데이터에 대해 수행할 연산에 따라 적절한 잠금을 먼저 획득해야 한다. 이를 병행 제어 관리자에게 잠금을 요청하면 관리자는 허용하거나 거부한다.
- 호환성 함수

	S	X
S	true	false
X	false	false

  즉, 공유 잠금이 된 데이터에 대해 공유 잠금 요청은 허용되지만 다른 요청은 허용되지 않는다. 허용되지 않는 요청은 호환되지 않는 잠금을 획득하고 있는 다른 모든 트랜잭션이 잠금을 해제할 때까지 대기하게 된다.
- 트랜잭션은 잠금을 획득한 후에 언제든지 그것을 해제할 수 있다. 그러나 그 데이터를 접근하는 동안에 잠금을 계속 유지해야 한다.
- 예1) 다음과 같은 두 개의 트랜잭션이 있다고 하자.

T_1 :
lock-X(B);
read(B);
$B := B - 5000$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 5000$;
write(A);
unlock(A);

T_2 :
lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display$(A + B)$;

이 두 트랜잭션의 다음과 같이 스케줄될 수 있다.

T_1	T_2	CCM
lock-X(B);		
		grant(B, T_1);
read(B);		
$B := B - 5000$;		
write(B);		
unlock(B);		
	lock-S(A);	
		grant(A, T_2);
	read(A);	
	unlock(A);	
	lock-S(B);	
		grant(B, T_2);
	read(B);	
	unlock(B);	
	display$(A + B)$;	
lock-X(A);		
		grant(A, T_1);
read(A);		
$A := A + 5000$;		
write(A);		
unlock(A);		

그러나 이렇게 스케줄되면 T_1과 T_2가 순차적으로 실행된 결과와 다른 결과를 T_2가 출력하게 된다. 이것은 T_1이 너무 일찍 B에 대한 잠금을 해제하였기 때문이다.

- 예2) 다음과 같이 잠금 해제를 트랜잭션의 끝으로 지연시킨 두 개의 트랜잭션이 있다고 하자.

T_1 :
lock-X(A);
read(A);
$A := A - 5000$;
write(A);
lock-X(B);
read(B);
$B := B + 5000$;
write(B);
unlock(A);
unlock(B);

T_2 :
lock-S(A);
read(A);
lock-S(B);
read(B);
display$(A + B)$;
unlock(A);
unlock(B);

이 두 트랜잭션의 예1처럼 스케줄될 수 없다. 이렇게 하면 항상 결과는 직렬성을 유지한다. 그러

나 만약 다음과 같이 스케줄되었다고 하자.

T_1	T_2	CCM
lock-X(B);		
		grant(B, T_1);
read(B);		
$B := B - 5000$;		
write(B);		
unlock(B);		
	lock-S(A);	
		grant(A, T_2);
	read(A);	
	lock-S(B);	
lock-X(A);		

이 경우 T_2의 B에 대한 공유 잠금은 허용될 수 없으며, 반대로 T_1의 A에 대한 배타 잠금도 허용될 수 없다. 즉, 교착상태가 발생하였다. 교착상태가 발생하면 하나의 트랜잭션을 롤백시켜야 한다.

- 교착상태가 발생하는 일관성이 위배되는 것보다는 선호된다.
- 잠금 기법의 또 다른 문제는 굶주림이다. 하나의 트랜잭션이 공유 잠금을 획득하고 있을 때 다른 트랜잭션이 배타 잠금을 요청하면 허용할 수 없어 대기하게 된다. 이 사이에 다른 트랜잭션들이 계속 공유 잠금을 요청하면 무한 대기할 수 있게 된다.