트랜잭션

- 일반 응용 프로그램의 구성
  - 명령어들의 집합
- 논리적인 작업 단위를 구성하는 연산들의 집합
- 실행 중 멈추거나 중단되지 않는 최소 작업 단위
- 데이터베이스 응용 프로그램은 트랜잭션의 집합

트랜잭션의 필요성

1. 데이터베이스 개발자는 작업 단위들을 트랜잭션으로 적절히 정의해야 함
2. 즉, 트랜잭션을 정의하는 것은 전적으로 개발자의 의무
3. 정의된 트랜잭션들에 대해서 위와 같은 문제가 발생하지 못하게 방지하는 것은 DBMS의 몫

ACID 특성

- 원자성 Atomicity
  - 트랜잭션은 중간에 멈출 수 없다. all or nothing
- 일관성 Consistency
  - 트랜잭션 실행 전후 데이터베이스 내용이 일관되어야 한다.
- 고립성 Isolation
  - 트랜잭션이 실행하는 과정에서 갱신한 데이터는 트랜잭션이 완료될 때까지 다른 트랜잭션이 참조할 수 없다.
- 지속성 Durability
  - 트랜잭션이 성공적으로 완료되면 그 트랜잭션이 갱신한 데이터베이스의 내용은 영구적으로 저장되어야 한다.

트랜잭션의 상태

- 동작 active : 트랜잭션이 시작되고 연산들이 정상적으로 실행 중인 상태

- 부분 완료 : 트랜잭션에 정의된 모든 연산의 실행이 끝난 상태 → 지속성은 확보되지 않은 상태
- 완료 committed : 트랜잭션이 성공적으로 종료된 상태 → 지속성까지 보장된 상태
- 실패 failed : 트랜잭션이 완료되지 못하고 더 이상 실행되지 못하는 상태
- 중단 aborted : 트랜잭션이 실패 한 후 시행되기 이전으로 복귀된 상태 → 롤백(처음으로 다시 돌아가라)

동시성 제어

- 사용자 수에 따른 DBMS 의 구분
- 다중 사용자 DBMS에서 필요한 기법
  - 하나의 트랜잭션이 완료되지 않은 상태에서 다른 트랜잭션 실행 가능
- 트랜잭션 간의 간섭이 발생하여 일관성이 깨지지 않도록 제어하는 기법

트랜잭션의 연산

- read(x) → select 연산
  - 이름이 x인 데이터베이스 항목을 트랜잭션의 지역변수 x로 읽어온다.
- write(x) →update 연산
  - 지역변수 x에 저장된 값을 데이터베이스 항목 x에 저장한다.
- x는 테이블, 레코드, 필드 등 데이터베이스를 구성하는 임의의 구성요소가 될 수 있음
- 단, write(x) 연산 수행할 때, 그 결과가 디스크에 즉시 저장될 수도 있고 아닐수도 있음
  - 대부분 주기억장치에 buffer를 유지

동시성 제어가 필요한 이유

트랜잭션 명령들 간의 끼어들기가 가능함

- 스케줄 → 운영체제 권한
  - 끼어들기 방식에 의해 실행되는 순서
  - 사용자는 어떠한 스케줄로 트랜잭션들이 실행되는지 미리 예측하기가 거의 불가능
- 끼어들기 방식은 서로간의 간섭에 의해서 잘못된 데이터를 생성할 수 있음

끼어들기로 인한 문제

- **갱신 분실**

    - T1에 의해 수행된 갱신(update)이 T2에 의해 사라짐 → 하나마나
- **연쇄 복귀**
    - T3가 롤백하면 아무문제 없는T4도 롤백해야함 → 완료되지 않은 T3
    - 더 심각한 문제 : T4가 이미 커밋으로 완료 시, 복귀 불가능 → 지속성 위배
- **불일치 분석**
    - 끼어들기로 인해 트랜잭션의 일관성이 유지되지 못하는 상황

⇒ 문제점 해결

- 순차적으로 실행 (가장 단순한 방법)
    - 그치만 말도 안됨. 현대 운영체제 프로세서의 장점을 다 버림
- 끼어들기를 최대한 허용하면서 직렬 스케줄과 동일한 결과를 갖도록 실행 순서를 제어
    - 동시성 제어의 목표 ⇒ 직렬 가능한 스케줄

직렬스케줄

- 각 트랜잭션의 연산들이 끼어들기 방식으로 실행되지 않고 순차적으로 실행되는 스케줄

  → CPU의 낭비가 심함 → 성능 저하

직렬 가능한 스케줄

: 직렬 스케줄과 실행 결과가 동일한 스케줄

- 직렬 가능한 스케줄인지 판단하는 방법
    - 연산들의 순서를 바꿔본다. → 전체적인 실행 결과에 영향을 미치지 않도록
    - 이때 주어진 스케줄이 직렬 스케줄로 변환되면 직렬 가능한 스케줄임
- 실행 순서를 바꿀 수 있는 경우
    - C1과 C2가 서로 다른 데이터 항목에 대한 read, write 연산 일 경우 → 교환 가능
    - C1과 C2가 같은 데이터 항목에 대한 read 연산일 경우
        - 모두 read 연산이면 교환 가능
        - 하나라도 write 연산이면 교환 불가능

직렬 가능한 스케줄이 되도록 하는 방법

- **잠금**
    - 트랜잭션의 실행 순서를 강제로 제어

- 하나의 트랜잭션이 수행하는 동안 특정 데이터 항목에 대해 다른 트랜잭션이 동시에 접근하지 못하도록 방지
- 잠금이 걸린 데이터는 잠금을 실행한 트랜잭션만 독점적으로 접근
  - 공유잠금
    - 트랜잭션 T가 데이터 항목x에 대해 S-lock을 걸면 T는 read(x)연산은 가능하지만 write(x)연산은 불가
    - 하나의 데이터 항목에 대해 여러 개의 공유 잠금 가능
  - 배타잠금
    - 트랜잭션 T가 데이터항목 x에 대해 X-lock을 걸면 T는read(x), write(x) 연산 모두 가능
    - 하나의 데이터 항목에 대해서는 하나의 배타잠금만 가능
      - 하나의 데이터에 대해 동시에 여러 개의 배타잠금은 불가능
- 잠금 설정 규칙
  - read(x) 실행을 위해서는 S-lock(x), X-lock(x) 중 하나 실행
  - write(x) 실행을 위해서는 X-lock(x) 실행
  - 연산 종류 후에는 unlock(x) 실행
  - S-lock(x), X-lock(x) 실행 후에만 unlock(x) 실행 가능
- 잠금의 한계
  - 단순한 잠금 연산만으로 직렬 가능한 스케줄을 보장하는 것은 아님
  - 교착상태 발생 가능
- 2단계 잠금 규약 → 2PL
  - 확장단계 : 트랜잭션이 lock 연산은 수행할 수 있으나 unlock연산은 수행할 수 없는 단계
  - 축소단계 : 트랜잭션이 unlock 연산은 수행할 수 있으나 lock연산은 수행할 수 없는 단계
- 2PL과 직렬 가능한 스케줄
  - 2PL을 준수하면 항상 직렬 가능한 스케줄이 됨
  - 모든 직렬 가능한 스케줄들이 2PL을 준수하는 것은 아님
  - 2PL은 직렬 가능한 스케줄의 충분조건

    - **2PL의 한계**
        - 교착상태 방지 불가능
            - 교착상태 회피 / 교착상태 탐지
        - 연쇄복귀문제가 발생할 수 있음
            - 해결 방안 : 엄격한 2PL 적용

              → 모든 X-lock에 대한 unlock연산은 트랜잭션 완료 후 실행
    - 잠금 단위
        - 잠금의 대상이 되는 데이터 객체의 크기
        - 잠금 단위 크면,
            - 동시성 수준이 낮아짐
            - 트랜잭션 제어가 간단해짐
        - 잠금 단위 작으면,
            - 동시성 수준 높아짐
            - 트랜잭션 제어가 복잡해짐
        - 잠금단위를 여러 단계로 정해놓고 필요에 따라 혼용하여 사용 ⇒ 블록 단위로 락을 걸어
- **타임스탬프**
    - 최대한 병행 수행을 보장
    - 직렬 가능 한 스케줄에 위배될 가능성이 있으면 실행 취소
- 대부분의 DBMS에서는 잠금 기법 사용

장애와 복구

- 트랜잭션 장애 : 트랜잭션 내의 논리적 오류나 잘못된 입력 데이터, 또는 시스템 내의 자원 부족으로 인한 트랜잭션 중단
- 시스템 장애 : 정전이나 하드웨어 결함으로 시스템 작동이 중단
- 미디어 장애 : 디스크와 같은 저장 장치의 일부 또는 전체가 손상
- 복구

  : 데이터베이스를 장애 발생 이전의 일관된 상태로 복원

  복구의 기본 원리는 데이터 중복 (redundancy)

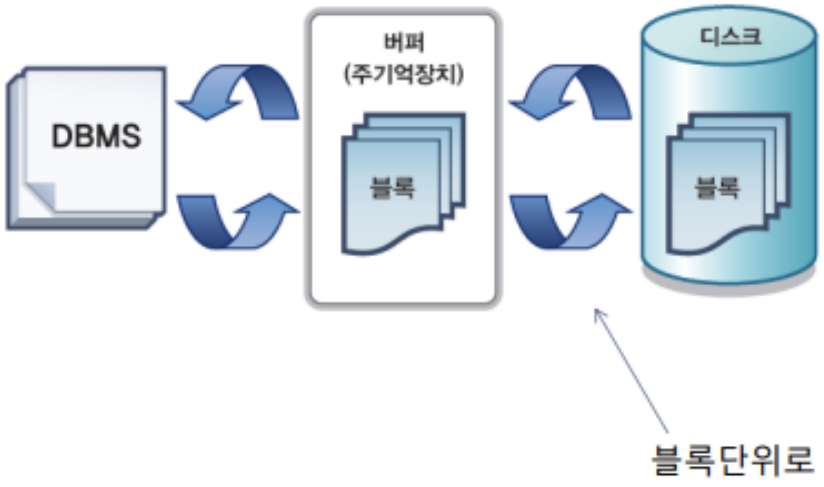
- 백업 파일 : DVD나 자기 테이프와 같은 저장장치를 이용
- Mirroring : 하나의 데이터베이스를 서로 다른 디스크에 복제 → 하나가 문제가 발생하면 복제 디스크로 계속 운용 가능 (비용이나 효과 면에서 안 좋음)

트랜잭션 장애나 시스템 장애의 대체

그림 9-1의 계좌이체에 대한 예

- 트랜잭션이 완료되지 못하고 장애발생 시, 재가동 되었을 때 갱신된 내용이 복구되어야 함
- 복구가 가능하려면?
  - 데이터베이스에 어떠한 순서로 갱신이 이루어졌는가를 나타내는 정보가 기록되어 있어야 함
  - 로그 라고 함
  - 로그는 비휘발성 저장장치에 보관
- **로그(log)**
  - 트랜잭션이나 시스템 장애에 대비한 데이터 베이스 갱신 이력 (history) 저장

데이터베이스 입출력 방식에 대한 가정



버퍼를 이용한 디스크 입출력

로그의 구성요소

로그 : 데이터베이스의 모든 갱신에 대한 기록 저장

→ 장애가 발생하면 이 기록을 보고 이전상태로 복귀가 가능

로그의 구성요소

- 트랜잭션의 시작 <T start>
- 트랜잭션의 완료 <T commit>
- 트랜잭션의 중단 <T abort>
- 데이터 항목에 대한 갱신 → write연산일 때만 사용
  - <T, x→데이터 항목 이름 , v1 → 원래 값 , v2→ 변경될 값>

로그 우선 기록 규약

write-ahead logging protocol

- 로그 레코드를 기록할 때 트랜잭션이 갱신한 데이터 항목을 기록하기 전에 로그 레코드를 먼저 기록해야 함
  - 데이터베이스에 이미 기록된 변경 내용을 취소하려면, 로그 레코드에 그 기록이 남아있어야 하기 때문
- 이미 완료된 트랜잭션에 대해서 데이터베이스의 갱신 내용을 디스크에 반영할 때도 유용하게 사용됨
  - 완료된 트랜잭션의 변경이 주기억장치의 버퍼에만 기록되어 있다면?
    - 재가동 후 로그를 이용하여 실제 데이터베이스에 갱신 기록을 반영할 수 있음

로그를 이용한 복구 기법

복구에 사용되는 기본 연산

- UNDO: 갱신된 값을 이전 상태로 되돌림
  - x값을 이전으로 돌려놔라 (<T,x,100,200>을 예시로 들자면,)
- REDO: 갱신을 재실행 함
  - x값을 200으로 재실행해라

로그를 이용한 복구 기법

- **지연 갱신을 기반으로 한 복구 기법**
  - 트랜잭션의 수행이 성공적으로 끝난 후 갱신 내용을 디스크에 반영
  - 완료 전(커밋 전)에는 주기억장치의 버퍼에만 기록을 저장

TIP: 트랜잭션이 완료되었다는 것은 디스크에 저장된 로그에 <T commit>이 기록되었다는 것을 의미

  - 이후의 시스템 장애에 대해서도 이 기록만으로 트랜잭션이 완료되었다는 것을 알 수 있음
  - 장애 후에 로그에 기록된 레코드를 이용하여 갱신 내용들을 복구 가능
  - 즉, 이 레코드가 로그에 기록이 되지 않으면, 트랜잭션 T가 완료된 것인지 아니면 실행 도중에 중단된 것인지 알 수 없음
    - UNDO 연산 불필요, REDO 연산 필요
    - 데이터 갱신에 의한 로그 레코드 형식 <T, x, v2> (v2:갱신 이후의 값)
  - 지연 갱신을 기반으로 한 복구 알고리즘은 UNDO연산이 필요 없으므로 NO-UNDO/REDO 알고리즘이라고 함
- **즉시 갱신을 기반으로 한 복구 기법**
  - 버퍼에서 갱신된 데이터들을 트랜잭션 실행 도중(완료되기 전)에 디스크에 저장 가능
  - 데이터 갱신에 의한 로그 레코드 형식 <T, x, v1, v2>
  - 즉시 갱신을 기반으로 한 복구 알고리즘을 UNDO/REDO 알고리즘이라고 함

검사점을 이용한 복구

- 로그의 크기가 커질 수록 복구 부담 증가
- 특히, 불필요한 REDO 연산이 다량 발생함

검사점 (checkpoint)

- 주기적으로 버퍼의 갱신 기록을 디스크에 기록

  → 현재까지 로그에 기록된 내용과 디스크에 저장된 데이터베이스의 내용을 일치시키는 과정
- 검사점 이전 로그에 대해서는 REDO, UNDO 불필요

검사점 작업 (즉시 갱신만을 가정)

- 모든 트랜잭션의 실행 중단

- 갱신된 모든 버퍼의 내용을 디스크로 출력
- 로그에 <checkpoint> 레코드 저장
- 중단된 트랜잭션을 다시 실행