Transaction

Transaction

- Transaction
  - ✓ 한번에 수행되어야 하는 논리적인 작업 처리의 단위
  - ✓ 데이터베이스에서 발생하는 여러 개의 SQL 명령문들을 하나의 논리적인 작업 단위로 처리하는데 이를 Transaction
  - ✓ 하나의 Transaction은 All-or-Nothing 방식으로 처리
  - ✓ 여러 개의 명령어의 집합이 정상적으로 처리되면 정상 종료하도록 하고 여러 개의 명령어 중에서 하나의 명령어라도 잘못되었다면 전체를 취소
  - ✓ 데이터베이스에서 작업의 단위로 Transaction이란 개념을 도입한 이유는 데이터의 일관성을 유지하면서 안정적으로 데이터를 복구시키기 위해서
  - ✓ 트랜잭션의 성질
    - 원자성(Atomicity)은 트랜잭션과 관련된 작업들이 부분적으로 실행되다가 중단되지 않는 것을 보장하는 능력
    - 일관성(Consistency)은 트랜잭션이 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 유지하는 것을 의미
    - 독립성(Isolation)은 트랜잭션을 수행 시 다른 트랜잭션의 연산 작업이 끼어들지 못하도록 보장하는 것을 의미
    - 지속성(Durability)은 성공적으로 수행된 트랜잭션은 영원히 반영되어야 함을 의미

Transaction

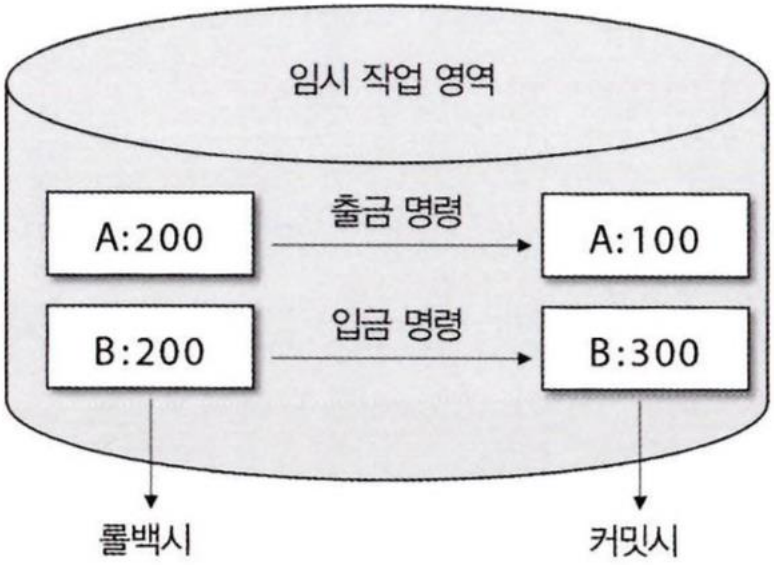
- 임시 작업 영역
  - ✓ 원자성을 구현하는 알고리즘은 복잡할 뿐만 아니라 성능에 지대한 영향을 주는 핵심 기술이어서 제품별로 고유한 방식을 사용하는데 제각각 다른 기술을 쓰지만 트랜잭션 처리 과정을 어딘가에 저장해 두는 방식에서 크게 벗어나지 않음
  - ✓ 처리 내역을 저장해 놓지 않으면 작업을 취소할 방법이 없음
  - ✓ 오라클은 이 영역을 Undo Segment 또는 Rollback Segment라고 부르며 SQL Server는 트랜잭션 로그(*.ldf)라고 부름
  - ✓ 내부적인 구조나 관리 방법은 확연히 다르겠지만 둘 다 레코드의 변경 사항을 저장하는 임시 작업 영역이라는 점은 동일
  - ✓ DBMS는 임시 영역에 데이터 변경 전, 후의 상태에 대한 복사본을 빠짐없이 기록해두는데 모든 명령이 성공하면 변경 후의 데이터를 기록하여 영구적으로 확정하는데 이를 Commit 이라고 함
  - ✓ 하나라도 실패하면 변경 전의 데이터를 다시 써넣어 안전하게 취소하는데 이를 RollBack 이라고 함

Transaction

❖ 임시 작업 영역

✓ 계좌 이체 작업을 트랜잭션으로 묶었을 경우 DBMS는 다음과 같이 작업 내역을 기록

- A, B 모두 트랜잭션 전에는 200 만원이 들어있었다고 가정
- A 계좌의 출금 전후, B 계좌의 입금 전후 상태를 기록
- 테이블에 바로 적용하는 것이 아니라 임시 영역에 먼저 기록하고 트랜잭션의 커밋 여부에 따라 다음 동작을 결정



Transaction

❖ 임시 작업 영역

- ✓ 계좌 이체 작업을 트랜잭션으로 묶었을 경우 DBMS는 다음과 같이 작업 내역을 기록
  - 이 상태에서 커밋하면 로그에서 처리 후의 레코드 상태를 가져와 적용하고 롤백하면 원래의 데이터를 유지
  - 이렇게 하면 둘 중 하나가 실패하더라도 데이터가 깨지지 않으며 안전하게 재시도할 수 있음
  - 업데이트하는 중에 전원이 차단되어도 실제 테이블을 조작하지 않았기 때문에 별 문제없음
- ✓ 모든 변경사항을 임시 영역에 일일이 기록한 후 적용하면 수정 속도는 느려지지만 고도로 지능화된 DBMS는 치밀한 알고리즘으로 오버헤드를 최소화하여 성능 문제를 극복함
- ✓ DBMS마다 또는 SQL 명령에 따라 로그를 작성하고 관리하는 방법은 다르지만 취소를 위한 완벽한 정보를 임시 작업 영역에 기록하는 방식은 동일

Transaction

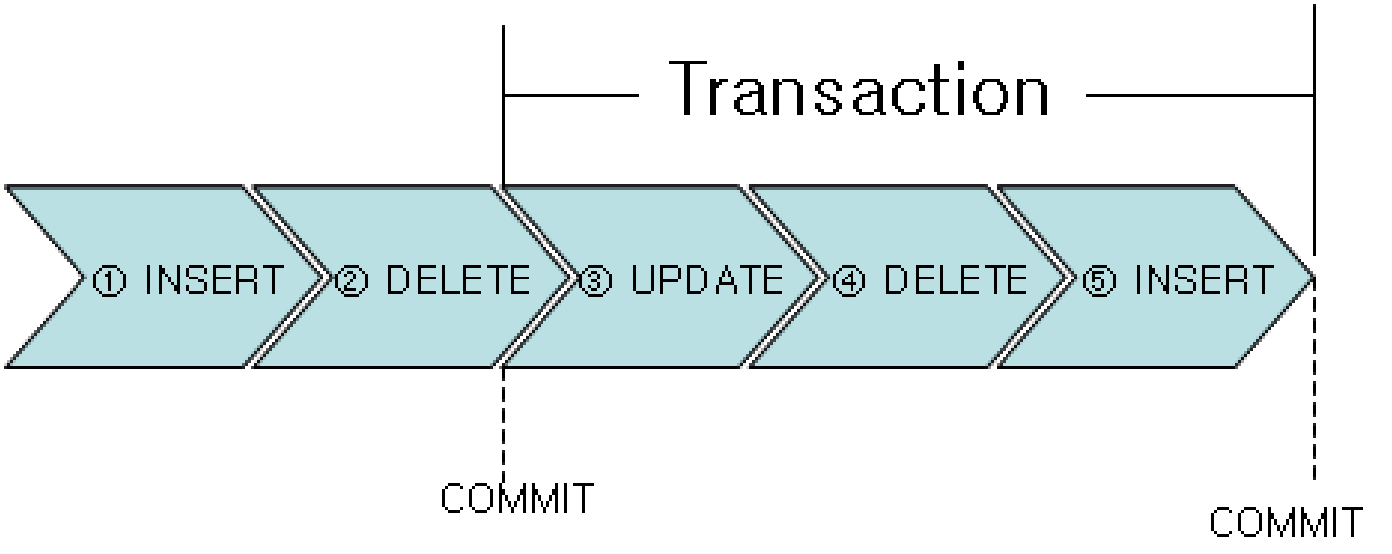
- Transaction 제어를 위한 명령어(Transaction Control Language)
  - ✓ COMMIT
  - ✓ SAVEPOINT
  - ✓ ROLLBACK

Transaction

- Transaction 제어를 위한 명령어(Transaction Control Language)
  - ✓ 데이터를 조작하는 명령어인 DML(Data Manipulation Language)은 실행됨과 동시에 Transaction이 시작
  - ✓ DML 작업이 성공적으로 처리되도록 하기 위해서는 COMMIT 명령을 작업을 취소하기 위해서는 ROLLBACK 명령으로 종료
  - ✓ COMMIT
    - COMMIT은 모든 작업들을 정상적으로 처리하겠다고 확정하는 명령어로 Transaction의 처리 과정을 데이터베이스에 모두 반영하기 위해서 변경된 내용을 모두 영구 저장
    - COMMIT 명령어를 수행하게 되면 하나의 Transaction을 종료
  - ✓ ROLLBACK
    - ROLLBACK은 작업 중 문제가 발생되어서 Transaction의 처리 과정에서 발생한 변경사항을 취소하는 명령어
    - ROLLBACK 명령어 역시 Transaction 과정을 종료
    - ROLLBACK은 Transaction 으로 인한 하나의 묶음 처리가 시작되기 이전의 상태로 되돌림
    - Transaction은 여러 개의 물리적인 작업(DML 명령어)들이 모여서 이루어지는데 이러한 과정에서 하나의 물리적인 작업이라도 문제가 발생하게 되면 모든 작업을 취소해야 하므로 이들을 하나의 논리적인 작업 단위(Transaction)로 구성
    - 문제가 발생하게 되면 이 논리적인 작업 단위를 취소하면 됨

Transaction

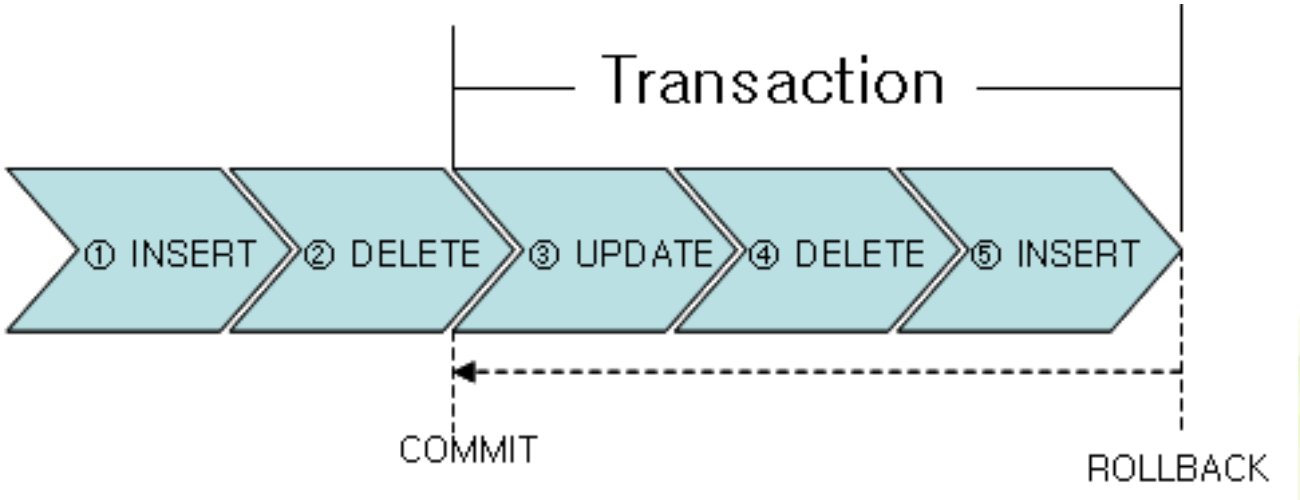
- Transaction 제어를 위한 명령어(Transaction Control Language)
  - ✓ 아래 그림에서 UPDATE 문으로 데이터를 갱신하고(③) DELETE 문으로 데이터를 삭제하고(④) INSERT 문을 사용해 데이터를 삽입(⑤)
  - ✓ 만약 이 모든 과정이 오류 없이 수행되었다면 지금까지 실행한 모든 작업(③, ④, ⑤)을 "데이터베이스에 영구 저장하라"는 명령으로 커밋을 수행



Transaction

❖ Transaction 제어를 위한 명령어(Transaction Control Language)

✓ Rollback 명령은 마지막으로 수행한 커밋 명령 까지만 정상 처리(①, ②)된 상태로 유지하고 그 이후에 수행했던 모든 DML 명령어 작업(③, ④, ⑤)들을 취소시켜 이전 상태로 원상 복귀



✓ Transaction은 이렇듯 All-or-Nothing 방식으로 DML 명령어들을 처리

Transaction

- Transaction 제어를 위한 명령어(Transaction Control Language)
  - ✓ COMMIT 과 ROLLBACK 명령어의 장점
    - 데이터 무결성이 보장
    - 영구적인 변경 전에 데이터의 변경 사항을 확인할 수 있음
    - 논리적으로 연관된 작업을 그룹화
  - ✓ COMMIT 명령어
    - Transaction(INSERT, UPDATE, DELETE) 작업 내용을 실제 DB에 저장
    - 이전 데이터가 완전히 UPDATE
    - 모든 사용자가 변경된 데이터의 결과를 확인할 수 있음
  - ✓ ROLLBACK 명령어
    - Transaction(INSERT, UPDATE, DELETE) 작업 내용을 취소
    - 이전 COMMIT한 곳 까지만 복구
  - ✓ 데이터베이스 사용자가 COMMIT이나 ROLLBACK 명령어를 명시적으로 수행시키지 않더라도 다음과 같은 경우에 자동 커밋 혹은 자동 Rollback이 발생
    - 접속 도구가 정상 종료되었다면 자동으로 COMMIT 되지만 비정상 종료되었다면 자동으로 ROLLBACK
    - DDL과 DCL 명령문이 수행된 경우 자동으로 COMMIT
    - 정전이 발생했거나 컴퓨터 Down시(컴퓨터의 전원이 끊긴) 자동으로 ROLLBACK

Transaction

- Transaction Mode
  - ✓ 자동, 수동모드
    - DBMS가 트랜잭션을 처리하는 모드는 크게 자동과 수동 2가지가 있음
    - 모드에 따라 트랜잭션 의 시작 여부와 유지 기간, 명령을 확정하는 시점이 다름
    - 트랜잭션 모드는 연결 단위 별로 지정하며 연결 중에도 언제든지 변경할 수 있음
      - ◆ 자동 커밋(Auto Commit)모드: 명령을 내리자마자 임시 영역을 거치지 않고 바로 확정하며 트랜잭션을 구성하지 않는데 수정 속도는 빠르지만 한 번 내린 명령을 취소할 수 없는데 SQL Server와 MariaDB, 대화식 프로그램은 이 모드가 디폴트
      - ◆ 수동 커밋(Manual Commit Mode) 모드: 명령 실행 시 트랜잭션이 즉시 시작되며 모든 결과를 임시 영역에 기록하는데 커밋 또는 롤백 명령을 내려야 확정 또는 취소를 결정하며 트랜잭션을 종료하는데 오라클과 DB2는 이 모드가 디폴트
    - 자동, 수동은 커밋을 적용하는 방식에 따라 붙인 이름이지 트랜잭션 시작 방식에 대해 붙인 이름이 아님
    - 자동은 명령 즉시 커밋해 버려 트랜잭션을 구성하지 않으며 수동은 변경 사항을 모으기 위해 항상 트랜잭션을 구성

Transaction

- Transaction Mode
  - ✓ 수동 모드 와 락
    - 자동 커밋 모드는 결과를 즉시 반영하여 편리한 반면 취소할 수 없어 위험
    - 수동 커밋 모드는 COMMIT이나 ROLLBACK 명령을 내리기 전에는 실제 데이터를 변경하지 않아 조금 느리지만 안전하기 때문에 좀 불편해도 수동 모드가 데이터 관리에는 더 유리
    - 수동 모드는 동시성이 떨어지는 약점이 있는데 임시 영역에 저장한 작업 기록을 소유자만 볼 수 있으며 다른 사용자는 보지 못함
    - 변경 중인 행은 확정 또는 취소할 때까지 다른 사람이 변경하지 못하도록 락을 걸어 두기 때문

Transaction

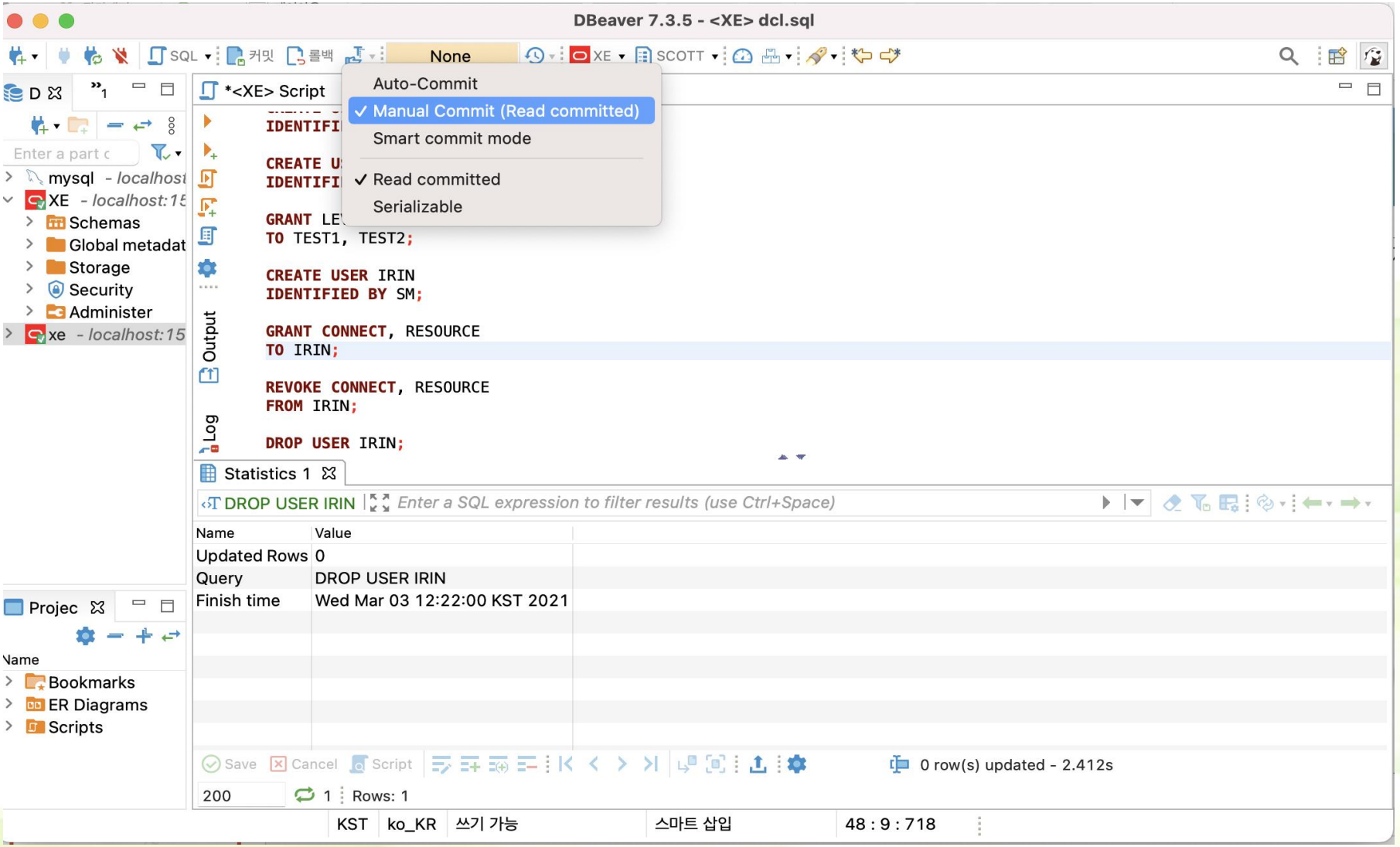
- Transaction Mode
  - ✓ 수동 모드 와 락
    - 2개의 데이터베이스 연결 세션이 있는 경우(SD1, SD2)
      - SD1 세션에서 서울의 인구를 1000으로 갱신하면 이 작업은 언두 세그먼트에 기록되지만 SD1에서 조회하면 언두 세그먼트까지 합쳐서 보여주므로 갱신한 인구수로 잘 보임
      - SD2 세션에서는 tCity 테이블을 바로 읽기 때문에 아직 갱신 사실을 알 수 없어 이전 인구수가 나타남
      - SD1 세션의 UPDATE 명령은 아직 트랜잭션 진행중이며 완결된 것이 아니어서 SD2가 조회하는 테이블에는 반영되어 있지 않는데 SD2는 변경 전의 서울 인구를 보고 있으며 이는 현재 값이 아닌데 때로는 이 짧은 시간 동안의 불일치도 큰 문제가 될 수 있음
      - SD1 세션은 변경 중인 행을 다른 사용자가 변경할 수 없도록 쓰기 락을 걸어둠
      - 내가 바꾸고 있는 중에 다른 사용자가 또 다른 값을 기록해 버리면 내가 쓴 값이 무시되기 때문에 트랜잭션 중에 독점권을 행사하는 것
      - 다행히 쓰기락이어서 SD2 세션에서 읽을 수는 있는데 이 상태에서 SD2 세션에서 서울의 인구를 1100으로 변경하고자 하면 스크립트 실행은 하지만 계속 작업 진행중이며 완료하지 함

Transaction

- Transaction Mode
  - ✓ 수동 모드 와 락
    - 2개의 데이터베이스 연결 세션이 있는 경우(SD1, SD2)
      - ◆ SD1 세션의 트랜잭션이 아직 진행 중이며 서울행에 대해 쓰기 락을 걸어 놓은 상태여서 락이 풀리기를 대기하고 있는 것인데 이 대기 상태는 SD1 세션에서 커밋하거나 롤백 할 때까지 계속됨
      - ◆ SD1에서 COMMIT를 실행하면 SD2의 대기가 풀리고 서울 인구수를 1100으로 갱신함
      - ◆ SD1의 트랜잭션은 종료되었지만 이번에는 SD2가 트랜잭션 진행중이며 역시 서울행을 붙잡고 있는 상태인데 이 때 각 연결에서 서울 인구수를 조회하면 자신이 수정한 데이터를 조회함
      - ◆ SD2 세션이 서울행에 락을 걸어 다른 사용자는 서울행을 읽기 만 할 수 있는데 락이 걸리지 않은 부산이나 춘천을 변경하는 것은 가능

Transaction

❖ DBeaver에서 수행하는 경우 트랜잭션 설정을 변경



Transaction

- 부서 번호가 10번인 부서에 대해서만 삭제하려고 했는데 테이블 내의 모든 데이터가 삭제되어 아무런 데이터도 찾을 수 없게 되었더라도 ROLLBACK 문을 사용하여 이전 상태로 되돌릴 수 있음

```
DROP TABLE DEPT01;

CREATE TABLE DEPT01    AS    SELECT *  FROM DEPT;

DELETE FROM DEPT01;

SELECT *    FROM DEPT01;

ROLLBACK;

SELECT *  FROM DEPT01;
```

Transaction

❖ 부서 번호가 20번인 부서만 삭제

✓ 부서 번호 20번 사원에 대한 정보만 삭제한 후 확인

```
DELETE FROM DEPT01
WHERE DEPTNO=20;

SELECT *
FROM DEPT01;
```

✓ 데이터를 삭제한 결과를 물리적으로 영구히 저장하기 위해서 커밋을 수행

```
COMMIT;

SELECT *
FROM DEPT01;
```

Transaction

- Auto Commit
  - ✓ DDL 문에는 CREATE, ALTER, DROP, RENAME, TRUNCATE 등이 있음
  - ✓ DDL문은 자동으로 커밋(AUTO COMMIT)이 발생
  - ✓ 오라클에서 set autocommit on 을 실행 한 경우 하나의 SQL 문장이 성공적으로 수행되면 바로 commit

Transaction

❖ CREATE문에 의한 자동 커밋에 의해서 이전에 수행했던 DML 명령어가 자동 커밋

1. 부서 번호가 40번인 부서를 삭제

```
DELETE FROM DEPT01 WHERE DEPTNO=40;
```

2. 삭제 후 부서 테이블(DEPT)과 동일한 내용을 갖는 새로운 테이블(DEPT03)을 생성

```
DROP TABLE DEPT02;


CREATE TABLE DEPT02 AS  SELECT * FROM DEPT;
```

3. DEPT02 테이블의 부서번호가 40번인 부서를 다시 되살리기 위해서 ROLLBACK 명령문을 수행하여도 이미 수행한 CREATE 문 때문에 자동으로 커밋이 발생하였으므로 되살릴 수 없음

```
ROLLBACK;


SELECT * FROM DEPT01;
```

Transaction

❖ TRUNCATE 문이 실패 되더라도 자동 커밋되어 이전에 수행했던 DML 명령어가 자동 커밋

1. 부서 테이블(DEPT03)에서 부서 번호가 20번인 부서를 삭제

```
DELETE FROM DEPT02 WHERE DEPTNO=30;
```

2. TRUNCATE 문을 실행시키되 테이블 명을 일부러 잘못 적어서 에러를 유도

```
TRUNCATE TABLE DEPTPPP;
```

3. 부서 번호가 20번인 부서를 다시 되살리기 위해서 ROLLBACK 명령문을 수행하여도 TRUNCATE 문이 수행되면서 자동으로 커밋이 발생하였으므로 되살릴 수 없음

```
ROLLBACK;

SELECT * FROM DEPT02;
```

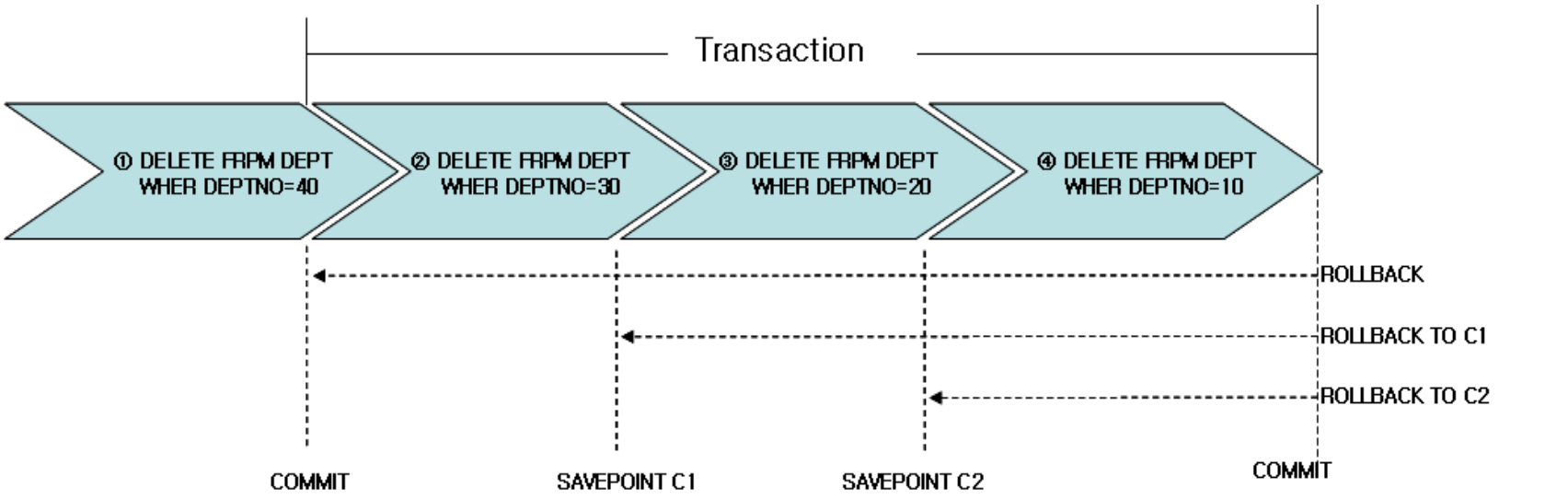
Transaction

- SAVEPOINT
  - ✓ 현재의 Transaction을 작게 분할
  - ✓ 저장된 SAVEPOINT는 ROLLBACK TO SAVEPOINT 문을 사용하여 표시한 곳까지 ROLLBACK할 수 있음
  - ✓ 여러 개의 SQL 문의 실행을 수반하는 Transaction의 경우 사용자가 Transaction 중간 단계에서 세이브포인트를 지정할 수 있음
  - ✓ Savepoint는 차후 Rollback과 함께 사용해서 현재 Transaction 내의 특정 세이브포인트까지 Rollback할 수 있음

Transaction

❖ SAVEPOINT

- ✓ COMMIT 명령이 내려진 후 다음 COMMIT 명령이 나타날 때까지가 하나의 Transaction으로 구성되므로 ②번에서 ④번까지가 하나의 Transaction
- ✓ 트랙잭션을 구성할 때 중간 중간 SAVEPOINT 명령으로 위치를 지정해 놓으면 (예를 들어 C) 하나의 Transaction 내에서도 ROLLBACK TO C(SAVEPOINT 문을 사용하여 표시한 곳)까지 ROLLBACK할 수 있음



Transaction

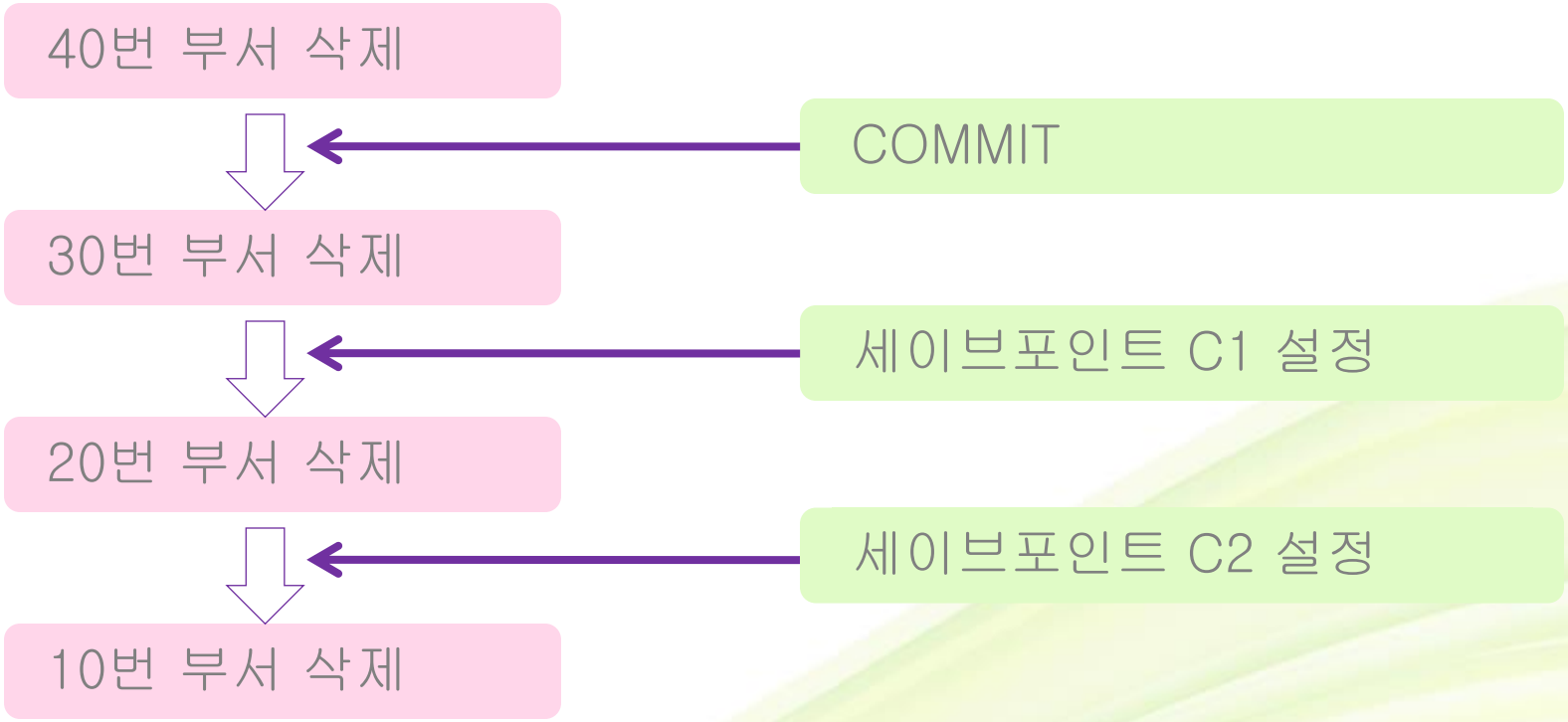
- SAVEPOINT
  - ✓ SAVEPOINT로 특정 위치를 지정하기 위한 형식

    SAVEPOINT *LABEL_NAME;*
  - ✓ SAVEPOINT로 지정해 놓은 특정 위치로 되돌아가기 위한 형식

    ROLLBACK TO *LABEL_NAME*;

Transaction

❖ Transaction 중간 단계에서 Savepoint를 지정



Transaction

1.부서번호가 40번인 부서를 삭제한 후에 커밋을 수행하여 새롭게 Transaction을 시작

```
DROP TABLE DEPT01;

CREATE TABLE DEPT01 AS SELECT * FROM DEPT;

DELETE FROM DEPT01 WHERE DEPTNO=40;

COMMIT;
```

2. 이번엔 부서번호가 30번인 부서를 삭제

```
DELETE FROM DEPT01 WHERE DEPTNO=30;
```

Transaction

3. 세이브포인트 C1를 설정한 후, 부서번호가 20번인 사원을 삭제

```
SAVEPOINT C1;

DELETE FROM DEPT01
WHERE DEPTNO =20;
```

4. 세이브포인트 C2를 설정한 후, 부서번호가 10번인 사원을 삭제

```
SAVEPOINT C2;

DELETE FROM DEPT01
WHERE DEPTNO =10;
```

Transaction

1. 지금 ROLLBACK 명령을 내리게 된다면 이전 COMMIT 지점으로 되돌아가므로 10, 20, 30번 부서의 삭제가 모두 취소되는데 10번 부서의 삭제 이전까지만 되돌리려면 다시 30, 20번의 부서를 삭제해야 함

2. Savepoint C2 지점으로 이동되어 10번 부서의 삭제 이전으로 되돌려진 것을 확인

```
ROLLBACK TO C2;

SELECT *  FROM DEPT01;
```

3. 마지막으로 이전 Transaction까지 Rollback 한 후의 결과 확인

```
ROLLBACK;

SELECT *  FROM DEPT01;
```

Transaction

❖ 다음 보기의 SQL문이 실행되면 조회되는 행 수는?

```
SAVEPOINT t1:

INSERT INTO EMP VALUES(10,20);

SAVEPOINT t2;

INSERT INTO EMP VALUES(20,30);

ROLLBACK TO t2;

COMMIT;

SELECT * FROM EMP;
```

LOCK

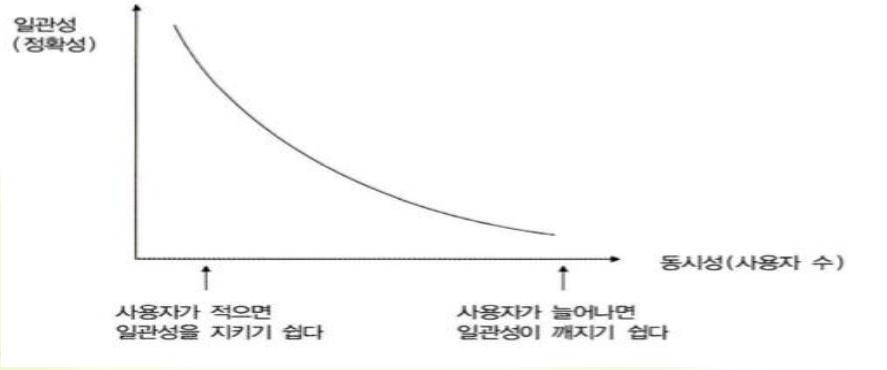
❖ 동시성 과 일관성

- ✓ 데이터베이스에 구축해 놓은 방대한 양의 데이터는 여러 사람이 같이 사용
- ✓ 예금 정보를 가지는 은행 DB는 모든 은행원이 공용으로 사용하며 열차 예약 DB도 모든 예약 담당자가 공유
- ✓ 집약되어 있고 공유하기 때문에 온라인 처리가 가능하며 가치 있는 정보가 됨
- ✓ 복수의 사용자가 같은 데이터를 동시에 액세스할 수 있는 것을 Concurrency(동시성)이라고 하는데 클라이언트는 네트워크로 연결되어 있고 서버는 전달받은 질의에 순차적으로 응답하면 되니 어려운 기술은 아님
- ✓ 최적의 환경에서는 동시 다발적으로 질의해도 순간적으로 반응할 만큼 성능이 뛰어남
- ✓ 일관성 이란 항상 정확한 정보를 가지는 것인데 은행 DB에 잔액을 질의하면 정확하게 알려주고 예약 시스템은 남은 좌석을 실시간으로 보여주는데 부정확한 정보는 없는 것보다 못함
- ✓ 이미 예약된 좌석을 비어 있는 것으로 조사하거나 방금 입금한 돈이 잔액에 즉시 반영되지 않으면 일관성이 없는 것
- ✓ DBMS가 제 성능을 발휘하려면 동시성과 일관성이 항상 유지되어야 하지만 안타깝게도 이 둘은 배치되는 성질이 있어 완벽하게 동시적이면서 완벽하게 일관적일 수 없음
- ✓ 공유로 인해 일관성을 유지하기 어렵고 일관성을 유지하려면 동시성이 떨어짐

LOCK

❖ 동시성 과 일관성

- ✓ 동시성과 일관성은 항상 반비례 관계
- ✓ 정확성을 희생하면 동시성을 높일 수 있고 정확성을 확보하려면 동시성을 제약해야 함
- ✓ 동시성과 일관성을 동시에 만족시키는 완벽한 방법은 없는데 DBMS는 적당한 수준에서 이 둘의 균형을 맞추는 여러 가지 장치와 옵션을 제공
- ✓ 그 장치가 바로 명령을 묶어서 처리하는 트랜잭션과 사용 중에 다른 사람의 접근을 막는 Lock
- ✓ Lock이 걸린 동안에는 동시에 접근할 수 없어 문제가 발생하지 않는데 일시적으로 동시성을 제한하여 그 반대급부로 일관성을 얻는 것
- ✓ DBMS는 동시성과 일관성을 유지하기 위한 필사의 노력을 하고 있으며 사용자는 내부 사정을 상세히 몰라도 상관없는데 이 두 가지 장치로도 완벽하지 않으며 또한 DBMS의 판단이 부적절한 경우도 있기 때문에 관리자나 개발자는 두 개념을 이해하고 DBMS의 동작에 개입하여 균형을 잡아야 함



LOCK

❖ 동시성의 부작용

✓ 손실 업데이트

- 15석의 좌석이 남은상태에서 A, B 두 명의 예약원이 근무중인데 DB 질의 결과 15석이 남아 있음을 확인하고 각각 10석, 7석을 예약하였고 예약한 자리 수 만큼 빼서 잔여 좌석 수를 갱신하는데 A가 먼저 15-10=5석으로 갱신하고 B가 다시 15-7=8로 갱신한 경우 잔여 좌석 은 15석 뿐이지만 17석이 초과 예약되었을 뿐만 아니라 아직도 8석 남은 것으로 잘못 기록됨

✓ 커밋되지 않은 읽기

- 예약원과 배차원이 동시에 작업하는 경우에는 배차원이 잔역 좌석을 조사해 보니 5석밖에 없어 100석짜리 객차 한량을 더 붙이기로 하고 잔여 좌석을 105석으로 만들었는데 예약원은 105석이 남아 있음을 확인하고 20석 예약을 받아 처리하였으며 잔여 좌석은 85석이 되었는데 예약이 완료된 후 배차원이 객차를 붙이려고 확인해 보니 여유 객차가 없어 트랜잭션을 롤백하면 잔여 좌석은 원래의 5석이 되는데 그 짧은 시간에 예약된 20석은 있지도 않는 좌석으로 예약된 것

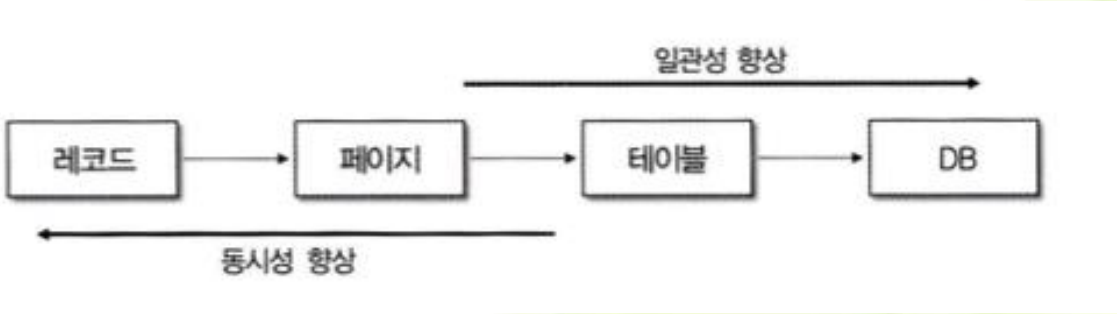
LOCK

- 동시성의 부작용
  - ✓ 반복하지 않은 읽기
    - 한 트랜잭션이 같은 값을 두 번 읽을 때 그 값이 달라지는 경우로 예약원이 15석이 남아 있음을 확인하고 고객과 예약 상담을 진행중인데 다른 예약원이 10석을 먼저 예약해 버렸다면 남은 좌석은 5석밖에 되지 않는데 이 때 고객이 예약을 결심하고 8석 예약을 요청했을 때 상담원이 다시 잔여 좌석을 확인해 보면 처음 읽은 15석과는 다른 5석밖에 없어서 결국 이 트랜잭션은 실패
    - 똑같은 데이터를 읽었음에도 불구하고 한 트랜잭션내에서 다른 결과가 리턴되었는데 이는 트랜잭션 진행 중에 다른 트랜잭션이 데이터를 변경하도록 내버려 두었기 때문
    - 잔여 좌석은 늘상 갱신되는 값이므로 읽을 때마다 그 결과가 달라지는 것이 당연하며 트랜잭션은 항상 이를 가정해야 함
  - ✓ 팬텀
    - 열차 시간이 2시 4시로 편성되어 있는데 예약원이 4시 열차가 있음을 확인하고 고객과 예약을 상담 중인데 그 사이에 배차원이 4시 열차를 편성표에서 빼버리는 경우 4시 열차는 DB상에서 사라져 버렸으며 예약원이 이미 확인한 4시 열차는 실제로 존재하지 않는 유령 열차인데 이처럼 다른 트랜잭션에 의해 삭제된 레코드를 Phantom이라고 함
    - 이 문제는 삭제 뿐만 아니라 삽입시에도 나타나는데 예약원은 2시, 4시 열차가 있음을 고객에게 알려 주고 상담을 진행중인데 배차원이 갑자기 3시 열차를 증편한다면 처음에 없던 레코드가 삽입되어 혼란을 초래하는데 팬텀을 해결하려면 데이터를 삽입, 삭제하는 동안에는 아예 읽지 못하도록 해야 함

LOCK

❖ 락의 범위

- ✓ 일관성을 얻으려면 동시성을 어느 정도 희생할 수밖에 없으며 동시성을 제한하는 장치가 바로 락인데 원리는 무척 간단
- ✓ 한 사람이 데이터를 사용하는 동안 다른 사용자가 이 데이터를 사용하지 못하도록 대기시키면 됨
- ✓ 공유 오브젝트를 일시적으로 잠궈 일관성을 유지
- ✓ A가 예약 트랜잭션을 진행중이면 B는 A가 예약을 Commit 하거나Rollback 할 때까지 대기
- ✓ 이렇게 하면 동시에 예약을 진행하지 않으므로 A가 보는 정보는 항상 정확하며 B가 업데이트하는 것도 항상 제대로 반영되는데 락을 거는 대상에 따라 사용 금지 범위가 달라짐



LOCK

- 락의 종류
  - ✓ 공유락: 쓰는 것은 금지하지만 읽기는 허용
  - ✓ 배타락: 쓰기 및 읽기도 금지

LOCK

- 격리 수준
  - ✓ 쿼리를 실행할 때 어떤 종류의 락을 얼마동안 걸 것인가를 지정하는 옵션
  - ✓ 동시성이 떨어지더라도 일관성이 중요하다면 높은 격리 수준을 설정하고 반대로 일관성을 희생하더라도 동시성을 높이려면 격리 수준을 낮춤
  - ✓ READ UNCOMMITTED - 커밋되지 않은 읽기를 허용하는 것으로 SELECT 문을 실행할 때 공유 락을 걸지 않아 한 트랜잭션에 서 데이터를 바꾸는 중에도 데이터를 읽을 수 있음
  - ✓ READ COMMITTED - 커밋된 읽기 만을 허용하는 것으로 SELECT 문을 실행할 때 공유 락을 걸어서 다른 트랜잭션이 데이터를 바꾸고 있는 중에는 읽을 수 없어 커밋되지 않은 읽기 현상은 발생하지 않음
  - ✓ REPETEABLE READ - 읽기를 마치더라도 공유락을 풀지 않으며 트랜잭션이 완전히 종료될 때까지 락을 유지하는 방식이라서 같은 값을 두 번 읽을 때 항상 같은 결과를 리턴
  - ✓ SERIALIZABLE - 가장 높은 격리수준으로 데이터를 읽는 동안 다른 트랜잭션이 이 데이터를 읽지도 쓰지도 못 할 뿐만 아니라 새로운 레코드를 추가하는 것도 허용하지 않음