Structured Query Language (SQL)

Part 5. Database Application Programs : Transactions and Stored Modules

Hyeokman Kim
School of Computer Science
Kookmin Univ.

ANSI/ISO SQL

SQL TRANSACTION

Transaction

- 트랜잭션(Transaction)
  - 데이터베이스의 논리적 연산단위.
    - 밀접히 관련되어 분리될 수 없는 한 개 이상의 데이터베이스 조작(SQL DML 문장)을 가리킴.
  - 트랜잭션은 분할할 수 없는 최소의 단위(atomic unit)임.
    - 한 트랜잭션에 포함된 모든 DML 문장은 전부 적용하거나 전부 취소해야 한다. (all-or-nothing)

- 예제: 계좌이체의 경우
  - 두 개의 업데이트가 모두 성공적으로 완료되었을 때 종료됨.
  - 둘 중 하나라도 실패할 경우, 두 계좌는 원래의 금액을 유지하고 있어야함.

```
-  STEP1. 100번 계좌의 잔액에서 10,000원을 뺀다.
-  STEP2. 200번 계좌의 잔액에서 10,000원을 더한다.
```

- 데이타베이스 응용 프로그램
  - 트랜잭션의 집합으로 볼 수 있음.

◘ 트랜잭션의 특징

특성	설명
원자성 (atomicity)	트랜잭션에서 정의된 연산들은 모두 성공적으로 실행되던지 아니면 전혀 실행되지 않은 상태로 남아 있어야 한다. (all or nothing)
일관성 (consistency)	트랜잭션이 실행되기 전의 데이터베이스 내용이 잘못 되어 있지 않다면 트랜잭션이 실행된 이후에도 데이터베이스의 내용에 잘못이 있으면 안 된다.
고립성 (isolation)	트랜잭션이 실행되는 도중에 다른 트랜잭션의 영향을 받아 잘못된 결과를 만들어서는 안 된다.
지속성 (durability)	트랜잭션이 성공적으로 수행되면 그 트랜잭션이 갱신한 데이터베이스의 내용은 영구적으로 저장된다.

◘ 데이터 잠금 (Locking)

- 트랜잭션의 특성(특히 원자성)을 충족하기 위해, 데이터베이스는 다양한 레벨의 잠금 기능을 제공함.
- 잠금은 기본적으로 트랜잭션이 수행되는 동안, 특정 데이터에 대해서 다른 트랜잭션이 동시에 접근하지 못하도록 제한하는 기법.
- 잠금이 걸린 데이터는 잠금을 실행한 트랜잭션만 독점적으로 접근할 수 있고, 다른 트랜잭션으로 부터 간섭이나 방해를 받지 않는 것이 보장됨.
- 잠금이 걸린 데이터는 잠금을 수행한 트랜잭션만이 해제할 수 있음.

Transaction Control

- 트랜젝션 제어를 위한 명령어
  - 1. COMMIT : 올바르게 반영된 데이터를 데이터베이스에 반영시킴.
  - 2. ROLLBACK : 트랜잭션 시작 이전의 상태로 되돌림.
  - 3. SAVEPOINT : SAVEPOINT를 설정하면, ROLLBACK할 때 트랜잭션에 포함된 전체 작업을 롤백하는 것이 아니라 현 시점에서 SAVEPOINT까지 '트랜잭션의 일부'만 롤백함.

- COMMIT과 ROLLBACK을 사용함으로써 다음과 같은 효과를 볼 수 있음.
  - 데이터 무결성 보장.
  - 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능.
  - 논리적으로 연관된 작업을 그룹핑하여 처리 가능.

1. COMMIT 명령어

- Commit
  - 입력/수정/삭제한 자료에 대해서 전혀 문제가 없다고 판단되었을 경우, COMMIT 명령어를 통해서 트랜잭션을 완료함.

- COMMIT이나 ROLLBACK 이전의 데이터 상태는 다음과 같음
  - 단지 메모리 버퍼만 영향을 받았기 때문에 데이터의 변경 이전 상태로 복구 가능하다.
  - 현재 사용자는 SELECT 문으로 결과의 확인이 가능하다.
  - 다른 사용자는 현재 사용자가 수행한 명령의 결과를 볼 수 없다.
  - 변경된 행은 잠금이 설정되어서, 다른 사용자가 변경할 수 없다.

- Oracle과 SQL Server에서의 차이점
  - Oracle은 DML 문장 수행 후에 사용자가 임의로 COMMIT 혹은 ROLLBACK을 수행해 주어야 트랜잭션이 종료됨.
  - SQL Server는 기본적으로 AUTO COMMIT 모드임.
    - DML 문장이 성공이면 자동으로 COMMIT이 되고, 오류가 발생할 경우 자동으로 ROLLBACK 처리됨.

예제: Oracle의 Commit

```
INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1997035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.

COMMIT;
커밋이 완료되었다.
```

```
UPDATE  PLAYER
SET     HEIGHT = 100;
480개의 행이 수정되었다.

COMMIT;
커밋이 완료되었다.
```

```
DELETE FROM PLAYER;
480개의 행이 삭제되었다.

COMMIT;
커밋이 완료되었다.
```

예제: SQL Server의 Commit (Auto Commit Mode)

```
INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1997035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.
```

```
UPDATE  PLAYER
SET     HEIGHT = 100;
480개의 행이 수정되었다.
```

```
DELETE FROM PLAYER;
480개의 행이 삭제되었다.
```

☞ SQL Server의 Commit 처리 방식

- ◆ Auto commit
- ◆ Implicit commit : 프로그램이 시작되면 트랜잭션이 묵시적으로 시작, 끝은 COMMIT/ROLLBACK 명령어 사용.
- ◆ Explicit commit : BEGIN/COMMIT/ROLLBACK TRANSACTION 명령어를 명시적으로 사용

2. ROLLBACK 명령어

- Rollback
  - 테이블 내 입력/수정/삭제한 데이터에 대하여, COMMIT 이전에는 언제나 변경 사항을 취소할 수 있음.
  - ROLLBACK은 데이터 변경 사항을 취소하고 데이터의 이전 상태로 복구되며, 관련된 행에 대한 잠금이 풀리어서 다른 사용자들이 데이터 변경을 할 수 있음.

예제: Oracle의 Rollback

```
INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1997035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.

ROLLBACK;
롤백이 완료되었다.
```

```
UPDATE  PLAYER
SET     HEIGHT = 100;
480개의 행이 수정되었다.

ROLLBACK;
롤백이 완료되었다.
```

```
DELETE FROM PLAYER;
480개의 행이 삭제되었다.

ROLLBACK;
롤백이 완료되었다.
```

예제: SQL Server의 Rollback

```
BEGIN TRAN
INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1997035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.

ROLLBACK;
롤백이 완료되었다.
```

```
BEGIN TRAN
UPDATE  PLAYER
SET     HEIGHT = 100;
480개의 행이 수정되었다.

ROLLBACK;
롤백이 완료되었다.
```

```
BEGIN TRAN
DELETE FROM PLAYER;
480개의 행이 삭제되었다.

ROLLBACK;
롤백이 완료되었다.
```

3. SAVEPOINT 명령어

□ SAVEPOINT

– SAVEPOINT를 설정하면, ROLLBACK할 때 트랜잭션에 포함된 전체 작업을 롤백하는 것이 아니라 현 시점에서 SAVEPOINT까지 '트랜잭션의 일부'만 롤백함.

– 복잡한 대규모 트랜잭션에서 에러가 발생했을 때, SAVEPOINT까지의 트랜잭션만 롤백하여, 실패한 부분에 대해서만 부분적으로 다시 실행함.

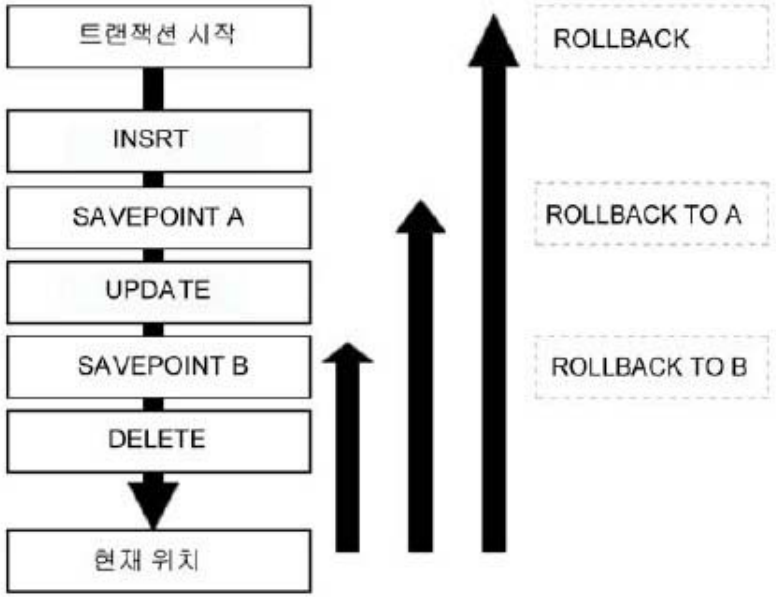
□ 명령어

– Oracle : SAVEPOINT 명령어

– SQL Server : SAVE TRANSACTION 명령어

◘ 특징

- 특정 저장점까지 ROLLBACK하면, 그 저장점 이후에 설정한 저장점은 무효가 됨.
  - ◆ 저장점 A로 되돌리고 나서, 다시 B와 같이 미래 방향으로 되돌릴 수는 없음.
- 저장점 지정 없이 ROLLBACK을 실행했을 경우, 반영 안된 모든 변경 사항을 취소하고 트랜잭션 시작 위치로 되돌아감.



예제: Oracle의 SAVEPOINT

□ 간단한 삽입, 수정, 삭제의 ROLLBACK

```
SAVEPOINT SVPT1;
저장점이 생성되었다.

INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.

ROLLBACK TO SVPT1;
롤백이 완료되었다.
```

```
SAVEPOINT SVPT2;
저장점이 생성되었다.

UPDATE  PLAYER
SET     WEIGHT = 100;
480개의 행이 수정되었다.

ROLLBACK TO SVPT2;
롤백이 완료되었다.
```

SAVEPOINT SVPT3;
저장점이 생성되었다.

DELETE FROM PLAYER;
480개의 행이 삭제되었다.

ROLLBACK TO SVPT3;
롤백이 완료되었다.

□ 예제

– 새로운 트랜잭션을 시작하기 전에, PLAYER 테이블의 데이터 건수와 몸무게가 100인 선수의 데이터 건수를 확인한다.

```
SELECT   COUNT(*)
FROM     PLAYER;

COUNT(*)
----------
       480
1개의 행이 선택되었다.

SELECT   COUNT(*)
FROM     PLAYER
WHERE    WEIGHT = 100;

COUNT(*)
----------
         0
1개의 행이 선택되었다.
```

– 새로운 트랜잭션을 시작하고, SAVEPOINT A와 SAVEPOINT B를 지정한다.

```
INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.

SAVEPOINT SVPT_A;
저장점이 생성되었다.

UPDATE  PLAYER
SET     WEIGHT = 100;
481개의 행이 수정되었다.

SAVEPOINT SVPT_B;
저장점이 생성되었다.

DELETE FROM PLAYER;
481개의 행이 삭제되었다.
```

– CASE1. SAVEPOINT B 저장점까지 ROLLBACK을 수행하고, 롤백 전후 데이터를 확인해본다.

```
SELECT   COUNT(*)
FROM     PLAYER;

COUNT(*)
----------
         0
1개의 행이 선택되었다.

ROLLBACK TO SVPT_B;
롤백이 완료되었다.

SELECT   COUNT(*)
FROM     PLAYER;

COUNT(*)
----------
       481
1개의 행이 선택되었다.
```

여기서 CASE 1,2,3을 순서대로 수행함.

– CASE2. SAVEPOINT A 저장점까지 롤백ROLLBACK을 수행하고,
롤백 전후 데이터를 확인해본다.

```
SELECT   COUNT(*)
FROM     PLAYER
WHERE    WEIGHT = 100;

COUNT(*)
-----------
        481
1개의 행이 선택되었다.

ROLLBACK TO SVPT_A;
롤백이 완료되었다.

SELECT   COUNT(*)
FROM     PLAYER
WHERE    WEIGHT = 100;

COUNT(*)
-----------
          0
1개의 행이 선택되었다.
```

– CASE3. 트랜잭션 최초 시점까지 ROLLBACK을 수행하고, 롤백 전후 데이터를 확인해본다.

```
SELECT    COUNT(*)
FROM      PLAYER;

COUNT(*)
----------
       481
1개의 행이 선택되었다.

ROLLBACK;
롤백이 완료되었다.

SELECT    COUNT(*)
FROM      PLAYER;

COUNT(*)
----------
       480
1개의 행이 선택되었다.
```

예제: SQL Server의 SAVEPOINT

□ 간단한 삽입과 수정의 ROLLBACK

```
SAVE TRAN SVPT1;
저장점이 생성되었다.

INSERT INTO PLAYER
(PLAYER_ID, TEAM_ID, PLAYER_NAME, POSITION, HEIGHT, WEIGHT, BACK_NO)
VALUES ('1999035', 'K02', '이운재', 'GK', 182, 82, 1);
1개의 행이 만들어졌다.

ROLLBACK TRAN SVPT1;
롤백이 완료되었다.
```

```
SAVE TRAN SVPT2;
저장점이 생성되었다.

UPDATE  PLAYER
SET     WEIGHT = 100;
480개의 행이 수정되었다.

ROLLBACK TRAN SVPT2;
롤백이 완료되었다.
```

ANSI/ISO SQL

SQL STORED MODULES: PROCEDURE, USER DEFINED FUNCTION, AND TRIGGER

Stored Module

- 저장 모듈(Stored module)
  - 개발자가 자주 실행하는 로직을 절차적인 언어를 이용하여 작성한 프로그램 모듈.
  - 데이터베이스 서버에 저장하여, 사용자와 애플리케이션들이 공유 가능한 일종의 SQL 컴포넌트 프로그램.
  - 독립적으로 실행되거나, 다른 프로그램으로부터 실행될 수 있는 완전한 실행 프로그램임.

- 종류
  - 1. Procedure
  - 2. User defined function
  - 3. Trigger

- 저장 모듈을 위한 절차적 프로그래밍 언어 (Procedural Language)
  - PL/SQL in Oracle
  - T-SQL in SQL Server

PL/SQL 개요

□ PL/SQL

– Block 구조

– Block 내에는 DML 문장, QUERY 문장, 그리고 절차형 언어(IF, LOOP) 등을 사용할 수 있으며, 절차적 프로그래밍을 가능하게 하는 트랜잭션 언어.

□ 특징

– Block 구조로 되어있어 각 기능별로 모듈화가 가능함.

– 변수, 상수 등을 선언하여 SQL 문장 간 값을 교환함.

– IF, LOOP 등의 절차형 언어를 사용하여 절차적인 프로그램이 가능함.

– DBMS 정의 에러나 사용자 정의 에러를 정의하여 사용할 수 있음.

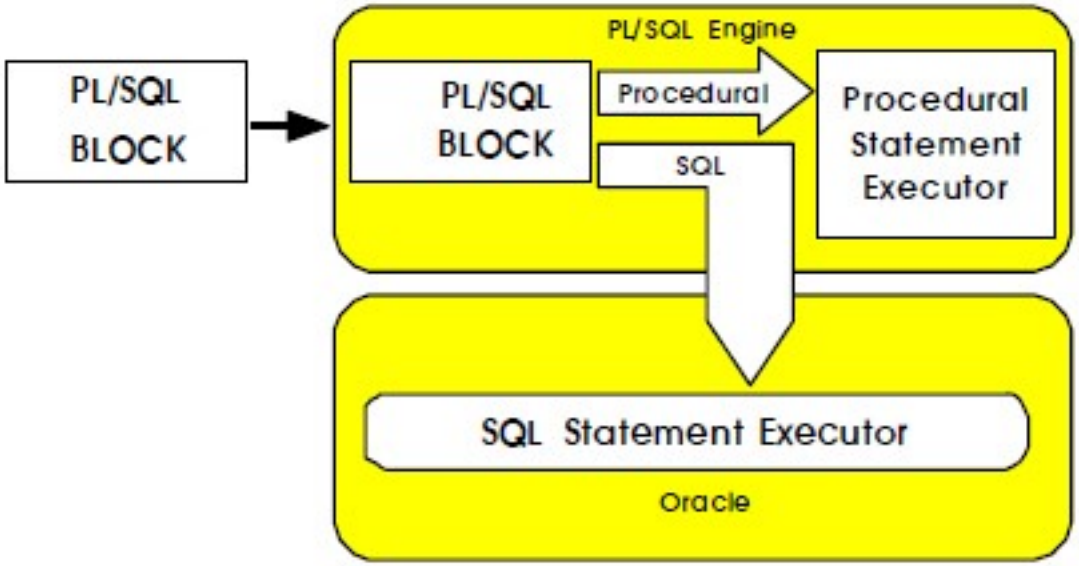
– Oracle에 내장되어 있으므로, Oracle과 PL/SQL을 지원하는 어떤 서버로도 프로그램을 옮길 수 있음.

– 응용 프로그램의 성능을 향상시킴.

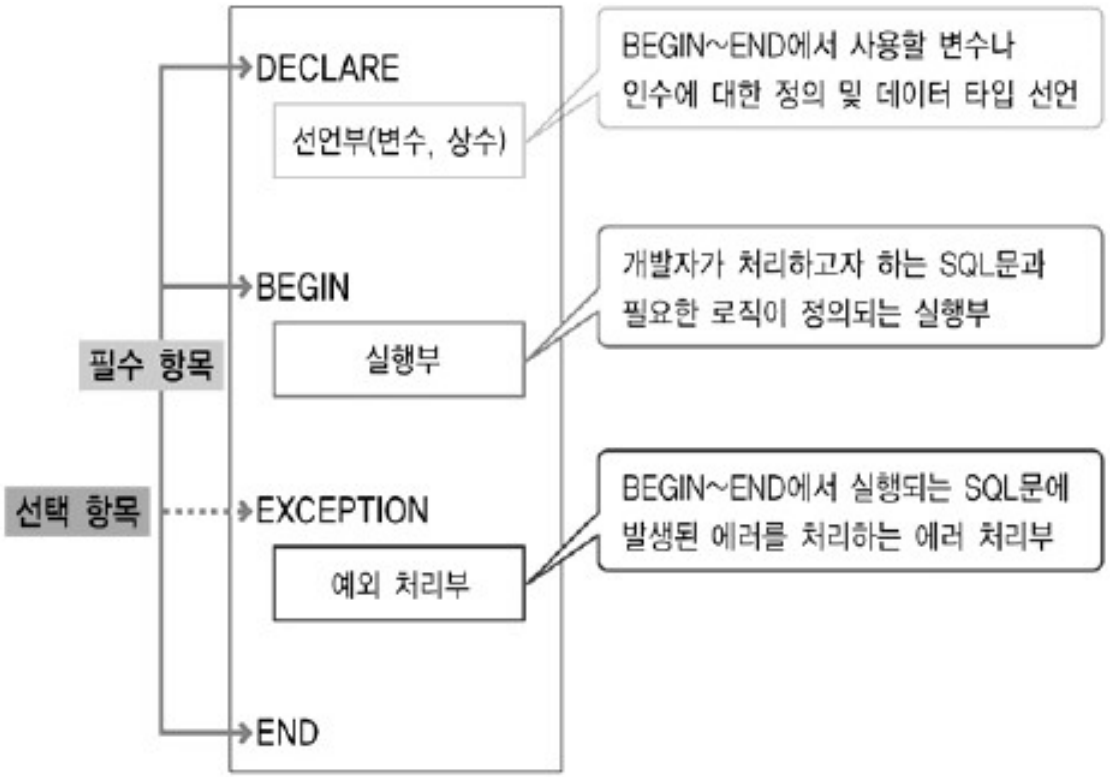
– 여러 SQL 문장을 Block으로 묶고 한 번에 Block 전부를 서버로 보내기 때문에 통신량을 줄일 수 있음.

PL/SQL의 실행

- PL/SQL engine (procedural statement executor)
  + SQL statement executor



PL/SQL의 블록 구조



PL/SQL의 기본 문법

□ Syntax

```
CREATE [OR REPLACE] PROCEDURE [Procedure_name]
( argument1 [IN|OUT|INOUT] data_type1,
  argument2 [IN|OUT|INOUT] date_type2,
  ... ... )
IS [AS]
 ... ...
BEGIN
 ... ...
EXCEPTION
 ... ...
END;
/                                   /* procedure를 컴파일하라는 명령어 */
```

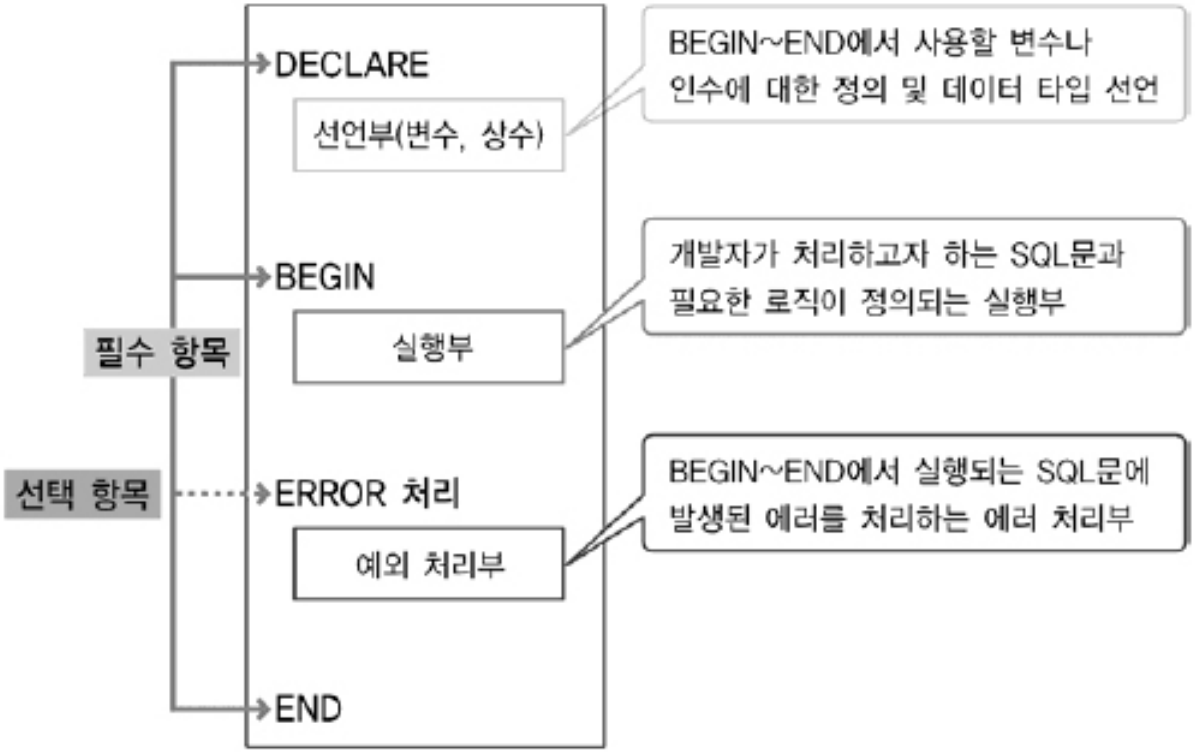
```
DROP PROCEDURE [Procedure_name];
```

T-SQL 개요

ㅁ 특징

- 변수 선언 기능
  - @@는 전역변수(시스템 함수), @는 지역변수 선언임.
  - 지역변수는 사용자가 자신의 연결 시간 동안만 사용하기 위해 만들어지는 변수이며 전역변수는 이미 SQL서버에 내장된 값임.
- int, float, varchar 등의 데이터 유형(Data Type)을 제공함.
- 산술연산자(+, -, *, /), 비교연산자(=, <, >, <>), 그리고 논리연산자(and, or, not) 사용이 가능.
- IF-ELSE, WHILE, CASE-THEN 등 흐름 제어 기능이 가능.
- 주석 기능
  - 한줄 주석 : -- 뒤의 내용은 주석
  - 범위 주석 : /* 내용 */ 형태를 사용하며, 여러 줄도 가능함.

T-SQL의 블록 구조



[그림 Ⅱ-2-20] T-SQL 구조

T-SQL의 기본 문법

□ Syntax

```
CREATE PROCEDURE [Schema_name.]Procedure_name
@parameter1 data_type1 [VARYING|DEFAULT|OUT|READONLY],
@parameter2 date_type2 [VARYING|DEFAULT|OUT|READONLY],
  … …
WITH RECOMPILE|ENCRYPTION|EXECUTE
AS
 … …
BEGIN
 … …
ERROR 처리
 … …
END;
```

```
DROP PROCEDURE [Schema_name.]Procedure_name;
```

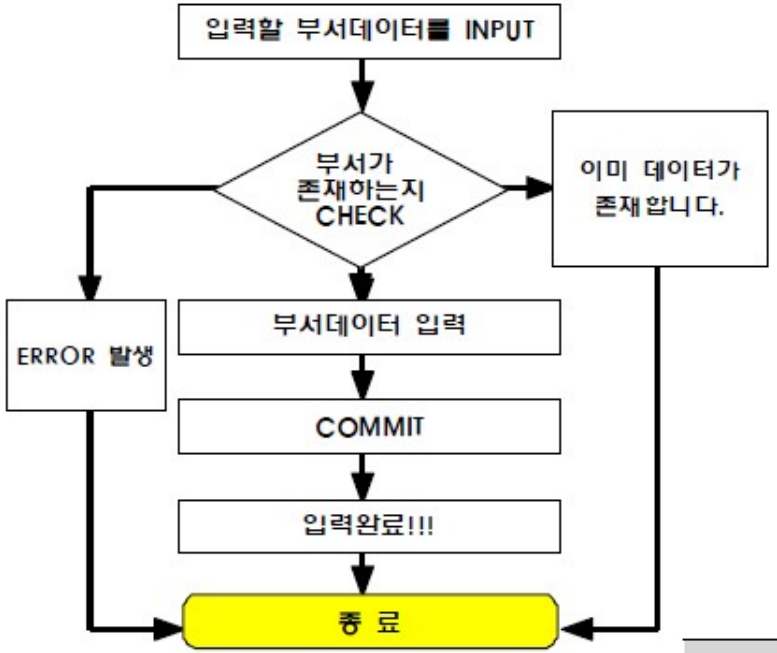
1. Procedure의 생성과 활용

- Procedure
  - 개발자가 자주 실행하는 로직을 절차적인 언어를 이용하여 작성한 프로그램 모듈

- EXECUTE 명령어로 실행

Procedure 예제

□ SCOTT가 소유하고 있는 DEPT 테이블에 새로운 부서를 등록하는 Procedure를 작성한다. DEPT 테이블의 구조는 다음과 같다.



[표 Ⅱ-2-14] DEPT 테이블 구조

DEPT			
칼럼ID	TYPE	길이	인덱스
DEPTNO	NUMBER	2	PK
DNAME	VARCHAR2	14	
LOC	VARCHAR2	13	

□ Oracle

```
CREATE OR REPLACE PROCEDURE p_DEPT_insert
(    v_DEPTNO in number,
     v_dname in varchar2,
     v_loc in varchar2,
     v_result out varchar2)
IS
     cnt number := 0;
BEGIN
     SELECT COUNT(*) INTO CNT
     FROM DEPT
     WHERE DEPTNO = v_DEPTNO AND ROWNUM = 1;
     if cnt > 0 then
          v_result := '이미 등록된 부서번호이다';
     else
          INSERT INTO DEPT (DEPTNO, DNAME, LOC)
          VALUES (v_DEPTNO, v_dname, v_loc);
          COMMIT;
          v_result := '입력 완료!!';
     end if;
EXCEPTION
     WHEN OTHERS THEN
          ROLLBACK;
     v_result := 'ERROR 발생';
END;
/
```

```
SQL> SELECT * FROM DEPT;                                    ----------①
    DEPTNO   DNAME          LOC
 --------   -------        ----------
        10   ACCOUNTING     NEW YORK
        20   RESEARCH       DALLAS
        30   SALES          CHICAGO
        40   OPERATIONS     BOSTON

SQL> variable rslt varchar2(30);                            ----------②

SQL> EXECUTE p_DEPT_insert(10,'dev','seoul',:rslt);         ----------③
PL/SQL 처리가 정상적으로 완료되었다.

SQL> print rslt;                                            ----------④
RSLT
--------------------------------
이미 등록된 부서번호이다

SQL> EXECUTE p_DEPT_insert(50,'NewDev','seoul',:rslt);      ----------⑤
PL/SQL 처리가 정상적으로 완료되었다.

SQL> print rslt;                                            ----------⑥
```

```
RSLT
--------------------------------
입력 완료!!

SQL> SELECT * FROM DEPT;                                    ----------⑦
    DEPTNO   DNAME          LOC
    ------   ---------      ----------
        10   ACCOUNTING     NEW YORK
        20   RESEARCH       DALLAS
        30   SALES          CHICAGO
        40   OPERATIONS     BOSTON
        50   NewDev         SEOUL
5개의 행이 선택되었다.
```

□ SQL Server

```
CREATE PROCEDURE dbo.p_DEPT_insert
      @v_DEPTNO int, @v_dname varchar(30),
      @v_loc varchar(30), @v_result varchar(100) OUTPUT
AS
      DECLARE @cnt int
      SET @cnt = 0
BEGIN
      SELECT @cnt=COUNT(*)
      FROM DEPT
      WHERE DEPTNO = @v_DEPTNO
      IF @cnt > 0
      BEGIN
            SET @v_result = '이미 등록된 부서번호이다'
            RETURN
      END ELSE
      BEGIN
            BEGIN TRAN
            INSERT INTO DEPT (DEPTNO, DNAME, LOC)
            VALUES (@v_DEPTNO, @v_dname, @v_loc)
            IF @@ERROR<>0
            BEGIN
                  ROLLBACK
                  SET @v_result = 'ERROR 발생'
                  RETURN
            END
            ELSE BEGIN
                  COMMIT
                  SET @v_result = '입력 완료!!'
                  RETURN
            END
      END
END
```

```
SELECT * FROM DEPT;                                          ----------①

   DEPTNO   DNAME          LOC
  -------   -----------    ----------
       10   ACCOUNTING     NEW YORK
       20   RESEARCH       DALLAS
       30   SALES          CHICAGO
       40   OPERATIONS     BOSTON

DECALRE @v_result VARCHAR(100)                               ----------②
EXECUTE dbo.p_DEPT_insert 10, 'dev', 'seoul',
        @v_result=@v_result OUTPUT                           ----------③
SELECT @v_result AS RSLT                                     ----------④
```

```
RSLT
--------------------------------
이미 등록된 부서번호이다

DECALRE @v_result VARCHAR(100)                               ----------⑤
EXECUTE dbo.p_DEPT_insert 50, 'dev', 'seoul',
        @v_result=@v_result OUTPUT                           ----------⑥
SELECT @v_result AS RSLT                                     ----------⑦

RSLT
--------------------------------
입력 완료!

SELECT * FROM DEPT;                                          ----------⑧

   DEPTNO   DNAME          LOC
  -------   --------       ----------
       10   ACCOUNTING     NEW YORK
       20   RESEARCH       DALLAS
       30   SALES          CHICAGO
       40   OPERATIONS     BOSTON
       50   NewDev         SEOUL
5개의 행에서 선택되었다.
```

2. User Defined Function의 생성과 활용

□ User Defined Function

– Procedure처럼 절차형 SQL을 로직과 함께 데이터베이스 내에 저장해 놓은 명령문의 집합.

□ Function과 Procedure와 차이점

– Function은 RETURN을 사용해서 하나의 값을 반드시 되돌려 줘야 함.

– Procedure는 EXECUTE 문을 사용하여 실행하나, Function은 DML 내에서 실행.

User Defined Function 예제

- 예제
  - 수의 절대값을 구하는UTIL_ABS 함수를 만들고, 이를 이용하여 K-리그 8월 경기결과와 두 팀간의 점수차를 UTIL_ABS 함수를 사용하여 절대값으로 출력한다.

□ Oracle

```
CREATE OR REPLACE FUNCTION UTIL_ABS
    (v_input in number)
    return NUMBER
IS
    v_return number := 0;
BEGIN
    if v_input < 0 then
            v_return := v_input * -1;
    else
            v_return := v_input;
    end if;
    RETURN v_return;
END;
/
```

```
SELECT   SCHE_DATE 경기일자,
         HOMETEAM_ID || ' - ' || AWAYTEAM_ID 팀들,
         HOME_SCORE || ' - ' || AWAY_SCORE SCORE,
         UTIL_ABS(HOME_SCORE - AWAY_SCORE) 점수차
FROM     SCHEDULE
WHERE    GUBUN = 'Y' AND
         SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER    BY SCHE_DATE;
```

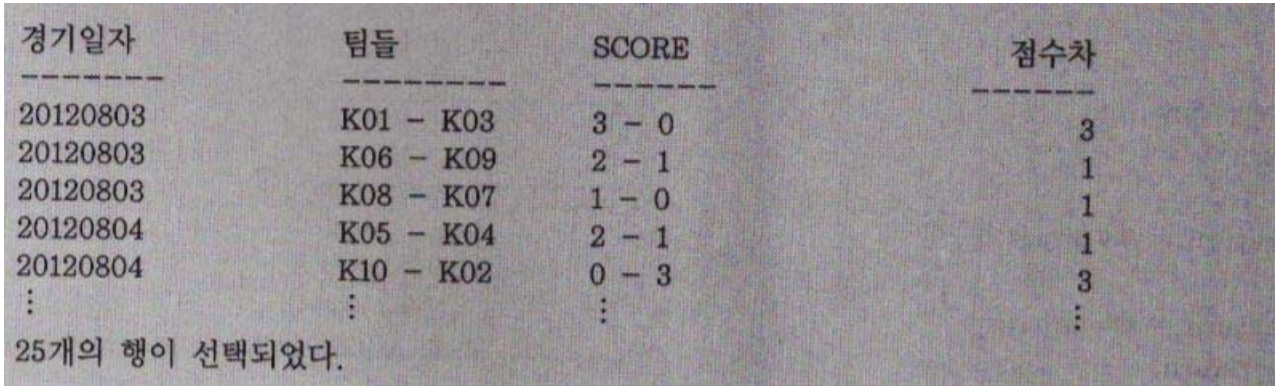
경기일자	팀들	SCORE	점수차
20120803	K01 - K03	3 - 0	3
20120803	K06 - K09	2 - 1	1
20120803	K08 - K07	1 - 0	1
20120804	K05 - K04	2 - 1	1
20120804	K10 - K02	0 - 3	3
⋮	⋮	⋮	⋮

25개의 행이 선택되었다.

□ SQL Server

```
CREATE FUNCTION dbo.UTIL_ABS
    (@v_input int)
    RETURNS int
AS
BEGIN
    DECLARE @v_return int
    SET @v_return=0
    IF @v_input < 0
            SET @v_return = @v_input * -1
    ELSE SET
            @v_return = @v_input
    RETURN @v_return;
END
```

```
SELECT   SCHE_DATE 경기일자,
         HOMETEAM_ID + ' - ' + AWAYTEAM_ID 팀들,
         HOME_SCORE + ' - ' + AWAY_SCORE SCORE,
         dbo.UTIL_ABS(HOME_SCORE - AWAY_SCORE) 점수차
FROM     SCHEDULE
WHERE    GUBUN = 'Y' AND
         SCHE_DATE BETWEEN '20120801' AND '20120831'
ORDER    BY SCHE_DATE;
```



경기일자	팀들	SCORE	점수차
20120803	K01 - K03	3 - 0	3
20120803	K06 - K09	2 - 1	1
20120803	K08 - K07	1 - 0	1
20120804	K05 - K04	2 - 1	1
20120804	K10 - K02	0 - 3	3
⋮	⋮	⋮	⋮

25개의 행이 선택되었다.

3. Trigger의 생성과 활용

◻ Trigger

- 특정한 테이블에 INSERT, UPDATE, DELETE와 같은 DML문이 수행되었을 때, 데이터베이스에서 자동으로 동작하도록 작성된 프로그램.
- 즉, 사용자가 직접 호출하여 사용하는 것이 아니라 데이터베이스에서 자동적으로 수행함.
- 테이블, 뷰, 데이터베이스 작업을 대상으로 정의할 수 있음.
- Trigger는 자체가 하나의 트랜젝션임.

◻ 종류

- 전체 트랜잭션 작업에 대해 발생되는 Trigger
- 각 투플에 대해서 발생되는 Trigger

- Procedure와 Trigger의 차이점
  - Trigger는 자체가 하나의 트랜젝션임.
    - Procedure는 BEGIN ~ END 절 내에 COMMIT, ROLLBACK과 같은 트랜잭션 종료 명령어를 사용할 수 있으나,
    - Trigger는 BEGIN ~ END 절 내에 사용할 수 없다.
  - Procedure는 EXECUTE 문으로 실행하나, Trigger는 조건이 충족되면 자동 실행됨.

Trigger 예제

□ 조건

- 어떤 쇼핑몰에 하루에 수만 건의 주문이 들어온다.
- 주문 데이터는 주문일자, 주문상품, 수량, 가격이 있으며,
- 사장을 비롯한 모든 임직원이 일자별, 상품별 총 판매수량과 총 판매가격으로 구성된 주문 실적을 온라인상으로 실시간 조회한다면, 한 사람의 임직원이 조회할 때마다 수만 건의 데이터를 읽어 계산해야 한다.

□ 문제점

- 빈번하게 조회작업이 일어난다면 조회작업에 많은 시간을 허비할 수 있다.

□ 해결책

- Trigger를 사용하여, 주문한 건이 입력될 때마다, 일자별 상품별로 판매수량과 판매금액을 집계하여 집계자료를 보관한다.

□ Oracle

– 관련 테이블 생성

```
CREATE TABLE ORDER_LIST (
    ORDER_DATE      CHAR(8) NOT NULL,
    PRODUCT         VARCHAR2(10) NOT NULL,
    QTY             NUMBER NOT NULL,
    AMOUNT          NUMBER NOT NULL);

CREATE TABLE SALES_PER_DATE (
    SALE_DATE       CHAR(8) NOT NULL,
    PRODUCT         VARCHAR2(10) NOT NULL,
    QTY             NUMBER NOT NULL,
    AMOUNT          NUMBER NOT NULL);
```

☞ PL/SQL에서 :NEW와 :OLD는 신규/기존 레코드를 갖고있는 구조체

구분	:OLD	:NEW
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

– Trigger 작성 (PL/SQL)

◆ ORDER_LIST 테이블에 주문 정보가 입력되면, 주문 정보의 주문 일자와 주문 상품을 기준으로, 판매 집계 테이블에 해당 주문 일자의 주문 상품 레코드가 존재하면, 판매 수량과 판매 금액을 더하고, 존재하지 않으면 새로운 레코드를 삽입한다.

```
CREATE OR REPLACE TRIGGER SUMMARY_SALES
    AFTER INSERT
    ON ORDER_LIST
    FOR EACH ROW
DECLARE
    o_date ORDER_LIST.order_date%TYPE;
    o_prod ORDER_LIST.product%TYPE;
BEGIN
    o_date := :NEW.order_date;
    o_prod := :NEW.product;
    UPDATE SALES_PER_DATE
    SET qty = qty + :NEW.qty, amount = amount + :NEW.amount
    WHERE sale_date = o_date AND product = o_prod;
    if SQL%NOTFOUND then
            INSERT INTO SALES_PER_DATE
            VALUES(o_date, o_prod, :NEW.qty, :NEW.amount);
    end if;
END;
/
```

– ORDER_LIST 테이블에 한나의 주문 정보(10개 주문)를 입력한다.

```
SQL> SELECT * FROM ORDER_LIST;
선택된 레코드가 없다.

SQL> SELECT * FROM SALES_PER_DATE;
선택된 레코드가 없다.

SQL> INSERT INTO ORDER_LIST VALUES('20120901', 'MONOPACK', 10, 300000);
1개의 행이 만들어졌다.

SQL> COMMIT;
커밋이 완료되었다.
```

– 주문 정보와 판매 집계 테이블에 같은 데이터가 들어왔는지 확인한다.

```
SQL> SELECT * FROM ORDER_LIST;
ORDER_DATE  PRODUCT          QTY      AMOUNT
----------  ---------  ---------  --------
20120901    MONOPACK          10      300000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
----------  ---------  ---------  --------
20120901    MONOPACK          10      300000
```

– 다른 주문 정보(20개 주문)를 입력하고, 확인한다.

```
SQL> INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000);
1개의 행이 만들어졌다.

SQL> COMMIT;
커밋이 완료되었다.

SQL> SELECT * FROM ORDER_LIST;
ORDER_DATE  PRODUCT             QTY      AMOUNT
----------  ----------  ------  -------
20120901    MONOPACK            10      300000
20120901    MONOPACK            20      600000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT             QTY      AMOUNT
---------   ----------  -----  -------
20120901    MONOPACK            30      900000
```

– 이번에는 다른 상품으로 주문 데이터를 입력한 후 두 테이블의 결과를 조회해 보고 트랜잭션을 ROLLBACK 수행한다.

– 판매 데이터의 입력 취소가 일어나면, 주문 정보 테이블과 판매 집계 테이블에 동시에 입력(수정) 취소가 일어나는지 확인해본다.

```
SQL> INSERT INTO ORDER_LIST VALUES('20120901','MULTIPACK',10,300000);
1개의 행이 만들어졌다.

SQL> SELECT * FROM ORDER_LIST;
ORDER_DA    PRODUCT          QTY      AMOUNT
--------    ---------      ------    -------
20120901    MONOPACK         10       300000
20120901    MONOPACK         20       600000
20120901    MULTIPACK        10       300000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
--------    ---------      ------    -------
20120901    MONOPACK         30       900000
20120901    MULTIPACK        10       300000

SQL> ROLLBACK;
롤백이 완료되었다.

SQL> SELECT *  FROM ORDER_LIST;
ORDER_DATE  PRODUCT          QTY      AMOUNT
--------    ---------      ------    -------
20120901    MONOPACK         10       300000
20120901    MONOPACK         20       600000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
--------    ---------      ------    -------
20120901    MONOPACK         30       900000
```

◻ SQL Server

– 관련 테이블 생성

```
CREATE TABLE ORDER_LIST (
    ORDER_DATE    CHAR(8) NOT NULL,
    PRODUCT       VARCHAR(10) NOT NULL,
    QTY           INT NOT NULL,
    AMOUNT        INT NOT NULL);

CREATE TABLE SALES_PER_DATE (
    SALE_DATE     CHAR(8) NOT NULL,
    PRODUCT       VARCHAR(10) NOT NULL,
    QTY           INT NOT NULL,
    AMOUNT        INT NOT NULL);
```

☞ T-SQL에서 deteted와 inserted는 신규/기존 레코드를 갖고있는 구조체

구분	deleted	inserted
INSERT	NULL	입력된 레코드 값
UPDATE	UPDATE되기 전의 레코드의 값	UPDATE된 후의 레코드 값
DELETE	레코드가 삭제되기 전 값	NULL

– Trigger 작성 (T-SQL)

◆ ORDER_LIST 테이블에 주문 정보가 입력되면, 주문 정보의 주문 일자와 주문 상품을 기준으로, 판매 집계 테이블에 해당 주문 일자의 주문 상품 레코드가 존재하면, 판매 수량과 판매 금액을 더하고, 존재하지 않으면 새로운 레코드를 삽입한다.

```
CREATE TRIGGER dbo.SUMMARY_SALES
    ON ORDER_LIST
    AFTER INSERT
AS
DECLARE
    @o_date DATETIME, @o_prod INT, @qty int, @amount int
BEGIN
    SELECT @o_date=order_date, @o_prod=product, @qty=qty,
           @amount=amount
    FROM inserted
    UPDATE SALES_PER_DATE
    SET qty = qty + @qty, amount = amount + @amount
    WHERE sale_date = @o_date AND product = @o_prod;
    IF @@ROWCOUNT=0
        INSERT INTO SALES_PER_DATE
        VALUES(@o_date, @o_prod, @qty, @amount)
END;
```

– ORDER_LIST 테이블에 주문 정보(10개)를 입력한다.

```
SELECT * FROM ORDER_LIST;
선택된 레코드가 없다.

SELECT * FROM SALES_PER_DATE;
선택된 레코드가 없다.

INSERT INTO ORDER_LIST VALUES('20120901', 'MONOPACK', 10, 300000);
1개의 행이 만들어졌다.
```

– 주문 정보와 판매 집계 테이블에 같은 데이터가 들어왔는지 확인한다.

```
SQL> SELECT * FROM ORDER_LIST;
ORDER_DATE  PRODUCT          QTY      AMOUNT
----------  ---------  ---------  --------
20120901    MONOPACK          10      300000

SQL> SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
----------  ---------  ---------  --------
20120901    MONOPACK          10      300000
```

– 다시 한 번 같은 주문 정보(20개)를 입력하고, 확인한다.

```
INSERT INTO ORDER_LIST VALUES('20120901','MONOPACK',20,600000);
1개의 행이 만들어졌다.

SELECT * FROM ORDER_LIST;
ORDER_DATE  PRODUCT          QTY      AMOUNT
----------  ---------- ----------  ----------
20120901    MONOPACK         10      300000
20120901    MONOPACK         20      600000

SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
--------    ---------- ----------  ----------
20120901    MONOPACK         30      900000
```

– 이번에는 다른 상품으로 주문 데이터를 입력한 후 두 테이블의 결과를 조회해 보고 트랜잭션을 ROLLBACK 수행한다.

– 판매 데이터의 입력 취소가 일어나면, 주문 정보 테이블과 판매 집계 테이블에 동시에 입력(수정) 취소가 일어나는지 확인해본다.

```
BEGIN TRAN
INSERT INTO ORDER_LIST VALUES('20120901','MULTIPACK',10,300000);
1개의 행이 만들어졌다.

SELECT * FROM ORDER_LIST;
ORDER_DATE  PRODUCT          QTY      AMOUNT
----------  ----------   ------   -------
20120901    MONOPACK          10      300000
20120901    MONOPACK          20      600000
20120901    MULTIPACK         10      300000

SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
---------   ----------   ------   -------
20120901    MONOPACK          30      900000
20120901    MULTIPACK         10      300000

ROLLBACK;
롤백이 완료되었다.

SELECT *  FROM ORDER_LIST;
ORDER_DATE  PRODUCT          QTY      AMOUNT
----------  ---------    ------   -------
20120901    MONOPACK          10      300000
20120901    MONOPACK          20      600000

SELECT * FROM SALES_PER_DATE;
SALE_DATE   PRODUCT          QTY      AMOUNT
---------   ---------    ------   -------
20120901    MONOPACK          30      900000
```