

# DML

강사 : 강병준

# SQL문의 종류

**DDL (Data Definition Language) : 데이터와 그 구조를 정의 합니다.**

SQL문	내 용
CREATE	데이터베이스 객체를 생성
DROP	데이터베이스 객체를 삭제
ALTER	기존에 존재하는 데이터베이스 객체를 다시 정의하는 역할

**DML (Data Manipulation Language) : 데이터의 검색과 수정등의 처리**

SQL문	내 용
INSERT	데이터베이스 객체에 데이터를 입력
DELETE	데이터베이스 객체에 데이터를 삭제
UPDATE	기존에 존재하는 데이터베이스 객체안의 데이터 수정
SELECT	데이터베이스 객체로부터 데이터를 검색

**DCL (Data Control Language) : 데이터베이스 사용자의 권한을 제어**

SQL문	내 용
GRANT	데이터베이스 객체에 권한을 부여
REVOKE	이미 부여된 데이터베이스객체의 권한을 취소

# 개 요

- INSERT문을 사용한 행(row)의 추가
- 변수 사용하기
- UPDATE문을 사용한 행(row)의 변경
- DELETE문을 사용한 행(row)의 제거
- 데이터베이스 무결성(Integrity)
- MERGE를 사용한 행(row)의 병합(merging)
- 데이터베이스 트랜잭션(transaction)
- 쿼리(query) 회귀(flashbacks)

# INSERT문을 사용한 행(row)의 추가

- 행(row)을 추가하기 위해 알아야 하는 필수 정보 I
  - 행(row)을 추가하고자 하는 테이블명(table name).
  - 추가되는 행(row)이 입력하기를 원하는 테이블의 컬럼명(column name).
  - 입력하기를 원하는 테이블의 컬럼명에 대한 컬럼의 입력값(value).

```
SQL> SELECT *  
2 FROM customers;
```

CUST_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Brown	01-JAN-70	800-555-1215

# INSERT문을 사용한 행(row)의 추가

- 행(row)을 추가하기 위해 알아야 하는 필수 정보 II
  - 행(row)의 추가시 기본키(primary key)나 NOT NULL로 정의된 컬럼(column)은 반드시 입력되어야 한다.

```
SQL> describe customers
```

Name	Null?	Type
-----	-----	-----
CUSTOMER_ID	NOT NULL	NUMBER(38)
FIRST_NAME	NOT NULL	VARCHAR2(10)
LAST_NAME	NOT NULL	VARCHAR2(10)
DOB		DATE
PHONE		VARCHAR2(12)

# 테이블에 새로운 행을 추가

- **INSERT** 문은 테이블에 새로운 데이터를 입력하기 위해 사용하는 데이터 조작어입니다.
- 다음은 **INSERT** 문의 기본 형식입니다.

```
INSERT INTO table_name  
(column_name, ...)  
VALUES(column_value, ...);
```

- 이미 사용하던 테이블과 동일한 이름이 테이블을 새로 만들려면 **DROP TABLE** 명령어로 테이블을 삭제한 후에 기존에 있던 부서 테이블(**DEPT**)과 동일한 구조를 갖되 데이터는 복사하지 않는 부서 테이블(**DEPT01**)을 생성하기 위해서 서브 쿼리문을 사용하여 테이블을 생성합니다.


```
DROP TABLE DEPT01;
```

```
CREATE TABLE DEPT01  
AS SELECT * FROM DEPT WHERE 1=0;
```

# 데이터 삽입

- 새로운 데이터를 추가하기 위해서 사용할 명령어 **INSERT INTO ~ VALUES ~**는 컬럼 명에 기술된 목록의 수와 **VALUES** 다음에 나오는 괄호에 기술한 값의 개수가 같아야 합니다.
- 컬럼 **DEPTNO**에 **10**을, 컬럼 **DNAME**에는 **'ACCOUNTING'**을, 컬럼 **LOC**에는 **'NEW YORK'**을 추가합니다.

```
INSERT INTO DEPT01  
(DEPTNO, DNAME, LOC)  
VALUES(10, 'ACCOUNTING', 'NEW YORK');
```



- **DEPT01**에 데이터를 추가합니다.

```
INSERT INTO DEPT01  
(DEPTNO, DNAME, LOC)  
VALUES(10, 'ACCOUNTING', 'NEW YORK');
```

# INSERT 구문에서 오류

- 컬럼 명에 기술된 목록의 수보다 **VALUES** 다음에 나오는 괄호 안에 기술한 값의 개수가 적으면 에러가 발생합니다.

```
INSERT INTO DEPT01  
(DEPTNO, DNAME, LOC)  
VALUES (10, 'ACCOUNTING');
```

- 컬럼 명에 기술된 목록의 수보다 **VALUES** 다음에 나오는 괄호에 기술한 값의 개수가 많으면 에러가 발생합니다.

```
INSERT INTO DEPT01  
(DEPTNO, DNAME, LOC)  
VALUES(10, 'ACCOUNTING', 'NEW YORK', 20);
```



# INSERT 구문에서 오류

- 컬럼 명이 잘못 입력되었을 때에도 에러가 발생합니다.

```
INSERT INTO DEPT01  
(NUM, DNAME, LOC)  
VALUES(10, 'ACCOUNTING', 'NEW YORK');
```

- 컬럼과 입력할 값의 데이터 타입이 서로 맞지 않을 경우에도 에러가 발생합니다.

```
INSERT INTO DEPT01  
(DEPTNO, DNAME, LOC)  
VALUES(10, ACCOUNTING, 'NEW YORK');
```

# INSERT문을 사용한 행(row)의 추가

## ■ 기본적인 행(row) 추가문장

```
SQL> insert into customers(customer_id, first_name,  
    last_name, dob, phone)  
2 values (6, 'Fred', 'Brown',  
3         '01-JAN-1970', '800-555-1215');
```

## ■ 컬럼 리스트의 생략

- 컬럼 리스트를 생략하는 경우 모든 컬럼의 값을 넣어 주어야 한다.

```
SQL> insert into customers  
2 values (7, 'Jane', 'Green',  
3         '01-JAN-1970', '800-555-1215');
```

```
INSERT INTO DEPT01  
VALUES (20, 'RESEARCH', 'DALLAS');
```

# INSERT문을 사용한 행(row)의 추가

- 널(null)값 기입하기

```
SQL> insert into customers  
2 values (8, 'Sophie', 'White', null, null);
```

- 입력값에 인용부호(quotes)가 포함된 경우

- 단일 인용부호는 두 개의 단일 인용부호를 연속으로 사용하여 표시

```
SQL> insert into customers  
2 values (9, 'Kyle', 'O''Malley', null, null);
```

- 이중 인용부호는 바로 사용 가능

```
SQL> insert into products (product_id,  
    product_type_id, name, description, price)  
2 values (13, 1, 'The "Great" Gatsby', null,  
3      12.99);
```

입력될 컬럼을 INSERT 문장에 기술하지 않는 방법이다  
`INSERT INTO DEPT(DEPTNO, DNAME) VALUES(91, '총무과');`;

두 번째 방법은 NULL 값 대신에 NULL이라는 키워드를 기술하는 방법이다  
`SQL> INSERT INTO DEPT VALUES(92, '경리과', NULL);`  
1 개의 행이 만들어졌습니다.

특수 값 입력

INSERT 문장에 SYSDATE, USER와 같은 특수 함수를 이용하면 테이블에 시스템의 날짜 또는 현재 사용자의 계정을 입력 할 수 있다.

`INSERT INTO EMP`

`(EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)`  
`VALUES(9000, USER, '연구원', 7839, SYSDATE, 5000, NULL, 90);`  
1 개의 행이 만들어졌습니다.

# INSERT문을 사용한 행(row)의 추가

- 한 테이블(table)에서 다른 테이블로 열(row) 복사(copy)하기
  - 컬럼의 입력값(value) 부분에 쿼리(query)를 사용하여 열을 복사한다.
  - 입력되어야 하는 컬럼과 쿼리의 결과 컬럼은 수와 타입이 일치해야 한다.

```
SQL> insert into customers (customer_id,  
    first_name, last_name)  
2  select 10, first_name, last_name  
3     from customers  
4  where customer_id = 1;
```

무결성 제약 조건(5 가지)이 위반되면 INSERT 시 ERROR 가 발생합니다...

```
SQL> INSERT INTO emp..  
  2  VALUES (7788, 'YOON', 'MANAGER', NULL, SYSDATE, NULL, NULL, 10);..  
INSERT INTO emp..  
      *..
```

ERROR at line 1:..

ORA-00001: unique constraint (SCOTT.EMP\_PRIMARY\_KEY) violated..

```
..  
SQL> INSERT INTO emp..  
  2  VALUES (1144, 'YOON', 'MANAGER', NULL, SYSDATE, NULL, NULL, 91);..  
INSERT INTO emp..  
      *..
```

ERROR at line 1:..

ORA-02291: integrity constraint (SCOTT.EMP\_FOREIGN\_KEY) violated - parent key not found..

```
..  
SQL> INSERT INTO emp..  
  2  VALUES (1144, 'YOON', 'MANAGER', 1234, SYSDATE, NULL, NULL, 10);..  
INSERT INTO emp..  
      *..
```

ERROR at line 1:..

ORA-02291: integrity constraint (SCOTT.EMP\_SELF\_KEY) violated - parent key not found..

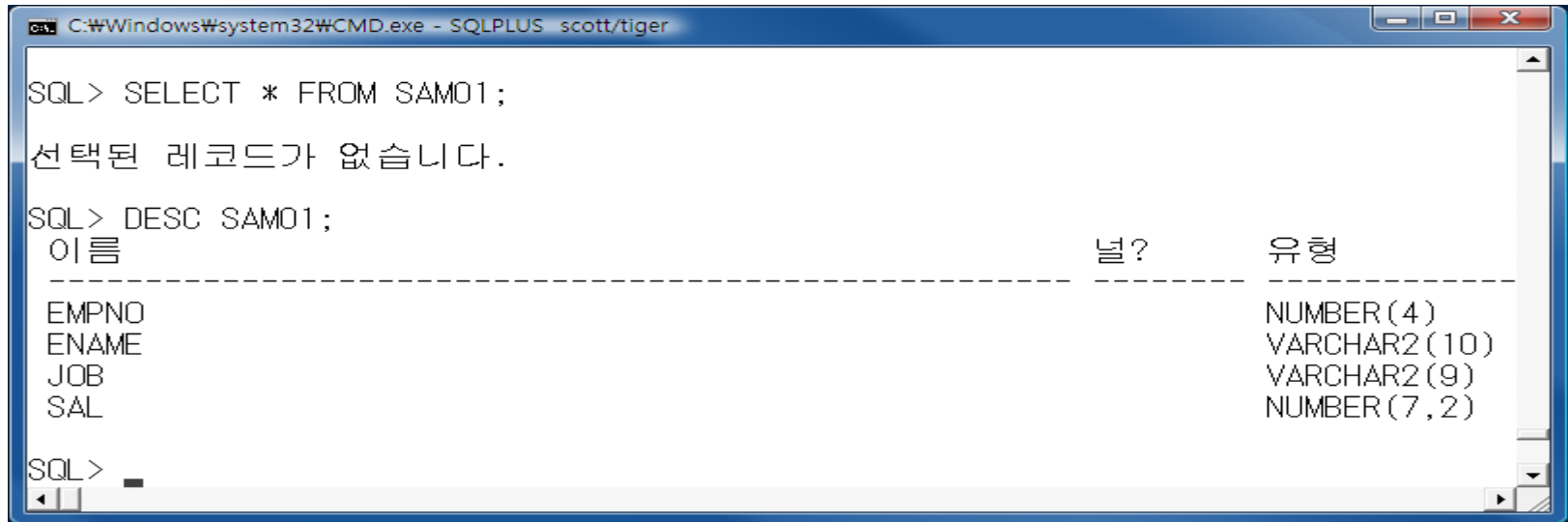
```
..  
SQL> INSERT INTO emp (empno,ename,job)..  
  2  VALUES (1144, 'YOON', 'MANAGER');..  
INSERT INTO emp (empno,ename,job)..  
      *..
```

ERROR at line 1:..

ORA-01400: mandatory (NOT NULL) column is missing or NULL during insert..

# 연습문제

1. 서브 쿼리문을 이용하여 다음과 같은 구조로 **SAM01** 테이블을 생성하시오.  
존재할 경우 **DROP TABLE**로 삭제 후 생성하시오.

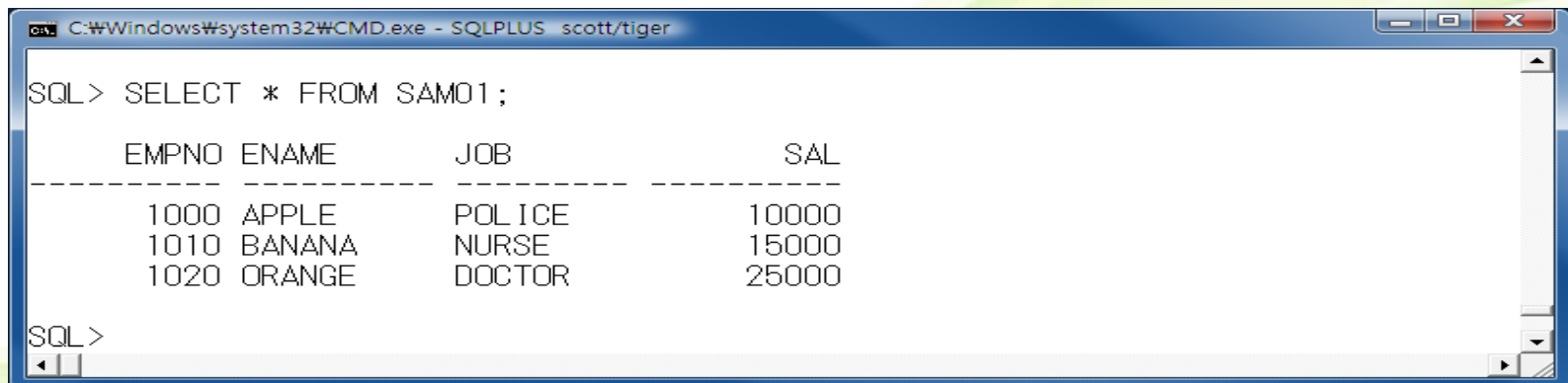


The screenshot shows a SQL\*Plus session with the following commands and output:

```
SQL> SELECT * FROM SAM01;
선택된 레코드가 없습니다.

SQL> DESC SAM01;
이름                                널?       유형
-----
EMPNO                                NUMBER(4)
ENAME                                VARCHAR2(10)
JOB                                  VARCHAR2(9)
SAL                                  NUMBER(7,2)
```

2. **SAM01** 테이블에 다음과 같은 데이터를 추가하시오.

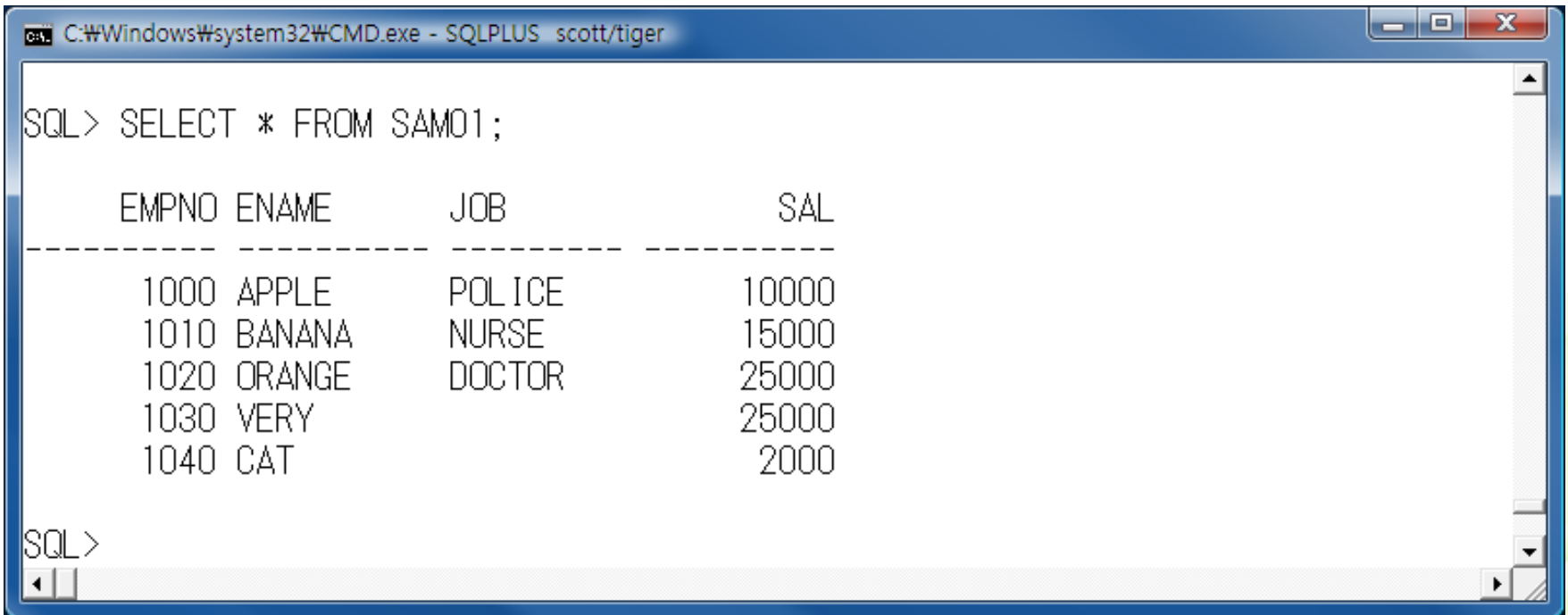


The screenshot shows a SQL\*Plus session with the following command and output:

```
SQL> SELECT * FROM SAM01;
EMPNO  ENAME      JOB          SAL
-----
1000    APPLE      POLICE       10000
1010    BANANA     NURSE        15000
1020    ORANGE     DOCTOR       25000
```

# 연습문제

**SAM01** 테이블에 다음과 같이 **NULL** 값을 갖는 행을 추가하시오.



```
SQL> SELECT * FROM SAM01;
```

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	10000
1010	BANANA	NURSE	15000
1020	ORANGE	DOCTOR	25000
1030	VERY		25000
1040	CAT		20000

```
SQL>
```



# Substitution Variable 사용하기

- 치환 (대체)변수
  - 값을 임시로 저장하기 위해서 SQL\*Plus 치환 변수를 사용
    - Single ampersand(&)
    - Double ampersand(&&)
    - DEFINE과 ACCEPT 명령어
  - SQL 문장간에 변수 값을 전달
  - 변수 지정 형식
    - 숫자는 : &변수명
    - 문자 / 날짜 : '&변수명'

# 변수 활용1

- &

- 특정값이나 컬럼명, 테이블명, WHERE절, ORDER BY절 또는 SELECT절 전체를 치환하여 임시로 저장하기 위해 사용

- &&

- 저장된 값을 해제하지 않는 한 해당 세션 동안 계속 저장

# & 를 이용한 변수 활용 예제

- &를 이용하여 변수 값 임시 저장
- Set verify on/off sqlplus에서 작동

SQL> insert into dept

values (&부서번호, '&부서명', '서울');

부서번호의 값을 입력하십시오: 55

부서명의 값을 입력하십시오: 경리부

구 2: values (&부서번호, '&부서명', '서울')

신 2: values (55, '경리부', '서울')

SET Verify ON

1 개의 행이 만들어졌습니다.

# &&를 이용한 변수 활용 예제

- &&를 이용하여 변수 값 저장

```
SQL> insert into dept  
      values (&부서번호, '&&부서명', '서울');
```

부서번호의 값을 입력하십시오: 55

부서명의 값을 입력하십시오: 경리부

구 2: values (&부서번호, '&&부서명', '서울')

신 2: values (55, '경리부', '서울')

1 개의 행이 만들어졌습니다.

```
SQL> define    -- 저장된 변수 값 확인 가능!
```

## ■ Define

- DEFINE에 의해 변수값 확인 및 설정 가능
- UNDEFINE에 의해 정의된 변수값 해제 가능

## ■ Accept

- 입력 라인을 읽고 주어진 변수에 값을 저장하는 기능
- 입력 내용을 숨길 수 있음
- 입력 값의 데이터 타입을 지정 가능
- 사용자가 입력받을 때 사용자가 원하는 프롬프트를 생성함

# Define을 이용한 변수 활용 예제

- Define를 이용하여 변수 값 지정 및 확인

```
SQL> define
```

```
DEFINE _CONNECT_IDENTIFIER = "ora9200" (CHAR)
```

```
DEFINE _SQLPLUS_RELEASE = "902000100" (CHAR)
```

```
DEFINE _EDITOR          = "Notepad" (CHAR)
```

```
DEFINE _O_VERSION       = "Oracle9i Enterprise  
Edition Release 9.2.0.1.0 – Production”
```

```
With the Partitioning, OLAP and Oracle Data Mining options  
JServer Release 9.2.0.1.0 - Production" (CHAR)
```

```
DEFINE _O_RELEASE       = "902000100" (CHAR)
```

```
DEFINE _RC               = "1" (CHAR)
```

- &&로 저장된 값이 있다면 define에서 변수명과 값 확인 가능
- 저장된 변수값 해제 : **undefine** 변수명

# Accept를 이용한 변수 활용 예제

- Accept를 이용하여 변수 지정 형식 지정

**ACCEPT** variable [datatype] [FORMAT format] [PROMPT string] {hide}

- **datatype** : char, number, date 데이터 타입 중 하나를 선택, 기본 값은 char
- **format** : 문자상수 형태이며 데이터 타입이 date일 경우 'yyyy/mm/dd' 형태도 가능
- **prompt** : 변수 값을 입력받기 전에 문자열을 보여줌
- **hide** : 입력하는 값이 화면에 보이지 않도록 함

```
SQL> ACCEPT department PROMPT 'Enter the department name : '  
Enter the department name : RESEARCH
```

☞ 출력 결과 확인

```
SQL> SELECT * FROM dept  
WHERE dname=UPPER('&department');
```

# 서브 쿼리로 데이터 삽입

- **INSERT INTO** 다음에 **VALUES** 절을 사용하는 대신에 서브 쿼리를 사용할 수 있습니다.
- 이렇게 하면 기존의 테이블에 있던 여러 행을 복사해서 다른 테이블에 삽입할 수 있습니다.
- 이 때 주의할 점은 **INSERT** 명령문에서 지정한 컬럼의 개수나 데이터 타입이 서브 쿼리를 수행한 결과와 동일해야 한다는 점입니다.

1. 서브 쿼리로 데이터 삽입하기 위해서 우선 테이블을 생성하되 데이터는 복사하지 않고 빈 테이블만 생성합니다.

```
DROP TABLE DEPT02;  
CREATE TABLE DEPT02  
AS  
SELECT * FROM DEPT WHERE 1=0;
```

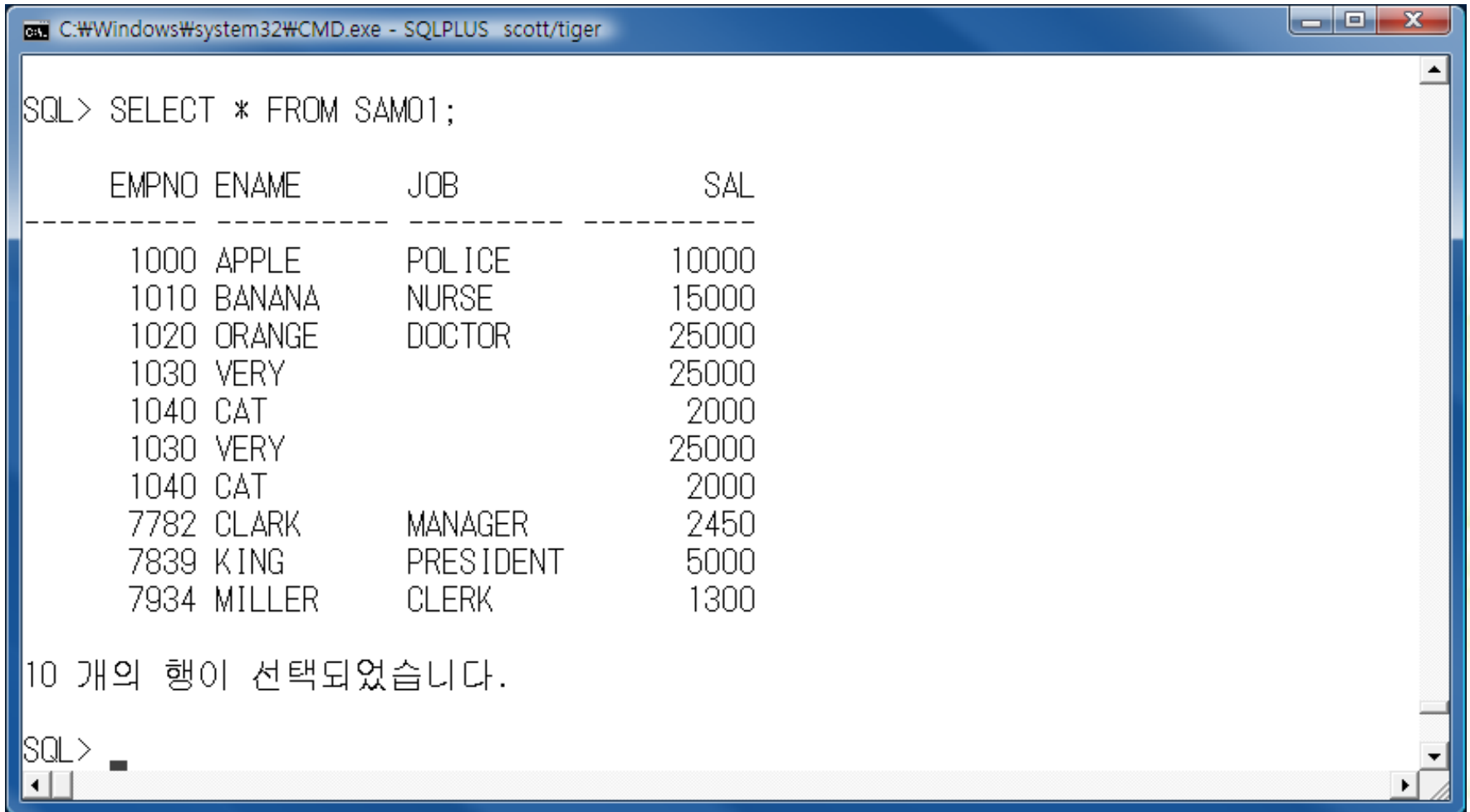
2. 테이블 구조만을 복사해서 내용을 갖지 않는 테이블에 서브 쿼리로 로우를 입력해 보시다.

```
INSERT INTO DEPT02  
SELECT * FROM DEPT;
```



# 연습문제

**SAM01** 테이블에 서브 쿼리문을 사용하여 **EMP** 에 저장된 사원 중 **deptno**가 **10**번 사원의 정보를 추가하시오.



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger". The user has entered the SQL command "SELECT \* FROM SAM01;". The output displays a table with four columns: EMPNO, ENAME, JOB, and SAL. The table contains 10 rows of data. Below the table, a message states "10 개의 행이 선택되었습니다." (10 rows selected). The prompt "SQL>" is visible at the bottom.

```
SQL> SELECT * FROM SAM01;
```

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	10000
1010	BANANA	NURSE	15000
1020	ORANGE	DOCTOR	25000
1030	VERY		25000
1040	CAT		2000
1030	VERY		25000
1040	CAT		2000
7782	CLARK	MANAGER	2450
7839	KING	PRESIDENT	5000
7934	MILLER	CLERK	1300

10 개의 행이 선택되었습니다.

```
SQL>
```

# 다중 테이블에 다중 행 입력

- **INSERT ALL**문을 사용하면 서브 쿼리의 결과를 조건 없이 여러 테이블에 동시에 입력할 수 있습니다.
- 사원번호, 사원명, 입사일자로 구성된 **EMP\_HIR** 테이블과 사원번호, 사원명, 해당관리자(상관))로 구성된 **EMP\_MGR** 테이블이 빈 테이블로 존재한다고 합시다.
- 사원 테이블(**EMP**)에서 부서 번호가 **20**인 사원들을 검색하여 **EMP\_HIR** 테이블에는 사원 번호, 사원 명, 급여를 **EMP\_MGR** 테이블에는 사원 번호, 사원 명, 해당관리자(상관)를 입력하고자 할 때 **INSERT ALL** 명령문을 사용하면 두 번의 쿼리문을 수행하지 않고도 하나의 쿼리문으로 두 개의 테이블에 원하는 컬럼 값을 삽입할 수 있습니다.
- **INSERT ALL** 명령문은 서브 쿼리의 결과 집합을 조건 없이 여러 테이블에 동시에 입력하기 위한 명령문입니다.
- 이때 주의할 점은 서브 쿼리의 컬럼명과 데이터가 입력되는 테이블의 컬럼명이 동일해야 한다는 점입니다.

```
INSERT ALL
  INTO EMP_HIR VALUES(EMPNO, ENAME, HIREDATE)
  INTO EMP_MGR VALUES(EMPNO, ENAME, MGR)
  SELECT EMPNO, ENAME, HIREDATE, MGR
  FROM EMP
 WHERE DEPTNO=20;
```

# 조건(WHEN)에 의해 다중 테이블에 다중 행 입력

- **INSERT ALL** 명령문에 **WHEN** 절을 추가해서 조건을 제시하여 조건에 맞는 행만 추출하여 테이블에 추가합니다.
- **EMP\_HIR02** 테이블에는 **1982** 년 **01** 월**01** 일 이후에 입사한 사원들의 번호, 사원 명, 입사일을 추가합니다.
- **EMP\_SAL** 테이블에는 급여가 **2000** 이상인 사원들의 번호, 사원 명, 급여를 추가합니다.

```
INSERT ALL
WHEN HIREDATE > '1982/01/01' THEN
  INTO EMP_HIR02 VALUES(EMPNO, ENAME,
    HIREDATE)
WHEN SAL > 2000 THEN
  INTO EMP_SAL VALUES(EMPNO, ENAME, SAL)
SELECT EMPNO, ENAME, HIREDATE, SAL FROM EMP;
```

# 실습하기

**PIVOTING INSERT**문을 실습하기 위해 한 주차 월요일부터 금요일까지 매일 매일의 판매 실적을 기록하는 테이블을 생성해 봅니다.

```
CREATE TABLE SALES(  
  SALES_ID NUMBER(4),  
  WEEK_ID NUMBER(4),  
  MON_SALES NUMBER(8, 2),  
  TUE_SALES NUMBER(8, 2),  
  WED_SALES NUMBER(8, 2),  
  THU_SALES NUMBER(8, 2),  
  FRI_SALES NUMBER(8, 2));
```

# 실습하기

2. **PIVOTING INSERT**문의 결과를 저장할 **SALES\_DATA** 테이블을 생성합니다.

```
CREATE TABLE SALES_DATA(  
    SALES_ID NUMBER(4),  
    WEEK_ID NUMBER(4),  
    DAILY_ID NUMBER(4),  
    SALES NUMBER(8, 2));
```

3. 새롭게 생성된 **SALES** 테이블에 주간 판매 실적을 추가합니다.

```
INSERT INTO SALES VALUES  
(1001, 1, 200, 100, 300, 400, 500);  
INSERT INTO SALES VALUES  
(1002, 2, 100, 300, 200, 500, 350);
```

# 실습하기

각 요일을 구분할 수 있는 컬럼을 추가하여 매일 매일의 판매 실적을 기록해봅니다.

```
INSERT ALL
  INTO SALES_DATA VALUES(SALES_ID, WEEK_ID, 1,
    MON_SALES)
  INTO SALES_DATA VALUES(SALES_ID, WEEK_ID, 2,
    TUE_SALES)
  INTO SALES_DATA VALUES(SALES_ID, WEEK_ID, 3,
    WED_SALES)
  INTO SALES_DATA VALUES(SALES_ID, WEEK_ID, 4,
    THU_SALES)
  INTO SALES_DATA VALUES(SALES_ID, WEEK_ID, 5,
    FRI_SALES)
SELECT SALES_ID, WEEK_ID, MON_SALES,
  TUE_SALES, WED_SALES,
  THU_SALES, FRI_SALES
FROM SALES;
```

# 테이블의 내용을 수정

- **UPDATE** 문은 테이블에 저장된 데이터를 수정하기 위해서 사용합니다.

```
UPDATE table_name  
SET column_name1 = value1, column_name2 = value2, ...  
WHERE conditions;
```

- **UPDATE** 문은 기존의 행을 수정하는 것입니다. 따라서 어떤 행의 데이터를 수정하는지 **WHERE** 절을 이용하여 조건을 지정합니다.
- **WHERE** 절을 사용하지 않을 경우는 테이블에 있는 모든 행이 수정됩니다.
- 테이블의 전체 행을 수정하려고 했던 것이 아니라면 **WHERE** 절의 사용 유무를 신중히 판단하여야 합니다.

# UPDATE문을 사용한 행(row)의 변경

- 행(row)을 변경하기 위해 알아야 하는 필수 정보
  - 행(row)을 변경하고자 하는 테이블명(table name).
  - 변경되는 행(row)을 명시하는 조건절(when clause).
  - 정의절(set clause)을 사용하여 명시되는 컬럼명과 변경 값.
- 기본적인 행(row) 변경 문장

```
SQL> update customers
2  set last_name = 'Orange'
3  where customer_id = 2;
```

```
SQL> update products
2  set price = price * 1.20,
   name = LOWER(name)
3  where price >= 20;
```



# UPDATE문을 사용한 행(row)의 변경

- 반환 절(returning clause)
  - 변경된 값에 대하여 집계 함수(aggregate function)를 이용한 단일 값을 반환 받을 수 있다.
  - 조건절(where clause)에 명시된 범위가 반환되는 값의 대상 범위이다.
  - 데이터 조작 문장(DML)에서 사용 가능

```
SQL> variable average_product_price NUMBER
2  update products
2  set price = price * 0.75
3  returning avg(price) into
      :average_product_price;
4  rollback;
5  print average_product_price
```

# 실습하기

1. 모든 사원의 부서번호를 **30**번으로 수정합시다.

```
UPDATE EMP01  
SET DEPTNO=30;
```

2. 이번엔 모든 사원의 급여를 **10%** 인상시키는 **UPDATE** 문을 보겠습니다.

```
UPDATE EMP01  
SET SAL = SAL * 1.1;
```

3. 모든 사원의 입사일을 오늘로 수정하려면 다음과 같이 합니다.

```
UPDATE EMP01  
SET HIREDATE = SYSDATE;
```

# 테이블의 특정 행만 변경

- **UPDATE** 문에 **WHERE** 절을 추가하면 테이블의 특정 행이 변경됩니다.
- 사원 테이블(**EMP01**)을 제거한 후 다시 기존에 있던 사원 테이블(**EMP**)과 동일한 구조와 데이터를 갖는 사원 테이블(**EMP01**)을 생성합니다.

1. 부서번호가 **10**번인 사원의 부서번호를 **30**번으로 수정합니다.

```
UPDATE EMP01  
SET DEPTNO=30  
WHERE DEPTNO=10;
```

2. 급여가 **3000** 이상인 사원만 급여를 **10%** 인상합니다.

```
UPDATE EMP01  
SET SAL = SAL * 1.1  
WHERE SAL >= 3000;
```

# 실습하기

**1982**년에 입사한 사원의 입사일을 오늘로 수정합니다. 사원의 입사일을 오늘로 수정한 후에 테이블 내용을 살펴봅시다.

```
UPDATE EMP01  
SET HIREDATE = SYSDATE  
WHERE SUBSTR(HIREDATE, 1, 2)='82';
```

# 연습문제

**SAM01** 테이블에 저장된 사원 중 급여가 **10000** 이상인 사원들의 급여만 **5000**원씩 삭감 하시오.

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> SELECT * FROM SAM01;
```

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	10000
1010	BANANA	NURSE	15000
1020	ORANGE	DOCTOR	25000
1030	VERY		25000
1040	CAT		2000
1030	VERY		25000
1040	CAT		2000
7782	CLARK	MANAGER	2450
7839	KING	PRESIDENT	5000
7934	MILLER	CLERK	1300

10 개의 행이 선택되었습니다.

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> SELECT * FROM SAM01;
```

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	5000
1010	BANANA	NURSE	10000
1020	ORANGE	DOCTOR	20000
1030	VERY		20000
1040	CAT		2000
1030	VERY		20000
1040	CAT		2000
7782	CLARK	MANAGER	2450
7839	KING	PRESIDENT	5000
7934	MILLER	CLERK	1300

10 개의 행이 선택되었습니다.

## 2개 이상의 컬럼 값 변경

테이블에서 하나의 컬럼이 아닌 복수 개 컬럼의 값을 변경하려면 기존 **SET** 절에 콤마를 추가하고 컬럼=값을 추가 기술하면 됩니다.

1. **SCOTT** 사원의 부서번호는 **20**번으로, 직급은 **MANAGER**로 한꺼번에 수정하도록 합시다.

```
UPDATE EMP01  
SET DEPTNO=20, JOB='MANAGER'  
WHERE ENAME='SCOTT';
```

2. **SCOTT** 사원의 입사일자는 오늘로, 급여를 **50** 으로 커미션을 **4000** 으로 수정합시다.

```
UPDATE EMP01  
SET HIREDATE = SYSDATE, SAL=50, COMM=4000  
WHERE ENAME='SCOTT';
```

# 서브 쿼리를 이용한 데이터 수정

UPDATE 문의 SET 절에서 서브 쿼리를 기술하면 서브 쿼리를 수행한 결과로 내용이 변경됩니다.

이러한 방법으로 다른 테이블에 저장된 데이터로 해당 컬럼 값을 변경할 수 있습니다

■ 서브 쿼리를 이용한 복수 컬럼 변경

UPDATE 문장에서 여러 개의 컬럼을 한 번에 변경 할 수 있으며 수도 있다.

```
SQL> UPDATE EMP SET
```

```
    JOB = (SELECT JOB FROM EMP WHERE EMPNO = 7900),
```

```
    SAL = (SELECT SAL FROM EMP WHERE EMPNO = 7844)
```

```
WHERE EMPNO = 9000;
```

1 행이 갱신 되었습니다.

20번 부서의 지역명을 40번 부서의 지역명으로 변경하기 위해서 서브 쿼리문을 사용해 봅니다.

```
UPDATE DEPT01 SET LOC=(SELECT LOC  
                        FROM DEPT01 WHERE DEPTNO=40)
```

```
WHERE DEPTNO=20;
```

# 서브 쿼리를 이용한 데이터 수정

- 다른 테이블을 기반으로 테이블의 행 변경

UPDATE 문장의 서브 쿼리에서 참조하는 테이블과 변경하고자 하는 테이블이 다를 수 도 있다.

```
SQL> UPDATE EMP10 SET  
      JOB = (SELECT JOB FROM EMP WHERE EMPNO = 7844)  
      WHERE EMPNO = (SELECT EMPNO FROM EMP WHERE  
                     ENAME = 'MILLER');
```

1 행이 갱신되었습니다



# 서브 쿼리를 이용한 데이터 수정

```
UPDATE table1 alias1
SET column = (SELECT expression
                FROM table2 alias2
                WHERE alias1.column = alias2.column);
```

상관관계 UPDATE 명령을 연습하기 위해 다음과 같이 테이블에 DNAME 컬럼을 추가한다.

```
ALTER TABLE EMP
ADD DNAME VARCHAR2(14);
```

부서 테이블을 기반으로 사원 테이블의 DNAME 컬럼에 부서명을 입력하는 방법은 다음과 같다.

```
SQL> UPDATE EMP E
2   SET E.DNAME = (SELECT DNAME
3                   FROM DEPT
4                   WHERE DEPTNO = E.DEPTNO);
```

14 행이 갱신되었습니다.

# 연습문제

서브 쿼리문을 사용하여 **EMP** 테이블의 저장된 데이터의 특정 컬럼만으로 구성된 **SAM02** 테이블을 생성하시오.

```
C:\Windows\system32\cmd.exe - sqlplus scott/tiger

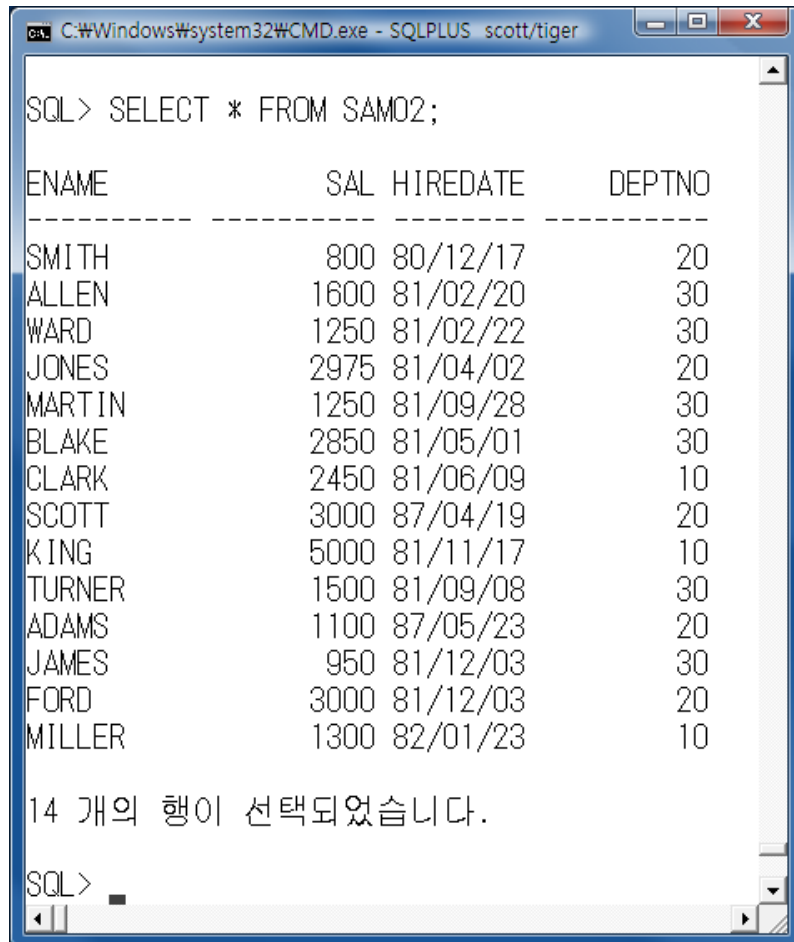
SQL> DESC SAM02
이름                                년?                                유형
-----
ENAME                               VARCHAR2(10)
SAL                                 NUMBER(7,2)
HIREDATE                            DATE
DEPTNO                              NUMBER(2)

SQL>
```

# 연습문제

생성 후 **DALLAS** 에 위치한 부서 소속 사원들의 급여를 **1000** 인상하시오.

■ [변경 전]



C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

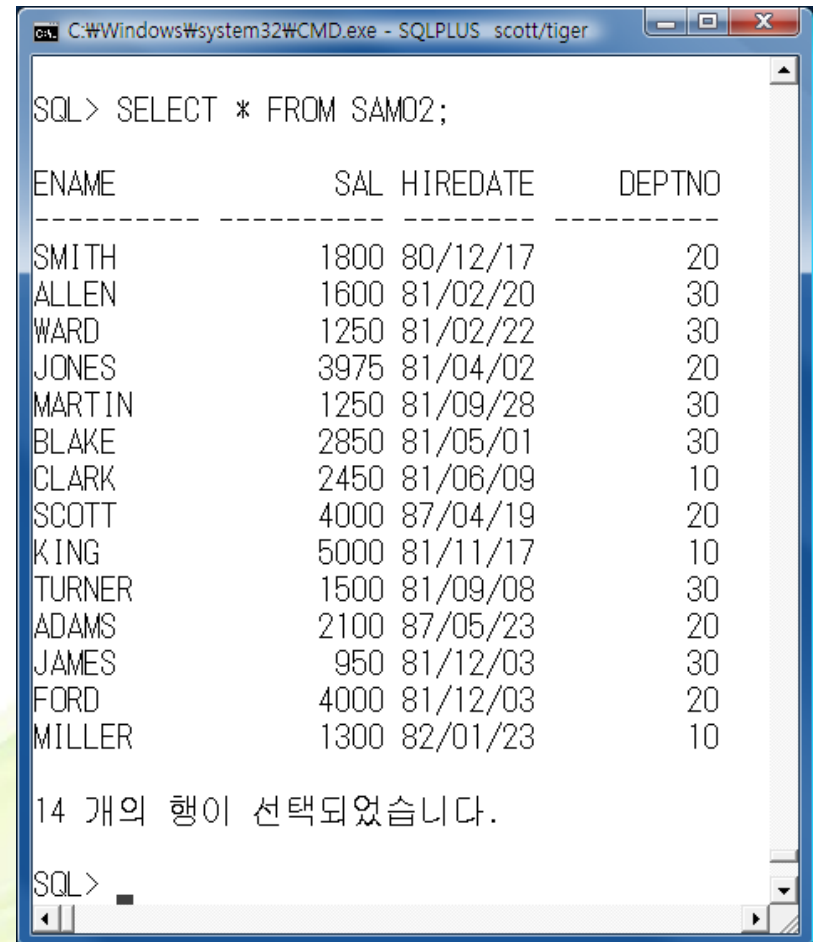
```
SQL> SELECT * FROM SAM02;
```

ENAME	SAL	HIREDATE	DEPTNO
SMITH	800	80/12/17	20
ALLEN	1600	81/02/20	30
WARD	1250	81/02/22	30
JONES	2975	81/04/02	20
MARTIN	1250	81/09/28	30
BLAKE	2850	81/05/01	30
CLARK	2450	81/06/09	10
SCOTT	3000	87/04/19	20
KING	5000	81/11/17	10
TURNER	1500	81/09/08	30
ADAMS	1100	87/05/23	20
JAMES	950	81/12/03	30
FORD	3000	81/12/03	20
MILLER	1300	82/01/23	10

14 개의 행이 선택되었습니다.

SQL>

[변경 후]



C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

```
SQL> SELECT * FROM SAM02;
```

ENAME	SAL	HIREDATE	DEPTNO
SMITH	1800	80/12/17	20
ALLEN	1600	81/02/20	30
WARD	1250	81/02/22	30
JONES	3975	81/04/02	20
MARTIN	1250	81/09/28	30
BLAKE	2850	81/05/01	30
CLARK	2450	81/06/09	10
SCOTT	4000	87/04/19	20
KING	5000	81/11/17	10
TURNER	1500	81/09/08	30
ADAMS	2100	87/05/23	20
JAMES	950	81/12/03	30
FORD	4000	81/12/03	20
MILLER	1300	82/01/23	10

14 개의 행이 선택되었습니다.

SQL>

# 연습문제

서브 쿼리를 이용해서 부서번호가 **20**인 부서의 부서명과 지역명을 부서번호가 **40**번인 부서와 동일하게 변경하도록 해 봅니다.

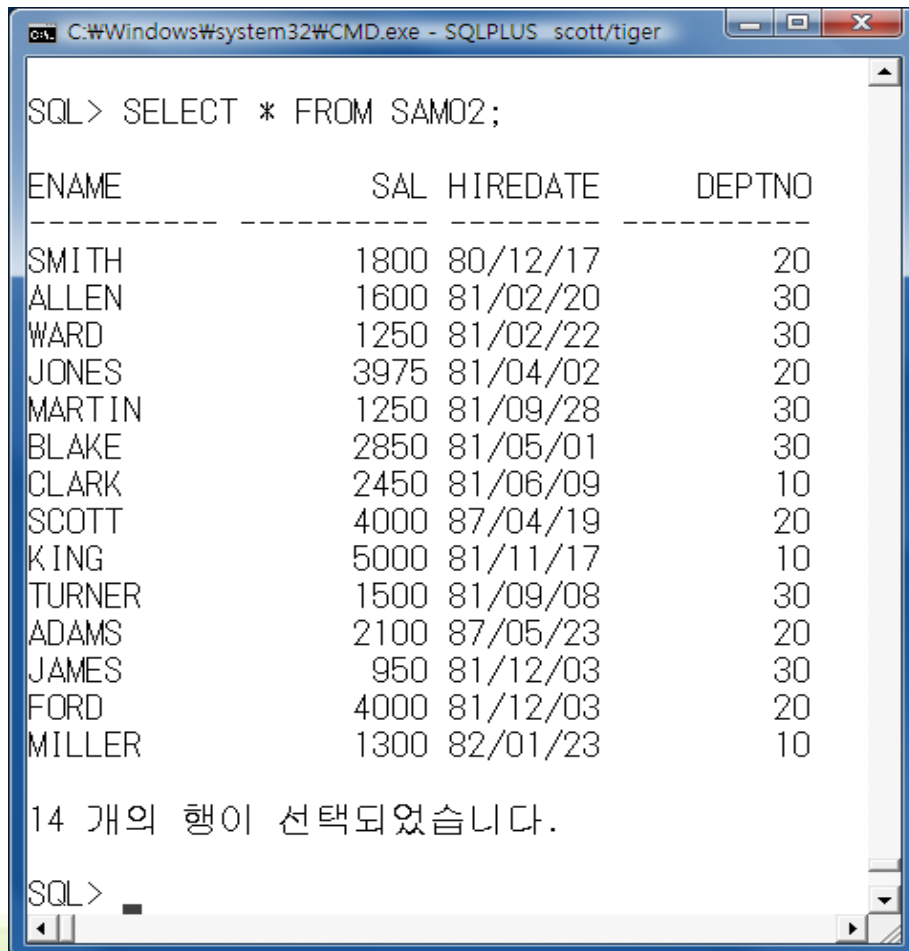
1. 부서 번호가 **20**번인 부서의 이름과 지역은 **RESEARCH**와 **DALLAS**입니다. 다음은 부서번호가 **20**인 부서의 부서명과 지역명을 부서 번호가 **40**번인 부서와 동일하게 변경하기 위한 **UPDATE** 명령문입니다.

```
UPDATE DEPT01
SET (DNAME, LOC)=(SELECT DNAME, LOC
                    FROM DEPT
                    WHERE DEPTNO=40)
WHERE DEPTNO=20;
```

# 연습문제

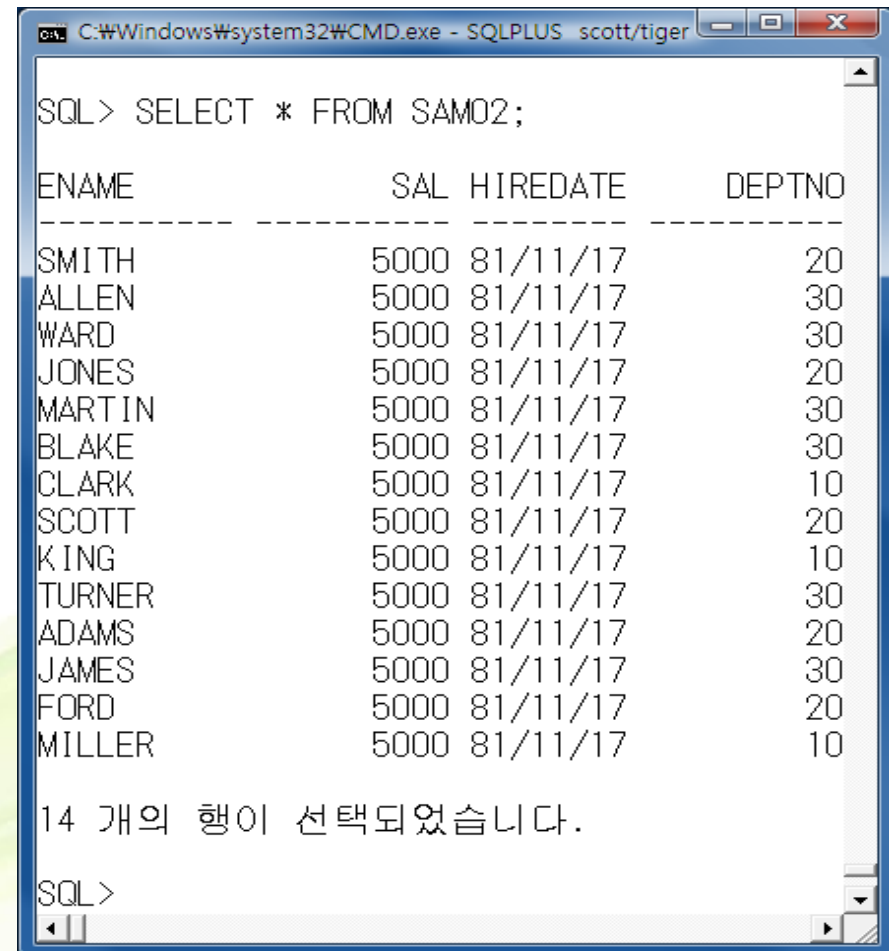
서브 쿼리문을 사용하여 **SAM02** 테이블의 모든 사원의 급여와 입사일을 이름이 **KING** 인 사원의 급여와 입사일로 변경하시오.

[변경 전]



ENAME	SAL	HIREDATE	DEPTNO
SMITH	1800	80/12/17	20
ALLEN	1600	81/02/20	30
WARD	1250	81/02/22	30
JONES	3975	81/04/02	20
MARTIN	1250	81/09/28	30
BLAKE	2850	81/05/01	30
CLARK	2450	81/06/09	10
SCOTT	4000	87/04/19	20
KING	5000	81/11/17	10
TURNER	1500	81/09/08	30
ADAMS	2100	87/05/23	20
JAMES	950	81/12/03	30
FORD	4000	81/12/03	20
MILLER	1300	82/01/23	10

[변경 후]



ENAME	SAL	HIREDATE	DEPTNO
SMITH	5000	81/11/17	20
ALLEN	5000	81/11/17	30
WARD	5000	81/11/17	30
JONES	5000	81/11/17	20
MARTIN	5000	81/11/17	30
BLAKE	5000	81/11/17	30
CLARK	5000	81/11/17	10
SCOTT	5000	81/11/17	20
KING	5000	81/11/17	10
TURNER	5000	81/11/17	30
ADAMS	5000	81/11/17	20
JAMES	5000	81/11/17	30
FORD	5000	81/11/17	20
MILLER	5000	81/11/17	10

# 행을 삭제

## DELETE 문장

- 1) DELETE 문장을 사용하여 테이블로부터 기존의 자료를 삭제할 수 있다...
- 2) WHERE 절을 명시하여 특정 행이나 행들을 삭제할 수 있다...
- 3) WHERE 절을 생략하면 테이블의 모든 행이 삭제 된다...

## Syntax

```
DELETE [FROM] table_name,  
[WHERE condition];
```

EMP 테이블에서 사원번호가 7499 인 사원의 정보를 삭제하여라...

```
SQL> DELETE emp;  
2  WHERE empno = 7499;  
..  
1 row deleted..
```

EMP 테이블에서 입사일자가 83 년인 사원의 정보를 삭제하여라...

```
SQL> DELETE emp;  
2  WHERE TO_CHAR(hiredate, 'YY') = '83';  
..  
1 row deleted..
```

# DELETE문을 사용한 행(row)의 삭제

- 행(row)을 삭제하기 위해 알아야 하는 필수 정보
  - 행(row)을 삭제하고자 하는 테이블명(table name).
  - 삭제되는 행(row)을 명시하는 조건절(where clause).
  - 조건절(where clause)이 없는 경우 모든 열(row)이 삭제되므로 주의.
- 기본적인 행(row) 삭제 문장

```
SQL> delete from customers  
2 where customer_id = 2;
```

```
SQL> delete from customers;
```

*All delete - attention*

# 행을 삭제

- **DELETE** 문은 테이블에 저장되어 있는 데이터를 삭제합니다.

```
DELETE FROM table_name  
WHERE conditions;
```

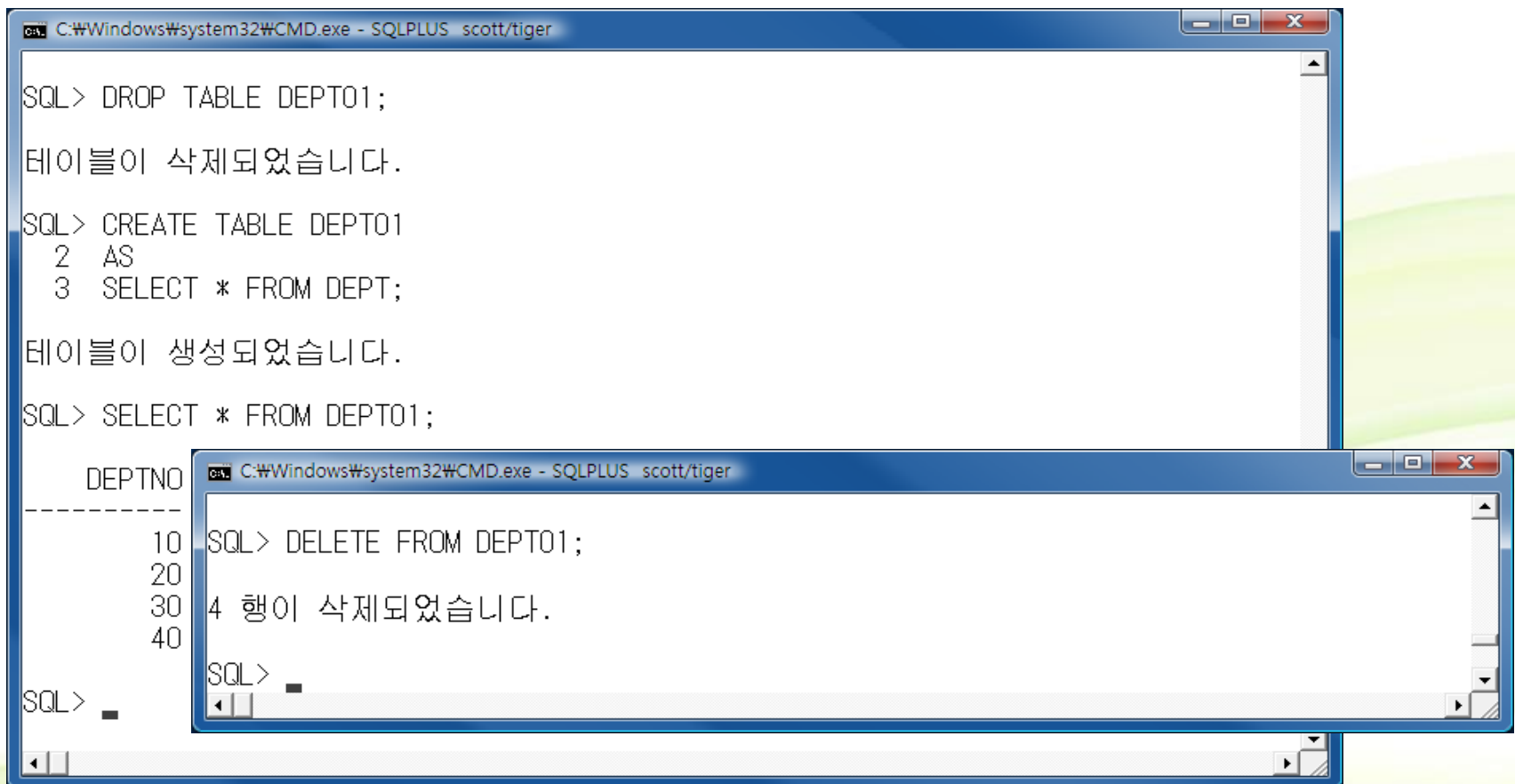
- **DELETE** 문은 테이블의 기존 행을 삭제하며 특정한 로우(행)을 삭제하기 위해서 **WHERE** 절을 이용하여 조건을 지정합니다.
- 만약 **DELETE** 문에 **WHERE** 절을 사용하지 않을 경우 테이블에 있는 모든 행이 삭제되므로 매우 신중하게 명령문을 사용해야 합니다.



# 실습하기

- **DELETE** 문으로 부서 테이블의 모든 행을 삭제합니다.

```
DELETE FROM DEPT01;
```



The image shows two overlapping SQL\*Plus command windows. The top window shows the process of creating a table DEPT01 with 4 rows of data. The bottom window shows the deletion of all data from DEPT01.

**Top Window:**

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> DROP TABLE DEPT01;
테이블이 삭제되었습니다.

SQL> CREATE TABLE DEPT01
  2 AS
  3 SELECT * FROM DEPT;
테이블이 생성되었습니다.

SQL> SELECT * FROM DEPT01;
```

DEPTNO
10
20
30
40

**Bottom Window:**

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

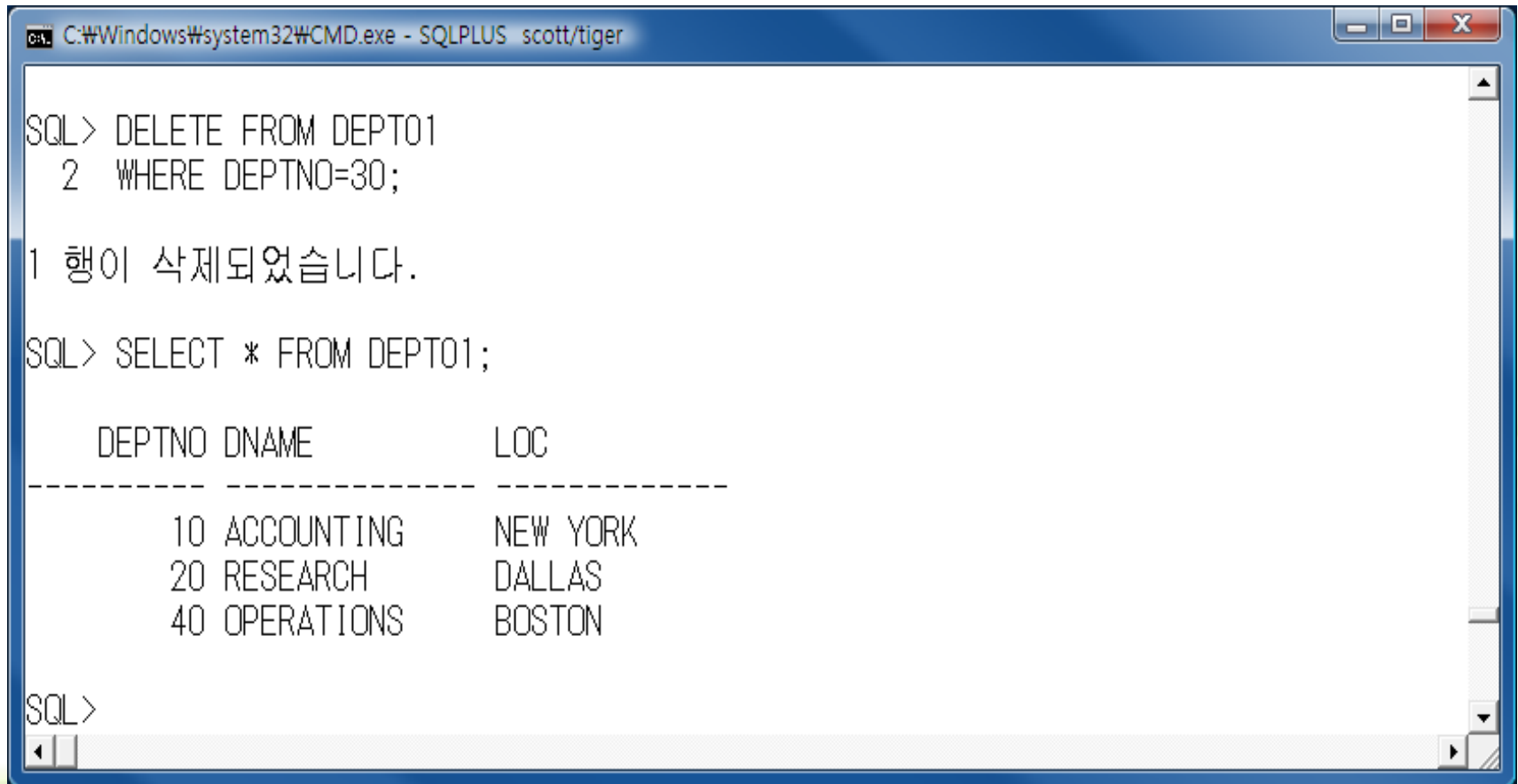
SQL> DELETE FROM DEPT01;
4 행이 삭제되었습니다.

SQL>
```

# 실습하기

- 부서 테이블에서 **30번** 부서만 삭제합니다.

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



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger". The user has entered the following SQL commands:

```
SQL> DELETE FROM DEPT01  
2  WHERE DEPTNO=30;  
  
1 행이 삭제되었습니다.  
  
SQL> SELECT * FROM DEPT01;
```

The output of the second command is a table with three columns: DEPTNO, DNAME, and LOC. The table contains three rows of data:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
40	OPERATIONS	BOSTON

The command prompt shows the prompt "SQL>" at the bottom, indicating the user is ready to enter another command.

# 연습문제

## 9. SAM01 테이블에서 직급이 정해지지 않은 사원을 삭제 하시오.

[삭제 전]

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> SELECT * FROM SAM01;
```

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	5000
1010	BANANA	NURSE	10000
1020	ORANGE	DOCTOR	20000
1030	VERY		20000
1040	CAT		2000
1030	VERY		20000
1040	CAT		2000
7782	CLARK	MANAGER	2450
7839	KING	PRESIDENT	5000
7934	MILLER	CLERK	1300

10 개의 행이 선택되었습니다.

[삭제 후]

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> SELECT * FROM SAM01;
```

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	5000
1010	BANANA	NURSE	10000
1020	ORANGE	DOCTOR	20000
7782	CLARK	MANAGER	2450
7839	KING	PRESIDENT	5000
7934	MILLER	CLERK	1300

6 개의 행이 선택되었습니다.

```
SQL>
```

# 서브 쿼리를 이용한 삭제

- **DELETE** 문을 사용하기에 앞서 사원 테이블을 복사합시다. 사원 테이블에서 부서명이 **SALES**인 사원을 모두 삭제해보도록 하겠습니다.
  - 사원 테이블에는 부서명이 기록되어 있지 않습니다.
  - 부서명은 부서 테이블에 기록되어 있으므로 부서 테이블에서 부서명이 **SALES**인 부서의 번호부터 알아내야 합니다.
  - 이렇게 알아낸 부서번호를 사원 테이블에 적용하기 위해서는 서브 쿼리를 이용해야 합니다.
- 
- 사원 테이블에서 부서명이 **SALES**인 사원을 모두 삭제해봅니다.

```
DELETE FROM EMP01
WHERE DEPTNO=(SELECT DEPTNO
                FROM DEPT
                WHERE DNAME='SALES');
```

# 연습문제

10. SAM02 테이블에서 RESEARCH 부서 소속 직원들만 삭제 하시오.

[변경 전]

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> SELECT * FROM SAM02;

ENAME          SAL  HIREDATE    DEPTNO
-----
SMITH          5000  81/11/17      20
ALLEN          5000  81/11/17      30
WARD           5000  81/11/17      30
JONES          5000  81/11/17      20
MARTIN         5000  81/11/17      30
BLAKE          5000  81/11/17      30
CLARK          5000  81/11/17      10
SCOTT          5000  81/11/17      20
KING           5000  81/11/17      10
TURNER         5000  81/11/17      30
ADAMS          5000  81/11/17      20
JAMES          5000  81/11/17      30
FORD           5000  81/11/17      20
MILLER         5000  81/11/17      10

14 개의 행이 선택되었습니다.

SQL>
```

[변경 후]

```
C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger

SQL> SELECT * FROM SAM02;

ENAME          SAL  HIREDATE    DEPTNO
-----
ALLEN          5000  81/11/17      30
WARD           5000  81/11/17      30
MARTIN         5000  81/11/17      30
BLAKE          5000  81/11/17      30
CLARK          5000  81/11/17      10
KING           5000  81/11/17      10
TURNER         5000  81/11/17      30
JAMES          5000  81/11/17      30
MILLER         5000  81/11/17      10

9 개의 행이 선택되었습니다.

SQL>
```

# 데이터베이스 무결성(Integrity)

- 무결성(integrity)
  - 어떠한 테이블 열(row)의 변경에도 기본 키(primary key)와 외래 키(foreign key)의 관계(relationship)를 유지하는 것.
- 기본 키 제약(primary key constraints)의 시행(enforcement)
  - 기본 키는 테이블에서 열을 구별 하는데 사용.
  - 최소 단위인 열의 구분을 위해 기본 키의 유일성(unique)을 보장하는 것.
  - 기본 키는 반드시 컬럼의 값을 가져야 한다.(not null)

CUST_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
-----	-----	-----	-----	-----
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Brown	01-JAN-70	800-555-1215

# 데이터베이스 무결성(Integrity)

```
SQL> insert into customers (  
  2  customer_id, first_name, last_name, dob, phone  
  3  ) values (  
  4  1, 'Jason', 'Price', '01-JAN-60', '800-555-1211'  
  5  );
```

```
SQL> ora-00001: unique constraint  
      (store.customers_pk) violated
```

```
SQL> update customers  
  2  set customer_id = 1  
  3  where customer_id = 2;
```

```
SQL> ora-00001: unique constraint  
      (store.customers_pk) violated
```

# 데이터베이스 무결성(Integrity)

- 외래 키 제약(foreign key constraints)의 시행(enforcement)
  - 외래 키는 한 테이블에서의 컬럼이 다른 테이블의 컬럼을 참조
  - 참조하는 테이블(**child-table**)의 컬럼 값을 참조 당하는 테이블(**parent-table**)의 컬럼 값으로 제한하고자 할 때 사용.
  - 참조하는 테이블(**child-table**)의 컬럼 값은 널(**null**) 값을 가질 수 있음.

```
SQL> select product_type_id, name
      2   from product_types;
SQL> select product_id, product_type_id
      2   from products;
```

PRODUCT_TYPE_ID	NAME
1	Book
2	Video
3	DVD
4	CD
5	Maga.

PRODUCT_ID	PRODUCT_TYPE_ID
1	1
2	1
3	2
4	2
	.



# 데이터베이스 무결성(Integrity)

```
SQL> insert into products (  
  2  product_id, product_type_id, name, description  
    , price) values (  
  3  13, 6, 'Test', 'Test', null);
```

```
SQL> ora-02291: integrity constraint  
      (store.products_fk_product_types) violated  
      - parent key not found
```

```
SQL> update products  
  2  set product_type_id = 6  
  3  where product_id = 1;
```

```
SQL> ora-02291: integrity constraint  
      (store.products_fk_product_types) violated  
      - parent key not found
```

# 데이터베이스 무결성(Integrity)

```
SQL> delete from product_types  
2   where product_type_id = 1;
```

```
SQL> ora-02292: integrity constraint  
      (store.products_fk_product_types) violated  
      - child record found
```

## ■ 제약(constraints)관련 정보 저장소(data dictionary)

```
SQL> select table_name, constraint_name,  
           constraint_type, status  
2   from user_constraints  
3   where table_name in ('PRODUCTS', 'PRODUCT_TYPES');
```

# 데이터베이스 무결성(Integrity)

TABLE_NAME	CONSTRAINT_NAME	C STATUS
-----	-----	-----
PRODUCTS	SYS_C005672	C ENABLED
PRODUCTS	PRODUCTS_PK	P ENABLED
PRODUCTS	PRODUCTS_FK_PRODUCT_TYPES	R ENABLED
PRODUCT_TYPES	SYS_C005670	C ENABLED
PRODUCT_TYPES	PRODUCT_TYPES_PK	P ENABLED

```
SQL> select table_name, column_name,  
           constraint_name  
2    from user_cons_columns  
3    where table_name in ('PRODUCTS','PRODUCT_TYPES');
```

TABLE_NAME	COLUMN_NAME	CONSTRAINT_NAME
-----	-----	-----
PRODUCT_TYPES	PRODUCT_TYPE_ID	PRODUCT_TYPES_PK
PRODUCT_TYPES	NAME	SYS_C005670

...

# MERGE를 사용한 행(row)의 병합

## ■ 병합(merging)

- **MERGE**는 두 개의 테이블에 대한 병합(merging)을 지원.
- 기준 테이블에 병합 조건의 맞는 경우 변경, 맞지 않는 경우 추가를 시행한다.

## ■ 병합(merging) 문장

- **MERGE INTO** : 병합의 기준 테이블을 명시한다.
- **USING ~ON** : 병합되는 테이블과 병합 조건을 명시한다.
- **WHEN MATCHED THEN** : 병합 조건이 참인 행(row)에 대한 변경을 처리한다.
- **WHEN NOT MATCHED THEN** : 병합 조건이 거짓인 행(row)에 대한 추가를 처리한다.

# MERGE를 사용한 행(row)의 병합

## MERGE

MERGE는 조건에 따라 데이터를 입력하거나 변경 할 수 있는 문장이다. 즉, 입력할 데이터가 대상 테이블에 존재하지 않으면 INSERT가 수행되며, 동일한 데이터가 대상 테이블에 존재하면 UPDATE가 수행된다. MERGE 문장의 문법은 다음과 같다

```
MERGE INTO table_name table_alias USING (table|view|sub_query) alias ON (join condition)
```

```
WHEN MATCHED THEN UPDATE SET
```

```
col1 = col1_val,
```

```
col2 = col2_val
```

```
WHEN NOT MATCHED THEN
```

```
INSERT (column_list) VALUES (column_values);
```

EMP10 테이블의 데이터를 EMP 테이블의 데이터로 갱신해보자.

즉, EMP 테이블에 저장된 행이 EMP10 테이블에 존재하면 EMP10 테이블의 해당 행을 EMP 테이블에 저장된 행으로 UPDATE하고, 존재하지 않으면 EMP10 테이블에 해당 행을 INSERT하는 것이다.

# MERGE를 사용한 행(row)의 병합

```
SQL> MERGE INTO EMP10 N USING EMP O  
ON (N.EMPNO = O.EMPNO) WHEN MATCHED THEN  
UPDATE SET N.ENAME = O.ENAME,  
N.JOB = O.JOB, N.SAL = O.SAL  
WHEN NOT MATCHED THEN  
INSERT (N.EMPNO, N.ENAME, N.JOB, N.SAL)  
VALUES (O.EMPNO, O.ENAME, O.JOB, O.SAL);
```

16 행이 병합되었습니다

위의 예를 보면 EMP10 테이블과 EMP테이블에서 동일한 행의 존재 여부는 두 테이블의 EMPNO 컬럼에 동일한 값이 존재하는지 여부로 판단한다. 즉, 동일한 사번이 있으면 EMP테이블의 데이터로 EMP10테이블을 변경하고, 동일한 사번이 없으면 EMP 테이블의 데이터를 입력한다.

# 데이터베이스 트랜잭션(transaction)

- 데이터 베이스 트랜잭션(transaction)
  - 단순히 작업에 대한 영구 반영이 아니다.
  - 논리적 작업 단위.
  - 하나의 트랜잭션(transaction)은 분리할 수 없는 **SQL**문장의 집합.
  - 거래함에 있어서 거래하는 양측 다 원하는 결과를 얻어야만 정상적으로 처리되는 것의 단위(transaction)로 온라인 송금이나, 현금 인출 등을 예로 들 수 있다.
- 트랜잭션(transaction)을 시행(commit)하고 취소(rollback)하기
  - 오라클에서는 암시적으로 트랜잭션이 작동 중이다.
  - **Commit** – 결과의 영구적 반영을 시행한다.
  - **Rollback** – 결과를 취소, 트랜잭션의 처음 시점으로 되돌린다.

# 트랜잭션

## ■ 은행 현금인출기(ATM)에서 돈을 인출하는 과정

현금인출을 하겠다고 기계에게 알려준다.

현금카드를 넣어서 본인임을 인증 받는다.

인출할 금액을 선택하면 은행 현금인출기는 돈을 내어준다.

계좌에서 인출된 금액만큼을 잔액에서 차감한다.



# 트랜잭션

- 이러한 거래에 있어서 지켜져야 할 중요한 것이 있습니다.
- 기계의 오동작 등으로 인하여 전산 상으로는 돈을 인출한 것으로 입력이 되었는데 돈은 안 나온다거나, 돈은 나왔는데 일련의 에러나 문제로 인하여서 돈을 인출한 것이 전산 상으로 입력이 안 되면 상당히 심각한 문제가 발생합니다.
- 이 때문에 전산 상으로도 입력이 정상적으로 잘 되고, 돈도 인출이 정상적으로 잘 났을 확인하고 나서야, 인출하는 하나의 과정이 정상적으로 처리되었음을 확인할 수 있습니다.
- 여기서 돈을 인출하는 일련의 과정이 하나의 묶음으로 처리되어야 한다는 것을 이해할 수 있을 것입니다.
- 그리고 혹시 처리도중 중간에 무슨 문제가 발생한다면 진행되던 인출과정 전체를 취소하고 다시 처음부터 시작해야 합니다.
- 이러한 작업을 위한 단위를 트랜잭션이라고 합니다.

# 데이터베이스 트랜잭션(transaction)

```
SQL> insert into customers  
2 values(6,'Fred', 'Green', '70/01/01',  
3      '800-555-1215');
```

```
SQL> commit;
```

```
SQL> update customers  
2 set first_name = 'Edward'  
3 where customer_id = 1;
```

```
SQL> rollback;
```

# 데이터베이스 트랜잭션(transaction)

```
SQL> select *  
1  from customers;
```

CUST_ID	FIRST_NAME	LAST_NAME	DOB	PHONE
1	John	Brown	01-JAN-65	800-555-1211
2	Cynthia	Green	05-FEB-68	800-555-1212
3	Steve	White	16-MAR-71	800-555-1213
4	Gail	Black		800-555-1214
5	Doreen	Blue	20-MAY-70	
6	Fred	Brown	01-JAN-70	800-555-1215

## ■ 실행 결과

- **Cust\_id = 6** 인 **Fred**의 열이 영구적으로 적용되었다.
- **Cust\_id = 1** 인 **John**의 이름 변경이 취소 되었다.

# 데이터베이스 트랜잭션(transaction)

- 트랜잭션(transaction)의 시작(start)
  - 데이터 베이스에 연결하고 첫 번째 **DML**문장을 수행할 때.
  - 앞선 트랜잭션을 끝내고 다시 **DML**문장을 수행할 때.
- 트랜잭션(transaction)의 끝(end)
  - **Commit** 이나 **Rollback**을 시행할 때.
  - 테이블 생성(**create table**)과 같은 데이터 정의 문장(**DDL**)을 시행할 때.
  - 권한 부여(**grant**)와 같은 데이터 제어 문장(**DCL**)을 사용할 때.
  - 데이터베이스와 접속을 끊은 경우. 정상적인 경우는 **commit**이, 비 정상적인 종료인 경우는 **rollback**이 적용된다.
  - 데이터 조작 문장(**DML**)을 시행 시 실패하면 시행된 문장에 대해서만 **rollback**이 적용된다.

# 데이터베이스 트랜잭션(transaction)

## ■ 저장점(savepoints)

- 트랜잭션(transaction)의 한 단위 안에 여러 개의 저장점(savepoints)을 배치할 수 있다.
- 대단히 긴 트랜잭션(transaction)에 주로 사용한다.
- 작은 단위로 트랜잭션(transaction)을 관리함으로써 효율적인 관리가 가능하다.

```
SQL> select product_id, price  
2   from products  
3   where product_id in (1,2);
```

PRODUCT_ID	PRICE
1	19.95
2	30

# 데이터베이스 트랜잭션(transaction)

```
SQL> update products  
2   set price = price * 1.20  
3   where product_id = 1;
```

```
SQL> savepoint save1;
```

```
SQL> update products  
2   set price = price * 1.30  
3   where product_id = 2;
```

```
SQL> select product_id, price  
2   from products where product_id in (1,2);
```

# 데이터베이스 트랜잭션(transaction)

PRODUCT_ID	PRICE
1	23.94
2	39

```
SQL> rollback to savepoint save1;
```

```
SQL> select product_id, price  
2   from products  
3   where product_id in (1,2);
```

PRODUCT_ID	PRICE
1	23.94
2	30

```
SQL> rollback;
```

# 데이터베이스 트랜잭션(transaction)

- ACID 트랜잭션(transaction) 속성들(properties)
  - 트랜잭션에 대한 보다 엄격한 4가지 속성 정의.
  - **Atomicity(원자성)** : 하나의 트랜잭션에 포함된 모든 SQL문장들은 하나의 불가분한 단위로 처리되어야 한다.
  - **Consistency(일관성)** : 트랜잭션이 시작하고 끝날 때의 상태는 항상 데이터베이스의 모든 제약을 만족하는 상태여야 한다.
  - **Isolation(고립성)** : 개별적인 트랜잭션은 각각의 트랜잭션에 간섭없이 작동함을 나타낼 수 있다.
  - **Durability(내구성)** : 트랜잭션이 시행(commit)되어지면, 어떠한 경우에 대해서도 데이터베이스의 변경은 보존되어야 한다.



# 데이터베이스 트랜잭션(transaction)

- 동시 트랜잭션(concurrent transaction)
  - 같은 시간에 많은 사용자들이 트랜잭션을 처리하는 것을 말한다.
  - 동일한 테이블에 대한 트랜잭션의 경우 요청의 순서에 따라 작업을 처리한다.
- 동시 트랜잭션의 테스트
  - 두 개의 연결을 생성하고 다음 페이지의 쿼리를 순서대로 시행하여본다.

# 데이터베이스 트랜잭션(transaction)

## Transaction 1 T1

```
Select *  
From customers;
```

```
Insert into customers(  
  Customer_id, first_name,  
  Last_name) value (  
  7, 'Jason', 'Price');
```

```
Update customers  
Set last_name = 'Orange'  
Where customer_id = 2;
```

```
Select *  
From customers;  
추가, 변경된 값을 모두 포함한다.
```

```
Commit;
```

## Transaction 2 T2

```
Select *  
From customers;
```

```
Select *  
From customers;  
T1에 의해 추가, 변경된 값이 포함되지  
않는다. 상단의 결과와 같다.
```

```
Select *  
From customers;  
T1에 의해 추가, 변경된 값이 모두  
포함된다.
```

# 데이터베이스 트랜잭션(transaction)

- 오라클에서의 동시 트랜잭션(concurrent transaction) 결과 요약
  - 트랜잭션은 시행(**commit**)이 이루어지기 전까지는 다른 사용자에게 영향을 미치지 않는다.  
오라클의 기본 트랜잭션 고립 수위로 변경 가능하다.
  - 동일한 테이블에 대한 트랜잭션의 경우 요청의 순서에 따라 작업을 처리한다.

# 데이터베이스 트랜잭션(transaction)

## ■ 트랜잭션 잠금(locking)

- 동시 트랜잭션을 지원하기 위해 테이블의 데이터에 대한 잠금(lock)을 사용한다.
- 오라클(oracle)은 열(row) 단위의 잠금(lock)을 지원한다.

Transaction 1 T1

```
Update customer  
Set last_name = 'jim'  
Where customer_id = 1;  
잠금 customer # 1
```

```
Commit;  
잠금 해제 customer # 1
```

Transaction 2 T2

```
Update customer  
Set last_name = 'james'  
Where customer_id = 1;  
T1이 잠금을 해제 할 때 까지 대기.
```

```
잠금 customer # 1
```

```
Commit;  
잠금 해제 customer # 1
```

# 데이터베이스 트랜잭션(transaction)

## ■ 트랜잭션 잠금(locking) 요약

- 트랜잭션이 열에 대한 잠금을 소유하고 있는 경우, 다른 트랜잭션은 동일한 열에 대한 잠금을 얻을 수 없다.
- 읽기(reader)는 읽기를 방해하지 않는다.
- 쓰기(writer)는 읽기를 방해하지 않는다.
- 쓰기만이 단지 동일한 열(row)에 대한 쓰기를 방해한다.

# 데이터베이스 트랜잭션(transaction)

- 트랜잭션 고립 단계(transaction isolation levels)
  - 한 트랜잭션에 의한 변화가 동시에 동작하는 다른 트랜잭션과 분리되는 정도.
  - **Read uncommitted**  
유령 읽기, 비반복적인 읽기, 오염된 읽기를 모두 허용
  - **Read committed**  
유령 읽기, 비반복적인 읽기는 허용. 오염된 읽기는 비 허용
  - **Repeatable read**  
유령 읽기는 허용. 비반복적인 읽기, 오염된 읽기는 비 허용
  - **Serializable**  
유령 읽기, 비반복적인 읽기 그리고 오염된 읽기를 모두 비 허용
  - 오라클은 **read committed** 와 **serializable** 만을 허용한다. 기본 적용은 **read committed** 이다.

```
SQL> set transaction isolation level  
      { SERIALIZABLE | READ COMMITTED }
```

# 데이터베이스 트랜잭션(transaction)

- 동시성 트랜잭션(concurrent transaction)에 의해 발생할 수 있는 문제들
  - 유령 읽기(phantom reads) : 트랜잭션의 행 추가로 인하여 다른 트랜잭션에서 동일한 질의에 대하여 다른 결과 집합을 가지게 되는 경우.

Transaction 1 T1

```
Select *  
From customers;
```

```
Select *  
From customers;
```

Transaction 2 T2

```
Insert into customers(  
  Customer_id, first_name,  
  Last_name) value (  
    9, 'dim', 'Price');  
Commit;
```

# 데이터베이스 트랜잭션(transaction)

- 비반복적인 읽기(**nonrepeatable reads**) : 트랜잭션의 행 변경으로 인하여 다른 트랜잭션에서 동일한 질의에 대하여 다른 행 결과를 가지게 되는 경우.

Transaction 1 T1

```
Select *  
From customers;
```

```
Select *  
From customers;
```

Transaction 2 T2

```
update customers  
Set last_name = 'doorian'  
Where customer_id = 1;  
Commit;
```



# 데이터베이스 트랜잭션(transaction)

- 오염된 읽기(**dirty reads**) : 트랜잭션의 행 변이 시행(**commit**)이나 취소(**rollback**)전에 다른 트랜잭션의 질의에 대하여 결과를 가지게 되는 경우.

Transaction 1 T1

```
Select *  
From customers;
```

```
Select *  
From customers;
```

Transaction 2 T2

```
update customers  
Set last_name = 'doorian'  
Where customer_id = 1;
```

```
rollback;
```

# 데이터베이스 트랜잭션(transaction)

## ■ 트랜잭션의 예제(Serializable)

Transaction 1 T1(Read committed)

```
Select *
From customers;
Insert into customers(
  Customer_id, first_name,
  Last_name) value (
  8, 'steve', 'button');
Update customers
Set last_name = 'yellow'
Where customer_id = 3;
Commit;
```

Select \*

```
From customers;
```

추가, 변경된 값을 모두 포함한다.

Transaction 2 T2(serializable)

```
Set transaction isolation
Level serializable;
Select *
From customers;
```

Select \*

```
From customers;
```

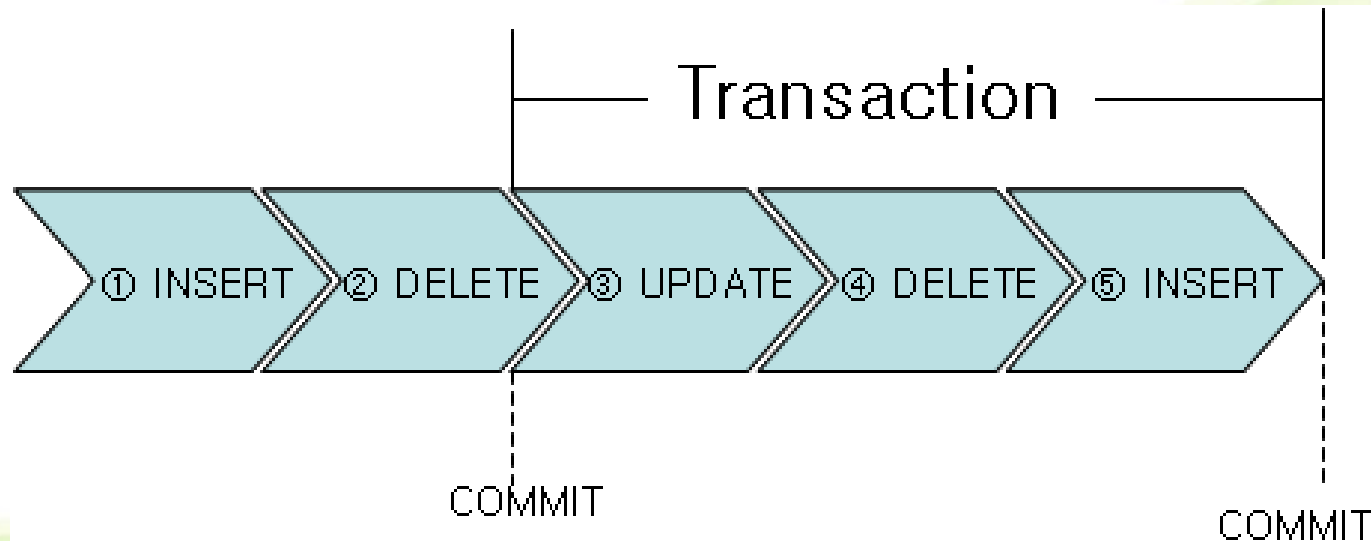
T2는 Serializable하기 때문에  
T1에 의해 추가, 변경된 값이 포함되지  
않는다.

# COMMIT과 ROLLBACK

- 데이터를 조작하는 명령어인 **DML(Data Manipulation Language)**은 이들이 실행됨과 동시에 트랜잭션이 진행됩니다.
- **DML** 작업이 성공적으로 처리되도록 하기 위해서는 **COMMIT** 명령을 작업을 취소하기 위해서는 **ROLLBACK** 명령으로 종료해야 합니다.
- **COMMIT**은 모든 작업들을 정상적으로 처리하겠다고 확정하는 명령어로 트랜잭션의 처리 과정을 데이터베이스에 모두 반영하기 위해서 변경된 내용을 모두 영구 저장합니다.
- **COMMIT** 명령어를 수행하게 되면 하나의 트랜잭션 과정을 종료하게 됩니다.
- **ROLLBACK**은 작업 중 문제가 발생되어서 트랜잭션의 처리 과정에서 발생한 변경사항을 취소하는 명령어입니다.
- **ROLLBACK** 명령어 역시 트랜잭션 과정을 종료하게 됩니다.
- **ROLLBACK**은 트랜잭션으로 인한 하나의 묶음 처리가 시작되기 이전의 상태로 되돌립니다.
- 트랜잭션은 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(트랜잭션)로 구성해 놓는다.
- 문제가 발생하게 면 이 논리적인 작업 단위를 취소해 버리면 되기 때문입니다.

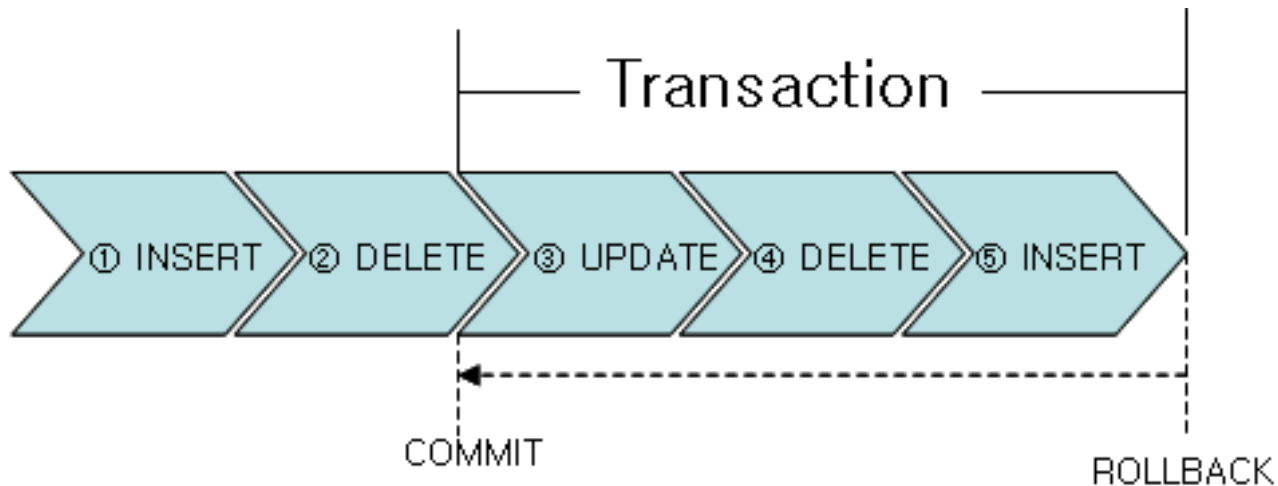
# COMMIT과 ROLLBACK

- 여러 개의 **DML** 명령어들을 어떻게 하나의 논리적인 단위인 트랜잭션으로 묶을 수 있을까?
- 트랜잭션은 마지막으로 실행한 커밋(혹은 롤백) 명령 이후부터 새로운 커밋(혹은 롤백) 명령을 실행하는 시점까지 수행된 모든 **DML** 명령들을 의미합니다.
- 아래 그림에서 **UPDATE** 문으로 데이터를 갱신하고(③), **DELETE** 문으로 데이터를 삭제하고(④), **INSERT** 문을 사용해 데이터를 삽입(⑤)합니다.
- 만약 이 모든 과정이 오류 없이 수행되었다면 지금까지 실행한 모든 작업(③, ④, ⑤)을 "데이터베이스에 영구 저장하라"는 명령으로 커밋을 수행합니다.



# COMMIT과 ROLLBACK

- 롤백 명령은 마지막으로 수행한 커밋 명령까지만 정상 처리(①, ②)된 상태로 유지하고 그 이후에 수행했던 모든 **DML** 명령어 작업(③, ④, ⑤)들을 취소시켜 이전 상태로 원상 복귀시킵니다.



- 트랜잭션은 이렇듯 **All-OR-Nothing** 방식으로 **DML** 명령어들을 처리합니다.

# COMMIT과 ROLLBACK

- **COMMIT과 ROLLBACK은 다음과 같은 장점이 있습니다.**
- **COMMIT 명령어과 ROLLBACK 명령어의 장점**
  - 데이터 무결성이 보장됩니다.
  - 영구적인 변경 전에 데이터의 변경 사항을 확인할 수 있습니다.
  - 논리적으로 연관된 작업을 그룹화할 수 있습니다.
- **COMMIT 명령어**
  - Transaction(INSERT, UPDATE, DELETE) 작업 내용을 실제 DB에 저장합니다.
  - 이전 데이터가 완전히 UPDATE 됩니다.
  - 모든 사용자가 변경된 데이터의 결과를 볼 수 있습니다.
- **ROLLBACK 명령어**
  - Transaction(INSERT, UPDATE, DELETE) 작업 내용을 취소합니다.
  - 이전 COMMIT한 곳 까지만 복구합니다.

# COMMIT과 ROLLBACK

- 데이터베이스 사용자가 **COMMIT**이나 **ROLLBACK** 명령어를 명시적으로 수행시키지 않더라도 다음과 같은 경우에 자동 커밋 혹은 자동 롤백이 발생합니다.
- 자동 **COMMIT** 명령과 자동 **ROLLBACK** 명령이 되는 경우
  - SQL\* PLUS가 정상 종료되었다면 자동으로 **COMMIT**되지만, 비정상 종료되었다면 자동으로 **ROLLBACK** 합니다.
  - DDL과 DCL 명령문이 수행된 경우 자동으로 **COMMIT** 됩니다.
  - 정전이 발생했거나 컴퓨터 **Down**시(컴퓨터의 전원이 끊긴) 자동으로 **ROLLBACK** 됩니다.

# 실습하기

■ 부서번호가 **10**번인 부서에 대해서만 삭제하려고 했는데 테이블 내의 모든 데이터가 삭제되어 아무런 데이터도 찾을 수 없게 되었다라도 **ROLLBACK** 문을 사용하여 이전 상태로 되돌릴 수 있습니다.

**DELETE** 문으로 테이블의 모든 데이터를 삭제합니다.

```
DELETE FROM DEPT01;
```

만일 부서번호가 **20**번인 부서에 대해서만 삭제하려고 했는데 위와 같은 명령을 수행했다면 테이블 내의 모든 데이터가 삭제되어 다음과 같이 아무런 데이터도 찾을 수 없게 됩니다. 이전 상태로 되돌리기 위해서 **ROLLBACK** 문을 수행합니다.

```
ROLLBACK;
```



# 실습하기

원래하려고 했던 부서번호가 **20**번인 부서만 삭제해 봅시다.

1. 이번에는 부서번호 **20**번 사원에 대한 정보만 삭제한 후, 확인합니다.

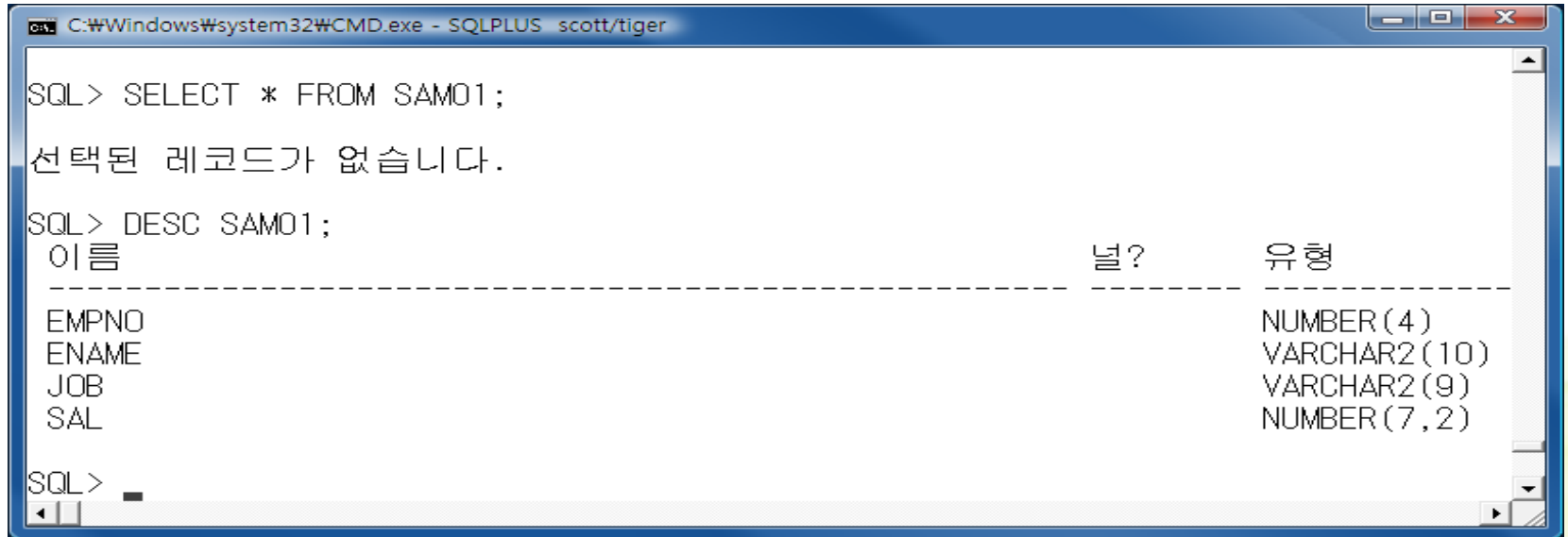
```
DELETE FROM DEPT01 WHERE DEPTNO=20;
```

2. 데이터를 삭제한 결과를 물리적으로 영구히 저장하기 위해서 커밋을 수행합니다.

```
COMMIT;
```

# 연습문제

1. 서브 쿼리문을 이용하여 다음과 같은 구조로 **SAM01** 테이블을 생성하시오. 존재할 경우 **DROP TABLE**로 삭제 후 생성하시오.



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\CMD.exe - SQLPLUS scott/tiger". The user enters the following SQL commands:

```
SQL> SELECT * FROM SAM01;
```

The output is: "선택된 레코드가 없습니다." (No records selected).

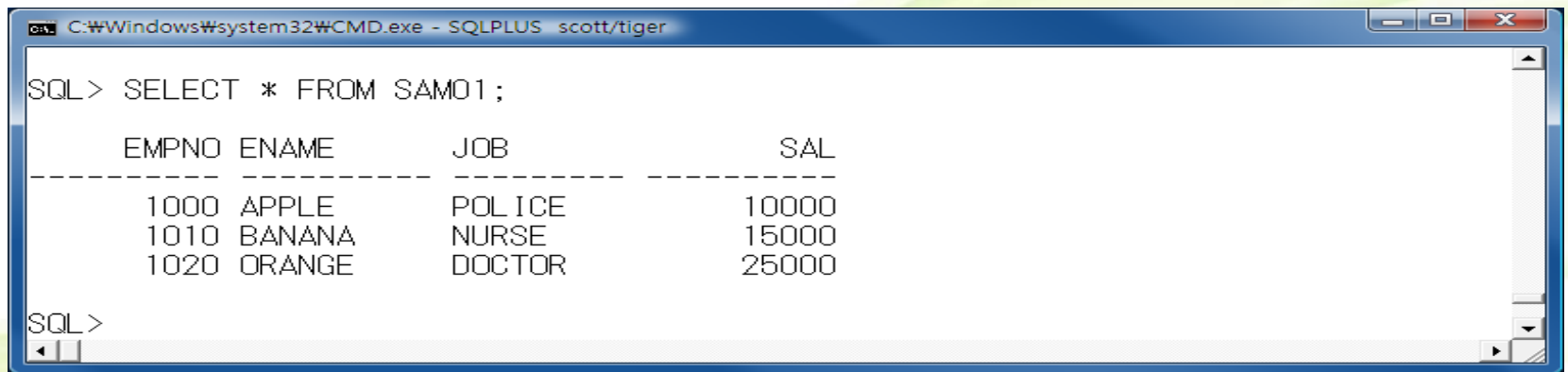
```
SQL> DESC SAM01;
```

The output shows the table structure:

이름	널?	유형
EMPNO		NUMBER (4)
ENAME		VARCHAR2 (10)
JOB		VARCHAR2 (9)
SAL		NUMBER (7,2)

The prompt "SQL>" is visible at the bottom.

2. **SAM01** 테이블에 다음과 같은 데이터를 추가 하시오.



The screenshot shows the same SQLPLUS command prompt window. The user enters the following SQL command:

```
SQL> SELECT * FROM SAM01;
```

The output shows the data inserted into the table:

EMPNO	ENAME	JOB	SAL
1000	APPLE	POLICE	10000
1010	BANANA	NURSE	15000
1020	ORANGE	DOCTOR	25000

The prompt "SQL>" is visible at the bottom.

# 자동 커밋

타입	설명
DML(Data Manipulation Language)	데이터베이스의 논리적 작업 단위로서 여러 개의 DML 문장으로 구성
DDL(Data Definition Language)	하나의 DDL 문장으로 구성
DCL(Data Control Language)	하나의 DCL 문장으로 구성

그렇다면 트랜잭션이 시작되는 시점과 종료되는 시점을 알아보자. 트랜잭션은 첫 번째 DML 문장이 실행되면 시작되고 다음과 같은 상황에서 트랜잭션은 종료된다.

- 사용자가 COMMIT 또는 ROLLBACK 명령을 실행한 경우
- CREATE 명령과 같은 DDL(CREATE, ALTER, DROP, RENAME, TRUNCATE) 문장을 실행한 경우
- DCL(GRANT, REVOKE) 문장을 실행한 경우
- 사용자가 SQL\*Plus 또는 iSQL\*Plus를 종료한 경우
- 하드웨어 고장 또는 시스템 오류

# 실습하기

**CREATE**문에 의한 자동 커밋에 의해서 이전에 수행했던 **DML** 명령어가 자동 커밋됨을 확인해 봅시다.

1. 부서 번호가 **40**번인 부서를 삭제합니다.

```
DELETE FROM dept02  
WHERE deptno=40;
```

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

```
CREATE TABLE DEPT03  
AS  
SELECT * FROM DEPT;
```

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

# 실습하기

**TRUNCATE** 문이 실패되더라도 자동 커밋되어 이전에 수행했던 **DML** 명령어가 자동 커밋됨을 확인해 봅시다

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

```
DELETE FROM DEPT03  
WHERE DEPTNO=20;
```

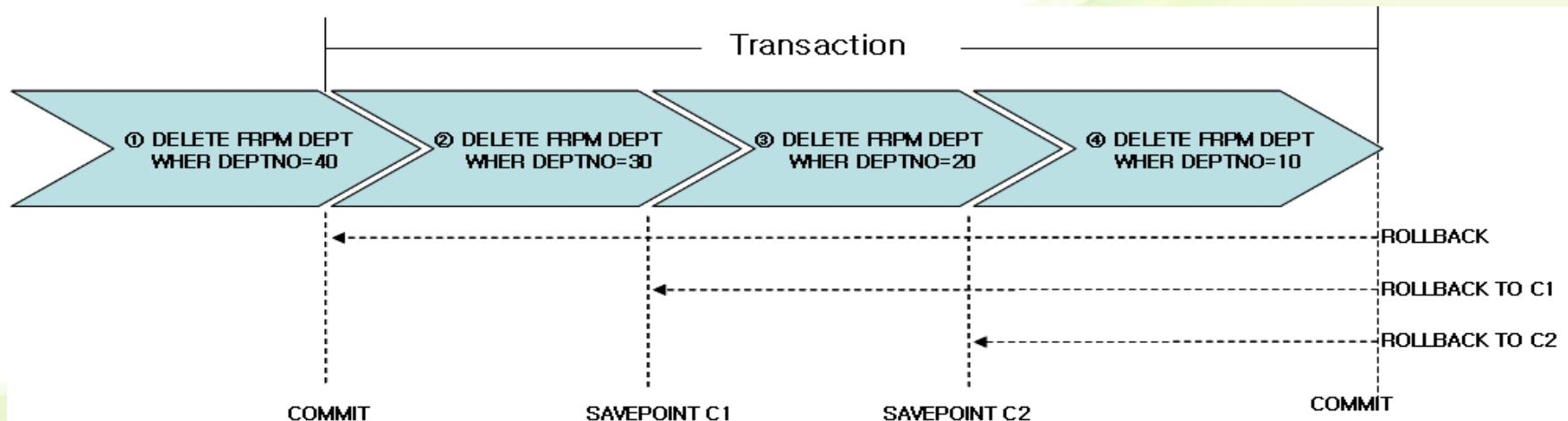
2. **TRUNCATE** 문을 실행시키되 테이블 명을 일부러 잘못 적어서 에러를 유도합니다.

```
TRUNCATE TABLE DEPTPPP;
```

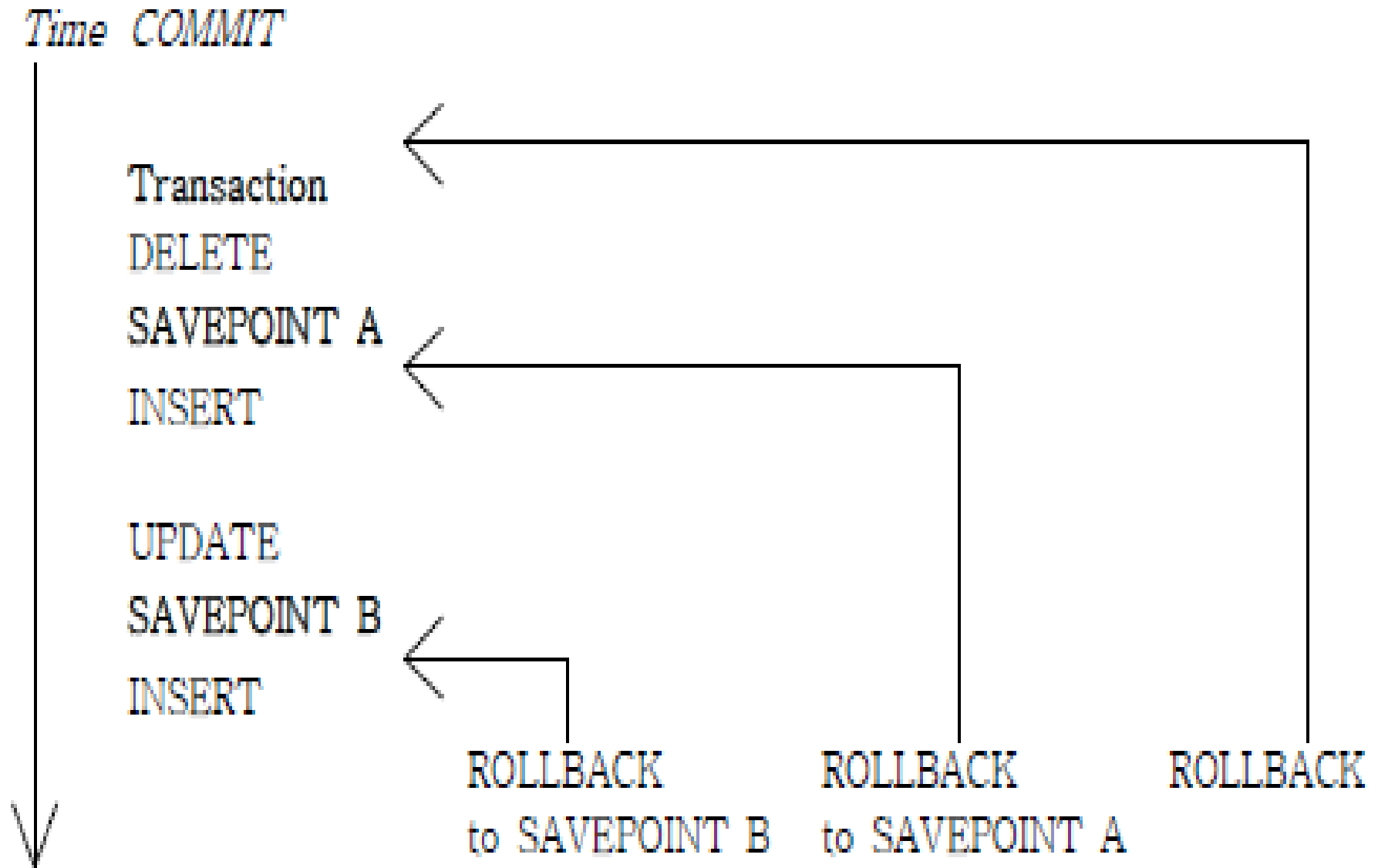
3. 부서번호가 **20**번인 부서를 다시 되살리기 위해서 **ROLLBACK** 명령문을 수행하여도 **TRUNCATE** 문이 수행되면서 자동으로 커밋이 발생하였으므로 되살릴 수 없습니다.

# SAVEPOINT

- SAVEPOINT 명령을 써서 현재의 트랜잭션을 작게 분할할 수 있습니다.
- 저장된 SAVEPOINT는 ROLLBACK TO SAVEPOINT 문을 사용하여 표시한 곳까지 ROLLBACK할 수 있습니다.
- 여러 개의 SQL 문의 실행을 수반하는 트랜잭션의 경우, 사용자가 트랜잭션 중간 단계에서 세이프포인트를 지정할 수 있습니다.
- 이 세이프포인트는 차후 롤백과 함께 사용해서 현재 트랜잭션 내의 특정 세이프포인트까지 롤백할 수 있게 됩니다.
- 아래 그림을 보면 COMMIT 명령이 내려진 후 다음 COMMIT 명령이 나타날 때까지가 하나의 트랜잭션으로 구성되므로 ②번에서 ④번까지가 하나의 트랜잭션이 됩니다.
- 이렇게 트랜잭션을 구성할 때 중간 중간 SAVEPOINT 명령으로 위치를 지정해 놓으면(예를 들어 C) 하나의 트랜잭션 내에서도 ROLLBACK TO C(SAVEPOINT 문을 사용하여 표시한 곳)까지 ROLLBACK할 수 있습니다.



# 트랜잭션



# SAVEPOINT

- 다음은 **SAVEPOINT**로 특정 위치를 지정하기 위한 사용 형식입니다.

```
SAVEPOINT LABEL_NAME;
```

- **SAVEPOINT**로 지정해 놓은 특정 위치로 되돌아가기 위한 사용 형식입니다.

```
ROLLBACK TO LABEL_NAME;
```



# 실습하기

- 다음과 같이 트랜잭션 중간 단계에서 세이브포인트를 지정해 보도록 하겠습니다.



# 실습하기

1. 부서번호가 **40**번인 부서를 삭제한 후에 커밋을 수행하여 새롭게 트랜잭션을 시작합니다.

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

2. 이번엔 부서번호가 **30**번인 부서를 삭제합니다.

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

3. 세이프포인트 **C1**를 설정한 후, 부서번호가 **20**번인 사원을 삭제합니다.

```
SAVEPOINT C1;  
DELETE FROM DEPT01 WHERE DEPTNO =20;
```

4. 세이프포인트 **C2**를 설정한 후, 부서번호가 **10**번인 사원을 삭제합니다.

```
SAVEPOINT C2;  
DELETE FROM DEPT01 WHERE DEPTNO =10;
```

# 실습하기

이제 부서번호가 **10**번인 사원을 삭제하기 바로 전으로 되돌리려면 어떻게 해야 할까요? 세이브 포인트를 이용해서 트랜잭션 중간 단계로 되돌려 봅시다.

1. 지금 **ROLLBACK** 명령을 내리게 된다면 이전 **COMMIT** 지점으로 되돌아가므로 **10, 20, 30**번 부서의 삭제가 모두 취소됩니다. 따라서 원했던 **10**번 부서의 삭제 이전까지만 되돌리려면 다시 **30, 20**번의 부서를 삭제해 주어야 할 것입니다.

```
ROLLBACK TO C2;
```

2. 위 결과 화면을 보면 세이브포인트 **C2** 지점으로 이동되어 **10**번 부서의 삭제 이전으로 되돌려진 것을 확인할 수 있습니다.

```
ROLLBACK TO C1;
```

3. 마지막으로 이전 트랜잭션까지 롤백한 후의 결과를 봅시다.

```
ROLLBACK;
```

## 읽기 일관성(Read Consistency)

데이터의 읽기 일관성이 필요한 이유는 다음과 같다.

- 데이터를 검색하는 사용자와 변경하는 사용자 사이에 일관적인 관점을 제공한다. 즉, 다른 사용자들이 변경 중인 데이터를 볼 수 없게 한다.
- 데이터를 변경하는 사용자들 사이에 일관적인 데이터베이스 변경 방법을 제공함으로써 동일한 데이터의 동시에 변경함으로써 발생 할 수 있는 혼란을 방지한다.

결과적으로 읽기 일관성의 목적은 각각의 사용자에게 다른 사용자들의 DML 작업이 시작되기 이전의 데이터 즉, 가장 최근에 커밋 된 데이터를 보여주는 것이다.

### ■ 읽기 일관성의 구현 원리

읽기 일관성은 Oracle에 의해 자동으로 관리되며, 데이터의 변경 이전 값을 UNDO 세그먼트에 저장함으로써 구현된다. 즉, 데이터베이스에 INSERT, UPDATE, DELETE 문장이 실행되면 Oracle 데이터베이스는 변경 전 데이터를 UNDO 세그먼트에 임시적으로 저장한다. 이러한 상황에서 해당 데이터를 변경한 사용자를 제외한 모든 사용자들은 UNDO 세그먼트의 변경 전 데이터를 보게 된다.

# 트랜잭션

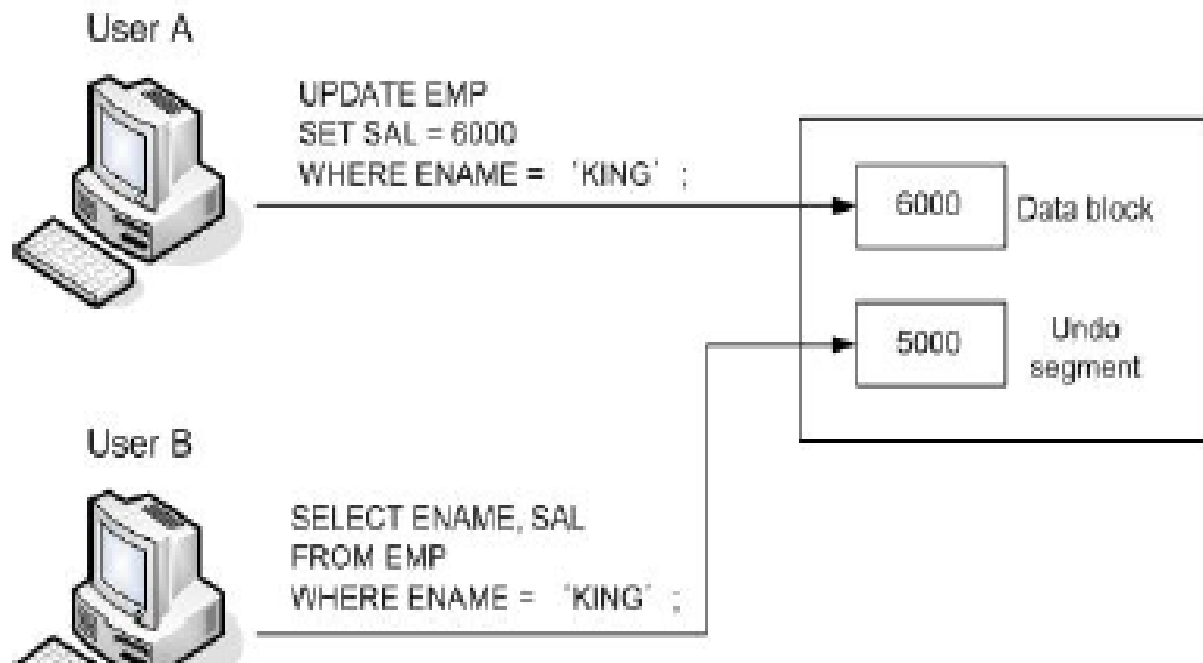


그림 9-2. 읽기 일관성의 원리

트랜잭션이 종료되기 전까지는 해당 데이터를 변경 중인 사용자만 변경 후의 결과를 보게 되며 다른 모든 사용자들은 모두 UNDO 세그먼트의 변경 전 데이터를 보게 된다. 즉, 이러한 메커니즘에 의해 사용자들은 가장 최근의 커밋 된 데이터만을 보게 됨으로써 읽기 일관성이 유지된다.

# 트랜잭션

1. 부서 테이블에 부서코드가 99,부서명은 ‘관리과’,위치는 ‘대구’인 행을 입력하시오.
2. 부서 테이블에 99번 부서의 부서명을 ‘회계과’로 변경하시오.
3. 부서 테이블의 99번 부서 행을 삭제하시오.
4. 트랜잭션이 자동 커밋 되는 시점을 모두 고르시오.
  - 1) DCL 문장의 실행
  - 2) DCL 문장의 실행
  - 3) 시스템 오류
  - 4) SQL\*Plus의 정상 종료
5. 현재 진행 중인 트랜잭션을 종료하고 모든 데이터 변경사항을 취소하는 명령어는 ( )이며, 데이터의 변경 변경사항을 데이터베이스에 영구히 반영하는 명령어는 ()이다.

# 연습문제

1. 아래의 구조를 만족하는 **MY\_DATA** 테이블을 생성하시오.

**SQL> DESC my\_data**

Name	Null?	Type
-----		
ID	NOT NULL	NUMBER(4)
NAME		VARCHAR2(10)
USERID		VARCHAR2(30)
SALARY		NUMBER(10,2)

2. 1번에 의해 생성된 테이블에 아래의 값을 입력하여라.

ID	NAME	USERID	SALARY
1	Scott	sscott	10,000.00
2	Ford	fford	13,000.00
3	Patel	ppatel	33,000.00
4	Report	rreport	23,500.00
5	Good	ggood	44,450.00

# 연습문제

3. 2번에서 입력한 자료를 확인 하여라
4. ID가 3번인 사람의 급여를 **65,000.00**으로 갱신하여라.
5. 이름이 **Ford**인 사원을 삭제하여라.
6. 급여가 **15,000**이하인 사람의 급여를 **15,000**로 변경하여라.
7. 1번에서 생성한 테이블을 삭제하여라.