데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

9차시: PL/SQL Advanced Feature

I. 시작

[학습목표]
PL/SQL의 향상된 기능을 학습합니다.

[학습목차]
9-1 Dynamic SQL
9-2 Dynamic CURSOR
9-3 AUTONOMOUS TRANSACTION
9-4 BULK COLLECT & FORALL

9-1 Dynamic SQL

실행시점(Run Time)에 실행 프로그램내에서 생성되는 SQL 또는 PL/SQL을 동적 SQL(Dynamic SQL) 이라고 합니다. 그동안 사용했던 SQL이나 PL/SQL을 보면 미리 프로그램내에서 작성 되어져 있다가 실행 시점에 수행되는 정적 SQL(Static SQL) 이었습니다.
정리하면 동적 과 정적의 의미는 SQL 또는 PL/SQL의 작성시점이 개발 시점에 하느냐 실행 시점에 하느냐의 차이 입니다.

버전	개발시점	실행 시점
정적 SQL	SQL 작성	SQL 실행
동적 SQL		SQL 작성➔ SQL 실행

장 점: 어플리케이션 프로그램에 유연성

< 참고> 개발시점에 실행되어야 할 SQL 이 미리 정의되어 있는 경우에는 기존 방식대로 정적 SQL (Static SQL)을 사용하면 되지만 실행될 SQL과 관련된 여러가지 사항들(수행조건,조인 대상 등)이 사전에 정의될 수 없으며 실행시점에 결정되는 경우에는 정적 SQL로는 대처가 불가능 하거나 이를 위해서 수많은 조건 처리를 해야 하기 때문에 매우 비효율적 입니다. 실행시점에 SQL을 정의하는 동적 SQL(Dynamic SQL)은 프로그램 개발시 유연성을 높일수 있는 방법이므로 기억을 해두셔야 합니다

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

사 용 방 법: ① DBMS_SQL

② NATIVE DYNAMIC SQL (EXECUTE IMMEDIATE 명령어)

< 참고> DBMS_SQL은 코딩 및 사용상 복잡성으로 인해 사용하기가 힘들었습니다.
ORACLE 8i 부터 PL/SQL에서 EXECUTE IMMEDIATE 를 지원하면서 부터
동적 SQL을 쉽고 효율적으로 사용할수 있게 되었습니다.
EXECUTE IMMEDIATE 가 사용하기 쉽고 성능상 더 효율적 입니다.

동적 SQL 대상 : ① DDL(Data Definition Language) , DCL(Data Control Language)
② DML(Data Manipulation Language)
③ SELECT
④ ANONYMOUS PL/SQL BLOCK

< 참고>

동적 SQL 대상중 DML 과 SELECT는 쉽게 이해가 가지만
DDL 과 DCL , ANONYMOUS PL/SQL BLOCK 낫설게 느껴 집니다.
PL/SQL BLOCK내에서 DDL,DCL 명령어는 사용할수 없습니다.
DDL 명령어를 예로 든다면 CREATE TABLE ~ 은 대표적인 DDL 계열 명령 입니다.
PL/SQL BLOCK내에서 CREATE TABLE ~ 을 사용할수 없습니다.
,사용하려면 동적 SQL 로 사용해야지만 PL/SQL BLOCK 내에서 테이블 생성이 가능하게 됩니다. ANONYMOUS PL/SQL BLOCK 도 프로그램 실행중 동적으로 작성하여 실행할수 있습니다.

동적 SQL을 사용하는 용도는 첫번째 예는 DDL, DCL을 PL/SQL block내에서 사용하기 위해서 이고 두번째 예는 유연성을 위한 용도 입니다. >

용 도: ① DDL(Data Definition Language) , DCL(Data Control Language),
SCL(Ssssion Control Language) 을 PL/SQL BLOCK내에서 사용하기 위해
② 개발시 유연성을 위해(More flexibility)

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

9_DYNAMIC_SQL_1.SQL

```
BEGIN
        CREATE TABLE BY_DYNAMIC(X DATE);        -- DDL계열의 CREATE TABLE 사용
END;
/


DECLARE
        V_SQL           VARCHAR2(2000);
BEGIN
        -- ------------------------------------------------------
        -- PL/SQL BLOCK내에서 DDL,DCL, 등의 계열은 동적으로만 가능 합니다.
        -- ------------------------------------------------------
        BEGIN
                V_SQL := 'DROP TABLE BY_DYNAMIC';
                EXECUTE IMMEDIATE V_SQL;                                        -- 변수에 SQL 저장후 사용
        EXCEPTION
                WHEN OTHERS THEN
                        DBMS_OUTPUT.PUT_LINE('DYNAMIC SQL  DROP => '||SUBSTR(SQLERRM,1,50));
        END;

        BEGIN
                EXECUTE IMMEDIATE 'CREATE TABLE BY_DYNAMIC(X DATE)';            -- 직접 SQL 문장  사용
        EXCEPTION
                WHEN OTHERS THEN
                        DBMS_OUTPUT.PUT_LINE('DYNAMIC SQL  CREATE =>'||SUBSTR(SQLERRM,1,50));
        END;
END;
/


DESC BY_DYNAMIC
```

① PL/SQL BLOCK내어서 DDL 을 직접 사용할수 없습니다. 실행시 Error가 발생 합니다.

② V_SQL 변수에 SQL문장을 저장한후 변수를 EXECUTE IMMEDIATE 사용

BY_DYNAMIC 테이블이 없는 경우 EXECUTE IMMEDIATE 실행시 EXCEPTION 발생

③ EXECUTE IMMEDIATE 에 직접 SQL문 사용

④ DESC 명령어들 통해 생성된 테이블 구조 확인

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

9_DYNAMIC_SQL_2.SQL

PL/SQL BLOCK내에서 DDL계열의 명령어를 사용하기 위해 동적 SQL을 사용하는 사례를 실습 해보았습니다. 이제는 개발의 유연성을 위해 동적 SQL을 사용하는 실습을 해봅시다.

```
DECLARE
        V_SQL                   VARCHAR2(2000);
        V_CONDITION_FLAG        BOOLEAN  := TRUE;
        R_EMP                   EMP%ROWTYPE;
BEGIN
        V_CONDITION_FLAG := FALSE;
        -- ---------------------------------------
        -- SELECT 문을  동적으로 작성
        -- ---------------------------------------
        BEGIN
                V_SQL := ' SELECT *  FROM EMP WHERE EMPNO = :V_EMPNO';

                IF V_CONDITION_FLAG THEN        -- 프로그램 실행시 상황에 따라 SELECT 문장이 달라진다.
                        V_SQL := V_SQL||'  AND JOB = ''SALESMAN'' ';
                END IF;

                EXECUTE IMMEDIATE  V_SQL INTO R_EMP  USING 7499;

                DBMS_OUTPUT.PUT_LINE('DYNAMIC SELECT  EMPNO='||R_EMP.EMPNO||'  ,ENAME='||R_EMP.ENAME);
        EXCEPTION
                WHEN  OTHERS THEN
                        DBMS_OUTPUT.PUT_LINE('DYNAMIC SQL SELECT => '||SUBSTR(SQLERRM,1,50));
        END;

        -- ---------------------------------------
        -- DDL , DML 문을  동적으로 작성
        -- ---------------------------------------
        BEGIN
                EXECUTE IMMEDIATE 'CREATE TABLE BONUS_LIST(EMPNO NUMBER(7), AMOUNT  NUMBER)';

                V_SQL :=  'INSERT INTO BONUS_LIST(EMPNO,AMOUNT) VALUES(:1,:2)';
                EXECUTE IMMEDIATE V_SQL USING 7499,7000;                        --  :1 ,:2는  어떤 의미?
                COMMIT;
        EXCEPTION
                WHEN  OTHERS THEN
                        DBMS_OUTPUT.PUT_LINE('DYNAMIC SQL DDL AND DML  => '||SUBSTR(SQLERRM,1,50));
        END;
END;
/

DESC BONUS_LIST
SELECT * FROM BONUS_LIST;
```

① 변수 V_CONDITION_FLAG는 BOOLEAN 타입으로 프로그램 실행중에 상황에 따라 TRUE or FALSE 의 값을 가질수 있습니다. 예제상에서는 실행시점에 FALSE 값을 가지게 된다.

② V_CONDITION_FLAG가 TRUE 일때는 조건절에 EMPNO 컬럼과 JOB 컬럼이 사용되고 FALSE 일때는 조건절에 EMPNO 컬럼만 사용 된다

실행 시점에 실행되는 상황에 따라 SQL이 달라 지게 됩니다.
실행시점의 여러가지 상황에 따라 SELECT 문의 조건절을 다르게 부여해야 할 경우에 이를 하나의 정적 SQL로 표현하는 것은 매우 어려울 것이며 또한 조건 분기(IF절)를 통해 여러 SQL을 조건에 따라 처리 하도록 작성하는 것은 많은 코딩 노력이 필요하다.

[질문] V_SQL에 사용된 SELECT 문중 :V_EMPNO 는 어떤 의미일까요 ?

[질문] EXECUTE IMMEDIATE 에서는 INTO 와 USING은 어떤 의미일까요 ?

INTO는 SELECT 한 결과를 INTO 이하의 변수(스칼라변수 OR 컴포짓 변수)에 저장하라는 의미이다,

USING은 바인드변수에 데이터 값을 매핑(대입) 하라는 의미 입니다.
USING 구문에 의해
WHERE EMPNO = :V_EMPNO ➔ WHERE EMPNO = 7499 로 바인딩 되어 실행

③ ②는 실행시점에 조건절이 결정되는 SELECT 문의 예를 보았습니다.
이번에는 INSERT 할 대상이 사전에 정의 되지 않고 실행시점에 정의 되어
DML 문을 동적으로 사용하는 예를 보겠습니다.
실행시점에 BONUS_LIST 테이블을 생성 합니다.
INSERT 문을 실행 시점에 동적으로 구성 합니다. VALUES절에
사용되는 :1, :2는 어떤 의미일까요 ?
앞에 (:)세미콜론이 붙었으니 바인드 변수를 의미하는 것이 겠죠!
일반적으로 변수명에 숫자가 사용되지 못하거나 좋지 않은 코딩 습관으로
알려져있지만 PL/SQL 개발시에는 편리한 방법 입니다. 변수의 이름 보다는 변수의
순서가 의미가 있는 경우에 여러 변수명을 일일이 정의하는것 보다 순서적으
로 :1 , :2 라고 변수명을 사용하는 것이 쉽고 직관적이기 때문 입니다.
USING 구문을 사용하여 7499 ➔ :1 , 7000 ➔ :2 에 매핑(대입)하여 INSERT를 수행

<참고> ANONYMOUS PL/SQL BLOCK 내에서 동적으로 테이블 생성하는 경우에는 EXECUTE IMMEDIATE 를 사용하여 쉽게 생성 하지만 STORED BLOCK 내에서 동적으로 테이블 생성시에는 에러가 발생합니다.잘못 알려져있는 공개된 방식은 해당 계정에게 CREATE ANY TABLE 권한을 주어서 실행하는 방법 입니다.
올바른 방법은 호출자 권한으로 (AUTHID CURRENT_USER 구문 삽입)으로 해당 Stored Block 을 생성/실행

```
CREATE OR REPLACE PROCEDURE  PROCEDURE_CREATE_TABLE
AUTHID CURRENT_USER
IS
                V_SQL           VARCHAR2(2000);
                R_EMP           EMP%ROWTYPE;
BEGIN
                V_SQL := 'CREATE TABLE  BY_DYNAMIC(X DATE)';
                EXECUTE IMMEDIATE V_SQL;

EXCEPTION
                WHEN OTHERS THEN
                        DBMS_OUTPUT.PUT_LINE('DYNAMIC...  '||SUBSTR(SQLERRM,1,50));
END;
/

>
```

9-2 Dynamic CURSOR

```
DECLARE
        CURSOR  CUR_EMP IS
                 SELECT ENAME,JOB,SAL,COMM FROM EMP WHERE  DEPTNO = 10;

        R_CUR_EMP       CUR_EMP%ROWTYPE;
BEGIN
        OPEN    CUR_EMP;
        LOOP
                FETCH  CUR_EMP  INTO  R_CUR_EMP;

                EXIT   WHEN  CUR_EMP%NOTFOUND;

                INSERT INTO BONUS(ENAME,JOB,SAL,COMM)
                      VALUES(R_CUR_EMP.ENAME,R_CUR_EMP.JOB,R_CUR_EMP.SAL,R_CUR_EMP.COMM);

        END LOOP;

        DBMS_OUTPUT.PUT_LINE('TOTAL '||TO_CHAR(CUR_EMP%ROWCOUNT)||' rows processed');

        CLOSE   CUR_EMP;
        COMMIT;
END;
/
```

선언부에서 CURSOR를 정의한후 실행부에서 CURSOR를 OPEN,FETCH,CLOSE 하여 처리 합니다. 선언부에서 커서가 미리 정의 되었습니다. 즉 정적 CURSOR입니다.
이전 CHAPTER에서 Dynamic SQL에서 학습하였듯이 만약 CURSOR의 정의가 실행 시점에 유연하게 변경되어야 한다면 선언부에서 정의하는 Static Cursor 대신에 실행 시점에 Cursor를 정의하는 Dynamic Cursor를 사용해야 합니다.

9_DYNAMIC_CURSOR.sql

```
DECLARE
        CUR_VAR         SYS_REFCURSOR;          -- 커서변수 선언
        R_CUR_EMP       EMP%ROWTYPE;
        V_SQL           VARCHAR2(200);
BEGIN

        V_SQL := 'SELECT ENAME,JOB,SAL,COMM FROM EMP WHERE  DEPTNO = :V_DEPTNO';

        OPEN    CUR_VAR FOR V_SQL USING 10;
        LOOP
                FETCH  CUR_VAR  INTO  R_CUR_EMP.ENAME, R_CUR_EMP.JOB,R_CUR_EMP.SAL,R_CUR_EMP.COMM;

                EXIT   WHEN  CUR_VAR%NOTFOUND;

                INSERT INTO BONUS(ENAME,JOB,SAL,COMM)
                      VALUES(R_CUR_EMP.ENAME,R_CUR_EMP.JOB,R_CUR_EMP.SAL,R_CUR_EMP.COMM);

        END LOOP;

        DBMS_OUTPUT.PUT_LINE('TOTAL '||TO_CHAR(CUR_VAR%ROWCOUNT)||' rows processed');

        CLOSE   CUR_VAR;
        COMMIT;
END;
/
```

① 위의 예제는 5차시에 학습한 5_CURSOR_3.SQL 를 Dynamic Cursor로 변환한 예제 이며 동일한 기능을 합니다. SYS_REFCURS`OR 타입의 CUR_VAR 라는 커서변수를 선언 한다.커서 변수는 C 언어의 Pointer 변수를 연상하시면 쉽게 이해 하실 수 있다.

개발언어에서 사용하는 변수는

(1) 새로운 값을 저장하거나 변수에 저장된 값을 바꿀수 있고
(2) PROCEDURE/FUNCTION간에 변수를 통해 값을 전달하거나 전달 받을수 있습니다.

PL/SQL에서의 커서 변수(CURSOR VARIABLE)는 커서를 변수 처럼 사용하겠다는 의미
(1) Dynamic Cursor 의 결과집합을 가리키는 Pointer 변수로 사용
(2) 커서의 결과 집합 데이타(Result Set)를 PL/SQL PROCEDRE/FUNCTION간에 파라미터 변수로 주고 받는 용도

커서 변수(CURSOR VARIABLE)를 선언 하는 방법은 2가지가 있습니다.
(1) SYS_REFCURSOR
(2) REF CURSOR

(1)는 PL/SQL의 데이터 타입으로 선언부에서 변수 선언 하듯이 선언하면 됩니다.
EX) CUR_VAR SYS_REFCURSOR;
(2)는 사용자 정의 타입입니다.
EX) TYPE ref_cursor _t IS REF CURSOR;
CUR_VAR ref_cursor_t;
< 참고> SYS_REFCURSOR 이 직관적이며 사용하기 편리 합니다.

② 실행 시점에 실행부(BEGIN ~ END)에서 V_SQL 변수에 동적으로 커서를 정의 한다.
실행시점의 상황에 따라 V_SQL 변수에 커서의 정의 구문인 SELECT문장의 조건을 임의적으로 변경 할수 있다

OPEN CUR_VAR FOR V_SQL USING 10;
정적인 CURSOR에서는 OPEN 문구 다음에 커서의 이름이 왔으나
동적인 CURSOR에서는 - 커서 변수(Cursor Variable)이 CUR_VAR 가 사용 됩니다..
- FOR 구문 다음에서 동적 커서의 정의 가 위치 하게 됩니다.
- USING은 바인드변수에 데이터값을 매핑(대입) 합니다.

FETCH,CLOSE에 커서변수가 사용된다는 점만 주의하면 정적 커서를 사용하는 방식과 동일 합니다.

③ 커서 속성자 사용도 정적 커서와 동일 합니다.

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

9-3 AUTONOMOUS TRANSACTION

AUTONOMOUS의 사전적 의미는 자치권 있는 독립적인 뜻을 가지고 있습니다.

AUTONOMOUS TRANSACTION은 독립적인 트랜잭션이라는 의미 입니다.
무엇으로부터 독립적인지 실습 해봅시다.

9_AUTONOMOUS _TRANSACTION_1.SQL

```
REM  ------------------------------------------
REM  EXCEPTION 발생을 기록하는 LOG 테이블 생성
REM  ------------------------------------------

DROP    TABLE  EXCEPTION_LOG;
CREATE  TABLE  EXCEPTION_LOG
(       LOG_DATE        VARCHAR2(8)   DEFAULT TO_CHAR(SYSDATE,'YYYYMMDD'),-- 로그 기록 일자 YYYYMMDD
        LOG_TIME        VARCHAR2(6)   DEFAULT TO_CHAR(SYSDATE,'HH24MISS'),-- 로그 기록 시간 HH24MISS
        PROGRAM_NAME    VARCHAR2(100),                                     --  EXCEPTION 발생 프로그램
        ERROR_MESSAGE   VARCHAR2(250),                                     --  EXCEPTION MESSAGE
        DESCRIPTION     VARCHAR2(250)                                      -- 비고 사항
);

REM  ------------------------------------------------------
REM  EXCEPTION을 기록하는 WRITE_LOG PROCEDURE 생성
REM  ------------------------------------------------------

CREATE OR REPLACE PROCEDURE
        WRITE_LOG(A_PROGRAM_NAME IN VARCHAR2,A_ERROR_MESSAGE IN VARCHAR2,A_DESCRIPTION IN VARCHAR2)
AS
BEGIN
        -- EXCEPTION을 LOG 테이블에 기록
        INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                            VALUES(A_PROGRAM_NAME,A_ERROR_MESSAGE,A_DESCRIPTION);
        -- COMMIT;
EXCEPTION
        WHEN OTHERS THEN
                NULL;
END;
/
```

학습을 열심히 하신 분들은 익숙하게 보신 테이블과 PROCEDURE이죠!
7차시에 학습한 ERROR 기록하는 테이블과 PROCEDURE를 생성 합니다.
WRITE_LOG PROCEDURE내에 COMMIT 부분이 주석으로 되어 있는 것을 주의 깊게
기억 해두시기 바랍니다

```
BEGIN
        BEGIN
                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_1','FIRST BLOCK INSERT 1','MAIN TRANSACTION');

                WRITE_LOG('TRANSACTION TEST_1','FIRST BLOCK  INSERT 2','SUB TRANSACGTION');

                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_1','FIRST BLOCK  INSERT 3','MAIN TRANSACTION');
                COMMIT;
        END;

        BEGIN
                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_2', 'SECOND BLOCK INSERT 1','MAIN TRANSACTION');
                WRITE_LOG('TRANSACTION TEST_2','SECOND BLOCK INSERT 2','SUB TRANSACGTION');
                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_2', 'SECOND BLOCK INSERT 3','MAIN TRANSACTION');

                ROLLBACK;
        END;
END;
/

COL  PROGRAM_NAME  FORMAT A20
COL  ERROR_MESSAGE FORMAT A20
COL  DESCRIPTION   FORMAT A20

SELECT * FROM EXCEPTION_LOG;
```

① TRANSACTION의 시작과 끝을 관찰 해봅시다.

Nested Block내의

첫번째 INSERT 시 TRANSACTION이 시작 됩니다.(MAIN TRANSACTION)

WRITE_LOG PROCEDURE를 호출하여 두번째 INSERT를 수행합니다.

(SUB TRANSACTION)

세번째 INSERT 로 TRANSACTION이 진행중입니다 .(MAIN TRANSACTION)

COMMIT로 TRANSACTION이 종료됩니다.

COMMIT의 범위는 INSERT 2개와 WRITE_LOG에 의한 INSERT 1개 이겠죠!

② Nested Block내의

첫번째 INSERT 시 TRANSACTION이 시작 됩니다.(MAIN TRANSACTION)

WRITE_LOG PROCEDURE를 호출하여 두번째 INSERT를 수행합니다.

(SUB TRANSACTION)

세번째 INSERT 로 TRANSACTION이 진행중입니다 .(MAIN TRANSACTION)

ROLLBACK으로 TRANSACTION이 종료됩니다.

ROLLBACK의 범위는 INSERT 2개와 WRITE_LOG에 의한 INSERT 1개 이겠죠!

③ 어떤결과가 나올까요?

첫번째 Nested Block 에서 INSERT 된 3개의 ROW가 나타나겠죠!

두번째 Nested Block 에서 INSERT 된 3개는 전부 ROLLBACK 되어 최소 되었죠!

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

첫번째 테스트 결과는 이미 Stored Block과 Transaction 관계 시간에 학습을 하였습니다.
WRITE_LOG는 문제발생시 상황을 기록해야 하는데 메인 트랜잭션이 ROLLBACK
되는 경우에는 함께 ROLLBACK 되기 때문에 LOG 기록이 지워지게 됩니다.
지워지는 것을 방지 하기 위해 WRITE_LOG 내에서 COMMIT을 수행하게 하면
무엇이 문제 일까요? 메인 트랜잭션에 영향을 미치게 되죠!

메인 트랜잭션의 종료 결과에 관계 없이 독립적인 서브 트랜잭션(Sub Transaction)을
처리해야 하는 상황에서 AUTONOMOUS TRANSACTION 이 필요 합니다.

9_AUTONOMOUS _TRANSACTION_2.SQL

```
CREATE OR REPLACE PROCEDURE
        WRITE_LOG(A_PROGRAM_NAME IN VARCHAR2,A_ERROR_MESSAGE IN VARCHAR2,A_DESCRIPTION IN VARCHAR2)
AS
        PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
        -- EXCEPTION을 LOG 테이블에 기록
        INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                        VALUES(A_PROGRAM_NAME,A_ERROR_MESSAGE,A_DESCRIPTION);
        COMMIT;
EXCEPTION
        WHEN OTHERS THEN
                NULL;
END;
/
```

① PRAGMA는 프로그램 컴파일 단계에서 실행되는 컴파일러 지시자 입니다.
즉 Block 실행전에 컴파일 단계에서 컴파일러 지시자인 PRAGMA를 발견하면
PRAGMA 이후의 지시자를 실행한후 컴파일을 수행 합니다.
Stored Block내에서 실행되는 Transaction은 독립적인
Transaction(AUTONOMOUS_TRANSACTION)으로 정의 하고 해당 Stored Block을
Compile 합니다.

② COMMIT을 사용합니다. 독립적인 Transaction으로 정의 되었으니 WRITE_LOG
에서 수행되는 COMMIT은 메인 트랜잭션에 영향을 주지 않고 독립적으로 수행 됩니다.

```
BEGIN
        BEGIN
                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_1','FIRST BLOCK INSERT 1','MAIN TRANSACTION');

                WRITE_LOG('TRANSACTION TEST_1','FIRST BLOCK  INSERT 2','SUB TRANSACGTION');

                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_1','FIRST BLOCK  INSERT 3','MAIN TRANSACTION');
                COMMIT;
        END;

        BEGIN
                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_2', 'SECOND BLOCK INSERT 1','MAIN TRANSACTION');
                WRITE_LOG('TRANSACTION TEST_2','SECOND BLOCK INSERT 2','SUB TRANSACGTION');
                INSERT INTO  EXCEPTION_LOG(PROGRAM_NAME,ERROR_MESSAGE,DESCRIPTION)
                                VALUES('TRANSACTION TEST_2', 'SECOND BLOCK INSERT 3','MAIN TRANSACTION');

                ROLLBACK;
        END;
END;
/
```

어떤 결과가 나올까요?

① 첫번째 Nested Block 에서 INSERT 된 3개의 ROW가 나타나겠죠!

② 흥미로운것은 두번째 Nested Block의 결과 입니다.
메인 트랜잭션이 ROLLBACK 되었지만 WRITE_LOG 내에서 수행된 INSERT는 AUTONOMOUS TRANSACTION에 의해서 로그 테이블에 저장되겠죠!

실행결과중 네번째 ROW를 관찰하시기 바랍니다. 메인 트랜잭션이 취소(ROLLBACK)되었지만 AUTONOMOUS TRANSACTION에 의해서 로그 테이블에 반영되었습니다.

9-4 BULK COLLECT & FORALL

BULK 의 사전적 의미는 대량의, 덩어리가 되다 등의 의미를 가지고 있습니다.
여러 개의 공 CD 를 묶어서 한꺼번에 싸게 판매하는 BULK CD 를 보신적 있으시죠!
BULK COLLECT 는 여러 개의 ROW 를 한꺼번에 FECTH 해서 처리하겠다는 의미입니다.
FORALL 은 여러 DML 을 묶어서 한꺼번에 실행 하겠다는 의미 입니다.
5 차시 CURSOR 시간에 1 건씩 FETCH 해서 1 건씩 INSERT 했던 방식 기억나시죠!

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

<5_CURSOR_3.SQL >

```
DECLARE
        CURSOR  CUR_EMP IS
                 SELECT ENAME,JOB,SAL,COMM FROM EMP WHERE  DEPTNO = 10;

        R_CUR_EMP       CUR_EMP%ROWTYPE;
BEGIN
        OPEN    CUR_EMP;
        LOOP
                FETCH  CUR_EMP  INTO  R_CUR_EMP;

                EXIT   WHEN  CUR_EMP%NOTFOUND;

                INSERT INTO BONUS(ENAME,JOB,SAL,COMM)
                      VALUES(R_CUR_EMP.ENAME,R_CUR_EMP.JOB,R_CUR_EMP.SAL,R_CUR_EMP.COMM);

        END LOOP;

        DBMS_OUTPUT.PUT_LINE('TOTAL '||TO_CHAR(CUR_EMP%ROWCOUNT)||' rows processed');

        CLOSE   CUR_EMP;
        COMMIT;
END;
/
```

<참고> 5차시에 CURSOR를 사용하여 데이터 처리하는 방법을 학습했습니다.

만약 CURSOR로 500백만 ROW를 처리해야 한다면

위의 처리 방식에서는 500백만번 LOOP를 돌면서

500백만번 FETCH를 하고

500백만번 INSERT 하게 됩니다.

1차시에 학습했던 PL/SQL 엔진 과 SQL 엔진에 대해서 기억을 되살려 보면

PL/SQL 엔진과 SQL 엔진 사이에 500백만번 CONTEXT SWITCH가 발생

빈번한 CONTEXT SWITCH 와 빈번한 SQL 실행은 성능을 저하 시킵니다..

한꺼번에 여러 ROW를 FETCH 하고 여러 ROW를 한꺼번에 처리해서

CONTEXT SWITCH 와 SQL 실행 횟수를 줄여 성능 향상 시키려는 방법이

BULK COLLECT 와 FORALL 입니다.

PL/SQL로 많은 데이터를 처리해야 하는 경우에 성능을 좌우하는

중요한 방법이므로 꼭 기억해두셔야 합니다.>

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

< 9_BULK_COLLECT.SQL 발췌>

```
REM  ***************************************************************************************************************
REM  **   NAME
REM  **          ANONYMOUS BLOCK)
REM  **   NOTES
REM  **          PL/SQL내에서 CURSOR  처리시  ARRAY PROCESSING (BULK-BINDING)
REM  **   MODIFIED
REM  **        홍길동  09/04/25  -  5_CURSOR_3.SQL 신규 작성
REM  **        홍길동  09/05/05  -  5_CURSOR_3.SQL 를 기반으로  9_BULK_COLLECT.SQL  수정
REM  ***************************************************************************************************************
TRUNCATE  TABLE   BONUS;
SET SERVEROUTPUT ON
DECLARE
        CURSOR  CUR_EMP_LARGE IS
                SELECT ENAME,JOB,SAL,COMM FROM EMP;
        TYPE T_RECORD_EMP IS TABLE OF  CUR_EMP%ROWTYPE  INDEX BY BINARY_INTEGER;
        R_CUR_EMP               T_RECORD_EMP;
        V_ARRAYSIZE             NUMBER(3) :=  3;  -- ArraySize:  1 , 10 , 100, 1000, 10000
BEGIN
        OPEN    CUR_EMP;
        LOOP
                --  Bulk-Collect (Array Fetch )
                FETCH  CUR_EMP  BULK COLLECT INTO R_CUR_EMP LIMIT V_ARRAYSIZE;
                --EXIT   WHEN  CUR_EMP%NOTFOUND;
                EXIT   WHEN  R_CUR_EMP.COUNT=0;
                --  FORALL (Array Processing)
                FORALL I IN R_CUR_EMP.FIRST.. R_CUR_EMP.LAST
                      INSERT INTO BONUS  VALUES R_CUR_EMP(I);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('TOTAL '||TO_CHAR(CUR_EMP%ROWCOUNT)||' rows processed');
        CLOSE   CUR_EMP;
       COMMIT;
END;
/
SELECT * FROM BONUS;
```

데이터베이스 프로그래밍-PL/SQL Advanced (9차시)

실습 데이터 생성

CREATE TABLE EMP_LARGE AS SELECT * FROM EMP;

INSERT INTO EMP_LARGE SELECT * FROM EMP_LARGE; //반복 실행 500만건 이상 생성

COMMIT;

① 5차시에 학습했던 CURSOR(5_CURSOR_3.SQL)을 BULK COLLECT 방식으로
를 재작성한 것 입니다.
CUR_EMP CURSOR를 정의 합니다.
CURSOR를 ROWTYPE 으로 참조하여 PL/SQL TABLE 유형을 정의 합니다.

C언어를 연상하며 해석을 해보면 CURSOR의 레코드를 한꺼번에 레코드형 배열에 저장하겠다는 의도죠!

② V_ARRAYSIZE 변수명과 데이터 3을 기억해두시기 바랍니다.

③ BULK COLLECT 는 여러 개의 ROW를 한꺼번에 FETCH 하겠다는 의미 입니다.
PL/SQL에서는 BULK COLLECT 또는 BULK-BINDING 이라고 불리우고
PRO*C 나 JAVA에서는 ARRAY FETCH 라고 부르지만 동일한 의미이므로 용어를 기억 해두시기 바랍니다.
INTO 이하에 사용된 변수는 스칼라(SCALAR)형 변수가 아니라 컴포지트(COMPOSITE) 유형의 변수인 PL/SQL TABLE 유형의 변수가 사용되었습니다. 즉 1개의 변수가 여러 개의 데이터값을 저장할수 있는 거죠!
LIMIT V_ARRAYSIZE 는 어떤 의미 일까요?
② 에서 따르면 LIMIT 3 의 의미이죠! 3개의 ROW씩 FECTH하라는 의미입니다.
LIMIT이 생략되면 전체 데이터를 전부 가져 옵니다. 무엇이 문제가 될수 있을까요?
시스템의 메모리를 과다하게 사용하여 문제를 일으킬수 있습니다.
BULK COLLECT는 빠른 성능을 제공하는 대신 좀더 많은 시스템 자원(Bulk 추출한 결과 집합을 시스템의 메모리에 저장하게됨) 을 사용하게 됩니다.
BULK COLLECT를 사용하여 개발하는 프로그램이 동시 사용성이 높은 경우라면 자원사용에 대해서 고려를 해야 합니다. 결론적으로 LIMIT를 사용하는 습관을 가지는 것이 좋습니다. 최적의 LIMIT 개수는 상황에 따라 달라지게 됩니다.
흔히 알려진 개수는 100개 ~ 1000개 정도 이지만 여러분이 사용하는 환경에 따라 달라지기 때문에 여러 번 테스트를 통해서 최적의 개수를 찾을 필요가 있습니다.

EXIT 부분에서도 주의 사항이 있습니다.
기존의 커서명%NOTFOUND를 사용하게 되면 LIMIT단위로 처리하다가

일부 잔여 데이터를 처리하지 않고 종료할수도 있습니다.
BULK COLLECT를 사용할때는 주의해서 기억해둘 사항 입니다.
R_CUR_EMP는 PL/SQL TABLE 변수 입니다. COUNT는 변수의 Method 입니다.
이전 차시에서 학습했던 내용이죠!
LIMIT을 3으로 하게 되니 이전에 비해 FETCH 횟수를 3배 줄이게 되는 거죠!

④ FORALL 구문과 DML 문이 같이 사용되어 DML 실행 횟수를 줄이게 됩니다.
빈번한 FETCH가 성능 저하를 일으키듯이 빈번한 DML 연산 역시 성능 저하를 일으키는 원인이 됩니다. FORALL 과 DML을 사용하면 위의 예제에 따르면 INSERT 횟수를 3배 줄이게 됩니다.
FORALL 구문은 FOR 구문을 연상하시면 쉽게 이해가 됩니다.
FOR 구문에서 IN 다음에 FOR 구문의 실행 범위가 나타나게 됩니다.
R_CUR_EMP.FIRST 와 R_CUR_EMP.LAST는 이전에 학습한 TABLE 변수의 Method들입니다. TABLE 변수에 저장된 모든 데이터를 FOR 구문동안 반복하게 됩니다.
3건의 데이터가 FETCH가 되면 FOR 구문을 3번 반복 하겠죠!
INSERT 를 보시면 VALUES 절에 개개의 값을 일일이 나열하지 않고 간결하게 TABLE 변수를 1개만 표시 합니다. 코딩이 간결해졌죠!
주의 하실 사항은 FORALL 구문을 반복하면서 INSERT를 3번 하는 것이 아니라 R_CUR_EMP 변수에 저장되어 있는 3개의 데이터(ROW)를 INSERT 문장과 바인딩을 한후 INSERT 구문은 1번만 실행 됩니다. 쉽게 생각하면 배열에 저장되어 있는 3개의 데이터를 한꺼번에 INSERT 한다고 생각하시면 이해 하기가 쉽습니다.

<참고> 실습은 여러분이 5백만건의 테스트 데이터를 만들어서
1건 단위
10건 단위
100건 단위
1000건 단위
를 각각 테스트 하여 처리시간을 비교해 보시기 바랍니다.
꼭 개인적으로 실습을 하고 성능 차이를 비교할수 있어야 합니다.

BULK COLLECT 를 사용할때는 DBA와 상의를 할 필요가 있습니다.
빠른 대신 많은 자원을 사용하는데 CLIENT의 자원이 아니라 DBMS 서버가 설치된 시스템의 자원을 사용하기 때문에 사용에 주의가 필요 합니다.

BULK COLLECT 와 FORALL 을 정리 해보면
Oracle 8i에서부터 지원하는 기능이며

BULK COLLECT는 여러행을 한꺼번에 FETCH
FORALL 은 여러 ROW를 한꺼번에 DML 연산 처리를 수행하므로
다음의 2가지 사항을 줄임으로 성능 향상을 가져 오게 됩니다.
(1) SQL엔진 과 PL/SQL 엔진 사이의 CONTEXT SWITCH(컨텍스트 전환) 횟수를 줄임
(2) SQL의 실행횟수를 줄임

[학습정리]

1. 정적 SQL(Static SQL)은 프로그램내에 사전에 정의되어 있다가 실행 시점(Run Time)에 실행되는 SQL 입니다.
2. 동적 SQL(Dynamic SQL)은 프로그램 실행시점에 정의되고 실행 되는 SQL 입니다.
   동적 SQL은 어플리케이션 개발의 유연성을 가질수 있다
3. CURSOR VARIABLE 을 선언하는 방법은
   (1) SYS_REFCURSOR
   (2) REF CURSOR
   2가지 방법이 있다.
4. AUTONOMOUS TRANSACTION은 독립적인 트랜잭션으로
   선언부에 PRAGMA AUTONOMOUS_TRANSACTION를 정의하여 사용한다.