애플리케이션 튜닝

웨어밸리 박태욱 대표컨설턴트

개요

시스템의 구조와 용량을 결정하는 요소는 시스템에 접속하는 동시 사용자 수, 사용되는 애플리케이션의 복잡성, 데이터 크기, 네트웍 환경 등을 들 수 있다. 이런 요소들을 적절히 판단하여 시스템이 정상적으로 만들어 졌다면 , 구성 요소들 중에서 시스템 성능에 가장 영향을 많이 주는 것은 애플리케이션이다. 잘 설계되지 못한 애플리케이션은 하드웨어의 구성과 용량에 관계없이 불만족스러운 응답속도를 보여주며 이로 인하여 같이 사용되는 다른 애플리케이션의 응답속도에도 영향을 주게 되어 전체적인 성능에 지대한 영향을 끼친다. 관계형 데이터베이스를 액세스 하는 애플리케이션의 처리속도를 최적화하기 위해서는 기본적으로 성능과 관련된 여러 가지 기능들을 충분히 검토하고 기준을 만들고 이를 관리하는 일련의 과정들이 반드시 있어야 한다. 그렇지 않은 상태에서 개발자 개개인의 능력에만 의지하게 된다면 종국에는 별도의 튜닝에 많은 시간을 할애해야 하고 재작성 해야 하는 프로그램이 많이 발생하게 된다. 여기에서는 프로그램 작성시 반드시 적용 해야 할 여러 가지 기능들을 소개하고 실제 응답속도가 좋지 못한 프로그램 사례들을 소개하고 이의 개선방법 을 제시하고자 한다.

업무기능 분석/ 프로세스 설계

대부분의 업무기능의 분석은 업무방식을 획기적으로 개선한다는 전제가 없다면 현재의 업무영역별로 나누어 기능을 분석한다. 대부분의 프로젝트에서는 기능분석도(FHD)나 프로세스 모델링, DFD(Data Flow Diagram)를 작성하지 않거나 하더라도 문서로서의 역할밖에 하지 못하는 경우가 많다. 이렇게 프로세스분석이 실질적으로 되지 않는 경우에는 데이터와 프로세스간의 상관관계를 정의하고 서로 불일치 하는 부분을 분석하는 절차가 생략되어 각 프로세스의 성능을 예측하는 부분은 불확실하게 측정 되거나 아예 실행하지 않아 결국에는 아무도 책임지지 못하는 시스템이 되고 만다. 특히 분산환경 이거나 병렬처리 (Oracle Parallel Server)환경에서는 더욱더 그러하다. 아래에는 분산환경과 병렬처리환경에서 특히 고려해야 할 여러 가지 요소들을 정리하여 보았다. 사실은 하나하나의 주제를 가지고 많은 내용들을 이야기 해야 하지만 여기에서는 몇 가지 대표적인 주제만 나열하는 것으로 마치려고 한다.

분산환경에서의 검증사항

- 고객을 중복할 것인가 원부를 중복시킬 것인가?
- 고객 이동시 원부를 이동할 것인가?
- 고객의 위치에 관계없이 동일한 서비스를 하려면?
- 조직별 통합실적을 어디서 어떻게 처리할 것인가?
- 이력관리는 어떻게 할 것인가?
- 백업 및 복구정책을 어떻게 할 것인가?
- 일부 시스템 장애시 대책은?
- 잠김현상(Locking)의 최소화 방안은?

OPS환경에서의 검증사항

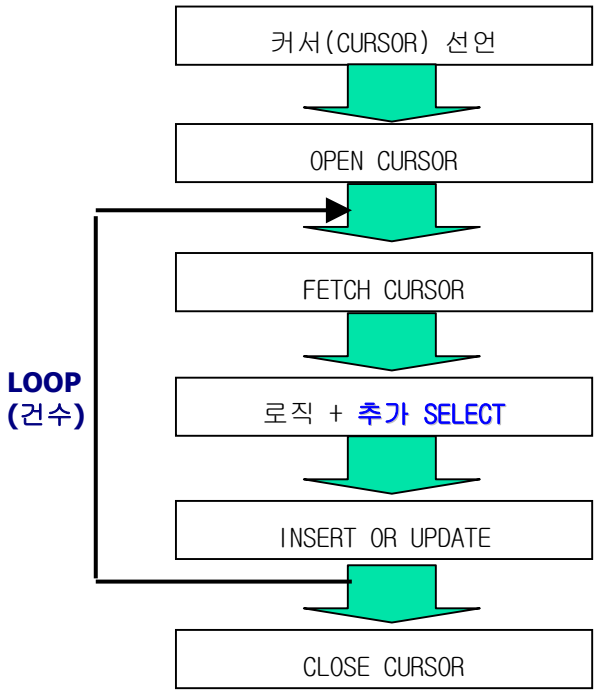
- 노드간 간섭이 일어나는 애플리케이션의 파티션(Partition)
- 노드간 간섭이 일어나는 데이터의 파티션

- ❖ 파티션 키가 설정되지 않는 데이터의 처리
- ❖ PCM LOCK의 최적화
- ❖ 간섭의 정도와 성능의 기대 수준

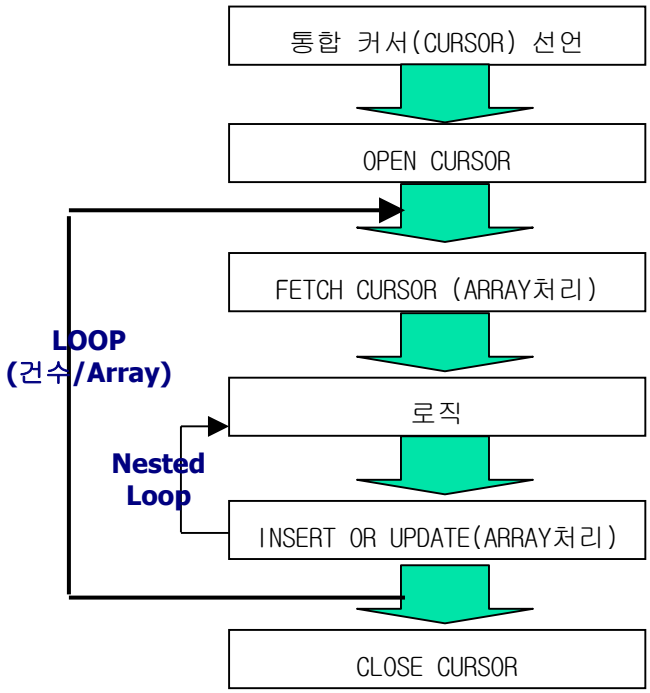
PRO*C에서 배열(Array)의 사용

테이블의 입력,삭제,변경과 조회 Fetch시에는 배열을 사용하여 데이터베이스와 클라이언트간의 콜(call) 횟수를 최소화 하여야 한다. 일반적으로 배열을 사용하는 경우와 그렇지 않은 경우는 작게는 2~3배에서 10배까지 응답속도의 차이가 나타날 수 있으며 배열의 크기는 네트웍환경과 레코드의 길이 등을 감안하여 적당한 크기를 설정하고 튜닝을 통하여 최적의 수치를 찾아 가도록 해야 할 것이며 대략 100~1000사이에서 결정되는 것이 일반적이다. 대개 개발자들은 일반적인 3GL형식의 로직을 구사하며 설령 배열을 사용하는 방법을 알더라도 로직을 추가로 구사하는 것이 귀찮아 사용하지 않는 경우가 많다. 따라서 배열을 사용하는 것을 전체적으로 관리하는 것이 매우 중요하다. 아래는 배열을 사용하는 로직과 사용하는 로직의 차이를 도표로 보여주고 있다.

❖ 일반적인 3GL로직구조



❖ 배열을 활용한 로직구조



PL/SQL에서 벌크(Bulk) 처리

일반적으로 3GL에서 SQL+로직을 사용하는 것보다 PL/SQL을 활용하면 다음과 같은 이점이 있다.

- ■ 로직과 SQL의 처리를 서버에서 하므로 서버의 성능을 최대한 활용할 수 있다. 특히 처리해야 할 데이터가 많은 경우에는 네트웍의 부하를 최소화 함으로써 전체적인 부하를 줄일 수 있다.
- ■ 업무로직이나 공통기능을 저장프로시져(Stored Procedure) 또는 패키지(Package)형태로 서버 한곳에서 유지보수 함으로써 업무로직이 변경 시에도 쉽게 대처할 수 있다.

이에 더하여 ORACLE 8i에서는 PRO*C에서만 가능했던 배열처리 기능을 PL/SQL에서도 벌크로 처리할 수 있도록 기능을 추가함으로써 서버와 로직간의 통신량을 최소화 하고 성능을 최대화 할 수 있도록 하였다. 아래에는 DML에서 사용하는 예와 조회시 FETCH하는 예를 보여주고 있다.

■ DML에서의 벌크처리 사용예

```
declare
        type t_deptno_rec  is table of varchar2(02);
        v_deptno_rec t_deptno_rec := t_deptno_rec('10','20','30');
begin
        forall  i  in  v_deptno_rec.first..v_deptno_rec.last
         update dept
           set dname = 'changed'
         where deptno = v_deptno_rec(i);
end;
/
```

■ 조회시 벌크처리 사용예

```
declare
         TYPE  t_arr  is TABLE OF  VARCHAR2(20);
        V_deptno_arr   t_arr;
        V_dname_arr   t_arr;
        V_num_fetch    integer :=0;
        V_batch_size    number(10) := 500;
        CURSOR  dept_cur  IS
        Select  deptno, dname  from dept where deptno > 20;
begin
        open  dept_cur;
        LOOP
             FETCH  dept_cur
                       BULK COLLECT INTO  v_deptno_arr, v_dname_arr
                        LIMIT v_batch_size;
             EXIT WHEN  dept_cur%NOTFOUND;
             V_num_fetch := v_num_fetch+1;
         END LOOP;
         dbms_output.put_line ('deptno = ' || v_deptno_arr(1));
         dbms_output.put_line ('dname = ' || v_dname_arr(1));
         CLOSE  dept_cur;
 End;
/
```

프리컴파일(PreCompile) 옵션

SQL을 사용하는 모든 프로그램에 대해서 기존의 컴파일전에 프리컴파일을 실행한다. 이때 사용되는 옵션 중 프로그램 파싱(Parsing)과 관련되는 옵션의 값이 올바르게 설정되었는지를 검증할 필요가 있다.

- Hold_cursor

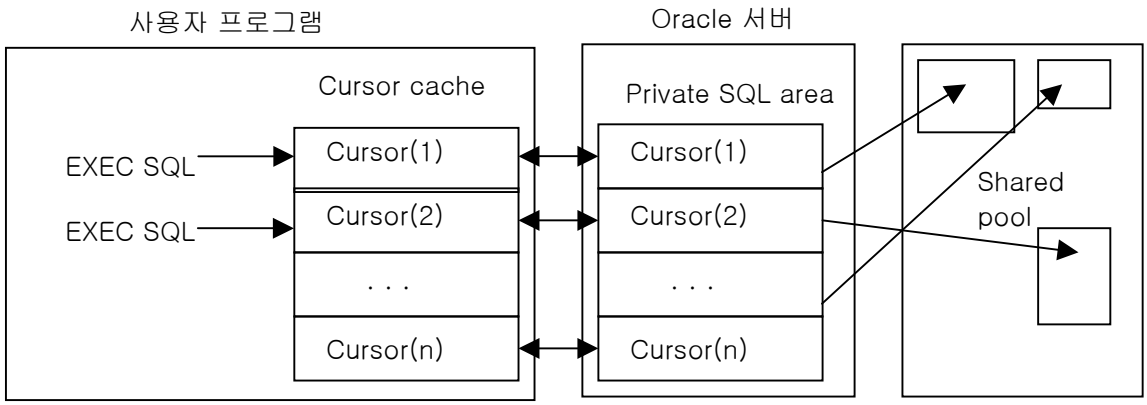
  Hold_cursor 는 프로그램내에 있는 SQL 들이 client context area 에 caching 되는가 를 지정하는 파라메터로 자주 수행되는 프로그램이고 성능을 최대한 높여야 한다면 "YES"로 설정해 주어야 한다. 만약 "NO"로 설정했다면 SQL 수행 시마다 cached sql age out 에 대한 오버헤드를 예상해야 한다. 같은 SQL을 작성했다 하더라도 SQL Cache 에서는 SQL이 사용된 프로그램에서의 위치가 다르다면 다른 SQL 로 인식한다.

- Release_cursor

  Release_cursor 는 수행한 SQL 이 서버쪽의 Private SQL Area 내에서 release 될 수 있음을 지정하는 파라메터로 메모리에 대한 부담이 없고 shared pool 이 충분히 크다면 "NO"로 지정하여 역시 파싱에 대한 부담을 없애야 한다.

- MAXOPENCURSORS

  Hold_cursor 를 "YES" 로 했다 하더라도 MAXOPENCURSORS 의 값이 작게 되어 있다면, SQL 캐쉬영역의 부족으로 또다시 SQL age out 혹은 새로운 SQL 캐쉬(cache) 할당 작업이 발생하며 새로운 할당작업은 코스트(Cost)가 비교적 큰 작업이다. Default 가 10인데 대부분의 프로그램은 10개 이상의 SQL을 가지고 있을 것으로 예상되며 50 정도의 충분히 큰 값으로 주어져야 한다. 그러나 데이터베이스 파라메터 OPEN_CURSORS는 세션별 오픈 할 수 있는 커서의 수로 이것보다 큰 값의 MAXOPENCURSORS는 의미가 없다.



- **파라메터 값에 따른 테스트 결과**

Hold_cursor	Release_cursor	Maxopencursors	parse	Client_time	cpu	elapsed
no	no	1	1704	3.23	0.66	0.39
no	no	10	1301	2.81	0.43	0.49
no	**no**	**20**	**15**	**1.49**	**0.25**	**0.17**
no	yes	1	1704	3.98	0.60	0.42
no	yes	10	1702	3.21	0.41	0.50
no	yes	20	1702	3.21	0.42	0.39
yes	no	1	17	1.83	0.24	0.24
yes	no	10	15	1.46	0.23	0.28
yes	**no**	**20**	**15**	**1.47**	**0.18**	**0.29**
yes	yes	1	1704	3.89	0.42	0.36
yes	yes	10	1702	3.35	0.99	0.44

yes	yes	20	1702	3.20	0.59	0.63
Client 응답시간별						
yes	no	10	15	1.46	0.23	0.28
yes	**no**	**20**	**15**	**1.47**	**0.18**	**0.29**
no	no	20	15	1.49	0.25	0.17
yes	no	1	17	1.83	0.24	0.24
no	no	10	1301	2.81	0.43	0.49
yes	yes	20	1702	3.20	0.59	0.63
no	yes	10	1702	3.21	0.41	0.50
no	yes	20	1702	3.21	0.42	0.39
no	no	1	1704	3.23	0.66	0.39
yes	yes	10	1702	3.35	0.99	0.44
yes	yes	1	1704	3.89	0.42	0.36
no	yes	1	1704	3.98	0.60	0.42
서버 CPU 시간별						
yes	**no**	**20**	**15**	**1.47**	**0.18**	**0.29**
yes	no	10	15	1.46	0.23	0.28
yes	no	1	17	1.83	0.24	0.24
no	no	20	15	1.49	0.25	0.17
no	yes	10	1702	3.21	0.41	0.50
yes	yes	1	1704	3.89	0.42	0.36
no	yes	20	1702	3.21	0.42	0.39
no	no	10	1301	2.81	0.43	0.49
yes	yes	20	1702	3.20	0.59	0.63
no	yes	1	1704	3.98	0.60	0.42
no	no	1	1704	3.23	0.66	0.39
yes	yes	10	1702	3.35	0.99	0.44

테스트 결과에서 알 수 있듯이 가장 성능이 바람직한 수행속도는 Hold_Cursor=Yes, Release_Cursor=No, Maxcopencursors=20인 경우이다. 따라서, 메모리에 대한 제약이 크지 않은 상황이라면 자주 사용되는 프로그램에 대해서 이렇게 설정을 해주는 것이 좋다. (실환경에서는 Maxopencursors=50 정도)

비효율적인 로직을 SQL로 통합

SQL을 많이 사용하지 않았거나 깊이 있게 알지 못하는 대부분의 경우 로직을 풀어쓰는 경우가 많다. 아래는대표적으로 많이 사용되는 로직중의 하나로서 범위가 중복되는 비슷한 내용의 SQL이 LOOP내에서 wm_cur 커서를 포함하여 3번이나 사용되었고 그에 대한 결과를 건마다 입력(Insert)하는 로직이다. 이러한 경우 대부분은 SQL을 모아서 하나의 SQL로 통합할 수 있다.

1

```
EXEC SQL DECLARE wm_cur CURSOR FOR
SELECT  gbond_kind, SUM(issue_amt),  SUM(redemp_amt)
   FROM bmssiiss
WHERE trade_date BETWEEN TO_CHAR(:s_date)  AND TO_CHAR(:e_date)
    AND dwm_gbn = 1
GROUP BY gbond_kind
ORDER BY gbond_kind;

EXEC SQL OPEN wm_cur;
While ( 1 )
{ EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL FETCH wm_cur
                INTO :siiss.gbond_kind,
                     :siiss.issue_amt,
```

```
                        :siiss.redemp_amt ;
 if ( sqlca.sqlcode == NOT_FOUND )   break;
 EXEC SQL WHENEVER SQLERROR DO sqlerror();
 EXEC SQL SELECT remain_amt,  remain_cnt
                    INTO :siiss.remain_amt,  :siiss.remain_cnt
                   FROM bmssiiss
                  WHERE trade_date = TO_CHAR(:e_date)
                     AND dwm_gbn    = 1
                     AND gbond_kind = :siiss.gbond_kind ;
dbl_amt = siiss.remain_amt;
```

2

```
EXEC SQL  SELECT remain_amt
                    INTO :bef_remain_amt
                   FROM bmssiiss
                  WHERE trade_date = TO_CHAR(:bef_e_date)
                     AND dwm_gbn    = 1
                     AND gbond_kind = :siiss.gbond_kind ;

if ( sqlca.sqlcode != NORMAL )
{  if ( sqlca.sqlcode == FETCH_NULL
        || sqlca.sqlcode == NOT_FOUND )
     {  bef_remain_amt = 0;    }
   else
     {  sqlerror();   }
}
dbl_bef_amt = bef_remain_amt;
if ( dbl_bef_amt > 0 )
{  dbl_amt_ch= ((dbl_amt - dbl_bef_amt) * 100)
                    / dbl_bef_amt;     }
else
{  dbl_amt_ch = 0;    }
siiss.remain_amt_ch = dbl_amt_ch;
DB_Insert();
  → insert into bmssiiss values ( …… );
}
```

3

첫번째 SQL의 범위는 두번째와 세번째 SQL의 범위를 포함한다. 따라서 별개로 각각 SQL을 사용하지 않고 첫번째 SQL을 실행하면서 나머지 SQL에 대한 결과값도 얻는 형태로 아래와 같이 SQL을 구사할 수 있다.

```
EXEC SQL DECLARE wm_cur CURSOR FOR
        SELECT  gbond_kind, SUM(issue_amt), SUM(redemp_amt),
                MAX(decode(trade_date/:e_date, 1,remain_amt,0)),
                MAX(decode(trade_date/:e_date, 1,remain_cnt,0)),
                MAX(decode(trade_date/:bef_e_date,1,remain_amt,0))
          INTO  :siiss.gbond_kind, :siiss.issue_amt, :siiss.redemp_amt,
                :siiss.remain_amt, :siiss.remain_cnt, :bef_remain_amt
          FROM bmssiiss
         WHERE trade_date BETWEEN TO_CHAR(:s_date)  AND TO_CHAR(:e_date)
```

```
        AND dwm_gbn = 1
    GROUP BY gbond_kind
    ORDER BY gbond_kind;
```

그 이후 전개되는 로직은 SQL실행결과에 대하여 산술적인 계산만 한다. 이렇게 산술적인 계산만 하는 로직은 복잡하지 않은 경우 SQL내에서 얼마든지 수행이 가능하다. 위의 경우에도 간단한 산술식이므로 아래와 같이 SQL내에서 사용될 수 있으며, INSERT를 별도로 실행할 필요 없이 INSERT ..... SELECT 구문을 이용하여 하나로 SQL을 구성하는 것이 전체적인 처리속도를 빠르게 할 수 있는 최선의 방법이다.

```
INSERT INTO bmssiiss
select TO_CHAR(:e_date),  :wm_bit,
       A.gbond_kind,          A.issue_amt,
       A.redemp_amt,          A.remain_amt,
       ((A.remain_amt – A.bef_remain_amt) * 100) / A.bef_remain_amt,
       A.remain_cnt
from ( SELECT gbond_kind, SUM(issue_amt)  issue_amt,
                  SUM(redemp_amt) redemp_amt,
                  MAX(decode(trade_date/:e_date,1,remain_amt,0)) remain_amt,
                  MAX(decode(trade_date/:bef_e_date,1,remain_amt,0))
                                                          bef_remain_amt,
                  MAX(decode(trade_date/:e_date,    1,remain_cnt,0)) remain_cnt
         FROM bmssiiss
       WHERE trade_date BETWEEN TO_CHAR(:s_date)  AND TO_CHAR(:e_date)
          AND dwm_gbn = 1
     GROUP BY gbond_kind )   A
ORDER BY gbond_kind;
```

SQL은 절차적인 언어가 아니라 집합적인 처리구조를 갖는 언어이다. 따라서 같은 데이터를 몇번씩 반복해서 스캔(SCAN)하고 이를 토대로 로직을 구사하여 결과를 얻는 것은 처리속도에 가장 큰 악영향을 끼친다. 위의 예는 3GL의 로직에는 익숙하지만 SQL의 집합적 처리방식에 대한 이해의 부족에서 비롯되었다. 비단 위의 예 뿐만 아니라 실제 프로젝트에서는 이보다 심하게 로직의 처리를 잘못하는 경우가 비일비재하다. 따라서 이와 같은 사례를 설정하고 이와 유사한 내용에 대해서는 서로 의논을 거쳐 될 수 있으면 하나의 SQL로 처리할 수 있도록 해야 하며, 유지.보수를 위해서 누가 보아도 쉽게 알 수 있도록 그에 대한 내용을 프로그램에 충분히 설명해 놓아야 한다.

DML Returning

ORACLE 8i에 추가된 기능으로 DML에서 실행했던 컬럼들의 값을 변수에 저장할 수 있도록 할 수 있다. 따라서 조회를 실행한 후에 다시 DML을 실행하는 경우 Returning절을 사용하여 이를 하나의 문장으로 SQL의 수를 줄임으로써 서버의 부담을 줄일 수 있다.

■ 한건씩 처리하는 로직에서의 사용예

아래는 테이블에 로우를 입력하기전 지점별 채번테이블에서 번호를 가져오는 로직으로, 테이블을 조회하고 데이터가 있으면 UPDATE를 실행하고 없으면 INSERT를 실행하도록 되어 있다.

```
EXEC SQL  SELECT  OrdNo + 1
                  INTO   :hOrdNo
                  FROM  CZSPOTORDNO
                 WHERE  BnhCd   =:pib->BnhCd   FOR UPDATE;
 if (SQLCODE == SQLEOQ)  {
        hOrdNo  = 1;
        EXEC SQL  INSERT INTO CZSPOTORDNO( BnhCd, OrdNo )
                          VALUES  ( :pib->BnhCd, :hOrdNo);
    }
 EXEC SQL
        UPDATE   CZSPOTORDNO
        SET        OrdNo   =:hOrdNo
        WHERE   BnhCd   =:pib->BnhCd;
```

Returning절을 활용하면 다음과 같이 로직을 변경하여 효율적으로 처리가 가능하다.

```
 EXEC SQL
            UPDATE  CZSPOTORDNO
            SET     OrdNo   = OrdNo + 1
            WHERE   BnhCd   =:pib->BnhCd
             RETURNING ORDNO INTO :hOrdNo;

      if (SQLCODE == SQLEOQ)
      {    hOrdNo  = 1;
            EXEC SQL
                  INSERT INTO CZSPOTORDNO( BnhCd, OrdNo )
                  VALUES  ( :pib->BnhCd, :hOrdNo);
```

■ BULK 처리하는 로직에서의 사용 예

PL/SQL에서 BULK로 처리 시에는 다음과 같이 사용된다.

```
DECLARE
        TYPE  Emplist IS VARRAY(100) OF NUMBER;
        TYPE  Numlist IS TABLE OF Emp_tab.Sal%TYPE;
        Empids EMPLIST := EMPLIST(7369, 7499, 7521, 7566, 7654, 7698);
        Bonlist NUMLIST;
BEGIN
        FORALL i IN Empids.FIRST..empIDs.LAST
        UPDATE Emp_tab SET Bonus = 0.1 * Sal
        WHERE Empno = Empids(i)
        RETURNING Sal BULK COLLECT INTO Bonlist;
```

맺음말

애플리케이션의 구조는 데이터의 구조와 깊은 연관이 있다. 아무리 좋은 방안의 애플리케이션의 구조를 구성하려 해도 데이터의 구조상 불가능하기 때문에 어쩔 수 없이 성능을 희생하는 경우는 비일비재 하다. 이는 전반적인 시스템성능을 상위단계에서부터 충분히 검토하지 않은 결과이다. 따라서 다시 한번 강조하지만 상위단계에서 시간을 많이 할애하여 전체적인 구조상의 문제를 제거하는데 초점을 맞추어야 한다. 그런 다음으로는 공통업무로직을 어떻게 구성할 것인가를 정하고 프로그래머들에 대한 로직의 표준을 정해주는 것이 중요하다. 다시말해 파싱을 최소화하고 배열의 처리를 기본으로 하며 여러 가지 성능에 관련된 지침들을 정해놓고 시작해야 하며, 이는 프로그램이 작성되는 시점 이후에 전반적으로 수정하는 것보다 몇 배 생산성이 좋은 것은 당연하다. 여기게 한가지를 덧붙인다면 관계형 데이터베이스의 성능관점에서 많은 경험을 한 리더의 필요성이다. 프로그램이 완료되고 나서의 튜닝은 한계가 있다. 따라서 프로젝트가 진행되는 시점에 같이 일하면서 전체적인 가이드를 해나가는 리더의 역할은 매우 중요하다.