
Hard Parsing 에 따른 성능 문제와 효과적인 SQL 작성법

1. 들어가며

많은 기업들이 정보 시스템의 근간으로 데이터베이스를 사용하고 있고 또 많은 사람들이 데이터베이스의 성능에 대해 불만을 토로한다. 데이터베이스의 성능 문제와 관련해 많은 원인과 해결책이 있지만 이 문제와 관련해 자주 언급되는 개념이 있다. **Hard Parsing** 이 그것이다. **Hard Parsing** 은 성능에 좋지 못한 영향을 준다는 기술 문서들은 상당히 많다. 하지만 아직도 많은 사람들이 **Hard Parsing** 이라는 것이 어떤 것이고 왜, 어떻게 나쁜 것인지 정확히 인식하고 있지 못한 것 같다. 여러 기업체에서 데이터베이스 성능 개선 활동을 수행하다 보면 **Hard Parsing** 을 유발하도록 프로그램을 개발한 개발자들도 그것이 얼마나 문제를 유발하는지를 알지 못한다. 그렇기 때문에 **Hard Parsing** 을 유발하는 프로그램이 만들어 지고 이것이 문제를 일으키는 경우를 많이 보았다.

이 문서에서 **Parsing** 이라는 작업이 무엇이고 그 중에서 **Hard Parsing** 이 얼마나 성능에 영향을 미치는 지를 알아보도록 하겠다. 하지만 이 문서에서는 단순히 개발툴에서 **Hard Parsing** 을 하지 않도록 프로그램을 작성하는 방법에 대한 논의는 다루지 않을 예정이며 구조적으로 **Hard Parsing** 을 하는 몇 가지 프로그램을 통해 보다 효율적인 **SQL** 작성법에 대한 논의를 하려고 한다.

2. Parsing과 Shared Pool

Parsing

Parsing이란 사용자가 수행을 요청한 SQL 문을 Database에서 실행 가능한 형태로 변경하기 위해 수행하는 일련의 과정을 일컫는다. Parsing은 크게 Sort Paring과 Hard Parsing으로 나누어질 수 있는데 Hard Parsing의 경우 더 많은 시스템 자원을 사용하게 된다.

Parsing이라는 작업은 실제로 사용자의 요구사항을 처리하는 단계가 아닌, 준비 단계로서 사용자 입장에서는 Overhead로 간주될 수 있는 부분이다. 하지만 다음 장에 언급되듯이 Parsing 작업은 내부적으로 상당히 많은 일을 수행할 수 있으며 이에 따라 CPU를 비롯한 시스템 자원의 사용과 그에 해당하는 만큼의 시간을 사용하기 때문에 Parsing 작업에 소요되는 시간은 가능한 최소화 되어야 한다. 오라클에서는 Shared Pool이라는 공간을 이용해 이 문제를 해결하고 있는데 Parsing시 많은 비용이 소요되는 일련의 작업을 수행 결과 생성한 정보를 버리지 않고 공유 메모리에 보관해 두었다가 차후 동일한 SQL문이 사용자에게 의해 요청될 경우 Parsing 작업을 다시 하지 않고 공유 메모리에 보관된 정보를 이용해 '실행' 함으로써 전체 수행속도를 빠르게 하고 보다 많은 CPU 자원이 실제 자료 처리에 사용될 수 있도록 하고 있다.

Shared Pool

Parsing을 설명하며 언급한 공유 메모리가 Shared Pool이다. Shared Pool은 SGA를 구성하고 있는 부분으로 크게 dictionary cache와 library cache로 구성된다.

- dictionary cache
data dictionary 정보를 보관하는 메모리 영역. SQL문에 대한 parsing 작업 또는 PL/SQL 코드에 대한 Compile 작업 시 이 부분의 내용을 참조한다.
- library cache
SQL 또는 PL/SQL 코드의 실행 가능한 형태를 보관하는 메모리 영역. Application에서 사용하는 SQL, PL/SQL문이 사용한다.

오라클에서 수행되는 SQL문과 Parsing된 정보를 공유하기 위해 Shared Pool의 library cache 영역을 사용한다. 오라클 서버가 SQL문에 대한 수행 요청을 받은 경우 library cache 영역에 수행하려는 SQL문에 대한 parsing된 형태가 있는지를 조사하고 그 형태가 존재하는 경우(library cache hit) 그 정보를 사용하는데 이를 soft parse라고 한다. 만일 수행하려는 SQL문이 library cache 영역에 존재하지 않는 경우(library cache miss) 새로 parsing 작업을 수행하며 이를 hard parse라 한다.

그러므로 SGA 영역 중 Buffer Cache가 Disk I/O에 대한 Cache 자원으로 사용되는 것처럼, Shared Pool은 Parsing에 소요되는 CPU Power에 대한 Cache라고 생각할 수 있다.

3. SQL문 수행 절차

SQL 문 수행은 크게 다음과 같은 절차로 이루어진다.

- **Parse** : SQL 문에 대해 문법, Object 존재 여부, 권한등의 검사를 수행한다. 또한 SQL 문에 대한 최적의 실행 방법을 결정
- **Execute** : 실제 자료 처리가 발생하는 부분으로 다양한 사용자의 요구가 이루어짐
- **Fetch** : Select 문장의 경우 처리된 자료가 사용자에게 보내지는 과정

위의 3 가지 수행 단계는 trace 의 결과에서도 나타나는 단계로 가장 기본적인 수행 단계라고 할 수 있다. 여기서는 Execute 나 Fetch 단계보다는 Parse 에 관심을 두고 있으므로 Parse-Execute 단계에서 발생하는 일들을 조금 더 자세히 살펴 보기로 하자.

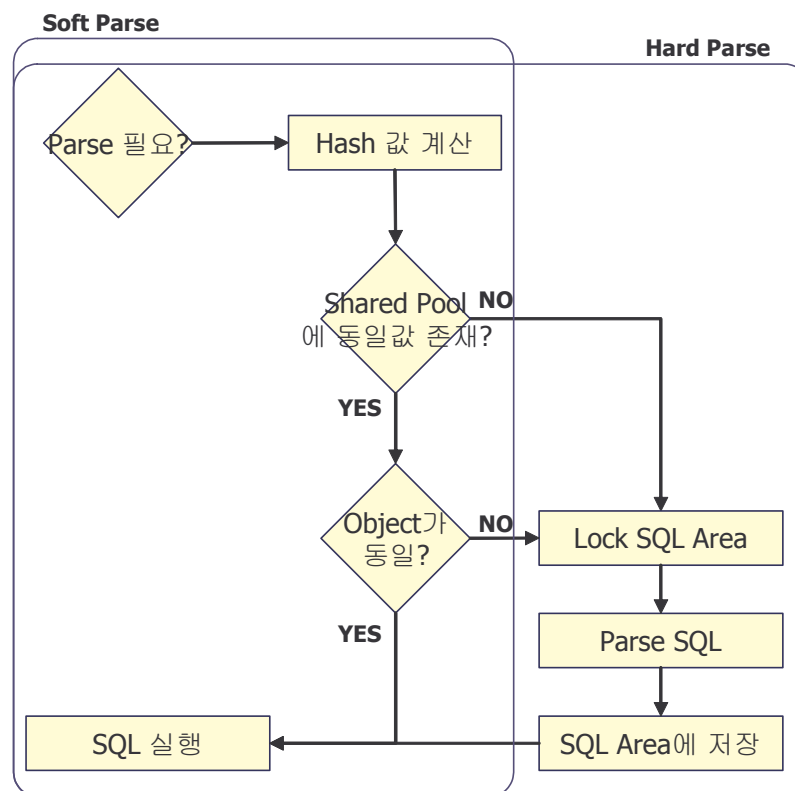


그림 1. Parse Step 개요

위의 그림에서 Hard Parse 로 표시되는 다음과 같은 절차로 이루어진다. (Local Node 에서 작업하는 경우)

- SQL 문에 대한 변환 및 문법 검증
- Data Dictionary 를 통해 SQL 문에서 사용되는 Table 과 Column 정보 검색
- SQL 문에서 사용되는 Object 들이 Parsing 과정 중 변경되는 일을 방지하기 위해 관련 Object 들에 Lock 설정.

-
- 참조되는 Object 들에 대해 SQL 문에서 요구한 작업에 대한 권한 유무 점검.
 - SQL 문에 대한 최적의 실행계획 생성.
 - 산출된 정보들을 library cache 에 저장.

최적의 실행계획 생성 단계는 매우 복잡한 계산을 거치게 되는데 이 과정에서는 (가능한 경우) Database 의 Parameter, Table/Index 의 통계정보 등을 이용해 여러 가지 경우(Table Join 순서 및 방법, Table Scan 및 Index Scan)에 대해 실행계획을 생성하고 이들 중 최적의 실행계획을 얻어낸다. 이 부분은 계산 및 처리과정이 많으므로 Parsing 작업 중 가장 많은 CPU 자원을 사용하는 부분이라고 할 수 있다. 이 부분에서 구체적으로 수행하는 일들은 10053 Event 에 대해 Trace 를 설정하면 볼 수 있으며 Reference 에 기술된 Site 의 문서를 보면 보다 구체적인 정보를 얻을 수 있다.

4. Literal SQL vs Bind SQL

- Literal SQL : SQL 문의 내용 중 특정 값을 지정하거나, 표현하는 부분에 문자, 숫자와 같은 상수값을 Hard 코딩해서 작성한 SQL
- Bind SQL : SQL 문의 내용 중 특정 값을 지정하거나, 표현하는 부분에 변수를 이용해 작성한 SQL

다음 표의 내용을 SQL*Plus 수행하면 수행된 SQL 문이 화면에 표시된다. Literal SQL 문은 변수가 값으로서(Literal) 사용되었으나 Bind SQL 문은 변수가 변수 자체로 사용되었다. 그렇기 때문에 Literal SQL 문은 각 SQL 문마다 1부터 4까지의 값이 값으로서 사용된 반면 Bind SQL 문은 :b0 라는 변수이름이 반복 사용되고 있다.

| | |
|--------------------|--|
| <p>Literal SQL</p> | <pre> set serveroutput on declare v_sql varchar2(1000); v_cnt number; begin for i in 1..4 loop v_sql:='select count(*) from user_objects where rownum<=' i; dbms_output.put_line(v_sql); execute immediate v_sql into v_cnt; end loop; end; / ** 결과로 수행되는 SQL select count(*) from user_objects where rownum<=1 select count(*) from user_objects where rownum<=2 select count(*) from user_objects where rownum<=3 select count(*) from user_objects where rownum<=4 </pre> |
| <p>Bind SQL</p> | <pre> set serveroutput on declare v_sql varchar2(1000); v_cnt number; begin for i in 1..4 loop v_sql:='select count(*) from dba_objects where rownum<=:b0'; dbms_output.put_line(v_sql); execute immediate v_sql into v_cnt using i; end loop; end; / ** 결과로 수행되는 SQL select count(*) from dba_objects where rownum<=:b0 select count(*) from dba_objects where rownum<=:b0 select count(*) from dba_objects where rownum<=:b0 select count(*) from dba_objects where rownum<=:b0 </pre> |

표 1. Literal SQL과 Bind SQL

앞에서 Shared Pool 의 사용 목적이 동일한 SQL 문에 대한 Parsing 정보를 재활용하기 위함이라고 했으나 Literal SQL 문으로 사용하면 조건에 사용된 값이 서로 틀리므로 오라클이 동일한 SQL 문으로 인식하지 않으므로 새롭게 Parsing 을 – Hard Parsing 을 – 수행하게 된다.

앞에서 Hard Parsing 이 자원을 상당히 많이 사용한다고 언급했다. 얼마나 많이 사용하는지를 확인하기 위해 다음과 같은 Test 를 수행하였다. 본 Test 에 사용된 Object 들은 Oracle 9i 에 포함된 demo schema 중 SH Schema 의 Object 를 사용하였다.

- Test 환경

| | |
|------------------------------|--|
| H/W : Pentium 4 1.4GHz | Oracle Version : Oracle9i E/E 9.0.1.4.0 |
| RAM : 512MB | SGA |
| OS : Windows XP Professional | Total System Global Area 126644544 bytes |
| | Fixed Size 282944 bytes |
| | Variable Size 83886080 bytes |
| | Database Buffers 41943040 bytes |
| | Redo Buffers 532480 bytes |

- Test Script

```

** CASE 1. Literal SQL

declare
    v_prod_name          varchar2(50);
    v_unit_price         number;

    v_sql                varchar2(500);

begin
    v_sql:='select /* literal */ prd.prod_name,cost.unit_price
              from sh.products prd, sh.costs cost
              where prd.prod_id=cost.prod_id(+)
              and cost.time_id(+) =to_date("20000101","YYYYMMDD")
              and prd.prod_id=';

    for i in 1..10000 loop
        execute immediate v_sql || i*5 into v_prod_name,v_unit_price;
    end loop;

end;
/



** CASE 2. Bind SQL
declare
    v_prod_name          varchar2(50);
    v_unit_price         number;

begin
    for i in 1..10000 loop
        select /* bind */ prd.prod_name,cost.unit_price
        into v_prod_name,v_unit_price
        from sh.products prd, sh.costs cost
        where prd.prod_id=cost.prod_id(+)
        and cost.time_id(+) =to_date('20000101','YYYYMMDD')
        and prd.prod_id=i*5;
    end loop;

```

end;
/

다음은 Windows 작업 관리자를 통해 측정한 각각의 CPU 사용률과 SQL*Plus 에서 표시된 경과시간 이다.

| | |
|--------------|--|
| <p>Case1</p> |  <p>경과 시간 : 16.04 초 소요</p> |
| <p>Case2</p> |  <p>경과 시간 : 1.00 초 소요</p> |

다음은 v\$sql performance view 에서 추출한 수행횟수 및 Memory 사용 정보이다.

| <p>Case1</p> | <pre>select substr(sql_text,1,20),avg(shareable_mem),count(*),max(executions) from v\$sql where lower(sql_text) like 'select /* literal */%' group by substr(sql_text,1,20);</pre> <table border="1"> <thead> <tr> <th>SUBSTR(SQL_TEXT,1,20)</th> <th>AVG(SHAREABLE_MEM)</th> <th>COUNT(*)</th> <th>MAX(EXECUTIONS)</th> </tr> </thead> <tbody> <tr> <td>select /* literal */</td> <td>16090.0436</td> <td>2112</td> <td>1</td> </tr> </tbody> </table> | SUBSTR(SQL_TEXT,1,20) | AVG(SHAREABLE_MEM) | COUNT(*) | MAX(EXECUTIONS) | select /* literal */ | 16090.0436 | 2112 | 1 |
|-----------------------|---|-----------------------|--------------------|----------|-----------------|----------------------|------------|------|-------|
| SUBSTR(SQL_TEXT,1,20) | AVG(SHAREABLE_MEM) | COUNT(*) | MAX(EXECUTIONS) | | | | | | |
| select /* literal */ | 16090.0436 | 2112 | 1 | | | | | | |
| <p>Case2</p> | <pre>select substr(sql_text,1,20),avg(shareable_mem),count(*),max(executions) from v\$sql where lower(sql_text) like 'select /* bind */%' group by substr(sql_text,1,20);</pre> <table border="1"> <thead> <tr> <th>SUBSTR(SQL_TEXT,1,20)</th> <th>AVG(SHAREABLE_MEM)</th> <th>COUNT(*)</th> <th>MAX(EXECUTIONS)</th> </tr> </thead> <tbody> <tr> <td>SELECT /* bind */ pr</td> <td>16095</td> <td>1</td> <td>10000</td> </tr> </tbody> </table> | SUBSTR(SQL_TEXT,1,20) | AVG(SHAREABLE_MEM) | COUNT(*) | MAX(EXECUTIONS) | SELECT /* bind */ pr | 16095 | 1 | 10000 |
| SUBSTR(SQL_TEXT,1,20) | AVG(SHAREABLE_MEM) | COUNT(*) | MAX(EXECUTIONS) | | | | | | |
| SELECT /* bind */ pr | 16095 | 1 | 10000 | | | | | | |

SQL 문은 10,000 번 수행됐으나 Case1 에 나타난 정보는 2,112 개만 존재하는 것으로 표시된다. 이는 Shared Pool 의 크기가 10,000 개의 SQL 문을 저장할 수 없어 먼저 수행된 SQL 문은 공간 확보를 위해 Shared Pool 에서 삭제되었음을 표시한다. 또 공유가 전혀 되지 않으므로 수행횟수를 표시하는 execution 값의 최대값은 1 을 표시하고 있다. SQL 문 1 개가 사용하는

Sharable Memory 의 크기는 약 16KB 이므로 10,000 개의 SQL 문에 대한 정보를 다 보관하는 경우 160MB 의 공간이 필요하다는 계산이 나온다.

반면 Case2 는 단 1 개의 형태만 존재하고 있으며 그것이 10,000 번 사용됐음을 보여주고 있다. 하지만 사용하는 Memory 량은 여전히 16KB 이므로 Case1 의 160MB 와 비교하면 엄청난 차이를 보이고 있다.

동일한 일을 수행하는 SQL 문을 Literal SQL 과 Bind SQL 로 작성한 경우 Computing 자원의 주요 Resource 인 CPU 와 Memory 를 사용하는 형태에서 상당한 차이가 발생한다는 것을 보았다. 그러면 오라클 내부적으로는 또 어떤 차이가 있을까? 다음 장에서 몇 가지 경우에 대한 소요시간 및 자원 사용량을 알아보기로 하자.

5. 7가지 PARSE 형태별 자원 사용량 비교

Parsing 시 얼마나 많은 자원과 시간이 소요되는지를 알아보기 위해 다음과 같은 TEST 를 수행하였다. 앞서 사용한 것과 마찬가지로 Oracle 9i 에 포함된 demo schema 중 SH Schema 의 Object 를 사용하였다. 이들 Object 들은 \$ORACLE_HOME/demo/schema 에 존재하는 mksample.sql 을 수행해 생성할 수 있다.

Cursor 를 선언하고 실행하기까지의 구현 방법에 꽤 많은 경우의 수로 구분할 수 있으나 모든 경우에 대한 Test 는 지면 관계상 어려우므로 다음과 같은 형태로 수행한 경우의 자원 사용량을 비교하였다.

- Case 1. Execute immediate 를 사용한 Literal SQL
- Case 2. DBMS_SQL 을 사용한 Literal SQL
- Case 3. cursor_sharing=force 설정 후 Execute immediate 를 사용한 Literal SQL
- Case 4. Execute immediate 를 사용한 Bind SQL
- Case 5. DBMS_SQL 을 사용한 Bind SQL(Open, Parse 1 회, Bind, Execute 반복)
- Case 6. Static SQL
- Case 7. DBMS_SQL 을 사용한 Bind SQL(Open 1 회, Parse, Bind, Execute 반복)

- Test 환경

| | |
|------------------------------|--|
| H/W : Pentium 4 1.4GHz | Oracle Version : Oracle9i E/E 9.0.1.4.0 |
| RAM : 512MB | SGA |
| OS : Windows XP Professional | Total System Global Area 126644544 bytes |
| | Fixed Size 282944 bytes |
| | Variable Size 83886080 bytes |
| | Database Buffers 41943040 bytes |
| | Redo Buffers 532480 bytes |

- Test 방식

PC 에 설치된 Oracle Server 에서 Script 수행 용 Session 1 개와 Performance 정보 수집용 Session 1 개만 접속 하였다. Script 수행 용 Session 에서는 경과 시간을 표시하기 위해 'set timing on'을 수행하였으며 Performance 정보 수집을 위해서 v\$sesstat, v\$latch 값을 Script 수행 전,후로 구해 그 차이를 기록하였다. 각각의 Test 수행 전 shared pool 에 대한 flush 를 수행해 Parsing 과 관련해 다른 Test 에서 생성된 정보를 삭제하였다.

- Test Script

```
** CASE 1. Literal SQL (Native Dynamic SQL)
declare
```

```

        v_prod_name          varchar2(50);
        v_prod_list_price number;
        v_unit_price         number;

begin
    v_sql                    varchar2(500);
    v_sql:='select
                prd.prod_name
                ,prd.prod_list_price
                ,cost.unit_price
            from sh.products prd, sh.costs cost
            where prd.prod_id=cost.prod_id(+)
            and cost.time_id(+)=to_date("20000101","YYYYMMDD")
            and prd.prod_id=';
    for i in 1..10000 loop
        execute immediate v_sql || i*5 into v_prod_name,v_prod_list_price,v_unit_price;
    end loop;
end;
/

** CASE 2. Literal SQL (DBMS_SQL)
declare
    v_prod_name          varchar2(50);
    v_prod_list_price number;
    v_unit_price         number;

    v_sql                    varchar2(500);
    cur_sql_exec          number;
    ignore                 number;
begin
    cur_sql_exec:=dbms_sql.open_cursor;
    v_sql:='select
                prd.prod_name
                ,prd.prod_list_price
                ,cost.unit_price
            from sh.products prd, sh.costs cost
            where prd.prod_id=cost.prod_id(+)
            and cost.time_id(+)=to_date("20000101","YYYYMMDD")
            and prd.prod_id=';
    for i in 1..10000 loop
        dbms_sql.parse(cur_sql_exec,v_sql || i*5,1);
        dbms_sql.define_column(cur_sql_exec,1,v_prod_name,50);
        dbms_sql.define_column(cur_sql_exec,2,v_prod_list_price);
        dbms_sql.define_column(cur_sql_exec,3,v_unit_price);
        ignore:=sys.dbms_sql.execute(cur_sql_exec);
        ignore:=sys.dbms_sql.fetch_rows(cur_sql_exec);
        dbms_sql.column_value(cur_sql_exec,1,v_prod_name);
        dbms_sql.column_value(cur_sql_exec,2,v_prod_list_price);
        dbms_sql.column_value(cur_sql_exec,3,v_unit_price);
    end loop;
    dbms_sql.close_cursor(cur_sql_exec);
end;
/

** CASE 3. Cursor Sharing 에 의한 공유
alter session set cursor_sharing = force;

declare

```

```

        v_prod_name          varchar2(50);
        v_prod_list_price number;
        v_unit_price         number;

begin
    v_sql                    varchar2(500);
    v_sql:='select
                        prd.prod_name
                        ,prd.prod_list_price
                        ,cost.unit_price
                    from sh.products prd, sh.costs cost
                    where prd.prod_id=cost.prod_id(+)
                    and cost.time_id(+) =to_date("20000101","YYYYMMDD")
                    and prd.prod_id=';
    for i in 1..10000 loop
        execute immediate v_sql || i*5 into v_prod_name,v_prod_list_price,v_unit_price;
    end loop;
end;
/

alter session set cursor_sharing = exact;

** CASE 4. Bind 변수 사용 (Native Dynamic SQL)
declare
    v_prod_name          varchar2(50);
    v_prod_list_price number;
    v_unit_price         number;

begin
    v_sql                    varchar2(500);
    v_sql:='select
                        prd.prod_name
                        ,prd.prod_list_price
                        ,cost.unit_price
                    from sh.products prd, sh.costs cost
                    where prd.prod_id=cost.prod_id(+)
                    and cost.time_id(+) =to_date("20000101","YYYYMMDD")
                    and prd.prod_id=';
    for i in 1..10000 loop
        execute immediate v_sql||':b1' into v_prod_name,v_prod_list_price,v_unit_price
using i*5;
    end loop;
end;
/

** CASE 5. Bind 변수 사용 (DBMS_SQL)
declare
    v_prod_name          varchar2(50);
    v_prod_list_price number;
    v_unit_price         number;

    v_sql                varchar2(500);
    cur_sql_exec         number;
    ignore               number;

begin
    cur_sql_exec:=dbms_sql.open_cursor;
    v_sql:='select
                        prd.prod_name

```

```

        ,prd.prod_list_price
        ,cost.unit_price
    from sh.products prd, sh.costs cost
    where prd.prod_id=cost.prod_id(+)
    and cost.time_id(+) = to_date('20000101','YYYYMMDD')
    and prd.prod_id='';
dbms_sql.parse(cur_sql_exec,v_sql||':b0',1);
for i in 1..10000 loop
    dbms_sql.define_column(cur_sql_exec,1,v_prod_name,50);
    dbms_sql.define_column(cur_sql_exec,2,v_prod_list_price);
    dbms_sql.define_column(cur_sql_exec,3,v_unit_price);
    dbms_sql.bind_variable(cur_sql_exec,'b0',i*5);
    ignore:=sys.dbms_sql.execute(cur_sql_exec);
    ignore:=sys.dbms_sql.fetch_rows(cur_sql_exec);
    dbms_sql.column_value(cur_sql_exec,1,v_prod_name);
    dbms_sql.column_value(cur_sql_exec,2,v_prod_list_price);
    dbms_sql.column_value(cur_sql_exec,3,v_unit_price);
end loop;
dbms_sql.close_cursor(cur_sql_exec);
end;
/

** CASE 6. Static SQL

declare
    v_prod_name          varchar2(50);
    v_prod_list_price number;
    v_unit_price         number;
begin
    for i in 1..10000 loop
        select prd.prod_name,prd.prod_list_price,cost.unit_price
        into v_prod_name,v_prod_list_price,v_unit_price
        from sh.products prd, sh.costs cost
        where prd.prod_id=cost.prod_id(+)
        and cost.time_id(+) = to_date('20000101','YYYYMMDD')
        and prd.prod_id=i*5;
    end loop;
end;
/

** CASE 7. Bind 변수 사용 (DBMS_SQL) + 반복 PARSE

declare
    v_prod_name          varchar2(50);
    v_prod_list_price number;
    v_unit_price         number;

    v_sql                varchar2(500);
    cur_sql_exec         number;
    ignore               number;
begin
    cur_sql_exec:=dbms_sql.open_cursor;
    v_sql:='select
            prd.prod_name
            ,prd.prod_list_price
            ,cost.unit_price
        from sh.products prd, sh.costs cost
        where prd.prod_id=cost.prod_id(+)
        and cost.time_id(+) = to_date('20000101','YYYYMMDD')
    ';
```

```

        and prd.prod_id='';
    for i in 1..10000 loop
        dbms_sql.parse(cur_sql_exec,v_sql||':b0',1);
        dbms_sql.define_column(cur_sql_exec,1,v_prod_name,50);
        dbms_sql.define_column(cur_sql_exec,2,v_prod_list_price);
        dbms_sql.define_column(cur_sql_exec,3,v_unit_price);
        dbms_sql.bind_variable(cur_sql_exec,'b0',i*5);
        ignore:=sys.dbms_sql.execute(cur_sql_exec);
        ignore:=sys.dbms_sql.fetch_rows(cur_sql_exec);
        dbms_sql.column_value(cur_sql_exec,1,v_prod_name);
        dbms_sql.column_value(cur_sql_exec,2,v_prod_list_price);
        dbms_sql.column_value(cur_sql_exec,3,v_unit_price);
    end loop;
    dbms_sql.close_cursor(cur_sql_exec);
end;
/

```

• Test 결과

| 구 분 | | Case 1 | Case 2 | Case 3 | Case 4 | Case 5 | Case 6 | Case 7 |
|----------------|------------------------------|---------|---------|--------|--------|--------|--------|--------|
| 소요시간 | | 15.07 | 16.04 | 2.05 | 1.08 | 1.06 | 1.01 | 2.08 |
| Stat | execute count | 10365 | 10415 | 10361 | 10226 | 10312 | 10216 | 10320 |
| | recursive calls | 15346 | 35902 | 15250 | 12788 | 33960 | 12261 | 44122 |
| | parse count (total) | 10182 | 10226 | 10184 | 10096 | 171 | 110 | 10172 |
| | parse count (hard) | 10042 | 10043 | 45 | 41 | 41 | 41 | 43 |
| | parse time cpu | 1299 | 1320 | 143 | 40 | 29 | 22 | 28 |
| | parse time elapsed | 1307 | 1327 | 145 | 43 | 29 | 31 | 30 |
| | opened cursors cumulative | 10150 | 173 | 10152 | 10078 | 117 | 86 | 119 |
| | CPU used by this session | 1560 | 1632 | 248 | 171 | 163 | 106 | 251 |
| Latch (get) | library cache | 497408 | 498217 | 135531 | 314887 | 5330 | 25022 | 315422 |
| | shared pool | 633460 | 635323 | 13387 | 23273 | 3510 | 2895 | 23606 |
| | row cache objects | 1412354 | 1412423 | 2121 | 2052 | 2060 | 1952 | 2140 |

결과는 크게 3 가지 부분으로 구분되며 다음과 같은 의미를 갖는다.

- 소요시간 : SQL*Plus 에서 set timing on 을 수행한 후 각 작업이 종료된 후 표시되는 시간. 실제로 Database 에서 수행되는 것과 약간의 오차가 있을 수 있으나 10046 Trace 를 수행한 경우 본 Test 환경의 File System 으로 부더의 Overhead 가 매우 크게 발생해 실제 값 자체를 상당히 왜곡시키기 때문에 이 지표를 사용했다. 하지만, 이 지표는 절대적으로 신뢰할 수 있는 값은 아니며 특히 소수점 이하로는 오차가 클 수 있기 때문에 단순 참고용으로 기재했다.
- exeute count : 각 Operation 수행을 위해 처리된 SQL 문의 개수.
- recursive calls : Oracle 이 내부적으로 요구되는 작업을 수행하기 위해 수행한 횟수와 PL/SQL 에서 호출된 SQL 문의 횟수이다. 내부적으로 요구되는 작업은 Hard Parsing 시 내부적으로 사용되는 SQL 문으로 볼 수 있으며 나머지는 PL/SQL 문에서 수행되는 횟수로 볼

수 있다. `execute count` 와 `recursive calls` 모두 자료 수집 방법에 따라 조금 다른 결과를 보여주므로 여기서는 절대적인 지표에 대해 언급하기 보다는 상대적인 지표로 언급한다.

- `parse count (total)` : 작업 수행 중 발생한 `Parse Count (Hard Parse + Soft Parse)`
- `parse count (hard)` : 작업 수행 중 발생한 `Hard Parse Count`
- `parse time cpu` : Parsing 작업 시 소요된 `CPU Time`. `Centi-second(1/100 초)` 단위로 표시되지만 측정 방식의 한계로 인해 약간의 오차가 존재할 수 있다.
- `parse time elapsed` : Parsing 작업 시 소요된 경과 시간. `Centi-second(1/100 초)` 단위로 표시된다.
- `CPU used by this session` : 사용자의 요청이 시작해서 종료될 때 까지 사용된 `CPU` 시간. `Centi-second(1/100 초)` 단위로 표시되지만 측정 방식의 한계로 인해 약간의 오차가 존재할 수 있다.
- `library cache` : `library cache latch` 에 접근한 횟수를 나타낸다. 몇 가지 요인이 영향을 미칠 수 있으나 본 Test 에서는 `(Hard) Parse Count` 가 변동 요인이므로 이들 값의 변화량에 대한 변동 값을 측정할 수 있다.
- `shared pool` : `shared pool latch` 에 접근한 횟수를 나타낸다.
- `row cache objects` : `row cache objects latch` 에 접근한 횟수를 나타낸다.

이상의 Test 결과 Parse 횟수, 특히 Hard Parse 횟수가 많을수록 수행 시간 및 Latch Access 횟수가 많아진다는 사실을 알 수 있다. 사용하는 SQL 문의 복잡도에 따라 다르지만 본 Test 에서는 수행시간이 최대 16 배의 차이가 나는 것으로 나타났다. 또한 주요 Latch 에 대한 접근 횟수 또한 최대 220 배 가량의 차이가 나는 것으로 나타나 Case 5, Case 6 과 같은 식의 프로그램이 다른 방식보다 우수하다는 것을 알 수 있다.

Test 결과에 의하면 어떤 방법을 사용하던 Literal SQL 문을 사용해 Hard Parse 가 많을 경우 Latch 자원 - 특히 row cache objects latch - 에 대한 상당한 Access 와 Performance 의 저하 현상이 발생하는 것을 알 수 있다. 또한 Case 5 과 Case 7 을 통해 Hard Parse 를 최소화 하더라도 Soft Parse 가 많으면 성능에 - 특히 Latch 자원에 - 문제를 줄 수 있다는 것을 알 수 있다. 또한 execute immediate 에 의해 수행되는 NDS(Native Dynamic SQL)문의 경우 cursor 를 SQL 문 1 개 수행 시 Cursor 를 반복적으로 Open 한다는 것을 opened cursors cumulative 값을 통해 알 수 있으며 DBMS_SQL Package 에 비해 사용하기 편하다는 장점이 있지만 Cursor Open 을 불필요하게 많이 수행한다는 것을 알 수 있다.

Case5 와 Case6 의 경우 어느 방법이 우수한지는 본 Test 만으로 판단하기는 어렵다. 일반적으로 Case6 가 조금 더 우수할 것으로 보이지만 library cache latch 의 Access 횟수가 Case5 보다 더 크기 때문에 library cache latch 가 심하게 문제가 되는 경우라면 Case5 가 우수할 수 있다. 한가지 분명한 점은 Case5 와 Case6 의 경우 Dynamic SQL 과 Static SQL 로 분명히 구분되는 영역이 존재한다는 사실이다. 때문에 Dynamic SQL 을 사용한다면 DBMS_SQL Package 를 사용하는 것이 유리하며 Static SQL 을 사용한다면 복잡한 DBMS_SQL Package 를 사용하기 보다는 위와 같은 방식으로 사용하는 것이 합리적이지 않을까 생각한다.

다음에는 실무에서 발생한 **Literal SQL** 문 사례와 그에 따른 성능문제 그리고 해결 사례를 소개하려고 한다. 하지만 단순히 개발 **Tool** 에서 **Bind SQL** 문을 사용하는 방법을 알지 못해 **Literal SQL** 문이 사용된 사례는 지양하고자 하며 구조적인 문제에서 발생하는 문제들에 대해 다루도록 하겠다.

6. Literal SQL 해결 사례

본 절에서는 전장에서 언급한 대로 Logic 구성, 또는 구조적 문제에서 발생하는 3 가지의 사례를 통해 진행하고자 한다. 사례에서 사용되는 자료는 Oracle 9i 에 포함된 demo schema 를 통해 구현하였다. 제한된 자료를 통해 구현했으므로 실제 업무적인 내용에서는 좀 미진할 수 있으므로 많은 양해 바란다. 각각의 Test 는 SQL 문 수행 전 Shared Pool 을 Flush 한 후 수행되었다.

6.1 Loop 구조로 반복 수행되는 Literal SQL

본 프로그램은 sh 스키마의 product table 내용 중 category 가 men 또는 women 이며 현재 재고가 있는 ('available, on stock') 제품 중 가격이 100 달러가 넘는 것은 판매금액과 판매량을 구하고, 100 달러가 되지 않은 것은 판매 금액만 구하는 logic 이다.

```
01 declare
02     v_sql varchar2(1000);
03     type refcurtype is ref cursor;
04     cur_sql refcurtype;
05     cursor cur_products is
06         select prod_id,prod_name,prod_list_price
07         from sh.products
08         where prod_status = 'available, on stock'
09         and prod_category in ('Men','Women');
10     rec_products cur_products%rowtype;
11     v_amount_sold    number;
12     v_quantity_sold  number;
13 begin
14
15     open cur_products;
16     loop
17         fetch cur_products into rec_products;
18         exit when cur_products%notfound;
19         if rec_products.prod_list_price>100 then
20             v_sql:='select sum(amount_sold),sum(quantity_sold)*' ||
rec_products.prod_list_price || ' from sh.sales where time_id>=to_date(''20000101'', 'YYYYMMDD')
and time_id<to_date(''20010101'', 'YYYYMMDD') and prod_id=' || rec_products.prod_id;
21         else
22             v_sql:='select sum(amount_sold),0 from sh.sales where
time_id>=to_date(''20000101'', 'YYYYMMDD') and time_id<to_date(''20010101'', 'YYYYMMDD') and
prod_id=' || rec_products.prod_id;
23         end if;
24         open cur_sql for v_sql;
25         fetch cur_sql into v_amount_sold,v_quantity_sold;
26 --         dbms_output.put_line('AMOUNT SOLD : ' || v_amount_sold || ', QUANTITY SOLD : '
|| v_quantity_sold);
27         close cur_sql;
28     end loop;
29     close cur_products;
30 end;
```

본 프로그램을 수행시키면 `cur_products cursor` 의 자료 개수만큼 `hard parsing` 이 발생한다. 왜 그럴까? 20 line 과 22 line 에서 `select` 문장을 문자열로 처리하고 있음을 알 수 있다. 이 경우 `rec_products.prod_id` 라는 변수는 상수화 되어 `shared pool` 의 입장에서는 서로 다른 SQL 문으로 수행되기 때문이다. 실무에서 사용하는 개발 툴의 종류를 불문하고 이런 식의 프로그램 방식이 많이 사용된다.

Literal SQL 문과 관련한 문제 해결방법은 보통 2 가지가 있다고 생각한다. 첫번째는 반복 수행되는 SQL 문이 Bind SQL 문을 사용하도록 변경하는 것이고 두번째는 SQL 문을 통합해 반복 수행하는 부분 자체를 없애는 방법이 그것이다.

다음의 Source 는 각각 Bind SQL 문을 사용한 경우와 SQL 문을 통합한 경우를 보여주고 있다.

```
01 declare
02     v_sql varchar2(1000);
03     type refcurtype is ref cursor;
04     cur_sql refcurtype;
05     cursor cur_products is
06         select prod_id,prod_name,prod_list_price
07             from sh.products
08             where prod_status = 'available, on stock'
09                and prod_category in ('Men','Women');
10     rec_products cur_products%rowtype;
11     v_amount_sold    number;
12     v_quantity_sold  number;
13 begin
14     open cur_products;
15     loop
16         fetch cur_products into rec_products;
17         exit when cur_products%notfound;
18         if rec_products.prod_list_price>100 then
19             v_sql:='select sum(amount_sold),sum(quantity_sold)*:b0 from sh.sales
where time_id>=to_date(''20000101'', ''YYYYMMDD'') and time_id<to_date(''20010101'', ''YYYYMMDD'')
and prod_id=:b1';
20             open cur_sql for v_sql using
rec_products.prod_list_price,rec_products.prod_id;
21         else
22             v_sql:='select sum(amount_sold),0 from sh.sales where
time_id>=to_date(''20000101'', ''YYYYMMDD'') and time_id<to_date(''20010101'', ''YYYYMMDD'') and
prod_id=:b1';
23             open cur_sql for v_sql using rec_products.prod_id;
24         end if;
25         fetch cur_sql into v_amount_sold,v_quantity_sold;
26 --         dbms_output.put_line('AMOUNT SOLD : ' || v_amount_sold || ', QUANTITY SOLD : '
|| v_quantity_sold);
27         close cur_sql;
28     end loop;
29     close cur_products;
30 end;
```

19line 과 22 라인에서 문자열을 구성하는 부분은 전과 유사하다. 다만 변수를 `||` Operator 를 통해 결합시키지 않고 `:b1` 라는 부분을 사용한 것과 `open cursor` 명령에서 `using` 절을 이용해 앞서 선언한 `:b1` 부분에 `rec_products.prod_id` 라는 변수를 입력하는 것이 다르다. 이 부분을

통해 각각의 경우 입력 받는 값이 1 개 또는 2 개라도 각각의 경우에 맞는 Bind SQL 문을 사용할 수 있으며 Hard Parsing 에 의한 성능 감소 현상을 거의 없앨 수 있다.

대부분의 경우 위와 같이 처리하면 큰 문제없이 수행이 가능하지만 5 장에서 살펴본 바와 마찬가지로 이 경우에도 Soft Parse 과다 현상 및 반복적인 Cursor 의 Open – Close 에 의한 부하는 여전히 존재한다는 문제가 있다. 따라서 보다 근원적인 문제 해결방식은 SQL 문의 반복 수행 자체를 막는 것이라고 할 수 있다.

아래 Source 를 보자.

```
01 declare
02     cursor cur_products(p_from_time in varchar2, p_to_time in varchar2) is
03         select
04             prd.prod_id
05             ,prd.prod_name
06             ,sum(sales.amount_sold) amount_sold
07             ,case
08                 when prd.prod_list_price>100 then
sum(sales.quantity_sold)*prd.prod_list_price
09             else
10                 0
11             end quantity_sold
12         from
13             sh.products prd
14             ,sh.sales sales
15         where
16             prd.prod_status = 'available, on stock'
17             and prod_category in ('Men','Women')
18             and sales.time_id(+) >= to_date(p_from_time,'YYYYMMDD')
19             and sales.time_id(+) < to_date(p_to_time,'YYYYMMDD')
20             and prd.prod_id = sales.prod_id(+)
21         group by
22             prd.prod_id
23             ,prd.prod_name
24             ,prd.prod_list_price;
25     rec_products cur_products%rowtype;
26 begin
27     open cur_products('20000101','20010101');
28     loop
29         fetch cur_products into rec_products;
30         exit when cur_products%notfound;
31 --         dbms_output.put_line('AMOUNT SOLD : ' || rec_products.amount_sold || ', QUANTITY
SOLD : ' || rec_products.quantity_sold);
32     end loop;
33     close cur_products;
34 end;
```

반복적으로 사용되던 부분이 cur_products cursor 부분에 흡수되었다. Group by 절이 추가되고 prod_id, prod_name,prod_list_price 가 추가되었다. SQL 문 통합 시 반복 수행되는 prod_id 만을 group by 에 지정하면 prod_name, prod_list_price 는 구할 방법이 없다. prod_name, prod_list_price 는 prod_id 에 종속적이므로 group by 절에 prod_id 뿐만 아니라 prod_name, prod_list_price 까지 추가해 원하는 자료를 가져올 수 있도록 하였다.

SQL 통합 시 고려해야 할 또 다른 중요한 점은 18~20 line 에 나타나 있다. 바로 Outer Join 의 사용이 그것이다. 원 source 를 보면 cur_products cursor 의 자료를 통해 반복 수행되는 cur_sql 부분이 호출됨을 알 수 있으나 cur_sql 에서 아무런 자료도 출력되지 않을 경우 그것을 사용하지 않겠다는 부분은 나타나지 않고 있다. 만일 'v_amount_sold 값이 0 보다 크다'라거나 또 다른 명시적 조건이 존재하지 않는 경우는 Child 관계에 해당하는 cur_sql cursor 의 결과 값이 없더라도 Parent 에 해당하는 cur_products cursor 부분의 자료는 유효하다는 것을 인지해야 한다.

다음의 표는 지난 호에서 사용된 주요 자원 사용량을 각각의 경우로 비교하고 있다.

| 구 분 | | Case 1 | Case 2 | Case 3 |
|-------------|---------------------------|---------|---------|--------|
| 소요시간 | | 38.00 | 18.05 | 4.01 |
| Stat | execute count | 5,078 | 5,125 | 303 |
| | recursive calls | 29,075 | 29,793 | 8,484 |
| | parse count (total) | 4,928 | 4,938 | 148 |
| | parse count (hard) | 4,722 | 29 | 25 |
| | parse time cpu | 709 | 46 | 24 |
| | parse time elapsed | 992 | 172 | 122 |
| | opened cursors cumulative | 4,887 | 4,897 | 104 |
| | CPU used by this session | 15.04 | 758 | 201 |
| Latch (get) | library cache | 273,091 | 152,812 | 5,093 |
| | shared pool | 301,141 | 13,094 | 3,098 |
| | row cache objects | 696,675 | 3,105 | 2,591 |

본 Table 에 나타난 수치는 시스템 환경에 따라 절대값 자체는 어느 정도 변경될 수 있으나 값의 비율은 어느 정도 유지되므로 상대적인 수치로 이해하는 것이 더 바람직하다. Case1 에서 Case2 로 가면서 Literal SQL 문의 감소로 인해 Hard Parsing 횟수가 감소하고 그로 인해 Parse Time CPU 가 떨어졌으며 그 시간이 소요시간에 반영되었음을 알 수 있으며 Latch 자원 사용량도 상당히 많이 감소했음을 보여주고 있다. Case2 에서 Case3 로의 변화로 인해 Soft Parse 횟수가 많이 감소되었고 Parsing 에 소요된 시간이 약간 감소했음을 알 수 있는데 Soft Parse 는 이미 Shared Pool 에 있는 정보를 활용하기 때문에 부담이 작다는 것을 알 수 있다. 하지만 SQL 문 통합에 따라 수행횟수가 상당히 감소되었고, 부담이 적기는 하지만 Soft Parse 의 감소 그리고 Latch 자원의 사용량 감소는 수행시간을 더 단축시켜주고 있다.

프로그램 설계 구조로 인해 Case3 을 적용할 수도 있고 그렇지 못할 수도 있으나 가능하면 Case3 과 같은 식으로 프로그램을 작성하는 것이 훨씬 더 효율적인 것이라고 알 수 있다.

6.2 자료 구조 차이에서 발생하는 Literal SQL

본 프로그램은 oe 스키마의 order_items table 의 내용을 신규로 생성한 order_summary 라는 table 로 자료를 이동시키는 사례이다. 각각의 Order 별로 주문번호,주문일,15 개까지의 제품번호, 15 개까지의 주문액을 1 개 Record 에 표시하고자 한다.

| ORDER_ID | LINE_ITEM_ID | PRODUCT_ID | UNIT_PRICE | QUANTITY |
|----------|--------------|------------|------------|----------|
| 2410 | 1 | 2976 | 46 | 10 |
| 2410 | 2 | 2982 | 40 | 5 |
| 2410 | 3 | 2986 | 120 | 6 |
| 2410 | 4 | 2995 | 68 | 8 |
| 2410 | 5 | 3003 | 2866.6 | 15 |
| 2410 | 6 | 3051 | 12 | 21 |



| ORDER_ID | ORDER_DATE | PRD_ID01 | PRD_ID02 | ... | PRD_AMT01 | PRD_AMT02 | ... |
|----------|------------|----------|----------|-----|-----------|-----------|-----|
| 2410 | 2000.05.24 | 2976 | 2982 | | 460 | 200 | |

Table, Column 과 같은 Database Object 이름은 bind 처리가 되지 않으므로(parse 단계가 bind 단계보다 먼저 수행되고, 그렇기 때문에 parse 단계에서 bind 변수의 내용을 알 수 없기 때문에 사용하는 Object 가 무엇인지 알 수 없기 때문이다.) 개발자는 다음과 같이 프로그램을 작성하였다.

```

01 declare
02     v_sql varchar2(200);
03     cursor cur_order_item is
04         select order_id,trim(to_char(line_item_id,'09'))
line_item_id,product_id,unit_price*quantity amt
05         from oe.order_items;
06 begin
07     insert into oe.order_summary(order_id,order_date) select
order_id,to_char(order_date,'YYYY.MM.DD') from oe.orders;
08     for rec_order_item in cur_order_item loop
09         if rec_order_item.line_item_id is NOT NULL then
10             v_sql:='update oe.order_summary set prd_id' ||
rec_order_item.line_item_id || '=' || rec_order_item.product_id ||
11                 ', prd_amt' || rec_order_item.line_item_id || '=' || rec_order_item.amt
||
12                 'where order_id=' || rec_order_item.order_id;
13             execute immediate v_sql;
14         end if;
15     end loop;
16 end;
```

order_summary table 에 order_id 와 order_date 를 입력한 후 order_items table 의 내용을 loop 를 수행하며 literal SQL 문을 만들어 수행하였다. prd_id, prd_amt 다음의 line_item_id 가 컬럼 이름이 hard coding 되어 있으므로 이 부분에 대한 bind 처리가 되지 않으므로 line 10,11 에서 line_item_id 를 추가시켜 literal SQL 로 사용하고 있음을 알 수 있다. SQL 문은 cur_order_item 의 건수만큼 수행되므로 order_items table 의 건수만큼 수행된다는 것을 알 수 있으며 이들 모두 서로 다른 값을 가지고 수행되므로 order_items table 건수만큼 Hard Parsing 이 발생할 것이라는 것을 짐작할 수 있다.

각각 15 개인 prd_id, prd_amt 컬럼에 대해서는 literal SQL 문의 사용이 불가피 하지만 입력되는 값 자체는 bind 처리가 가능하므로 프로그램을 다음과 같이 작성할 수 있다.

```

01 declare
02     v_sql varchar2(200);
```

```

03     cursor cur_order_item is
04         select order_id,trim(to_char(line_item_id,'09'))
line_item_id,product_id,unit_price*quantity amt
05         from oe.order_items;
06 begin
07     insert into oe.order_summary(order_id,order_date) select
order_id,to_char(order_date,'YYYY.MM.DD') from oe.orders;
08     for rec_order_item in cur_order_item loop
09         if rec_order_item.line_item_id is NOT NULL then
10             v_sql:='update oe.order_summary set prd_id' ||
rec_order_item.line_item_id || '=:b0' ||
11             ', prd_amt' || rec_order_item.line_item_id || '=:b1 where
order_id=:b2';
12             execute immediate v_sql using
rec_order_item.product_id,rec_order_item.amt,rec_order_item.order_id;
13         end if;
14     end loop;
15 end;

```

line 10 과 line11 에서 literal SQL 문을 생성하는 과정은 전과 동일하다. 하지만 실제로 입력되는 prd_id,prd_amt 는 bind 변수를 사용했음을 알 수 있다. 이렇게 할 경우 어느 정도 Literal SQL 문은 감소하지만 prd_id,prd_amt 의 개수인 15 번의 Hard Parsing 은 피할 길이 없다.

이 경우도 다음과 같이 SQL 문을 통합해 1 개의 SQL 문으로 수행하도록 할 수 있다.

```

1 insert into oe.order_summary
2 select ord.order_id,to_char(order_date,'YYYY.MM.DD')
3 ,max(decode(line_item_id,1,product_id)),...
4 ,max(decode(line_item_id,1,unit_price*quantity)),...
5 from oe.order_items ordi, oe.orders ord
6 where ord.order_id=ordi.order_id
7 group by ord.order_id,to_char(order_date,'YYYY.MM.DD');

```

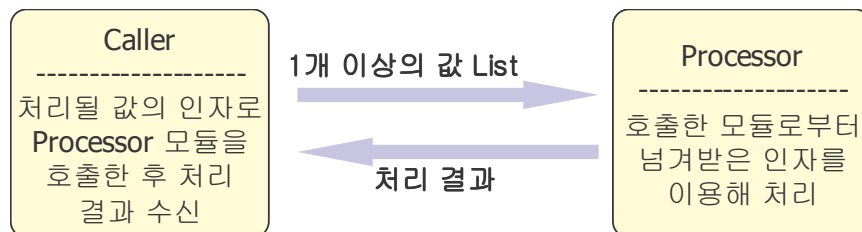
PL/SQL Logic 이 사라지고 insert 문장으로 대체되었다 - line3 과 line4 에서는 line_item_id 숫자만 증가하며 동일한 구조로 사용된다. 이렇게 할 경우 SQL 문은 단 1 회만 수행되며 Parsing 작업도 단 1 회면 충분하다 - 본 Test 결과에서는 shared pool truncate 에 따른 내부작업으로 인해 execute count, parse count 가 더 큰 값으로 표시되므로 유의하기 바란다.

| 구 분 | | Case 1 | Case 2 | Case 3 |
|--------------------------|---------------------------|--------|--------|--------|
| 소요시간 | | 1.02 | 1.01 | 0.03 |
| Stat | execute count | 795 | 785 | 55 |
| | recursive calls | 3,539 | 3,179 | 780 |
| | parse count (total) | 723 | 735 | 14 |
| | parse count (hard) | 672 | 43 | 6 |
| | parse time cpu | 36 | 7 | 6 |
| | parse time elapsed | 86 | 73 | 22 |
| | opened cursors cumulative | 714 | 712 | 14 |
| CPU used by this session | | 60 | 35 | 7 |
| Latch (get) | library cache | 19,174 | 23,225 | 1,090 |
| | shared pool | 21,964 | 3,854 | 890 |
| | row cache objects | 6,476 | 1,012 | 655 |

Case2 는 주어진 구조에서 Bind SQL 을 사용한 형태이다. 앞서 언급했듯이 값에 대한 부분은 Bind SQL 을 통해 공유가 가능하지만 컬럼 이름에 대한 부분은 여전히 컬럼 이름의 개수 만큼 Literal SQL 문으로 작성되어야 하므로 어느 정도의 Hard Parsing 은 피하지 못하고 있다. 하지만 구조는 그대로 사용하면서 일부만 Bind SQL 로 사용한 것은 많은 향상을 일으키지는 못하고 있다. 물론 더 많은 자료에 대해 수행할 경우 Case1 과 Case2 의 격차는 좀 더 커질 수 있지만 큰 효과는 없어 보인다. Case3 는 PL/SQL 로 작성된 프로그램을 SQL 문으로 변경하면서 구조를 변경한 사례이다. 실행횟수, 구조를 변경한 Case3 은 parse 횟수, latch 자원 사용량 등 Case1 은 물론 Case2 와 비교해서도 상당한 향상이 있었음을 알 수 있다.

6.3 정해지지 않은 개수의 변수를 사용해 발생하는 Literal SQL

앞의 두 가지 사례는 모두 SQL 문을 통합해 상당한 성능 향상을 이룩한 사례를 제시하였다. 하지만 실무에서는 프로그램의 구조상 SQL 문의 통합이 불가능한 경우가 많이 존재한다. 예를 들어 전사 개발 표준이 각 단위별로 작성된 작은 모듈을 호출해 사용하도록 한 경우이거나 패키지 프로그램을 이용해 또 다른 업무 구축을 하고자 할 경우 등 이런 제약사항이 발생하는 경우는 주변에서 많이 찾아볼 수 있다. 본 예제는 기능별로 프로그램이 모듈화 한 경우이며 각 모듈간 기능의 통합을 불가능한 경우를 다룬다. 본 프로그램은 다음과 같은 2 개의 모듈로 구성되어 있으며 Caller 모듈은 처리 대상을 추출해 Processor 모듈로 넘겨주고, Processor 모듈은 Caller 모듈에서 넘겨준 처리 대상으로 값을 처리한 후 Caller 모듈로 결과를 보낸다. Caller 모듈은 넘겨받은 결과값으로 후속 처리 - 여기서는 화면 출력 - 을 하는 구조를 갖는다.



Processor 모듈을 개발한 개발자는 Processor 모듈을 호출하는 모듈이 얼마나 많은 값 List 를 넘겨줄지 알 수 없으므로 입력되는 값 List 를 문자열로 구성한 후 IN 을 이용해 처리하는 다음과 같은 방식을 사용하였다. (편의상 1 개의 Package 로 Caller 와 Processor 모듈을 표현하였다.) 본 예제는 Caller 모듈에서 외부에서 입력받은 기간 중 회계적으로 1/4, 2/4 기에 있었던 프로모션 행사 중 Internet 을 통해 수행됐으며 직접 판매점을 대상으로 한 프로모션 행사의 정보를 구하고 Processor 모듈에서는 Caller 모듈에서 구해진 프로모션 행사에 대해 일자별로 프로모션 비용을 구해 Caller 모듈로 넘겨지는 구조를 갖는다.

```

-- Package Spec
1 create or replace package sh.lit_case3 AUTHID CURRENT_USER as
2     TYPE typ_promo IS TABLE OF promotions.promo_id%TYPE index by binary_integer;
3
4     procedure proc_caller(p_begin_date in varchar2, p_end_date in varchar2);
5     procedure proc_processor(p_promo_list in typ_promo, p_time in varchar2, p_day_name out
varchar2, p_sum_promo out number);
6 end lit_case3;
  
```

```

-- Package Body
01 create or replace package body sh.lit_case3 as
02     procedure proc_caller(p_begin_date in varchar2, p_end_date in varchar2) is
03         cursor cur_times(i_begin_date varchar2,i_end_date varchar2) is
04             select to_char(time_id,'YYYYMMDD') time_id
05             from sh.times
06             where fiscal_quarter_number in (1,2)
07             and time_id between to_date(i_begin_date,'YYYYMMDD') and
to_date(i_end_date,'YYYYMMDD');
08         cursor cur_sales(i_date varchar2) is
09             select distinct prom.promo_id
10             from
11                 sh.channels      chan
12                 ,sh.sales        sales
13                 ,sh.promotions  prom
14             where
15                 chan.channel_id=sales.channel_id
16                 and sales.promo_id=prom.promo_id
17                 and time_id=to_date(i_date,'YYYYMMDD')
18                 and chan.channel_class = 'Direct'
19                 and prom.promo_category = 'internet';
20         rec_time cur_times%rowtype;
21         tab_promo_id typ_promo;
22         v_sum_promo  number;
23         v_day_name   varchar2(20);
24     begin
25         open cur_times(p_begin_date,p_end_date);
26         loop
27             fetch cur_times into rec_time;
28             exit when cur_times%NOTFOUND;
29             open cur_sales(rec_time.time_id);
30             v_sum_promo:=0;
31             fetch cur_sales bulk collect into tab_promo_id;
32             if cur_sales%ROWCOUNT>0 then
33
proc_processor(tab_promo_id,rec_time.time_id,v_day_name,v_sum_promo);
34 --             dbms_output.put_line(rec_time.time_id || '[' || v_day_name ||
'] : ' || v_sum_promo);
35                 end if;
36                 close cur_sales;
37             end loop;
38             close cur_times;
39         end;
40
41     procedure proc_processor(p_promo_list in typ_promo, p_time in varchar2, p_day_name out
varchar2, p_sum_promo out number) is
42         v_param_list varchar2(1000);
43         v_sql  varchar2(1000);
44         i pls_integer:=1;
45     begin
46         v_param_list:='';
47         while p_promo_list.exists(i) loop
48             if v_param_list is null then
49                 v_param_list:='(' || to_char(p_promo_list(i));
50             else
51                 v_param_list:=v_param_list || ',' || to_char(p_promo_list(i));
52             end if;
53             i:=i+1;

```



```

54         end loop;
55         v_param_list:= v_param_list || ' ';
56         v_sql:='
57             declare
58                 p_time  varchar2(100);
59                 p_sum_promo      number;
60             begin
61                 :p_sum_promo:=0;
62                 select max(tim.day_name),nvl(sum(promo_cost),0)
into :p_day_name,:p_sum_promo
63                 from sh.promotions prom, sh.times tim
64                 where promo_id in ' || v_param_list || ' and
to_date('' ||p_time||'', 'YYYYMMDD') between promo_begin_date and promo_end_date and
tim.time_id=to_date('' ||p_time||'', 'YYYYMMDD');
65             exception
66                 when others then
67                     :p_sum_promo := -1;
68             end;';
69         execute immediate v_sql using in out p_sum_promo,in out p_day_name;
70     end;
71 end lit_case3;

```

proc_processor 모듈은 p_promo_list 를 통해 입력되는 값의 개수를 알지 못하므로 line 46~line55 까지 입력되는 값으로 문자열을 구성하고 있다. 그리고 이것을 사용해 line64 에서와 같이 'IN' 연산자를 이용해 정해지지 않은 개수의 인수 값을 처리하고 있으며 실제 처리되는 부분에서는 'declare ... begin ... exception ... end'의 구조를 취하고 있다. 입력받은 값 List 를 문자열로 구성하였으므로 Literal SQL 로 수행되며 Processor 모듈은 수행하는 SQL 문이 모두 Hard Parsing 을 유발하므로 이 부분이 많이 호출될 경우 성능 상의 문제가 발생할 수 있다.

위의 구조를 그대로 사용할 경우 특별히 문제를 개선할 수 있는 방법은 없어 보인다. Oracle Version 8.1.6 부터 Literal SQL 문이 사용된 경우라도 내부적으로 Bind SQL 문으로 변환해 수행할 수 있는 방식을 제공하며 cursor_sharing 이라는 Parameter 값을 조정해 수행할 수 있다. 1 차적으로 생각할 수 있는 방법은 Processor 모듈에서 cursor_sharing Parameter 값을 조정해 프로그램의 구조변경 없이 Bind SQL 을 사용하는 방식을 생각해 볼 수 있다. 하지만 Processor 모듈의 어느 부분에 execute immediate 'alter session set cursor_sharing=force'를 사용하더라도 Bind SQL 로 변경되지 않았다는 것을 알 수 있을 것이다. Test 결과 PL/SQL 구문에서 cursor_sharing 은 PL/SQL 내부에 'declare ~ begin ~ end' 구조내부에 존재하는 Literal SQL 문에 대해서는 적용되지 않으며 SQL 내부에 bind 변수가 일부 사용될 경우도 적용되지 않는 것으로 나타났다. 따라서 프로그램 구조를 일부 변경해 다음과 같이 재 작성해 볼 수 있다.

```

-- Package Spec
위와 동일

-- Package Body
01 create or replace package body sh.lit_case3 as
02     procedure proc_caller(p_begin_date in varchar2, p_end_date in varchar2) is
03         cursor cur_times(i_begin_date varchar2,i_end_date varchar2) is
04             select to_char(time_id,'YYYYMMDD') time_id
05             from sh.times
06             where fiscal_quarter_number in (1,2)
07             and time_id between to_date(i_begin_date,'YYYYMMDD') and
to_date(i_end_date,'YYYYMMDD');

```

```

08         cursor cur_sales(i_date varchar2) is
09             select distinct prom.promo_id
10             from
11                 sh.channels      chan
12                 ,sh.sales        sales
13                 ,sh.promotions  prom
14             where
15                 chan.channel_id=sales.channel_id
16                 and sales.promo_id=prom.promo_id
17                 and time_id=to_date(i_date,'YYYYMMDD')
18                 and chan.channel_class = 'Direct'
19                 and prom.promo_category = 'internet';
20         rec_time cur_times%rowtype;
21         tab_promo_id typ_promo;
22         v_sum_promo      number;
23         v_day_name      varchar2(20);
24     begin
25         open cur_times(p_begin_date,p_end_date);
26         loop
27             fetch cur_times into rec_time;
28             exit when cur_times%NOTFOUND;
29             open cur_sales(rec_time.time_id);
30             v_sum_promo:=0;
31             fetch cur_sales bulk collect into tab_promo_id;
32             if cur_sales%ROWCOUNT>0 then
33
34                 proc_processor(tab_promo_id,rec_time.time_id,v_day_name,v_sum_promo);
35                 dbms_output.put_line(rec_time.time_id || '[' || v_day_name ||
36                 ']' : ' || v_sum_promo);
37             end if;
38             close cur_sales;
39         end loop;
40     end;
41     procedure proc_processor(p_promo_list in typ_promo, p_time in varchar2, p_day_name out
42     varchar2, p_sum_promo out number) is
43         v_param_list varchar2(1000);
44         v_sql      varchar2(1000);
45         i pls_integer:=1;
46     begin
47         v_param_list:='';
48         while p_promo_list.exists(i) loop
49             if v_param_list is null then
50                 v_param_list:='(' || to_char(p_promo_list(i));
51             else
52                 v_param_list:=v_param_list || ',' || to_char(p_promo_list(i));
53             end if;
54             i:=i+1;
55         end loop;
56         v_param_list:= v_param_list || ')';
57         execute immediate 'alter session set cursor_sharing=force';
58         begin
59             v_sql:='select max(tim.day_name),nvl(sum(promo_cost),0)
60                 from sh.promotions prom, sh.times tim
61                 where promo_id in ' || v_param_list || ' and
62                 to_date('' || p_time || ''',''YYYYMMDD'') between promo_begin_date and promo_end_date and
63                 tim.time_id=to_date('' || p_time || ''',''YYYYMMDD'')';
64             execute immediate v_sql into p_day_name,p_sum_promo;

```

```

62         exception
63             when others then
64                 p_sum_promo := -1;
65         end;
66         execute immediate 'alter session set cursor_sharing=exact';
67     end;
68 end lit_case3;

```

위와 같이 프로그램을 작성하면 다음 <그림 2>와 같이 사용된 Literal SQL 문이 모두 Bind SQL 문으로 변경되어 수행됨을 알 수 있다.

| SQL Text | Module | Version Count | Optimizer Mode | Execs |
|--|----------|---------------|----------------|-------|
| select max(tim.day_name),sum(promo_cost) from sh.promotions prom, sh.times tim where promo_id in (:SYS_B_00, :SYS_B_01, :SYS_B_02, :SYS_B_03) | SQL*Plus | 1 | CHOOSE | 4 |
| SELECT max(tim.day_name),sum(promo_cost) from sh.promotions prom, sh.times tim where promo_id in (331) and to_date('19980602','YYYYMMDD') between | SQL*Plus | 1 | CHOOSE | 2 |
| select max(tim.day_name),sum(promo_cost) from sh.promotions prom, sh.times tim where promo_id in (:SYS_B_0, :SYS_B_1, :SYS_B_2, :SYS_B_3) | SQL*Plus | 1 | CHOOSE | 2 |
| SELECT max(tim.day_name),sum(promo_cost) from sh.promotions prom, sh.times tim where promo_id in (83,119,215,407,431) and to_date('19990120','YYYYMMDD') | SQL*Plus | 1 | CHOOSE | 1 |
| SELECT max(tim.day_name),sum(promo_cost) from sh.promotions prom, sh.times tim where promo_id in (31,283,307,319,331,415) and to_date('19980623','YYYYMMDD') | SQL*Plus | 1 | CHOOSE | 1 |

그림 2. Shared Pool을 통해 본 Literal SQL과 Cursor_sharing으로 Bind 처리된 SQL

이외에는 대안이 없을까? cursor_sharing 을 사용하지 않고 다음과 같이 Bind SQL 문을 작성할 수도 있다.

```

-- Package Spec
위와 동일

-- Package Body
01 create or replace package body sh.lit_case3 as
02     procedure proc_caller(p_begin_date in varchar2, p_end_date in varchar2) is
03         cursor cur_times(i_begin_date varchar2, i_end_date varchar2) is
04             select to_char(time_id, 'YYYYMMDD') time_id
05             from sh.times
06             where fiscal_quarter_number in (1,2)
07             and time_id between to_date(i_begin_date, 'YYYYMMDD') and
to_date(i_end_date, 'YYYYMMDD');
08         cursor cur_sales(i_date varchar2) is
09             select distinct prom.promo_id
10             from
11                 sh.channels      chan
12                 ,sh.sales        sales
13                 ,sh.promotions  prom
14             where
15                 chan.channel_id=sales.channel_id
16                 and sales.promo_id=prom.promo_id
17                 and time_id=to_date(i_date, 'YYYYMMDD')
18                 and chan.channel_class = 'Direct'

```

```

19             and prom.promo_category = 'internet';
20         rec_time cur_times%rowtype;
21         tab_promo_id typ_promo;
22         v_sum_promo      number;
23         v_day_name      varchar2(20);
24     begin
25         open cur_times(p_begin_date,p_end_date);
26         loop
27             fetch cur_times into rec_time;
28             exit when cur_times%NOTFOUND;
29             open cur_sales(rec_time.time_id);
30             v_sum_promo:=0;
31             fetch cur_sales bulk collect into tab_promo_id;
32             if cur_sales%ROWCOUNT>0 then
33
34                 proc_processor(tab_promo_id,rec_time.time_id,v_day_name,v_sum_promo);
35             --             dbms_output.put_line(rec_time.time_id || '[' || v_day_name ||
36             ']' : ' || v_sum_promo);
37             end if;
38             close cur_sales;
39         end loop;
40         close cur_times;
41     end;
42
43     procedure proc_processor(p_promo_list in typ_promo, p_time in varchar2, p_day_name out
44     varchar2, p_sum_promo out number) is
45         v_sql      varchar2(1000);
46         v_day_name  varchar2(100);
47         v_sum_promo  number;
48
49         i pls_integer:=1;
50         j pls_integer;
51         cur_sql_exec      number;
52         ignore           number;
53     begin
54         cur_sql_exec:=dbms_sql.open_cursor;
55         v_sql:='select max(tim.day_name),nvl(sum(promo_cost),0)
56             from sh.promotions prom, sh.times tim
57             where to_date(:p_time,'YYYYMMDD') between promo_begin_date
58 and promo_end_date and tim.time_id=to_date(:p_time,'YYYYMMDD') and promo_id in (';
59         while p_promo_list.exists(i) loop
60             v_sql:=v_sql || ':b' || i || ',';
61             i:=i+1;
62         end loop;
63         v_sql:=substr(v_sql,1,lengthb(v_sql)-1) || ')';
64         dbms_sql.parse(cur_sql_exec,v_sql,1);
65     begin
66         dbms_sql.define_column(cur_sql_exec,1,p_day_name,100);
67         dbms_sql.define_column(cur_sql_exec,2,p_sum_promo);
68         dbms_sql.bind_variable(cur_sql_exec,'p_time',p_time);
69         i:=1;
70         while p_promo_list.exists(i) loop
71             dbms_sql.bind_variable(cur_sql_exec,'b' || i,p_promo_list(i));
72             i:=i+1;
73         end loop;
74         ignore:=sys.dbms_sql.execute_and_fetch(cur_sql_exec);
75         dbms_sql.column_value(cur_sql_exec,1,p_day_name);
76         dbms_sql.column_value(cur_sql_exec,2,p_sum_promo);
77         if ignore=0 then

```

```

74             p_sum_promo:=0;
75             end if;
76         exception
77             when others then
78                 p_sum_promo := -1;
79             end;
80             dbms_sql.close_cursor(cur_sql_exec);
81     exception
82         when others then
83             if dbms_sql.is_open(cur_sql_exec) then
84                 dbms_sql.close_cursor(cur_sql_exec);
85             end if;
86     end;
87 end lit_case3;

```

Processor 로 입력되는 p_promo_list 변수를 이용해 동적으로 Bind SQL 문을 구성하는 것이다. 이런 방식을 사용하면 cursor_sharing 사용에 따르는 내부적으로 수행되는 Literal Check 부분을 수행하지 않으므로 유리하지만 dbms_sql Package 를 과다하게 호출하는 부담도 발생하므로 어느 것이 유리하다고 잘라 말하기는 어려울 듯 하다.

두번째와 세번째 방법을 이용하면 분명히 Bind SQL 문이 사용되지만 IN List 에 들어오는 변수의 개수가 서로 틀리기 때문에 그 효율이 떨어진다고 할 수 있다. 위에 첨부된 <그림 2>에서도 알 수 있듯이 2,4,6 번 수행된 SQL 문도 존재하므로 효율이 좋다고 할 수는 없다. 이 문제를 해결하기 위해 가장 IN List 에 들어가는 변수 중 가장 많이 들어가는 변수를 구해서 모든 SQL 문의 IN List 를 그 개수로 구성하는 방법을 떠올릴 수 있다.

```

-- Global 변수를 저장하는 Package
1 create or replace package sh.global_variable AUTHID CURRENT_USER as
2     g_maxcnt number:=15;
3 end global_variable;

-- Package Spec
위와 동일

-- Package Body
01 create or replace package body sh.lit_case3 as
02     procedure proc_caller(p_begin_date in varchar2, p_end_date in varchar2) is
03         cursor cur_times(i_begin_date varchar2,i_end_date varchar2) is
04             select to_char(time_id,'YYYYMMDD') time_id
05                 from sh.times
06                 where fiscal_quarter_number in (1,2)
07                    and time_id between to_date(i_begin_date,'YYYYMMDD') and
to_date(i_end_date,'YYYYMMDD');
08         cursor cur_sales(i_date varchar2) is
09             select distinct prom.promo_id
10                 from
11                 sh.channels      chan
12                 ,sh.sales        sales
13                 ,sh.promotions  prom
14             where
15                 chan.channel_id=sales.channel_id
16                 and sales.promo_id=prom.promo_id
17                 and time_id=to_date(i_date,'YYYYMMDD')
18                 and chan.channel_class = 'Direct'
19                 and prom.promo_category = 'internet';

```

```

20         rec_time cur_times%rowtype;
21         tab_promo_id typ_promo;
22         v_sum_promo      number;
23         v_day_name      varchar2(20);
24     begin
25         open cur_times(p_begin_date,p_end_date);
26         loop
27             fetch cur_times into rec_time;
28             exit when cur_times%NOTFOUND;
29             open cur_sales(rec_time.time_id);
30             v_sum_promo:=0;
31             fetch cur_sales bulk collect into tab_promo_id;
32             if cur_sales%ROWCOUNT>0 then
33
34                 proc_processor(tab_promo_id,rec_time.time_id,v_day_name,v_sum_promo);
35                 -- dbms_output.put_line(rec_time.time_id || '[' || v_day_name ||
36                 ']' : ' || v_sum_promo);
37                 end if;
38                 close cur_sales;
39             end loop;
40             close cur_times;
41         end;
42     procedure proc_processor(p_promo_list in typ_promo, p_time in varchar2, p_day_name out
43     varchar2, p_sum_promo out number) is
44         v_sql      varchar2(1000);
45         v_day_name  varchar2(100);
46         v_sum_promo  number;
47
48         i pls_integer:=1;
49         j pls_integer;
50         cur_sql_exec      number;
51         ignore            number;
52     begin
53         cur_sql_exec:=dbms_sql.open_cursor;
54         v_sql:='select max(tim.day_name),nvl(sum(promo_cost),0)
55             from sh.promotions prom, sh.times tim
56             where to_date(:p_time,'YYYYMMDD') between promo_begin_date
57             and promo_end_date and tim.time_id=to_date(:p_time,'YYYYMMDD') and promo_id in (';
58             for i in 1..sh.global_variable.g_maxcnt loop
59                 if i=sh.global_variable.g_maxcnt then
60                     v_sql:=v_sql || ':b' || i || ')';
61                 else
62                     v_sql:=v_sql || ':b' || i || ',';
63                 end if;
64             end loop;
65             dbms_sql.parse(cur_sql_exec,v_sql,1);
66             begin
67                 dbms_sql.define_column(cur_sql_exec,1,p_day_name,100);
68                 dbms_sql.define_column(cur_sql_exec,2,p_sum_promo);
69                 dbms_sql.bind_variable(cur_sql_exec,'p_time',p_time);
70                 while p_promo_list.exists(i) loop
71                     dbms_sql.bind_variable(cur_sql_exec,'b' || i,p_promo_list(i));
72                     i:=i+1;
73                 end loop;
74                 j:=i;
75                 while j<=sh.global_variable.g_maxcnt loop
76                     dbms_sql.bind_variable(cur_sql_exec,'b' || j,-1);
77                     j:=j+1;

```

```

75         end loop;
76         ignore:=sys.dbms_sql.execute(cur_sql_exec);
77         ignore:=sys.dbms_sql.fetch_rows(cur_sql_exec);
78         dbms_sql.column_value(cur_sql_exec,1,p_day_name);
79         dbms_sql.column_value(cur_sql_exec,2,p_sum_promo);
80         if ignore=0 then
81             p_sum_promo:=0;
82         end if;
83     exception
84         when others then
85             p_sum_promo := -1;
86     end;
87     dbms_sql.close_cursor(cur_sql_exec);
88 exception
89     when others then
90         if dbms_sql.is_open(cur_sql_exec) then
91             dbms_sql.close_cursor(cur_sql_exec);
92         end if;
93     end;
94 end lit_case3;

```

이 방법은 SQL 문의 공유를 극대화 했다는 점에서는 장점을 갖지만 IN List 를 통해 입력되는 변수의 최대 개수를 구하는 부분이 또 다른 성능의 병목이 될 수 있다는 점을 무시할 수 없다. 또 최대 개수 만큼 Dummy 로 Bind 값을 구성해 주어야 하므로 많이 사용되는 경우 이 부분 또한 무시할 수 없는 부분이 될 수 있다. 여기서는 최대 개수가 미리 정해진 것으로 가정하고 사용했으나 실무에서는 미리 지정된 값을 사용하기가 어려울 수 있다. 또 최대 개수를 절대 발생하지 않을 만큼 넉넉하게 지정하면 앞서 언급한 대로 Processor 모듈에서 SQL 문 구성에 따른 부담으로 나타나 오히려 시간이 더 많이 소요될 수 있으므로 이 또한 바람직하지 않다.

| 구 분 | | Case 1 | Case 2 | Case 3 | Case 4 |
|-------------|---------------------------|--------|--------|--------|--------|
| 소요시간 | | 4.04 | 3.07 | 3.02 | 3.05 |
| Stat | execute count | 1,227 | 1,427 | 885 | 893 |
| | recursive calls | 8,564 | 8,648 | 7,808 | 9,055 |
| | parse count (total) | 641 | 829 | 339 | 347 |
| | parse count (hard) | 412 | 45 | 57 | 28 |
| | parse time cpu | 96 | 28 | 21 | 27 |
| | parse time elapsed | 213 | 140 | 110 | 103 |
| | opened cursors cumulative | 596 | 790 | 300 | 304 |
| | CPU used by this session | 297 | 242 | 223 | 236 |
| Latch (get) | library cache | 28,724 | 16,516 | 33,371 | 47,519 |
| | shared pool | 28,972 | 5,189 | 6,900 | 6,631 |
| | row cache objects | 20,232 | 4,076 | 4,436 | 2,870 |

수행시간은 Case1 을 제외하고는 모두 큰 차이 없는 것으로 보인다. 다만 Case4 는 IN List 에 사용되는 최대 변수의 개수가 이미 정해져 있다는 가정 하에 작성되었으며 동적으로 이 값을 구하는 경우는 많은 부분을 검색해 결과를 가져와야 하므로 최소 5 초 정도의 Overhead 가 더 발생한다는 문제가 있기 때문에 이와 같은 환경에서는 Case4 는 고려대상이 될 수 없다. 여러 가지 지표를 통해 볼 때 Case2 보다는 Case3 이 더 바람직해 보이지만 둘 사이에 큰 차이가 있지는 않으므로 개발자가 사용하기 편한 방법을 사용해도 좋을 듯 하다. 하지만 IN List 를

통해 넘어오는 값의 개수가 많지 않으며 대부분 동일한 개수의 값이 사용된다면 Case3 이 좀 더 유리할 것으로 보인다.

7. 결 론

Parsing이라는 작업이 무엇인지 알아보았고 **Hard Parsing**이라는 것이 얼마나 성능에 좋지 못한 영향을 미치는지 알아보았다. 또 몇 가지 형태별로 동일한 작업을 수행했을 때 오라클 내부적으로 어떤 자원들이 얼마나 사용되는 지 알아보았다.

SQL 문 작성 시 **Hard Parsing**을 피하면서 보다 더 좋은 성능으로 작동될 수 있는 몇 가지 방법을 알아보았다. 본 문서를 처음 작성할 때는 **Literal SQL**의 제거를 통한 성능 향상 기법까지만 알아보려 했으나 **SQL** 사용 시 그것보다 더 중요하다고 생각되는 것이 바로 **SQL**적인 표현 기법이기에 때문에 **SQL** 통합을 통한 성능개선까지 알아보았다.

본 문서에 언급되어 있는 방법 외에도 여러 가지 방법으로 **Test**를 수행해 보면 더 다양한 경험을 할 수 있을 것으로 생각된다. 이 문서의 내용이 많은 분들에게 도움이 됐으면 하는 바람이다.

8. Reference

1. Oracle9i Database Performance Tuning Guide and Reference Release 2 (9.2)
2. Oracle9i Database Concepts Release 2 (9.2)
3. Oracle9i Supplied PL/SQL Packages and Types Reference 2 (9.2)
4. Oracle9i Reference Release 2 (9.2)
5. Metalink NOTE 32895.1 : SQL Parsing Flow Diagram
6. Metalink NOTE 62143.1 : Understanding and Tuning the Shared Pool
7. <http://technet.oracle.com> : Efficient use of bind variables, cursor_sharing and related cursor parameters(Bjørn Engsig)
8. <http://www.hotsos.com> : A LOOK UNDER THE HOOD OF CBO: THE 10053 EVENT(Wolfgang Breitling)