개발자 교육자료 II

for 농협중앙회

2004-12-02

최세훈(SeHoon.Choi@oracle.com)
DB 기술팀
한 국 오 라 클(주)

목 차

Optimizer 관련 권장사항 요약

다음은 일부 Optimizer와 관련된 Parameter의 주요 변경사항을 요약하였다. 가장 중요한 것은 'workarea_size_policy '를 AUTO로 운영할 것인지, MANUAL로 운영할 것인지 이며, AUTO로 운영하게 되면 *_AREA_SIZE를 이용하지 않고 가능한 Temp I/O 없이 Sort,Hash Memory를 이용하므로 Perfoamce의 상당한 효과를 준다. 또한 Plan에 영향을 주는 주요항목이다. 기본적으로 아래의 Parameter를 설정한 환경하에서 Tuning하도록 하며, Full Table Scan의 규모가 자주 나서, 가능한 Index Scan위주로 변경하고자 한다면, optimizer_index_caching=0 ->20 ~ 40 ,optimizer_index_cost_adj=100 -> 40~80정도로 조정해 볼 필요가 있다. 기존의 시스템에 비해 DB Block Size가 늘어나고, db_file_multiblock_read_count도 늘어난다면 Full Table Scan이 기존 시스템보다 커질 가능성이 아주 높다. 그러므로 이 파라메터에 주의를 기울여야 한다.

항목	권 장 사항 내용
1. Oracle Parameter	✓ db_file_multiblock_read_count=16 or 32 ✓ hash_join_enabled=TRUE ✓ **optimizer_index_caching=0 (OPEN이후 Full Table 비중이 너무 높을 경우 20 ~ 40으로)** ✓ **optimizer_index_cost_adj=100 (OPEN이후 Full Table 비중이 너무 높을 경우 40 ~ 80으로)** ✓ **pga_aggregate_target= (OS Memory – SGA) * 0.2 (==>SGA를 제외한 OS Memory의 20%로 Start)** ✓ **query_rewrite_enabled=TRUE** ✓ session_cached_cursors=0 (Literal SQL을 공유화 이후 100정도 설정) ✓ **shared_pool_reserved_size=0** ✓ **shared_pool_size=(기존 값의 1.5배)** ✓ transaction_auditing=FALSE ✓ **workarea_size_policy=AUTO (==> *_AREA_SIZE는 필요 없음)** ✓ optimizer_dynamic_sampling=(1(=>9i), 2(=>10g) ✓ skip_unusable_indexes=TRUE (10g Only) ✓ **statistics_level=TYPICAL (9i,10g)**
2. SQL 유형 및 Tuning	✓ 1회성 Literal SQL유형 전면 수정(Bind변수로 공유화) ==> PRO*C의 Method 2,3,4를 사용. ✓ 비효율적인 SQL에 대해 Tuning. ✓ 실행빈도가 높은 SQL은 가능한 더 정교하게 Tuning (V$SQL의 Execution의 역순에 의해 확인) ✓ V$SORT_USAGE를 통해 TEMP과다 발행 SQL은 Tuning ✓ Row Chaining%가 높은 Table은 신 시스템 구축시 PCTFREE를 기존보다 크게 준다. ✓ HASH Join등을 적극 활용 ✓ 신기능 및 새로운 SQL 기법 적용. (SQL 1999, GROUPING SET등)

	✓ LOOP Query 는 가능한 줄일 수 있는 방법으로 변경 ✓ 계산 용도의 SQL 제거. APP단에서 처리 ✓ 불필요한 Function 제거 ✓ 불필요한 Hint제거. Hint는 Optimizer의 판단을 방해한다. ✓ PREFETCH, Bulk Binding,Bulk Collecting,Array Processing등 적극 활용
3. PRO*C Compile Option	✓ PRO*C의 PreCompile option의 Prefetch=100, release_cursor=no, hold_cursor=no를 Default로 적용 (**단 관련 Module이 Bind변수화 되어 있다면 HOLD_CURSOR=YES로 적용**) ✓ DBMS_APPLICATION_INFO Package를 이용한 SQL의 실행 Source를 확인할 수 있는 체제 구축

성능향상을 위한 개발자 고려사항

1. **집중적인 SQL문장이 실행되는 시간(OLTP) 동안에 Literal을 사용한 SQL을 집중적으로(초당 수백개)실행되는 Application은 Bind변수를 사용하여야 합니다.**
   - ✓ 집중적인 기준은 짧은 시간(수초 내외)동안 100여개 이상의 비공유 SQL을 실행 시키는 기준
   - ✓ 주로 Loop 내의 Literal의 Concatenation으로 SQL문장을 만들어 실행시키는 형태
   - ✓ select문장 뿐만 아니라 insert/update/delete문장도 포함된다.

2. **사용자의 증가 또는 처리 Data의 수의 증가에 따른 SQL실행 수의 변화가 급격히 증가되는 Application은 Bind변수를 사용하여야 합니다.**
   - ✓ 사용자 또는 Data의 증가에 따라 실행 SQL수가 달라지면 단위 시간동안 Parsing을 해야 하는 수가 증가되며, 급격히 시스템 Resource를 고갈시키며, Internal lock(latch)경합 발생으로 Hang현상이 발생할 수 있다.

3. **불필요한 Function(Oracle,User Defined)은 사용하지 않도록 합니다.**
   - ✓ Not Null Constraint가 걸려 있는 Column에 NVL() 을 사용하는 경우 및 SUM(NVL())과 같이 비효율적으로 사용하는 경우.
   - ✓ NVL() Function인 0.0001초의 CPU를 소모한다면 1000000 Row를 처리할 경우 10초의 CPU를 소모

```
select NVL(COL1,0),Col2,Col3  <<<==== COL1 컬럼은 Not Null Constraint로 Null이 올 수 없음

      from TAB...
```

4. **PRO*C 로 개발된 경우 BIND변수를 사용한 형태로 개발된 경우라면, REALSE_CURSOR=NO, PREFETCH=100 (Data량이 많다면 1000)를 이용합니다. OLTP에서 사용되는 SQL들은 Bind변수를 사용하는 형태로 수정되어야 합니다.**

5. **JDBC Application개발 본에 대해서 BIND변수를 사용하도록 합니다.**

   -JDBC로 개발되는 Application은 DB에 접근하는 SQL은 Bind변수를 사용한 형태로 수정되어야 합니다.

```
     pstmt = conn.prepareStatement ("select ename from emptest where empno = " + ii);

     rset  = pstmt.executeQuery();
```

```
     pstmt = conn.prepareStatement ("select ename from emptest where empno = ?");

     pstmt.setInt (1, ii); // Set the Bind Value

     rset  = pstmt.executeQuery();
```

6. JDBC로 개발된 경우 PREFETCH=100정도를 사용합니다.

PREFETCH란 Array Fetch(한번에 DB에서 읽어서 Client로 가져오는 단위)기능을 Driver단에서 처리해 주는 기능으로 DB와의 Roundtrip을 줄이고 성능향상을 가져올 수 있는 기능입니다.

default=> 10 이며 OLTP는 100정도 설정하는 것이 효과적입니다. 그렇다고 너무 큰 값을 설정하면 안됩니다.

예) PREFETCH 확인 방법

```
int default_row_prefetch = ((OracleConnection)conn).getDefaultRowPrefetch ();

System.out.println ("The Default RowPrefetch for the connection is: "  + default_row_prefetch);
```

예) PREFETCH설정 방법

```
((OracleStatement)stmt).setRowPrefetch (100);
```

공유/비공유 SQL의 장단점

	장점	단점	장점주요 사용 환경
공유 SQL	Parsing을 다시 하지 않는다. 사용자의 증가나 SQL실행규모의 증가에도 시스템 Resource가 완만하게 증가한다.	분포도가 편향되어 있는 경우 값에 따른 다른 Plan이 결정되지 못하는 단점	OLTP, 높은 SQL실행규모의 시스템
비공유SQL	Parsing시 Plan을 결정하는데 Bind변수를 사용하는 것보다 정확하다.	Library Cache Contention유발 library cache latch, shared pool latch 경합 발생 Parsing CPU 사용 SQL실행규모가 커지면 급격한 Resource(CPU,Memory)감소 Memory Fragmentation발생 (Shared Pool)	DW,야간 Batch, 소수의 사용자 시스템

JDBC를 이용한 공유/비공유 SQL의 실행 TEST결과

다음은 아래의 SQL문장을 Bind변수를 사용한 공유 SQL과, 상수를 결합한 형태로 SQL문장을 만들어 실행시키는 비공유SQL의 9999회 실행시켜 Oracle의 Shared pool Memory사용현황과 Parsing시 CPU사용시간을 TEST한 상황이다. (단 아래의 내용은 실행 Server별로 차이는 있음)

결론적으로 보면, 사용 Memory와 CPU 사용률이 실행 규모에 비례에 증가되고, 실행된 SQL문장이 기존의 Cache화 되어있는 SQL문장들을 밀어내는 역할을 한다.

	SQL 유형	Shared Pool의 Memory사용	Parsing 수	Exec 수	Parsing CPU Usage
공유 SQL	select ename from emptest where empno = :1	9807	1	9999	0.01 sec
비공유SQL	select ename from emptest where empno = 1 select ename from emptest where empno = 2 select ename from emptest where empno = 3 …… ……	93,219,148 (92 MB)	9999	9999	14.33 sec

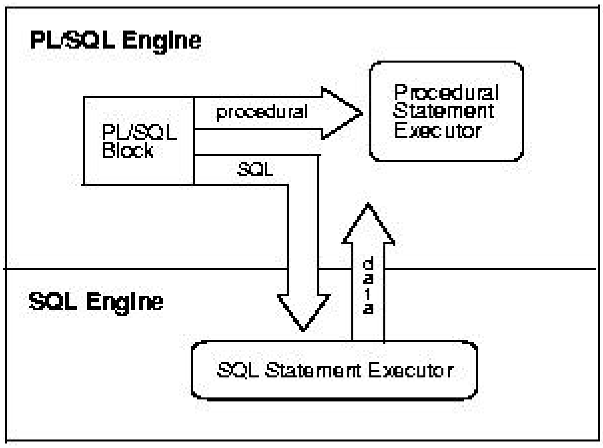
Literal을 사용한 비공유 SQL (재사용할 수 없는 비공유 SQL – JDBC Sample)

```
import java.sql.*;

import oracle.jdbc.driver.*;

class StatementLiteral
{
   public static void main (String args [])  throws SQLException
   {

        int ii;

       // Load the Oracle JDBC driver
       DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

       // Connect to the database
       Connection conn =DriverManager.getConnection
("jdbc:oracle:thin:@192.168.139.153:1521:ORA9iR2L", "scott", "tiger");

        // SQL Trace ON
        PreparedStatement pstmt = conn.prepareStatement ("alter session set events '10046
trace name context forever, level 1'");
        ResultSet rset = pstmt.executeQuery();



        ii = 1;
       while(ii <= 9999)
       {
             pstmt = conn.prepareStatement ("select ename from emptest where
empno = " + ii);

           try{
                   rset = pstmt.executeQuery();
           } catch (SQLException e) {
               System.out.println ("Expected exception thrown: " + e.getMessage());
           }
           ii = ii + 1;
           while(rset.next ())
           {
               System.out.println (rset.getString (1));
           }
           rset.close ();

           pstmt.close ();
        }

        // SQL Trace OFF
        pstmt = conn.prepareStatement ("alter session set events '10046 trace name context
forever, level 8'");
        rset = pstmt.executeQuery();


        pstmt.close ();
   }
}
```

Bind변수를 사용한 공유 SQL (재사용 가능한 공유 SQL – JDBC Sample)

```
import java.sql.*;

import oracle.jdbc.driver.*;

class StatementBind
{
   public static void main (String args [])  throws SQLException
   {

        int ii;

        // Load the Oracle JDBC driver
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        // Connect to the database
        Connection conn =DriverManager.getConnection
("jdbc:oracle:thin:@192.168.139.153:1521:ORA9iR2L", "scott", "tiger");

        // SQL Trace ON
        PreparedStatement pstmt = conn.prepareStatement ("alter session set events '10046
trace name context forever, level 1'");
        ResultSet rset = pstmt.executeQuery();

        pstmt = conn.prepareStatement ("select ename from emptest where
empno = ?");

        ii = 1;
        while(ii <= 9999)
        {
           pstmt.setInt (1, ii); // Set the Bind Value
           try{
               rset = pstmt.executeQuery();
           } catch (SQLException e) {
               System.out.println ("Expected exception thrown: " + e.getMessage());
           }
           ii = ii + 1;
           while(rset.next ())
           {
               System.out.println (rset.getString (1));
           }
           rset.close ();
        }
        // SQL Trace OFF
        pstmt = conn.prepareStatement ("alter session set events '10046 trace name context
forever, level 8'");
        rset = pstmt.executeQuery();


       pstmt.close ();
   }
}
```

PL/SQL Engine과 SQL Engine의 Overhead를 줄이기 위한 방안



위의 그림은 일반적인 PL/SQL engine이 SQL engine과 어떻게 상호작용을 하는가를 보여주고 있다. 정상적인 PL/SQL 프로그램에서 SQL을 수행해야 하는 경우 이를 SQL engine에 보내게 되고 SQL engine은 이를 처리하고 만약 필요할 경우 Data를 반환하게 된다. 이러한 PL/SQL engine과 SQL engine 사이의 context switching은 overhead를 야기한다. 그러므로 불필요한 SQL문장(계산용 SQL)의 실행은 PL/SQL문장의 자체 기능으로 처리하며, 또한 loop성 Query는 Bulk Binding형태로 처리하는 기능을 제공한다.

예1) <<< Bulk Binding 을 이용한 Insert Sample >>>

```
SET SERVEROUTPUT ON
CREATE TABLE parts (pnum NUMBER(4), pname CHAR(15));


DECLARE
   TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY_INTEGER;
   TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY_INTEGER;
   pnums NumTab;
   pnames NameTab;
   t1 NUMBER(5);
   t2 NUMBER(5);
   t3 NUMBER(5);
   PROCEDURE get_time (t OUT NUMBER) IS
   BEGIN SELECT TO_CHAR(SYSDATE,'SSSSS') INTO t FROM dual; END;
BEGIN
   FOR j IN 1..5000 LOOP -- load index-by tables
          pnums(j) := j;
          pnames(j) := 'Part No. ' || TO_CHAR(j);
   END LOOP;

   get_time(t1);

   FOR i IN 1..5000 LOOP -- use FOR loop
          INSERT INTO parts VALUES (pnums(i), pnames(i));
   END LOOP;

   get_time(t2);

   FORALL i IN 1..5000 -- use FORALL statement
          INSERT INTO parts VALUES (pnums(i), pnames(i));

   get_time(t3);
   dbms_output.put_line('Execution Time (secs)');
   dbms_output.put_line('---------------------');
   dbms_output.put_line('FOR loop: ' || TO_CHAR(t2 - t1));
   dbms_output.put_line('FORALL: ' || TO_CHAR(t3 - t2));
END;
/
```

예2) <<< Bulk Collect & Bulk Binding 을 이용한 Update Sample >>>

```
drop table emp2;

CREATE TABLE emp2 (deptno NUMBER(2), job VARCHAR2(15));

INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(10, 'Clerk');
INSERT INTO emp2 VALUES(20, 'Bookkeeper');
INSERT INTO emp2 VALUES(30, 'Analyst');
INSERT INTO emp2 VALUES(30, 'Analyst');


drop table emp3;

CREATE TABLE emp3 (deptno NUMBER(2), job VARCHAR2(15));

INSERT INTO emp3 VALUES(10, '');
INSERT INTO emp3 VALUES(10, '');
INSERT INTO emp3 VALUES(20, '');
INSERT INTO emp3 VALUES(30, '');
INSERT INTO emp3 VALUES(30, '');

commit;


DECLARE
    TYPE EMP2_JOB IS TABLE OF EMP2.JOB%TYPE;
    TYPE EMP2_DEPTNO IS TABLE OF EMP2.DEPTNO%TYPE;
    EMP2_JOB_C EMP2_JOB;
    EMP2_DEPTNO_C EMP2_DEPTNO;
BEGIN

   SELECT JOB,DEPTNO BULK COLLECT INTO EMP2_JOB_C,EMP2_DEPTNO_C FROM EMP2;

   FORALL i IN 1..EMP2_JOB_C.COUNT
     UPDATE emp3 set job = EMP2_JOB_C(i) where deptno  = EMP2_DEPTNO_C(i);

EXCEPTION
   WHEN OTHERS THEN
   ROLLBACK; -- or COMMIT;

END;
/
```

SQL Tuning을 위한 Guide

SQL Tuning시 고려사항

1. 가능한 Hint는 사용하지 않는다. Hint를 많이 구사한 Application들은 지난 호에 설명했던 것과 같이 기준(Parameter 및 통계정보)이 잘 못 설정된 경우가 많다. Hint는 최후에 어쩔 수 없는 경우에 사용하도록 한다.
   - ✓ 1차적으로 Plan이 원하는 경우가 아닐 경우 통계정보 및 Init.ora의 Parameter값들을 확인해 본다.

     [USER|ALL|DBA]_TABLES,[USER|ALL|DBA]_INDEXES,[USER|ALL|DBA]_TAB_COLUMNS 등의 Dictionary정보 확인, 최종 Analyze시간. Block수,Row수, Column의 Distinct값, Index의 Clustering Factor, Sample Size등을 확인하다. 이들 값들이 현실 Data와 비슷한지 확인한다. 통계정보가 없거나, 너무 오래됐거나 Sampling Size가 너무 작은 경우 현실 Data와 다를 수 있다. 이러한 경우는 통계정보를 다시 생성한다.
   - ✓ Column에 대한 통계정보(Histogram)는 안 돌리는 것을 원칙으로 한다. 그러나 편향된 Data분포도를 가지고 있다면 Histogram을 운영한다. 또한 이들 Column에 대한 Where절의 사용되는 값들은 Bind변수를 사용하지 않도록 한다. Histogram을 사용하는 곳에는 Literal값을 사용하여야 Plan이 효과적으로 풀린다.
   - ✓ Hint를 지정할 경우는 가능한 Tight하게 주도록 한다. 그렇지 않을 경우 향후 plan이 변경될 가능성이 많기 때문이다.

     예) /*+ USE_NL(a b) */ ==> /*+ ORDERED USE_NL(a b) .... */
   - ✓ Hint는 Hint의 의미를 정확히 이해하고 합당한 Hint를 주도록 한다. 어설픈 힌트는 오히려 역효과가 발생되는 경우가 많다.

2. 통계정보는 운영 중에 직접 돌리지 않는다.
   - ✓ 집중적인 운영시기에 통계정보 수집을 위한 실행은 Library Cache Contention을 유발하며, 관련 SQL 및 PL/SQL들을 Invalid시켜 성능저하 및 문제의 원인이 되기도 한다.
   - ✓ 저녁시간의 한가한 시간을 이용해서 돌린다. (특히 집중적인 시스템의 사용시기 등은 조심)
3. WORKAREA_SIZE_POLICY=AUTO 이면 *_AREA_SIZE는 이용하지 않으며, 설정해봐야 의미가 없다. 즉 Optimizer는 *_AREA_SIZE에 의해 Plan을 결정하지 않는다.
4. Tuning시 Plan적인 Tuning뿐만 아니라 구조적인 Tuning에도 집중한다.
   - ✓ Execution이 높은 것은 과다한 Loop Query가 아닌지 검토한다. Loop Query 보완, 최적화.
   - ✓ 1회성(Literal을 사용) 비공유 SQL, 특히 집중적으로 실행되는 Literal SQL은 Bind변수 기법을 사용한다.
   - ✓ Parsing을 줄이는 방법으로 JAVA의 "Statement Cache", PRO*C의 RELEASE_CURSOR=NO 등 Language의 효과적인 기법들을 사용한다.
   - ✓ Array Processing기능을 사용한다. 기본적으로 거의 대부분의 DB접속 방식에서 이들 기법들을 제공하고 있으며, Coding상의 특별한 처리 없이 PREFETCH 기능이 이와 같은 기능이며, Application시 특별한 구현이 필요 없이 Option의 설정만으로 가능하다. (ODBC,JDBC,OO4O,ADO,PRO*C등)
   - ✓ PL/SQL의 Bulk Binding/Bulk Collecting 기능을 이용한다.
   - ✓ Aggregate Function등 향상되고 효과적인 여러 Query기능을 활용한다.
5. Tuning시 Plan은 상수(Literal)로 TEST하지만 실제는 bind변수로 운영되는 경우 Plan이 다를 수 있다. Program에 bind변수로 되어 있다면 bind변수로 Plan을 확인해 봐야 한다.
6. 다음의 사항도 고려한다.
   - ✓ Hash Join을 사용할 경우 Driving 순서, Row Set을 고려하여 사용한다.

- ✓ SQL문장에서 반드시 필요한 Column에 대해서만 Select한다. 불필요한 Column들은 Sort,Hash Operation에서 Memory에 Loading되어야 하므로 Temp Disk I/O의 원인이 되기도 한다.
- ✓ Chaining % 비율을 항상 검토하고 Row Chaining비율이 높은 table에 대해서는 Column의 Data Type 조정 및 Block의 PCTFREE등을 늘리도록 한다. Table의 구조적인 문제, 또는 업무적인 형태를 고려하여 Reorg(CTAS,MOVE,Exp/Imp등 이용)를 하도록 한다.
- ✓ 평균 Row길이 와 Block당 Row수도 항상 주의 깊게 관찰하여 문제점이 없는지 검토한다. (통계정보 이용 DBA_TABLES.NUM_ROWS/DBA_TABLES.BLOCKS) Block당 Row가 적은 경우는 DELETE가 많이 됐다는 경우이므로 Full Table Scan이 자주 발생된다면 Reorg대상이 될 수 있다. 그러나 RAC환 경에서는 Block경합을 줄이기 위해 인위적으로 PCTFREE를 키워 Block당 Row수를 적게 가져가는 경우도 있다.
- ✓ Hash Join과 Sort Merge Join시 TEMP쪽에 I/O가 발생하지 않도록 한다.
- ✓ PRO*C Application일 경우 BIND변수의 사용 여부 ,RELEASE_CURSOR=NO, PREFETCH=1000(batch), PREFETCH=100(OLTP)를 권장한다.
- ✓ PL/SQL의 Batch Job일 경우 Bulk Binding, Bulk Collecting을 이용하도록 유도한다.

Execution Plan 보기

Execution Plan이란 Optimizer가 Query Optimizer단계에서 RBO또는 CBO에 의해 결정해낸 Optimal Access Path정보를 가지고 QEP Generator가 만들어낸 실행계획을 나타낸다. 이 Execution Plan은 SQL문장이 실행될 때 필요한 모든 정보를 포함하고 있다.

① Access Path : 어떠한 방법으로 Data에 접근할 것인가? (Index Scan , Index Fast Full Scan, Full Table Scan 등)

② Join Method : 어떠한 Join Method가 사용될 것인가?

③ Join Order : 어떠한 Join의 순서로 풀릴 것인가?

다음과 같은 Execution Plan을 가정하자.

```
 ID PID Execution Plan
----------------------------------------------------------
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=5 Bytes=250)
 1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=1 Card=5 Bytes=160)
 2    1     NESTED LOOPS (Cost=3 Card=5 Bytes=250)
 3    2       TABLE ACCESS (BY INDEX ROWID) OF 'DEPT' (TABLE) (Cost=2 Card=1 Bytes=18)
 4    3         INDEX (RANGE SCAN) OF 'DEPT_DEPTNO' (INDEX) (Cost=1 Card=1)
 5    2       INDEX (RANGE SCAN) OF 'EMP_DEPTNO' (INDEX) (Cost=0 Card=5)
```

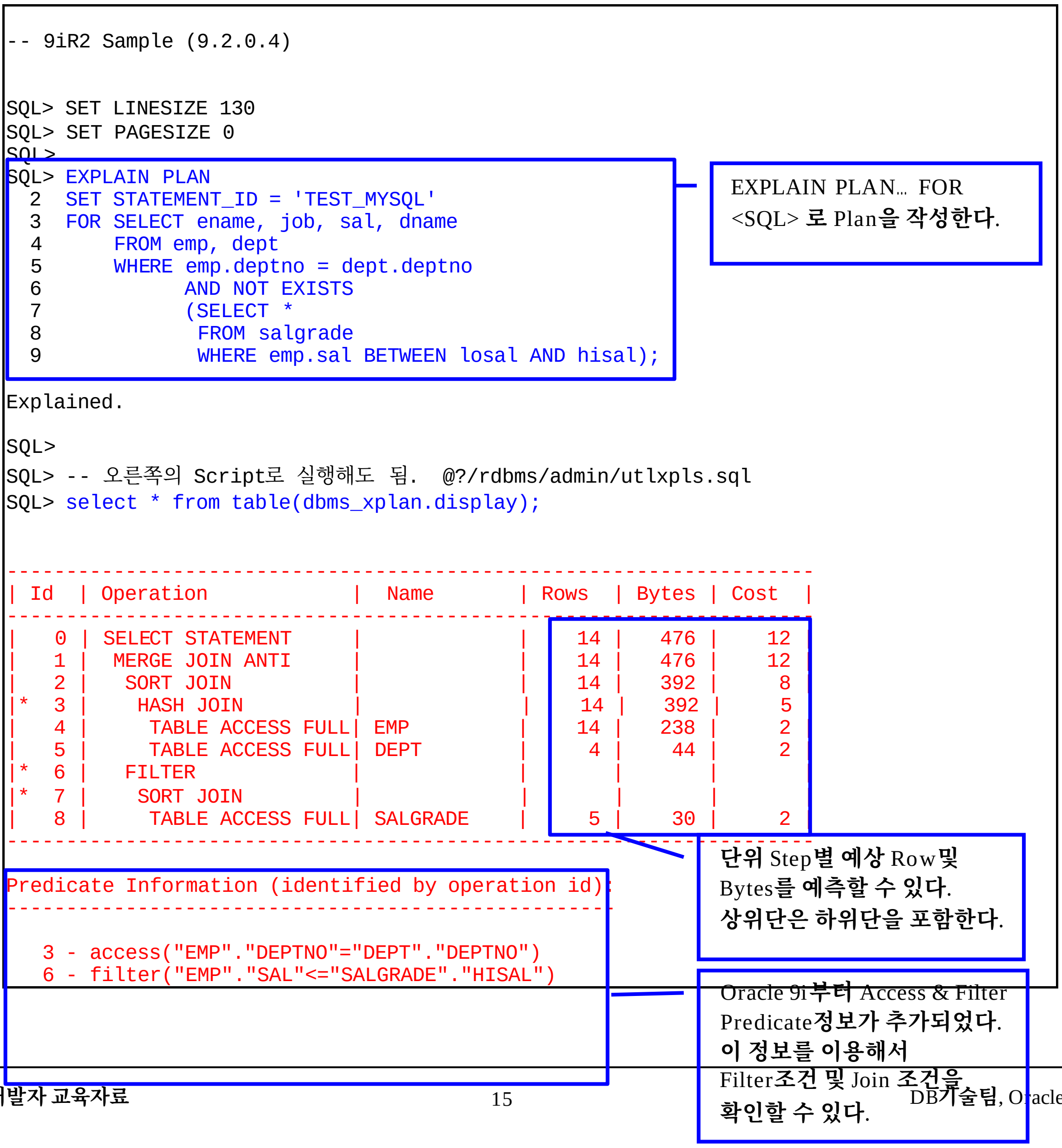
Plan에서 나오는 각 line을 우리는 Row Source라고 한다. Plan을 보면서 처리 순서를 판단하는 것은 간단하다. Plan은 Tree형태로 되어 있으며, 자신보다 하위 Level이 있으면 하위 Level부터, 같은 Level이라면 위의 Row Source부터 실행된다고 보면 된다. 위의 Plan에서 “Optimizer=CHOOSE”에서 알 수 있듯이 해당 SQL문장은 Optimizer mode가 CHOOSE에서 Plan이 만들어진 것이다. 또한 Plan에서 “Cost=”의 항목이 나오면 CBO로 풀렸다는 것이다. RBO인지 CBO인지의 판단은 Optimizer mode의 항목으로 판단하는 것이 아니라 “Cost=”로 판단한다는 것에 주의하자. 우리는 위의 Plan에서 2개의 Table “DEPT”와 “EMP” Table의 각각 ① Access Path를 확인할 수 있다. 모두 Index를 사용하고 있는 것이다. 또한 ② Join Method로는 “NESTED LOOP” Join이 사용되었다. ③ Join Order로는 자신보다 하위 Level이 있으면 하위 Level부터, 같은 Level이라면 위의 Row Source부터 실행된다는 법칙을 적용해 보면 ID를 기준으로 4→3→5→2→1→0 의 순서로 처리된다. 단 Join Method가 Nested Loop이기 때문에 3에서 Return되는 Row수 만큼 다음 단계가 반복된다. 그러므로 Join Order는 DEPT → EMP순으로 Nested Loop로 처리될 것이라는 것을 알 수 있다. 또한 “Card=5”는 Computed Cardinality를 나타내며 몇 건의 Row가 Return될 것인지를 CBO가 통계정보를 이용해서 계산해 낸 값이다. “Bytes=250”는 Return될 Row들의 Bytes를 나타내므로 5 Row에 250 Bytes정도의 Return

Row가 발생할 것이라는 것을 예측할 수 있다. 통계정보만 정확하다면 이들 값도 상당히 정확하다고 보면 된다.

Execution Plan을 보기 위해서는.

- 각 Database User마다 "PLAN_TABLE"이 있어야 하며, PLAN_TABLE은 Oracle Version마다 다르다. 해당 Oracle Version의 $ORACLE_HOME/rdbms/admin/utlxplan.sql을 실행하면 만들어 진다.
- SQL을 실행하지 않고 Trace만 보는 방법
  "EXPLAIN PLAN", SQL*Plus의 "SET AUTOTRACE TRACEONLY EXPLAIN"
- PLAN을 PLAN_TABLE에서 확인하는 방법
  (< 8i)이전까진 Plan_Table에서 직접 Select
  (>= 8i) 이상부터는 Plan_Table에서 직접 Select 하거나,
  "$ORACLE_HOME/rdbms/admin"위치에서 utxpls.sql(Serial Plan) 또는 utlxplp.sql(Parallel Plan) Script실행
  (>=9i) utxpls.sql, utlxplp.sql 외에 "select * from table(dbms_xplan.display);" Query

Execution Plan보기(EXPLAIN PLAN) 실행 예 (Oracle 9i, 10g)



```
-- 9iR2 Sample (9.2.0.4)


SQL> SET LINESIZE 130
SQL> SET PAGESIZE 0
SQL>
SQL> EXPLAIN PLAN
  2  SET STATEMENT_ID = 'TEST_MYSQL'
  3  FOR SELECT ename, job, sal, dname
  4      FROM emp, dept
  5      WHERE emp.deptno = dept.deptno
  6            AND NOT EXISTS
  7            (SELECT *
  8             FROM salgrade
  9             WHERE emp.sal BETWEEN losal AND hisal);
Explained.

SQL>
SQL> -- 오른쪽의 Script로 실행해도 됨.  @?/rdbms/admin/utlxpls.sql
SQL> select * from table(dbms_xplan.display);


--------------------------------------------------------------------
| Id  | Operation              |  Name         | Rows  | Bytes | Cost  |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT       |               |   14 |   476 |    12 |
|   1 |  MERGE JOIN ANTI       |               |   14 |   476 |    12 |
|   2 |   SORT JOIN            |               |   14 |   392 |     8 |
|*  3 |    HASH JOIN           |               |   14 |   392 |     5 |
|   4 |     TABLE ACCESS FULL| EMP           |   14 |   238 |     2 |
|   5 |     TABLE ACCESS FULL| DEPT          |    4 |    44 |     2 |
|*  6 |   FILTER               |               |      |       |       |
|*  7 |    SORT JOIN           |               |      |       |       |
|   8 |     TABLE ACCESS FULL| SALGRADE      |    5 |    30 |     2 |
--------------------------------------------------------------------

Predicate Information (identified by operation id):
--------------------------------------------------------

   3 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
   6 - filter("EMP"."SAL"<="SALGRADE"."HISAL")
```

EXPLAIN PLAN… FOR <SQL> 로 Plan을 작성한다.

단위 Step별 예상 Row및 Bytes를 예측할 수 있다. 상위단은 하위단을 포함한다.

Oracle 9i부터 Access & Filter Predicate정보가 추가되었다. 이 정보를 이용해서 Filter조건 및 Join 조건을 확인할 수 있다.

```
   7 - access("EMP"."SAL">="SALGRADE"."LOSAL")
       filter("EMP"."SAL">="SALGRADE"."LOSAL")

Note: cpu costing is off

24 rows selected.
```

```
-- 10g Sample (10.1.0.2)

SQL> -- @?/rdbms/admin/utlxpls.sql
SQL> select * from table(dbms_xplan.display);

---------------------------------------------------------------------------------
| Id  | Operation            | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |           |    14 |   532 |    18  (17)| 00:00:01 |
|   1 |  MERGE JOIN ANTI     |           |    14 |   532 |    18  (17)| 00:00:01 |
|   2 |   SORT JOIN          |           |    14 |   420 |    12  (17)| 00:00:01 |
|*  3 |    HASH JOIN         |           |    14 |   420 |    11  (10)| 00:00:01 |
|   4 |     TABLE ACCESS FULL| DEPT      |     4 |    52 |     5   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| EMP       |    14 |   238 |     5   (0)| 00:00:01 |
|*  6 |   FILTER             |           |       |       |            |          |
|*  7 |    SORT JOIN         |           |     5 |    40 |     6  (17)| 00:00:01 |
|   8 |     TABLE ACCESS FULL| SALGRADE  |     5 |    40 |     5   (0)| 00:00:01 |
---------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
   6 - filter("EMP"."SAL"<="HISAL")
   7 - access("EMP"."SAL">="LOSAL")
       filter("EMP"."SAL">="LOSAL")
```

Oracle 10g부터 Default로 CPU반영 비중과 예측시간을 확인할 수 있다. 10g부터는 Cost기준이 Time이므로 가능하다.

Execution Plan보기(SQL*Plus의 SET AUTOTRACE) 실행 예 (9i)

```
9iR2 Sample (9.2.0.4) ======================>

SQL> SET AUTOTRACE ON
SQL>
SQL> SELECT ename, job, sal, dname
  2    FROM emp, dept
  3   WHERE emp.deptno = dept.deptno
  4         AND NOT EXISTS
  5         (SELECT *
  6            FROM salgrade
  7           WHERE emp.sal BETWEEN losal AND hisal);

no rows selected
```

① SQL문장이 실행 되었다. 실행은 하지 않고 Plan만 확인 하려면 "SET AUTOT TRACEONLY EXPLAIN"으로 처리한다.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=14 Bytes=756)
   1    0   MERGE JOIN (ANTI) (Cost=13 Card=14 Bytes=756)
   2    1     SORT (JOIN) (Cost=8 Card=14 Bytes=392)
   3    2       HASH JOIN (Cost=5 Card=14 Bytes=392)
   4    3         TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=238)
   5    3         TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=44)
   6    1     FILTER
   7    6       SORT (JOIN)
   8    7         TABLE ACCESS (FULL) OF 'SALGRADE' (Cost=2 Card=409 Bytes=10634)
```

② 내부적으로 EXPLAIN PLAN… 으로 처리된 PLAN정보를 보여준다. 실제 Runtime Plan은 아니다.

```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         21  consistent gets
          0  physical reads
          0  redo size
        376  bytes sent via SQL*Net to client
        372  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

③ 실제 실행된 Execution Statistics를 보여 준다. 전체 읽어들인 Block수는 db block gets + consistent gets.

Cached Execution Plan(V$SQL_PLAN)

Explain Plan으로 보는 Plan과 실제 Runtime시 Plan이 다를 수 있다. 이는 Explain Plan은 단지 SQL문장에 대한 구조적인 분석 하에 예상 Plan을 만들어 내게 된다. 예를 들어 "select * from emp where empno = :B1 " 의 SQL문장을 실행한다고 생각해 보자. Empno가 Index가 설정되어 있고 Number Type으로 되어 있으며, Index를 이용하는 것이 효과적이라고 가정하면 Explain plan은 empno의 index를 이용해서 풀릴 것이다. 그러나 실제 Runtime시 Bind변수인 ":B1"에 Number Type이 아니라 Character Type으로 Bind되었다면 Index를 이용할 수가 없는 것이다. 또한 Oracle 9i의 Bind Peeking과 같은 기능인 처음 Binding되는 Literal에 의해 Plan을 결정하는 방식에 의해서도 그러한 상황이 있을 수 있다. 이와 같이 Explain Plan을 보는 것과 실제 예측 실행시간이 너무 차이가 난다면 Runtime Plan을 확인해 볼 필요가 있다. Oracle 9i부터 V$SQL_PLAN의 Performance View를 제공하며, 현재 Cache화 되어 있는 SQL문장들에 대한 Runtime Plan을 확인해 볼 수 있다. V$SQL_PLAN은 PLAN_TABLE과 Column항목이 거의 같다.

```
 SELECT hash_value,  (select sql_text from v$sql s      where s.hash_value = p.hash_value and s.address =
p.address and rownum <= 1),    child_number,ID ,PARENT_ID ,    LPAD('
',2*(depth))||OPERATION||DECODE(OTHER_TAG,NULL,'','*')||      DECODE(OPTIONS,NULL,'','
('||OPTIONS||')')||DECODE(OBJECT_NAME,NULL,'',' OF '"'||OBJECT_NAME||'"')||
DECODE(OBJECT#,NULL,'','(Obj#'||TO_CHAR(OBJECT#)||')')||DECODE(ID,0,DECODE(OPTIMIZER,NULL,'','
Optimizer='||OPTIMIZER))||DECODE(COST,NULL,'',' (Cost='||COST||DECODE(CARDINALITY,NULL,'','
Card='||CARDINALITY)||DECODE(BYTES,NULL,'',' Bytes='||BYTES)||')') SQLPLAN,OBJECT_NODE,
PARTITION_START ,PARTITION_STOP, PARTITION_ID, CPU_COST, IO_COST, TEMP_SPACE, DISTRIBUTION,
OTHER , ACCESS_PREDICATES , FILTER_PREDICATES FROM v$sql_plan p
START WITH ID=0 and hash_value = [XXXXXXXXXX]
CONNECT BY PRIOR ID=PARENT_ID AND
                PRIOR hash_value=hash_value AND
                PRIOR child_number=child_number
ORDER BY hash_value,child_number,ID,POSITION
```

SQL_TRACE와 TKPROF를 이용한 SQL Tuning

SQL_TRACE or 10046 Trace Enable/Disable

SQL_TRACE는 Application이 SQL문장들을 처리되는 과정을 Trace로 남기게 하는 기능이다. 10046 Trace기능은 SQL_TRACE의 기능에 추가적인 정보를 기록한다. Level 1은 SQL_TRACE기능과 같으며, Level 4는 Bind변수 정보, Level 8은 Wait Event정보, Level 12는 Bind변수 정보 와 Wait Event정보를 같이 보여 준다. 주의할 사항으로 Trace를 On시켰으면 반드시 Monitoring후 Off를 해야 한다는 것이다. 그렇지 않을 경우 Disk Full이 발생할 수 있으므로 반드시 주의해야 한다. Trace는 init.ora의 user_dump_dest에서 지정된 곳에 생성된다.

- At the instance level : (init.ora Parameter이용)

```
sql_trace = {TRUE  |  FALSE}
```

```
또는
event = "10046 trace name context forever, level {1|4|8|12}"
```

- At the Session level : (SQL*Plus 또는 Application Routine내)

```
ALTER SESSION SET SQL_TRACE= {True|False};
또는 10046 Trace On
alter session set events '10046 trace name context forever, level {1|4|8|12}';
10046 Trace Off
alter session set events '10046 trace name context off';
또는
EXECUTE dbms_session.set_sql_trace({True|False});
또는
EXECUTE dbms_system.set_sql_trace_in_session(session_id, serial_id,
{True|False});
```

Execution Plan보기(SQL_TRACE,10046 Trace와 TKPROF) 실행 예 (9i)

Oracle 9i Release 2(9.2.x)부터는 Tuning에 대해 아주 유용하고 Tuning대상이 어느 곳(Plan상의 STEP) 인지를 판단하기 쉽게 획기적으로 개선되었으며, 초보자도 Tuning대상을 쉽게 찾을 수가 있다. 또한 사용자가 SQL문장을 실행해서 결과 값을 받는 서비스 Time은 "Service Time = DB실행시간 + Wait Time" 이며 9i부터는 SQL TRACE의 Level에 따라 Wait정보의 Summary정보도 같이 보여 주므로 어느 곳에 병목현상이 있는지 판단하기가 쉬워졌다. 다음은 Oracle 9i Release 2(9.2.x)부터 개선된 사항이다.

- 10046 Trace level에 따라 Wait(level 8, level 12일 경우)정보도 표시
- 각 Row Source (Plan상의 STEP) 마다 Statistics 표시
- 9i에서는 time=xxxxxxxxxx 정보가 1/1000000초 단위. 8i까지는 1/100초
- Run Time Plan & TKPROF실행시 Plan 주의
- TKPROF …. EXPLAIN=xxxx/yyyy 일 경우 Plan이 2개 (Runtime Plan & Tkprof 실행 시점의 Plan)

```
-- Oracle 9iR2 Sample (9.2.0.4)
-- ==============================


-- alter session set sql_trace=true;
alter session set events '10046 trace name context forever, level 12';


SELECT ename, job, sal, dname
  FROM emp, dept
 WHERE emp.deptno = dept.deptno
       AND NOT EXISTS
       (SELECT *
          FROM salgrade
         WHERE emp.sal BETWEEN losal AND hisal);

ORA9iR2L@oracle> tkprof ora9ir2l_ora_2137.trc ora9ir2l_ora_2137.prf explain=scott/tiger
width=132

TKPROF: Release 9.2.0.4.0 - Production on Mon Nov 22 16:30:54 2004

Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
```

```
ORA9iR2L@oracle> vi ora9ir2l_ora_2137.prf


SELECT ename, job, sal, dname
  FROM emp, dept
 WHERE emp.deptno = dept.deptno
       AND NOT EXISTS
       (SELECT *
          FROM salgrade
         WHERE emp.sal BETWEEN losal AND hisal)
```

①

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.06       0.09          0         21          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.06       0.09          0         21          0           0
```

②

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 59  (SCOTT)
```

③

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  MERGE JOIN ANTI (cr=21 r=0 w=0 time=94397 us)
     14   SORT JOIN (cr=14 r=0 w=0 time=92823 us)
     14    HASH JOIN  (cr=14 r=0 w=0 time=92237 us)
     14     TABLE ACCESS FULL OBJ#(30627) (cr=7 r=0 w=0 time=860 us)
      4     TABLE ACCESS FULL OBJ#(30628) (cr=7 r=0 w=0 time=275 us)
     14   FILTER  (cr=7 r=0 w=0 time=1238 us)
     40    SORT JOIN (cr=7 r=0 w=0 time=856 us)
      5     TABLE ACCESS FULL OBJ#(30630) (cr=7 r=0 w=0 time=383 us)
```

④

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   MERGE JOIN (ANTI)
     14    SORT (JOIN)
     14     HASH JOIN
     14      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMP'
      4      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'DEPT'
     14    FILTER
     40     SORT (JOIN)
      5      TABLE ACCESS (FULL) OF 'SALGRADE'
```

⑤

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1        1.57          1.57
```

① SQL문장 처리 정보

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.06       0.09          0         21          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.06       0.09          0         21          0           0
```

실행 수 Sec 처리된 Disk Block수 처리된 Row수

- 'query' : Consistent Read(CR), 'current' : Current Read(SCUR), 'disk' : Physical Read
- 'Parse': parsing(parse Request 수, 실제 Hard Parsing은 아님), 'Execute': SQL문장 실행 수, 'Fetch': Fetch회수
- 'cpu','elapsed': CPU 사용시간 및 전체 처리시간
- Logical Read = query + current (Logical Read는 Physical Read를 포함한다. 그러므로 Logical Read >= Physical Read이다. 그러나 반대의 경우라면 Temporary Tablespace로 Disk I/O(Sort,Hash,Bitmap Operation등에 의해)가 발생했다고 보면 된다. 여기서의 관점은 Logical Read는 전체 Block Buffer Operation의 량을 나타내므로 이들 처리 Block수를 줄이는 것이 Tuning의 관건이다.
- Fetch회수와 Row수와의 관계에서 Fetch=Rows일 경우는 Single Row Fetch의 처리를 하였고, Fetch <= Rows일 경우는 Array Fetch 또는 Prefetch처리를 한 경우이다. 여기서의 관점은 Performance향상을 위해 Array Fetch 또는 Prefetch를 효과적으로 사용하고 있느냐?의 관점이다.
- Parse수와 Execute관점에서 본다면 같은 SQL문장을 매번 실행할 때 마다 Parse Request를 할 필요가 없다는 것이다. 이를 줄이는 방법으로 Pro*C의 "RELEASE_CURSOR=NO"(Dynamic SQL은 해당 안됨), Java의 "Statement Cache" 등이 있다.

② SQL문장 Parsing정보

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 59  (SCOTT)
```

- 'Misses in library cache during parse'의 값이 0이면 Soft parsing, 1이면 SGA Cache에 없어서 Hard Parsing이 발생한 경우이다.
- Optimizer mode정보 및 Parsing Schema정보를 확인할 수 있다.
- Recursive SQL일 경우 Recursive depth정보도 나타난다.

③,④ SQL문장 Execution Plan정보(Runtime Plan/TKPROF실행시 Plan)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  MERGE JOIN ANTI (cr=21 r=0 w=0 time=94397 us)
     14   SORT JOIN (cr=14 r=0 w=0 time=92823 us)
     14    HASH JOIN  (cr=14 r=0 w=0 time=92237 us)
     14     TABLE ACCESS FULL OBJ#(30627) (cr=7 r=0 w=0 time=860 us)
      4     TABLE ACCESS FULL OBJ#(30628) (cr=7 r=0 w=0 time=275 us)
     14   FILTER  (cr=7 r=0 w=0 time=1238 us)
     40    SORT JOIN (cr=7 r=0 w=0 time=856 us)
      5     TABLE ACCESS FULL OBJ#(30630) (cr=7 r=0 w=0 time=383 us)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   MERGE JOIN (ANTI)
     14    SORT (JOIN)
     14     HASH JOIN
     14       TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMP'
      4       TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'DEPT'
     14    FILTER
     40     SORT (JOIN)
      5       TABLE ACCESS (FULL) OF 'SALGRADE'
```

Oracle 9i Release 2부터는 각 Row Source(Step)별 일량을 확인할 수 있는 Statistics정보를 보여준다.

- Tkprof의 explain=xxx/xxx을 주게 되면 2개의 Plan이 만들어진다. 처음 것이 Runtime Plan이며, 두번 째가 Tkprof utility를 돌린 시점의 EXPLAIN PLAN에 의해 만들어진 Plan이다. 이들 Plan은 tkprof를 돌린 시점이 다를 수 있으므로 다를 수가 있는 것이다.

- ‘Rows’: Oracle 8.0까지는 Access했던 즉 찾아 들어갔던 Row수로 나타나며, 8i 부터는 조건에 의해 Filter되어 Return된 Row수를 나타낸다.
- Oracle 9i Release 2부터는 각 Row Source(Step)별 일량을 확인할 수 있는 Statistics정보를 보여준다. 상위 Step은 하위 Step의 값들을 포함한다. 그러므로 전체 일량 중에서 병목현상이 어느 곳에서 가장 많이 발생된지를 명확히 알 수 있다. 즉 Tuning의 point를 정확히 확인 할 수 있다.
- ‘cr=’: Consistent Read 총 Block 수, ‘r=’(Oracle 10g는 ‘pr=’): Physical Read 총 Block수, ‘w=’(Oracle 10g는 ‘pw=’): Physical Write 총 Block수 , ‘time=’: 전체 처리시간 (micro second(1/1000000초 단위)
- 여기서 가장 상위의 “cr=21”은 “SQL문장 처리정보”에서 확인 한 Logical Read (= query + current)의 값과 같다는 것을 확인 할 수 있다. 또한 전체 처리시간은 “time=94397” 이므로 약 0.09초 걸렸다는 것을 알 수 있다. 그럼 전체 처리 일량 중 병목현상을 찾고자 한다면 전체 일량 중 가장 많은 Block처리 및 시간으로 Top/Down식으로 찾아내려 가면 누구든지 쉽게 찾을 수 있을 것이다. 해당 부분을 찾은 뒤, Join Method의 변경 /Join Order의 변경 / Access path의 변경에 Tuning을 실시하면 될 것이다.

TKPROF Output의 Row값의 Version별 변화.

TKPROF의 Output형태가 8i이전 까지는 Table 또는 Index를 찾아 들어간 Row수를 나타내며, 8i에서부터 Rows 값은 Filtering되어 Return된 Row수를 나타낸다. TKPROF output에 Rows값이 0으로 나오는 경우는 해당 SQL문장의 Cursor가 Close되기 전에 Trace가 종료되었거나, 끊긴 경우이다. Runtime Plan은 Cursor가 Close되는 시점에 기록되므로 이 값이 0으로 나올 수 있다.

```
-- Oracle 8.0.x의 Plan에
select /*+ ORDERED USE_NL(d e) */ *
from dept d, emp e
where e.deptno = d.deptno and d.deptno = 10

Rows     Execution Plan
-------  -----------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      3   NESTED LOOPS
      4    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'DEPT'
     14    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMP'
```

DEPT Table에서 Full Table Scan에 의해 4 Row가 Access되었으며, EMP Table도 Full Table Scan에 의해 14 Row가 Access되었다. Nested loop Join에 의해 3 Row가 Return 되었다.

```
-- Oracle 9.2.x의 Plan에
select /*+ ORDERED USE_NL(d e) */ *
from dept d, emp e
where e.deptno = d.deptno and d.deptno = 10

Rows     Execution Plan
-------  -----------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      3   NESTED LOOPS
      1    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'DEPT'
      3    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMP'
```

DEPT Table에서 Full Table Scan에 의해 1 Row가 Return되었으며, EMP Table도 Full Table Scan에 의해 3 Row가 Return되었다. Nested loop Join에 의해 3 Row가 Return 되었다.

⑤ SQL문장 Wait 정보

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
```

```
 SQL*Net message to client                          1        0.00          0.00
 SQL*Net message from client                        1        1.57          1.57
```

- 이 부분은 10046 Event의 8 또는 12 Level에 의해 생성되며, SQL문장을 실행하는데 발생되었던 Wait정보의 Summary값을 나타낸다. 사용자가 느끼는 실행시간은 SQL문장의 실행시산 과 Wait시간의 합이므로 이들 Wait시간의 주의 깊은 관찰도 필요하다.
- "Times Waited" : Wait Event가 발생했던 회수
- "Max. Wait" : 최대로 Wait했던 시간. (단위 sec)
- "Total Waited" : 전체 Wait했던 시간. (단위 sec)

Dynamic Sampling

✓ 더 낳은 Plan을 경정하기 위한 목적으로 더 정확한 Selectivity & Cardinality를 구하기 위한 방법.

✓ 추가적인 Recursive SQL 발생. 전체 Query실행 시간 대비 적은 Sampling 시간일 경우 사용

✓ 더 정확한 single-table predicate selectivities를 확인하기 위해서 10053 trace와 병행해서 사용

✓ 통계정보가 없거나 너무 오래된 경우 table cardinality를 예측해 볼 경우

✓ Table Level로 지정하지 않고 SQL문장 단위로 지정

✓ How Dynamic Sampling Works

? OPTIMIZER_DYNAMIC_SAMPLING= 0 ~ 10(init.ora), DYNAMIC_SAMPLING(0 ~ 10) Hint

✓ When to Use Dynamic Sampling

? A better plan can be found using dynamic sampling.

? The sampling time is a small fraction of total execution time for the query.

? The query will be executed many times.

✓ How to Use Dynamic Sampling to Improve Performance

? OPTIMIZER_DYNAMIC_SAMPLING = 0 : dynamic sampling disable. (9.0.x default)

? OPTIMIZER_DYNAMIC_SAMPLING = 1 (9i R2 default) 다음의 조건이 모두 만족할 경우 Sampling

✓ Query에 1개 또는 그 이상의 Table이 왔을 경우

✓ 일부 Table이 Index가 없고 통계정보가 없을 경우

✓ 통계정보가 없는 Table이 상대적으로 고 비용의 Table일 것으로 Optimizer가 판단한 경우

? OPTIMIZER_DYNAMIC_SAMPLING >1 (~ 10): more aggressive application of dynamic sampling (analyzed or unanalyzed) & Sampling할 I/O 의 level을 결정

```
>>> DYNAMIC_SAMPLING Hint 의 10053 TRACE
QUERY
select /*+ dynamic_sampling(7) */ deptno from emp where sal *5/8>300
…..
…..
*** 2003-05-28 18:06:58.000
** Performing dynamic sampling initial checks. **
** Dynamic sampling initial checks returning TRUE (level = 7).
*** 2003-05-28 18:06:58.000
** Generated dynamic sampling query:
   query text :
SELECT /*+ ALL_ROWS IGNORE_WHERE_CLAUSE */ NVL(SUM(C1),0), NVL(SUM(C2),0)
FROM (SELECT /*+ IGNORE_WHERE_CLAUSE NOPARALLEL("EMP") */ 1 AS C1,
   CASE WHEN "EMP"."SAL"*5/8>300 THEN 1 ELSE 0 END AS C2
   FROM "EMP" "EMP") SAMPLESUB
*** 2003-05-28 18:06:58.000
** Executed dynamic sampling query:
   level : 7
   sample pct. : 100.000000
   actual sample size : 14
   filtered sample card. : 14
   orig. card. : 14
   block cnt. : 1
   max. sample block cnt. : 256
   sample block cnt. : 1        <<<<<<<< _OPTIMIZER_DYN_SMP_BLKS 와
                              OPTIMIZER_DYNAMIC_SAMPLING의 level에 의해 Sampling Block수가 결정됨
   min. sel. est. : 0.0500
** Using dynamic sel. est. : 1.00000000
  TABLE: EMP     ORIG CDN: 14  ROUNDED CDN: 14  CMPTD CDN: 14
  Access path: tsc  Resc:  2  Resp:  2
  BEST_CST: 2.00  PATH: 2  Degree:  1
```

Using System Statistics (>= 9i)

- ✓ System statistics enable the CBO to use CPU and I/O characteristics.
- ✓ System statistics must be gathered on a regular basis; this does not invalidate cached plans.
- ✓ Gathering system statistics equals analyzing system activity for a specified period of time.
- ✓ import_system_stats으로 업무 유형별 dictionary에 반영
- ✓ Procedures of the dbms_stats package used to collect system statistics:

  gather_system_stats,set_system_stats,get_system_stats

- ✓ Automatic gathering

Collect statistics for OLTP:

```
    SQL> EXECUTE dbms_stats.gather_system_stats -
      2  (interval => 120, stattab => 'mystats', statid => 'OLTP');
```

Collect statistics for OLAP:

```
    SQL> EXECUTE dbms_stats.gather_system_stats -
      2  (interval => 120, stattab => 'mystats', statid => 'OLAP');
```

- ✓ Manual Gathering (start / stop)

```
SQL> EXECUTE dbms_stats.gather_system_stats(gathering_mode => 'START');

SQL> EXECUTE dbms_stats.gather_system_stats (gathering_mode => 'STOP');
```

```
SQL> EXECUTE dbms_stats.gather_system_stats -
>  (gathering_mode => 'START');
PL/SQL procedure successfully completed.
SQL>
SQL> select * from aux_stats$ ;

SNAME                PNAME             PVAL1 PVAL2
-----------------  ----------- ---------- -------------------
SYSSTATS_INFO      STATUS                  MANUALGATHERING
SYSSTATS_INFO      DSTART                  05-29-2003 17:08
SYSSTATS_INFO      DSTOP                   05-29-2003 17:08
SYSSTATS_INFO      FLAGS                 1
SYSSTATS_TEMP      SBLKRDS            1044
SYSSTATS_TEMP      SBLKRDTIM          9000
SYSSTATS_TEMP      MBLKRDS             205
SYSSTATS_TEMP      MBLKRDTIM          2740
SYSSTATS_TEMP      CPUCYCLES        285852
SYSSTATS_TEMP      CPUTIM          2095618
SYSSTATS_TEMP      JOB                   0
SYSSTATS_TEMP      MBRTOTAL           3067

12 rows selected.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------
--------------------------------------------------------------------
| Id  | Operation            |  Name        | Rows  | Bytes | Cost  |
--------------------------------------------------------------------
|   0 | SELECT STATEMENT     |              | 24591 |   768K|    43 |
|*  1 |  TABLE ACCESS FULL   | TESTEMP10    | 24591 |   768K|    43 |
--------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("TESTEMP10"."DEPTNO"=10)
Note: cpu costing is off         <<<<<<<<<< System Stat을 STOP하기 전까지는
                                                     사용안함

14 rows selected.


SQL> EXECUTE dbms_stats.gather_system_stats -
>   (gathering_mode => 'STOP');
PL/SQL procedure successfully completed.
SQL> explain plan for
  2  select * from testemp10 where deptno = 10;
Explained.
```

```
SQL>  select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
--------------------------------------------------------------------------
| Id  | Operation             |  Name        | Rows   | Bytes | Cost (%CPU)|
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |              | 24591 |   768K|     52  (18)|
|*  1 |  TABLE ACCESS FULL    | TESTEMP10    | 24591 |   768K|     52  (18)|
--------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("TESTEMP10"."DEPTNO"=10)
```

PRO*C Precompile Option권장 안

1. **User의 입력조건에 따라 SQL문장이 만들어 지는 Dynamic SQL이라도 Bind변수를 사용한 공유형태의 SQL문장으로 작성한다. 특히 사용빈도가 아주 높은 곳에는 반드시 적용한다. 단. 변경이 없는 상수(Literal)는 문제가 없다. (Application)**

2. **Bind변수를 사용하지 않는 곳에는 Pro*C Compile Option중 RELEASE_CURSOR=YES, HOLD_CURSOR=NO 의 Option으로 Compile한다. RELEASE_CURSOR=NO로 Compile할 경우는 모든 부분이 Bind변수를 사용하여 공유화 할 경우만 사용한다. (Application)**

3. **PRO*C의 경우 Oracle 8i부터 제공되는 PREFETCH를 Compile Option을 100정도로 사용한다. Array Fetch를 사용한 기존의 Program은 그대로 사용한다. ODBC,JDBC,OLEDB,OO4O와 같은 다른 DB접속 방식도 PREFETCH기능을 제공하므로 이 기능을 충분히 활용한다. (Application)**

PRO*C 개발자의 주의 사항

- ✓ MAXOPENCURSORS :maxopencursors가 Application에서 실행되는 SQL을 수용할 정도로 충분하고 OPEN_CURSORS(init.ora)이내라면 “hold_cursor=yes / release_cursor=no ”와 “hold_cursor=no / release_cursor=no ”의 차이는 없음.

- ✓ hold_cursor=yes,release_cursor=no : SQL문장이 대부분 Literal SQL(비공유 SQL)일 경우 Cursor Cache 의 크기가 무한정 늘어나서 max open cursor문제 주의.

- ✓ release_cursor=yes : CPU사용률 증가 및 성능저하가 많이 발생하기 때문에 가능하면 사용하지 않도록 한다. 즉 Literal SQL을 수정하고 공유화 한 후에 release_cursor=no를 사용한다.

- ✓ session_cached_cursors(init.ora) : hold_cursor 와 유사한 역할을 하며, Bind변수를 사용한 경우만 적용한다.

- ✓ Dynamic SQL (Method #1 ~ #4) :compile option 과 상관없이 무조건 parse call 이 발생. 즉 Statistics SQL처럼 고정된 형태가 아니라 매번 SQL문장이 변경될 수 있으므로 Parse Call은 발생한다. 그러나 Dynamic SQL도 Hard Parsing만큼은 일어나지 않도록 하기 위해 Bind변수를 사용해야 한다.

Precompile Option

Release Cursor (default : NO)

✓ 정의 : SQL문장이 실행되어질 때 연관된 cursor cache와 private SQL area간의 link를 control하는 Option.

✓ If release_cursor=Yes Then : SQL 문장 실행 후 cursor가 close되고 link가 remove되며 메모리가 free되어진다.

✓ If release_cursor=No and hold_cursor=Yes Then : link가 유지되고 precompliler는 open cursor수가 MAXOPENCURSORS를 넘지 않는 한 link를 재사용하지 않는다.

✓ 일반적으로 빈번하게 수행되는 OLTP application에서는 NO로 사용하여 처리 성능을 높힐 수 있다. 단 반드시 Bind변수를 사용한 경우에 가능하다. (단 전제 조건으로 반드시 Bind변수를 사용한 곳에 적용을 한다. Literal을 사용한 곳은 YES로 사용한다.)

Hold cursor (default : NO)

✓ 정의 : SQL문장이 실행되어질 때 연관된 cursor와 cursor cache entry간의 link 상태를 control하는 Option. (cursor cache entry는 문장을 processing할 때 필요한 정보를 저장하는 곳이다).

✓ If hold_cursor=no Then : SQL문장을 실행한 후 cursor가 close되고 precompiler는 link를 reusable상태로 mark, 그리고는 다른 SQL문장을 위해 그 link를 바로 사용할 수 있고 Private SQL area에 할당된 메모리를 free시킨다.

✓ If hold_cursor=Yes and release_cursor=No Then : link가 유지되고 precompliler는 다른 SQL을 위해 그 link를 다시 사용하지 않는다. reparsing할 필요가 없고 private sql area에 메모리를 할당할 필요가 없어 performance가 좋아진다.

✓ Release_cursor=Yes 가 Hold_cursor=Yes에 우선하고 Hold_cursor=NO는 release_cursor=NO에 우선한다.

✓ Release cursor option과 마찬가지로 빈번하게 수행되는 OLTP,BATCH application에서는 YES로 사용하여 처리 성능을 높힐 수 있다. 단 해당 SQL을 재사용이 자주 되는 경우만 지정한다. (단 전제 조건으로 반드시 Bind변수를 사용한 경우만 YES로 적용을 한다. Literal을 사용한 곳은 NO로 사용한다.)

MaxOpenCursors (Deafult : 10)

✓ Precompiler가 cache하려는 동시에 open되는 cursor의 수를 지정하는 option이다.

✓ Maxopencusors는 SQLLIB cursor cache의 initial size를 지정한다. Free cache 엔트리가 존재하지 않을 때 새로운 cursor가 필요하다면 entry를 reuse할 것이다. 그러나 reuse하는 것은 hold_cursor, release_cursor 파라미터와 관련한 cursor cache entry의 상태에 따라 불가능할 수도 있어 reuse가 불가할 시에는 새로운 cursor cache entry을 할당한다.

✓ 필요하다면 open_cursors에 지정된 limit에 도달하거나 메모리를 다 사용할 때까지 cursor cache entry를 추가할 것이다.

✓ Maxopencursors 파라미터는 open_cursors 수보다 작아야 한다.

- ✓ 프로그램에서 필요한 동시 open cursor 수가 증가함에 따라 maxopencursors를 재지정하고자 할 것이지만 대부분의 프로그램에서 10정도를 지정하는 것이 적정하다.

- ✓ 특별히 application에서 동시에 open하는 커서가 많지 않으면 default로 사용한다. HOLD_CURSOR=YES를 지정하고 Open된 Cursor가 많을 경우는 값을 Default이상으로 지정한다.

Unsafe_null (Default : NO)

- ✓ Indicator 변수를 사용하지 않고 null을 fetch할 때 발생하는 ora-1405 에러를 유발시키지 않도록 하는 option이다. 1405 에러가 유발되지 않도록 하기위해서는 unsafe_null을 yes로 설정하여 application을 compile한다.

- ✓ MODE가 ORACLE이거나 DBMS=V7,V8 혹은 V6_CHAR인 경우에 한해서 적용된다.

- ✓ 그러나 unsafe_null option은 embedded PL / SQL block내에서는 적용되지 않아 PL / SQL block을 사용할 때는 반드시 indicator 변수를 사용하여야만 null fetch 시 발생하는 1405 에러를 피할 수 있다.

- ✓ Unsafe_null option은 YES로 설정하여 사용하고 Null value 에러 유발 방지를 위해 사용했던 NVL 함수는 생략하도록 한다. Application의 작성형태에 맞게 고려한다.

Prefetch (Default : 1)

- ✓ Server와의 Roundtrip수를 줄이고 Memory의 효과적인 사용측면에 효과적이다. Server와의 Roundtrip을 줄임으로서 성능의 획기적인 향상을 가져온다.

- ✓ Array Fetch 기능을 내부적으로 App에서 하지 않더라도 내부적으로 Pre Fetch해서 App뿐만 아니라 DBMS에 성능향상을 가져온다.

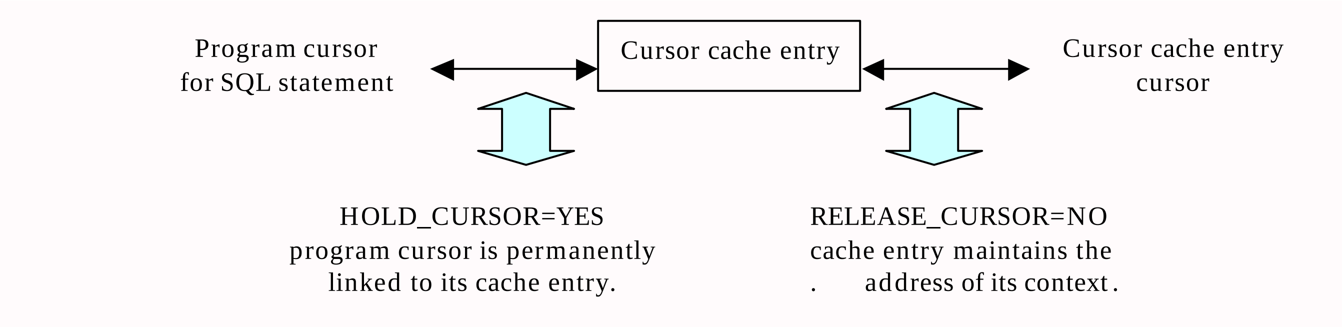
- ✓ OLTP에서는 100정도로 설정하며, Fetch해오는 Row수가 적다면(주로 1건 단위 Fetch)일 경우는 성능에 별 효과는 없다.

- ✓ 야간 Batch등에 많은 Row수를 처리할 경우는 1000 ~ 5000정도로 큰 값을 지정하여 사용한다.

- ✓ ODBC Driver, JDBC, Pro*C등에 사용될 수 있다.

- ✓ PREFETCH=100 정도로 OLTP의 App에 설정하여 사용한다.

RELEASE_CURSOR의 Option에 따른 성능차이


Program cursor
for SQL statement
Cursor cache entry
Cursor cache entry
cursor
HOLD_CURSOR=YES
program cursor is permanently
linked to its cache entry.
RELEASE_CURSOR=NO
cache entry maintains the
. address of its context.

OLTP의 Bind변수를 사용한 Application에 대해서 PRO*C 의 Compile Option을

HOLD_CURSOR=YES, RELEASE_CURSOR=NO,PREFETCH=100 (test를 통한 적정치) 사용

여러 Row를 Return할 경우 Cursor를 선언하여 이용하게 된다. 또한 SQL문장이 Bind값만 바뀌어 처리를 하게 되며(성능을 위해서 OLTP에서는 필수적으로 Bind변수 사용. 즉 Literal을 사용한 Dynamic SQL자제) 이 경우 RELEASE_CURSOR의 Option NO,YES에 따라 SQL실행 성능뿐 아니라 전체적으로 Oracle System에 대한 Parsing Request가 줄어들어 안정적인 시스템을 유지하기에 필수적인 사항이다. RELEASE_CURSOR의 Option에 따른 성능 TEST를 한 Instance에서 TEST한 결과이다. 결과적으로 TEST시나리오의 경우는 3.5배의 효과가 있었다. 그러나 실행되는 SQL문장의 규모가 크다면 더욱 더 큰 결과를 가져올 것이다.

Sample PRO*C Pgm (sample1.pc)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>

#define     UNAME_LEN      20
#define     PWD_LEN        40

VARCHAR     username[UNAME_LEN];  /* VARCHAR is an Oracle-supplied struct */
varchar     password[PWD_LEN];    /* varchar can be in lower case also. */

VARCHAR   emp_name[UNAME_LEN];
long      salary;
long      salary1;



void sql_error(msg)
    char *msg;
{
    char err_msg[128];
    size_t buf_len, msg_len;

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s\n", msg);
    buf_len = sizeof (err_msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);
    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_FAILURE);
}



void main()
{
    strncpy((char *) username.arr, "SCOTT", UNAME_LEN);
    username.len = (unsigned short) strlen((char *) username.arr);
    strncpy((char *) password.arr, "TIGER", PWD_LEN);
    password.len = (unsigned short) strlen((char *) password.arr);

    EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error--\n");

    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    EXEC SQL ALTER SESSION SET SQL_TRACE=TRUE;
    EXEC SQL ALTER SESSION SET TIMED_STATISTICS=TRUE;
    /*=================================================================*/

    EXEC SQL DECLARE emp_cursor CURSOR FOR
         SELECT ename, sal from emp where sal > :salary and
         sal <= :salary + 1000;

    salary = 0;
    while (salary < 5000)
    {
         EXEC SQL OPEN emp_cursor;
         while (sqlca.sqlcode != 0 )
         {
              EXEC SQL FETCH emp_cursor INTO :emp_name, :salary1;
         }
         salary += 10;
         EXEC SQL CLOSE emp_cursor; /* <== MODE=ANSI 에서는 Loop내에 둬야 함 */
    }
    /*EXEC SQL CLOSE emp_cursor; */


    /*=================================================================*/

    exit(EXIT_SUCCESS);
}
```

RELEASE_CURSOR=NO

SQL Trace를 확인해 보면 처음에 한번만 DBMS에게 Parse Request를 하므로 Parsing Overhead가 없다. 그러나 Bind변수를 사용하지 않은 경우라면 Cursor라면 RELEASE_CURSOR=NO를 피해야 한다. OLTP에서와 또한 집중적으로 변경이 이루어지는 Batch Job에서 loop성 SQL문장에 대해서는 Bind변수를 사용하고 RELEASE_CURSOR=NO를 지정하여야 한다.

```
=====================
PARSING IN CURSOR #1 len=74 dep=0 uid=190 oct=3 lid=190 tim=3842865443 hv=842476701
ad='3567d670'
select ename ,sal  from emp where (sal>:b0 and sal<=(:b0+1000))
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
```

```
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=1,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842865443
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842865443




select ename ,sal
from
 emp where (sal>:b0 and sal<=(:b0+1000))


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute    500      0.03       0.00          0          0          0           0
Fetch      500      0.01       0.04          0        500       2000         400
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1001      0.04       0.04          0        500       2000         400
```

RELEASE_CURSOR=YES

SQL Trace를 확인해 보면 매번 DBMS에게 Parse Request를 하므로 Parsing Overhead(Roundtrip증가, library cache latch Contention, CPU증가)가 발생한다. 그러므로 아래와 같은 결과가 나타나게 된다. Bind변수를 사용한 경우에 반복적으로 실행되는 Cursor라면 RELEASE_CURSOR=YES를 피해야 한다.

```
=====================
PARSING IN CURSOR #1 len=74 dep=0 uid=190 oct=3 lid=190 tim=3842884939 hv=842476701
ad='3567d670'
select ename ,sal  from emp where (sal>:b0 and sal<=(:b0+1000))
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842884939
STAT #1 id=1 cnt=1 pid=0 pos=0 obj=16369 op='TABLE ACCESS FULL EMP '
=====================
PARSING IN CURSOR #1 len=74 dep=0 uid=190 oct=3 lid=190 tim=3842884939 hv=842476701
ad='3567d670'
select ename ,sal  from emp where (sal>:b0 and sal<=(:b0+1000))
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842884939
STAT #1 id=1 cnt=1 pid=0 pos=0 obj=16369 op='TABLE ACCESS FULL EMP '
=====================
PARSING IN CURSOR #1 len=74 dep=0 uid=190 oct=3 lid=190 tim=3842884939 hv=842476701
ad='3567d670'
select ename ,sal  from emp where (sal>:b0 and sal<=(:b0+1000))
END OF STMT
PARSE #1:c=1,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842884939
STAT #1 id=1 cnt=1 pid=0 pos=0 obj=16369 op='TABLE ACCESS FULL EMP '
=====================
PARSING IN CURSOR #1 len=74 dep=0 uid=190 oct=3 lid=190 tim=3842884939 hv=842476701
ad='3567d670'
select ename ,sal  from emp where (sal>:b0 and sal<=(:b0+1000))
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=3842884939
```

```
FETCH #1:c=0,e=0,p=0,cr=1,cu=4,mis=0,r=1,dep=0,og=4,tim=3842884939


select ename ,sal
from
 emp where (sal>:b0 and sal<=(:b0+1000))


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse      500       0.10        0.03          0          0          0          0
Execute    500      0.02       0.04          0          0          0           0
Fetch      500      0.02       0.02          0        500       2000         400
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1500      0.14       0.09          0        500       2000         400
```

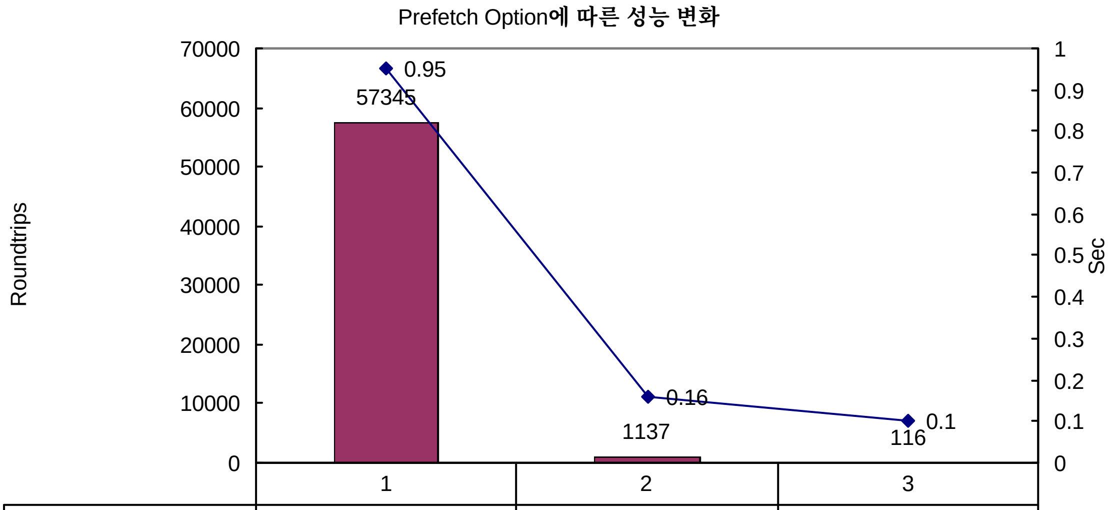
PREFETCH의 효율

- ✓ Prefetch란 PRO*C에서 Oracle 8i 이상의 version에서 제공하며 Server와의 Rountrip수를 줄이고 Memory의 효과적인 사용측면에 효과적이다. Server와의 Roundtrip을 줄임으로서 성능의 획기적인 향상을 가져온다. 이러한 기능은 ODBC,OLE DB, OO4O,JDBC Driver에서도 제공된다.
- ✓ 지정하지 않으면 Default가 1이며, 9i에서는 실제 Fetch단위가 PREFETCH + 1로 가져온다.
- ✓ Array Fetch 기능을 내부적으로 App에서 하지 않더라도 내부적으로 Pre Fetch해서 App뿐만 아니라 DBMS에 성능향상을 가져온다.
- ✓ OLTP에서는 100정도로 설정하며, Fetch해오는 Row수가 적다면(주로 1건 단위 Fetch)일 경우는 성능에 별 효과는 없다.
- ✓ 야간 Batch등에 많은 Row수를 처리할 경우는 1000 ~ 5000정도로 큰 값을 지정하여 사용한다.
- ✓ ODBC Driver, JDBC, Pro*C등에 사용될 수 있다.
- ✓ PREFETCH=100 정도로 OLTP의 App에 설정하여 사용한다.

Prefetch의 변화에 따른 성능 비교표

- ✓ 총 Row수 : 114688
- ✓ 실행 SQL : "select ename from bigemp"
- ✓ 대상 DBMS : Oracle 9i, HANFIS2 DB

	PREFETCH	ROUNDTRIP Count	Elapse Time
1	1	57345	0.95
2	100	1137	0.16
3	1000	116	0.1



	1	2	3
▬ ROUNDTRIP Count	57345	1137	116
◆ Elapse Time	0.95	0.16	0.1

Prefetch=1

```
select ename
from
 bigemp

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch    57345      0.79       0.95          0      57716          0      114688
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    57347      0.80       0.95          0      57716          0      114688

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 20

Rows     Row Source Operation
-------  ---------------------------------------------------
 114688  TABLE ACCESS FULL BIGEMP


=====================
PARSING IN CURSOR #1 len=37 dep=0 uid=20 oct=3 lid=20 tim=1037762838818267 hv=2376501895
ad='9185be8'
select ename  from bigemp
END OF STMT
PARSE #1:c=976,e=976,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=4,tim=1037762838818267
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1037762838818267
FETCH #1:c=976,e=977,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=4,tim=1037762838819244
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838829010
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838835846
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838842682
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838850494
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838857330
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838864166
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838871979
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838878815
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838885651
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=2,dep=0,og=4,tim=1037762838893463
………………….
```

Prefetch=100

```
select ename
from
 bigemp

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch     1137      0.12       0.16          0       1808          0      114688
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     1139      0.12       0.16          0       1808          0      114688

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 20

Rows     Row Source Operation
-------  ---------------------------------------------------
 114688  TABLE ACCESS FULL BIGEMP



=====================
PARSING IN CURSOR #1 len=37 dep=0 uid=20 oct=3 lid=20 tim=1037763404850719 hv=2376501895
ad='9185be8'
select ename  from bigemp
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1037763404850719
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1037763404850719
FETCH #1:c=976,e=976,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=4,tim=1037763404851695
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=101,dep=0,og=4,tim=1037763404851695
FETCH #1:c=976,e=977,p=0,cr=2,cu=0,mis=0,r=101,dep=0,og=4,tim=1037763404861461
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=101,dep=0,og=4,tim=1037763404879039
FETCH #1:c=0,e=0,p=0,cr=2,cu=0,mis=0,r=101,dep=0,og=4,tim=1037763404880992
FETCH #1:c=0,e=0,p=0,cr=1,cu=0,mis=0,r=101,dep=0,og=4,tim=1037763404882945
FETCH #1:c=0,e=0,p=0,cr=2,cu=0,mis=0,r=101,dep=0,og=4,tim=1037763404884898
………………….
```

Prefetch=1000

```
select ename
from
 bigemp

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch      116      0.10       0.10          0        790          0      114688
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      118      0.10       0.10          0        790          0      114688

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 20
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
 114688  TABLE ACCESS FULL BIGEMP
```

```
====================
PARSING IN CURSOR #1 len=37 dep=0 uid=20 oct=3 lid=20 tim=1037763577016020 hv=2376501895
ad='9185be8'
select ename  from bigemp
END OF STMT
PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1037763577016020
EXEC #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1037763577016020
FETCH #1:c=0,e=0,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=4,tim=1037763577016020
FETCH #1:c=1952,e=1953,p=0,cr=6,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577018950
FETCH #1:c=976,e=977,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577030669
FETCH #1:c=976,e=976,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577042387
FETCH #1:c=976,e=976,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577054106
FETCH #1:c=0,e=0,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577064848
FETCH #1:c=976,e=977,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577079497
FETCH #1:c=976,e=976,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577101958
FETCH #1:c=976,e=976,p=0,cr=6,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577126372
FETCH #1:c=1952,e=977,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577150787
FETCH #1:c=976,e=976,p=0,cr=7,cu=0,mis=0,r=1001,dep=0,og=4,tim=1037763577165435
…………………….
```

Scrollable Cursors (Oracle 9i R2)

- ✓ DECLARE SCROLL CURSOR:

  DECLARE <cursor name> SCROLL CURSOR

- ✓ OPEN: OPEN statement in the same way

- ✓ FETCH: fetch rows up or down, first or last row directly, or fetch any single row in a random manner.

  - ■ FETCH FIRST : Fetches the first row from the result set.
  - ■ FETCH PRIOR : Fetches the row prior to the current row.
  - ■ FETCH NEXT : Fetches the next row from the current position. This is same as the non-scrollable cursor FETCH.
  - ■ FETCH LAST : Fetches the last row from the result set.
  - ■ FETCH CURRENT : Fetches the current row.
  - ■ FETCH RELATIVE n : Fetches the nth row relative to the current row, where n is the offset.
  - ■ FETCH ABSOLUTE n : Fetches the nth row, where n is the offset from the start of the result set.

```
<< Sample SQL >>
```

```
SQL>  SELECT empno, ename, sal FROM emp;

     EMPNO ENAME                       SAL
---------- -------------------- ----------
      7369 SMITH                       800
      7499 ALLEN                      1600
      7521 WARD                       1250
      7566 JONES                      2975
      7654 MARTIN                     1250
      7698 BLAKE                      2850
      7782 CLARK                      2450
      7788 SCOTT                      3000
      7839 KING                       5000
      7844 TURNER                     1500
      7876 ADAMS                      1100
      7900 JAMES                       950
      7902 FORD                       3000
      7934 MILLER                     1300

14 rows selected.
```

```
/*
 *  A Sample program to demonstrate the use of scrollable
 *  cursors with host arrays.
 *
 *  This program uses the scott/tiger schema.Make sure
 *  that this schema exists before executing this program
 */


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

#define ARRAY_LENGTH   4

/* user and passwd */
char *username = "scott";
char *password = "tiger";

/* Declare a host structure tag. */
struct emp_rec_array
{
    int    emp_number;
    char   emp_name[20];
    float  salary;
} emp_rec[ARRAY_LENGTH];

void print_rows()
{
    int i;

    for(i=0; i<ARRAY_LENGTH; i++)
        printf("%d    %s  %8.2f\n", emp_rec[i].emp_number,
             emp_rec[i].emp_name, emp_rec[i].salary);

}

void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    printf("\n%s", msg);
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_FAILURE);
}
```

```
void main()
{
    int noOfRows; /* Number of rows in the result set */

    /* Error handle */
    EXEC SQL WHENEVER SQLERROR DO sql_error("Connect error:");

    /* Connect to the data base */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    /* Error handle */
    EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error:");

    /* declare the cursor in scrollable mode */
    EXEC SQL DECLARE c1 SCROLL CURSOR FOR
        SELECT empno, ename, sal FROM emp;

    EXEC SQL OPEN c1;

    EXEC SQL WHENEVER SQLERROR DO sql_error("Fetch Error:");

    /* This is a dummy fetch to find out the number of rows
       in the result set */
    EXEC SQL FETCH LAST c1 INTO :emp_rec;

    /* The number of rows in the result set is given by
       the value of sqlca.sqlerrd[2] */

    noOfRows = sqlca. sqlerrd[2];
    printf("Total number of rows in the result set %d:\n",
             noOfRows);

    /* Fetch the first ARRAY_LENGTH number of rows */
    EXEC SQL FETCH FIRST c1 INTO :emp_rec;
    printf("******************** DEFAULT : \n");
    print_rows();

    /* Fetch the next set of ARRAY_LENGTH rows */
    EXEC SQL FETCH NEXT c1 INTO :emp_rec;
    printf("******************** NEXT  : \n");
    print_rows();

    /* Fetch a set of ARRAY_LENGTH rows from the 3rd row onwards */
    EXEC SQL FETCH ABSOLUTE 3 c1 INTO :emp_rec;
    printf("******************** ABSOLUTE 3 : \n");
    print_rows();

    /* Fetch the current ARRAY_LENGTH set of rows */
    EXEC SQL FETCH CURRENT c1 INTO :emp_rec;
    printf("******************** CURRENT : \n");
    print_rows();

    /* Fetch a set of ARRAY_LENGTH rows from the 2nd offset
       from the current cursor position */
    EXEC SQL FETCH RELATIVE 2 c1 INTO :emp_rec;
    printf("******************** RELATIVE 2 : \n");
    print_rows();

    /* Again Fetch the first ARRAY_LENGTH number of rows */
    EXEC SQL FETCH ABSOLUTE 0 c1 INTO :emp_rec;
    printf("******************** ABSOLUTE 0 : \n");
    print_rows();

    /* close the cursor */
    EXEC SQL CLOSE c1;

/* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(EXIT_SUCCESS);
}
```

```
ORA9iR2L@oracle:/home/oracle/oracle9/precomp/demo/edu> scrollable2
Total number of rows in the result set 14:
******************** DEFAULT :
7369    SMITH                  800.00
7499    ALLEN                 1600.00
7521    WARD                  1250.00
7566    JONES                 2975.00
******************** NEXT  :
7654    MARTIN                1250.00
7698    BLAKE                 2850.00
7782    CLARK                 2450.00
7788    SCOTT                 3000.00
******************** ABSOLUTE 3 :
7521    WARD                  1250.00
7566    JONES                 2975.00
7654    MARTIN                1250.00
7698    BLAKE                 2850.00
******************** CURRENT :
7698    BLAKE                 2850.00
7782    CLARK                 2450.00
7788    SCOTT                 3000.00
7839    KING                  5000.00
******************** RELATIVE 2 :
7876    ADAMS                 1100.00
7900    JAMES                  950.00
7902    FORD                  3000.00
7934    MILLER                1300.00
******************** ABSOLUTE 0 :
7876    ADAMS                 1100.00
7900    JAMES                  950.00
7902    FORD                  3000.00
7934    MILLER                1300.00
```

Application의 Module명 및 단위 Routine명 표시하기

Oracle은 실행되는 SQL문장을 Shared Pool에 Cache화 하며, Cache화 되는 내용은 SQL문장, 실행횟수, Parsing횟수,Disk IO및 Cache에서 읽어 들인 Block수, Sorting여부,Parsing User정보, Module명,Action명 등을 관리한다.

여기서 Application의 어떤 Module(Tuxedo의 어떤 Service)의 어떤 Action(특정 처리루틴)에서 처리된 SQL문장인가를 확인하기 위해서는 별도의 추가적인 Coding이 필요하다.

"DBMS_APPLICATION_INFO" Package가 그것이며, 오라클의 모든 App(SQL*Plus,OEM,ERP,...)등도 이 Package를 이용하여 해당 SQL 문장이 어느 Module의 어떤 Action에서 들어온 것인지를 운영관리자가 확인할 수 있도록 하였다.

비효율적인 문제의 SQL문장의 찾더라도 정확이 어떤 Module, 어떤 처리루틴에서 들어온 것인지를 찾기 어려우며, 또한 개발자도 유지,보수 단계에 들어가게 되면 담당자도 해당 Routine을 찾기 어려운 경우가 발생한다. 그래서 "DBMS_APPLICATION_INFO" Package를 이용한 Module명과 Action명을 기술하는 Routine을 개발 단계일 때 추가하는 것을 강력히 권고한다.

DBMS_APPLICATION_INFO Package사용의 장점

1. 비효율적인 부분을 찾아서 Tuning및 결과 적용하는데 대한 유지 / 보수 절차가 쉬워진다. 문제된 부분을 찾아서, Source부분을 신속히 찾을 수 있다.

2. 업무별로 시스템의 부하정도를 판단할 수 있다. 즉 SQL문장의 실행 규모, Parsing 규모, SQL처리 유형 등을 판단하고, 이 자료를 근거로 DB를 관리하는데 효과적인 자원관리를 할 수 있다.

3. 보안의 용도로도 사용할 수 있다. 예를 들어 ACTION명에 루틴명과 처리 담당자의 ID등을 기록하면, 추후 Auditing정보로 추적이 가능하다. (TRANSACTION_AUDITING=true, 인 환경에서 Log Miner이용)

DBMS_APPLICATION_INFO Package사용의 단점

1. DBMS_APPLICATION_INFO를 실행하기 위한 App의 처리단계가 추가된다. 그러나 처리단위를 LOGIN시 1번만으로 조절하며, 실행시 Bind변수를 사용한 공유형태로 처리하면, DBMS에 부하가 거의 없다.

2. 개발자들의 코딩을 위한 루틴 추가의 일부 Source의 변경이 필요하다.

DBMS_APPLICATION_INFO Package의 사용 예 및 Cache화 되는 내용

- DBMS_APPLICATION_INFO Package를 사용하기 위한 Sample

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>

#define      UNAME_LEN      20
#define      PWD_LEN        40
#define      MODULE_LEN     65

VARCHAR      username[UNAME_LEN];  /* VARCHAR is an Oracle-supplied struct */
varchar      password[PWD_LEN];    /* varchar can be in lower case also. */

varchar      szModuleName[MODULE_LEN];
varchar      szActionName[MODULE_LEN];

VARCHAR    emp_name[UNAME_LEN];
long       salary;

varchar     szDbmsApplicationSQL[100];
char szSQLDBMSAPP[] ="BEGIN DBMS_APPLICATION_INFO.SET_MODULE(:A,:B); END;";

………….

void main()
{

     username.len = sprintf((char*)username.arr, "SCOTT");
    password.len = sprintf((char*)password.arr, "TIGER");


    EXEC SQL WHENEVER SQLERROR DO sql_error("ORAC

    EXEC SQL CONNECT :username IDENTIFIED BY :pas


    /*============================================

      szDbmsApplicationSQL.len = sprintf((char*)szDbmsApplicationSQL.arr,
szSQLDBMSAPP);

       EXEC SQL PREPARE STMT FROM :szDbmsApplicationSQL;

     /* App Module명 */
      szModuleName.len = sprintf((char*)szModuleName.arr, "sample1.pc");

      /* App 단위 Action명 */
      szActionName.len = sprintf((char*)szActionName.arr, "SELECT EMP TABLE");

       /* DBMS_APPLICATION_INFO 도 bind변수 사용 */
       EXEC SQL EXECUTE STMT USING  :szModuleName,  :szActionName

    /*==========================================================

    EXEC SQL DECLARE emp_cursor CURSOR FOR
         select ename, sal from emp;

    EXEC SQL OPEN emp_cursor;
    while (sqlca.sqlcode != 0 )
```

DBMS_APPLICATION_INFO
실행형태

DBMS_APPLICATION_INFO를 사용하기 위해 추가되는 루틴. 주로 Login다음정도에 두며, Function입구점에 Action명을 지정한다.

Module명

Action 명

```
    {
         EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    }
    EXEC SQL CLOSE emp_cursor;


    exit(EXIT_SUCCESS);
}
```

- Oracle Cache(Shared Pool)의 SQL및 Module,Action정보

SQLTEXT	BUFGETSPEREXEC	EXECUTIONS	PARSE_CALLS	DISK_READS	BUFFER_GETS	ROWS_PROCESSED	MODULE	ACTION
select ename ,sal from emp	27	1	1	0	27	1	**sample1.pc**	**SELECT EMP TABLE**

위의 정보로 해당 SQL이 I/O를 얼마나 일으켰으며, 어떤 Module과 어떤 처리 Routine에서 실행한 SQL인지를 확인할 수 있다.

PRO*C 에서의 Dynamic SQL의 사용에 의한 SQL공유화 방안

OLTP에서는 Bind변수를 사용하여 SQL문장 등이 공유하도록 구성되어야 한다. SQL문장이 공유되지 않을 경우는 다음과 같은 문제를 야기하게 된다.

- ● 매번 같은 기능의 SQL문장을 parsing하게 된다. 그러므로 Parsing의 Overhead가 있게 된다.
- ● SQL문장이 Cache로 부터 수시로 Load / Unload되므로 Memory Fragmentation이 유발되며, 새로운 SQL등을 Memory에 할당하기 위한 Overhead가 발생한다.
- ● 업무의 증가에 비례하여 SQL이 급격히 증가되므로 Memory의 관리의 Overhead로 인한 Latch의 Contention을 유발하게 되며, 추가적인 Memory의 비용이 야기된다.
- ● Bind변수를 사용하지 않으므로 SQL을 재사용할 수 없으며, 그러므로 매번 SQL을 Parsing을 해야 하므로 시스템의 CPU,Memory를 급격히 사용하는 고비용 구조가 되며, 사용자나 업무의 증가에 따른 시스템 Resource의 고갈이 급격히 나타나게 된다. 그러므로 OLTP의 집중적으로 실행되는 업무에서는 반드시 Bind변수를 사용하여야 한다.

Dynamic SQL이란 SQL구문, TABLE명 혹은 입력 Host 변수명 등의 COMPONENTS가 프로그램 수행시점에서 결정되어 Static SQL 보다 유연한 처리구조로서, 주로 사용자의 입력을 받아서 조건에 맞는 SQL문장을 만들어 처리하는 방식이며, 순수하게 입력된 String을 Concatenation해서 SQL문장을 만드는 방식과, SQL문장의 구성은 Bind변수를 지정하고 Using절을 이용해 Parameter로 넘기는 방식이 있다. SQL 문장이 집중적으로 실행되는 업무의 OLTP에서는 두 번째 방식으로 구성되어야 한다.

개요

다양한 업무적 요구에 의해 dynamic SQL을 사용하여야 하는 응용프로그램이 존재한다. Dynamic SQL은 오라클 내부적으로 static SQL과 같은 방법으로 수행되어지며, 다만 dynamic SQL을 구현하는데 있어 host binding변수를 사용하지 않는다면 SQL문장의 re-parsing으로 인한 성능의 저하가 일어날 수 있다.

Re-parsing의 예

```
strcpy((char *)sql.arr,“select * from emp where empno = 2783”);
sql.len = (int)strlen((char *)sql.arr);

EXEC SQL PREPARE STMT FROM :stmt;
EXEC SQL DECLARE CUR CURSOR FOR STMT;
EXEC SQL OPEN CUR;
while (1) {
    EXEC SQL FETCH INTO :emprec;
.
.
.
```

재사용 예

```
strcpy((char *)sql.arr,“select * from emp where empno = :a”);
sql.len = (int)strlen((char *)sql.arr);

EXEC SQL PREPARE STMT FROM :stmt;
EXEC SQL DECLARE CUR CURSOR FOR STMT;
EXEC SQL OPEN CUR USING :host_empno;
while (1) {
    EXEC SQL FETCH INTO :emprec;
.
```

Dynamic SQL의 구현

Dynamic SQL을 구현하는 방법은 아래와 같이 4가지가 존재하며, 각각의 방법마다 제한과 용도가 제시된다. 응용프로그램에서는 2번과 3번 방법이 권장된다.

Method	SQL문장의 종류
1	host변수를 사용하지 않는 non query
2	host변수를 사용하는 non query
3	select-list의 item과 입력 host변수를 갖는 query
4	가변적인 select-list item과 입력 host변수를 갖는 query

Method #1

응용프로그램에서 SQL문장을 ***EXECUTE IMMEDIATE***문을 사용하여 즉시 수행 가능하다.
SQL문장은 query문장(SELECT문)을 포함하지 않으며, 입력 host변수를 위한 placeholder를 가지지 않는다.

```
'DELETE FROM EMP WHERE DEPTNO = 20'

'GRANT SELECT ON EMP TO scott'
```

SQL문장은 매번 수행할 때 마다 parsing이 된다.

Method 2를 이용한 Dynamic SQL 구현 예

```
EXEC SQL EXECUTE IMMEDIATE
```

```
                    "CREATE TABLE dyn1 (col1 VARCHAR2(4))";
```

Method #2

응용프로그램에서 SQL문장을 ***PREPARE***와 ***EXECUTE***명령에 의해 수행한다.

PREPARE 문장:

```
EXEC SQL PREPARE statement_name
               FROM { :host_string | string_literal };
```

statement_name	precomiler에의해 사용되어지는 identifier일뿐이며 프로그램의 host변수가 아니다.
host_string	SQL문을 가리키는 host변수
string_literal	SQL문을 표현한 문자열

EXECUTE 문장:

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

statement_name	PREPARE문에서 정의된 identifier
host_variable_list	:host_variable1[:indicator1][,host_variable2[:indicator2], ...]

SQL문장은 query(SELECT문)가 아니고, DML(UPDATE,INSERT,DELETE) 문장이며, 입력 host변수의 수와 datatype은 precomile이전에 결정되어져야 한다.

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'

'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

실행시 Host변수에 매번 다른 값이 올 수 있으며, SQL문장은 최초 한번만 parsing된다.
DDL문장(CREATE,GRANT,DROP,…)은 PREPARE명령만으로 수행된다.

Method 2를 이용한 Dynamic SQL 구현 예

```
sprintf( (char *) vcSql.arr,
          "UPDATE TB_CCSTBASICINFO%s \
          SET CNTC_TEL_NO = DECODE(CNTC_TEL_NO, :a, :b, CNTC_TEL_NO ), \
             CNTC_FAX_NO = DECODE(CNTC_FAX_NO, :c, :d, CNTC_FAX_NO ) \
          WHERE CUST_ID = :e  \
          AND ( CNTC_TEL_NO = :f  \
          OR CNTC_FAX_NO = :g)",
       szLinkName);
vcSQL.len = (int)strlen((char *)vcSQL.arr);

EXEC SQL PREPARE STMT FROM :vcSQL;

EXEC SQL EXECUTE STMT USING
               :gstPstnInfo.vcOldTelNo, :gstPstnInfo.vcNewTelNo,
                    :gstPstnInfo.vcOldTelNo, :gstPstnInfo.vcNewTelNo,
                    :gstPstnInfo.vcCustId, :gstPstnInfo.vcOldTelNo, :gstPstnInfo.vcOldTelNo;
```

Method #3

응용프로그램에서 query문장을 PREPARE와 DECLARE, OPEN, FETCH, CLOSE cursor 명령 순으로 실행 한다.

```
PREPARE statement_name FROM { :host_string | string_literal };

DECLARE cursor_name CURSOR FOR statement_name;

OPEN cursor_name [USING host_variable_list];

FETCH cursor_name INTO host_variable_list;

CLOSE cursor_name;
```

Select-list item의 수, 입력 host변수의 placeholder의 수와 입력 host변수의 datatype은 precompile이전에 결정되어 있어야 한다.

Method 3를 사용하여 Parallel Degree를 동적으로 지정하는 예

```
sprintf(dynstmt.arr,
   "SELECT /*+parallel (emp,%d)*/ ename FROM emp WHERE deptno = :v1",degree);
dynstmt.len = strlen(dynstmt.arr);

EXEC SQL PREPARE S FROM :dynstmt;
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C USING :deptno;

EXEC SQL WHENEVER NOT FOUND DO break;

/* Loop until the NOT FOUND condition is detected. */
for (;;) {
       EXEC SQL FETCH C INTO :ename;
                    .
}
```

Method #4

응용프로그램에서 SQL문장을 descriptor (SQLDA구조체)를 이용하여 아래의 순서로 수행한다.

```
EXEC SQL PREPARE statement_name
   FROM { :host_string | string_literal };

EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
   INTO bind_descriptor_name;

EXEC SQL OPEN cursor_name
   [USING DESCRIPTOR bind_descriptor_name];

EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
   INTO select_descriptor_name;

EXEC SQL FETCH cursor_name
   USING DESCRIPTOR select_descriptor_name;

EXEC SQL CLOSE cursor_name;
```

Select-list의 item,입력 host변수의 수와 datatype은 응용프로그램 수행시(runtime)에 결정되며, SQL문장에 담길 select-list item들과 host변수들의 개수를 미리 알 수 없을 때 사용한다.

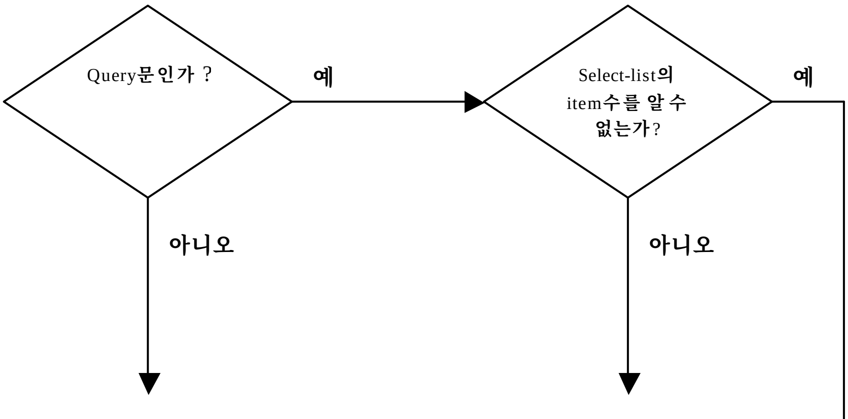
```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'

'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

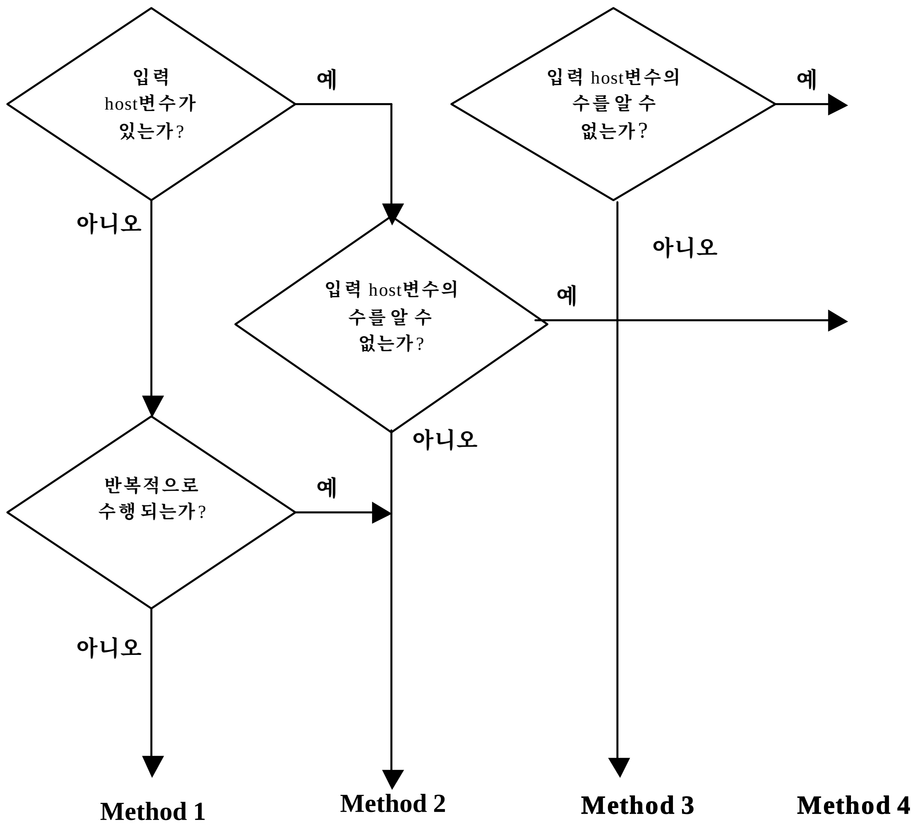
단순한 Method 4 사용 예

```
/*
 * A very simple program that demonstrates how to do
 * array fetches using dynamic SQL Method 4.
 *
 * Make sure to precompile with MODE=ORACLE.
 */

#include <stdio.h>
#include <sqlca.h>
#include <sqlda.h>

#include <sqlcpr.h>
#include <stdlib.h>

#define MAX_SELECT_ITEMS    8
#define FETCH_SIZE          5   /* Fetch in 5-row chunks. */
#define MAX_CHARS          10
#define MAX_NAME_SIZE       8   /* Maximum size of a select-list item name. */

SQLDA   *selda;

/* Data buffer. */
char c_data[MAX_SELECT_ITEMS][FETCH_SIZE][MAX_CHARS];

void print_rows(n)
   int n;
{
   int row, sli;
   for (row = 0; row < n; row++)
   {
      for (sli = 0; sli < selda->N; sli++)
      {
         printf("%.10s ", c_data[sli][row]);
      }
      printf("\n");
   }
}

int array_size = FETCH_SIZE;  /* needs to be a host var for FOR */
char *username = "scott/tiger";
char *stmt = "select ename, empno, sal, hiredate from emp";

/* This is a minimal program, with little error checking,
 * since the SQL statement is hard-coded. If you were to
 * substitute 'comm' for 'sal' in the statement below, the
 * program would fail with a -1405 on Oracle7, as there are
 * no indicator variables.
 */

void sql_error()
{
   char msgbuf[512];
   size_t msgbuf_len, msg_len;

   msgbuf_len = sizeof(msgbuf);
   sqlglm(msgbuf, &msgbuf_len, &msg_len);

   printf ("\n\n%.*s\n", msg_len, msgbuf);

   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(EXIT_FAILURE);
}

void main()
{
   int row_count;
   int sli;     /* select-list item */

   EXEC SQL CONNECT :username;
   if (sqlca.sqlcode == 0)
      printf("Connected.\n");
   else
   {
      printf("Cannot connect as SCOTT.\n");
      exit(EXIT_FAILURE);
   }

   EXEC SQL WHENEVER SQLERROR DO sql_error();
```

```
    selda = sqlald(MAX_SELECT_ITEMS, MAX_NAME_SIZE, 0);

    EXEC SQL PREPARE S FROM :stmt;
    EXEC SQL DECLARE C CURSOR FOR S;
    EXEC SQL OPEN C;
    EXEC SQL DESCRIBE SELECT LIST FOR S INTO selda;

    selda->N = selda->F;   /* Assumed not negative. */
    for (sli = 0; sli < selda->N; sli++)
    {
       /* Set addresses of heads of the arrays in the V element. */
       selda->V[sli] = c_data[sli][0];
       /* Convert everything to varchar on output. */
       selda->T[sli] = 1;
       /* Set the maximum lengths. */
       selda->L[sli] = MAX_CHARS;
    }

    for (row_count = 0; ;)
    {
       /* Do the fetch. The loop breaks on NOT FOUND. */
       EXEC SQL FOR :array_size FETCH C USING DESCRIPTOR selda;

       print_rows(sqlca.sqlerrd[2] - row_count);
       row_count = sqlca.sqlerrd[2];
       if (sqlca.sqlcode == 1403)
          break;
    }

/*    if (sqlca.sqlerrd[2] - row_count > 0)
       print_rows(sqlca.sqlerrd[2] - row_count); */

    printf("\n%d rows retrieved\n", sqlca.sqlerrd[2]);

    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_SUCCESS);
}
```

Dynamic SQL Method의 선택





지침

- 모든 SQL문장과 host변수의 placeholder는 character string으로 처리된다.
- Method 2와 3에서 host변수와 host변수의 placeholder의 수,datatype은 precompile단계에서 결정되어져야 한다.
- Dynamic SQL Method는 응용프로그램의 유연성을 제공하지만 일반적인 embedded SQL의 사용보다는 구현이 어렵다.
- Method 1과 2를 과도하게 반복해서(looping) 사용한다면 실행시 마다 parsing이 일어난다.
- Method 4는 최고의 유연성을 제공하지만 code의 복잡도로 인해 dynamic SQL의 개념을 완전히 이해하여야만 구현이 가능하다. 일반적으로 Method 1,2,3을 완벽하게 구현할 수 있을 때 Method 4의 구현이 가능하다.

JDBC에서의 SQL공유화 방법

4. JDBC Application개발 본에 대해서 BIND변수를 사용하도록 한다.

- JDBC로 개발되는 Application은 DB에 접근하는 SQL은 Bind변수를 사용한 형태로 수정되어야 합니다.

```
pstmt = conn.prepareStatement ("select ename from emptest where empno = " + ii);

rset  = pstmt.executeQuery();
```

```
pstmt = conn.prepareStatement ("select ename from emptest where empno = ?");

pstmt.setInt (1, ii); // Set the Bind Value

rset  = pstmt.executeQuery();
```

2. JDBC로 개발된 경우 PREFETCH=100정도를 사용합니다.

PREFETCH란 Array Fetch(한번에 DB에서 읽어서 Client로 가져오는 단위)기능을 Driver단에서 처리해 주는 기능으로 DB와의 Roundtrip을 줄이고 성능향상을 가져올 수 있는 기능입니다.

default=> 10 이며 OLTP는 100정도 설정하는 것이 효과적입니다. 그렇다고 너무 큰 값을 설정하면 안됩니다.

예) PREFETCH 확인 방법

```
int default_row_prefetch = ((OracleConnection)conn).getDefaultRowPrefetch ();

System.out.println ("The Default RowPrefetch for the connection is: "  + default_row_prefetch);
```

예) PREFETCH설정 방법

```
((OracleStatement)stmt).setRowPrefetch (100);
```

OLTP환경에서의 비공유 SQL(Literal SQL)의 문제점

공유/비공유 SQL의 장단점

	장점	단점	장점주요 사용 환경
공유 SQL	Parsing을 다시 하지 않는다. 사용자의 증가나 SQL실행규모의 증가에도 시스템 Resource가 완만하게 증가한다.	분포도가 편향되어 있는 경우 값에 따른 다른 Plan이 결정되지 못하는 단점	OLTP, 높은 SQL실행규모의 시스템
비공유SQL	Parsing시 Plan을 결정하는데 Bind변수를 사용하는 것보다 정확하다.	Library Cache Contention유발 library cache latch, shared pool latch 경합 발생 Parsing CPU 사용 SQL실행규모가 커지면 급격한 Resource(CPU,Memory)감소 Memory Fragmentation발생 (Shared Pool)	DW,야간 Batch, 소수의 사용자 시스템

공유/비공유 SQL의 실행 TEST결과

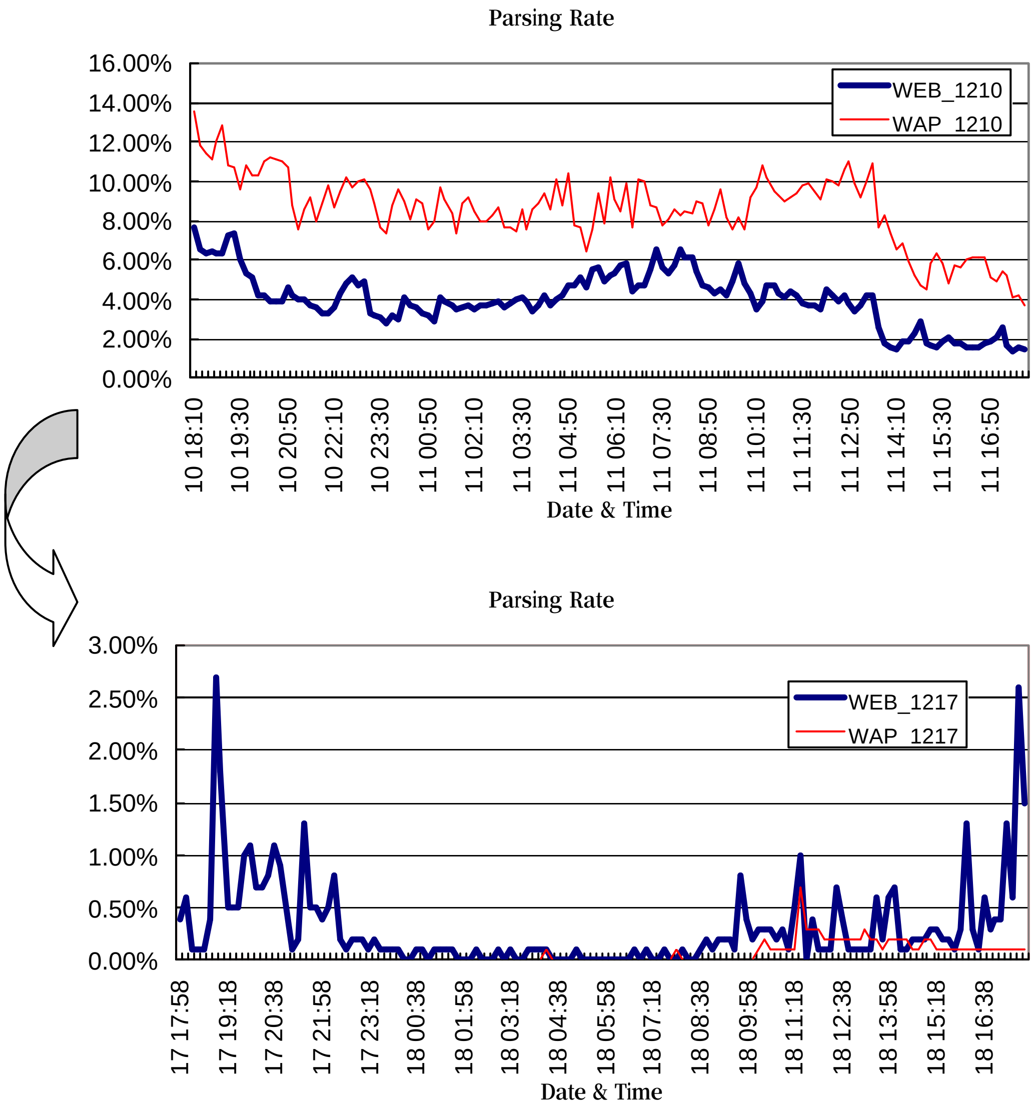
다음은 아래의 SQL문장을 Bind변수를 사용한 공유 SQL과, 상수를 결합한 형태로 SQL문장을 만들어 실행시키는 비공유SQL의 9999회 실행시켜 Oracle의 Shared pool Memory사용현황과 Parsing시 CPU사용시간을 TEST한 상황이다. (단 아래의 내용은 실행 Server별로 차이는 있음)

결론적으로 보면, 사용 Memory와 CPU 사용률이 실행 규모에 비례에 증가되고, 실행된 SQL문장이 기존의 Cache화 되어있는 SQL문장들을 밀어내는 역할을 한다.

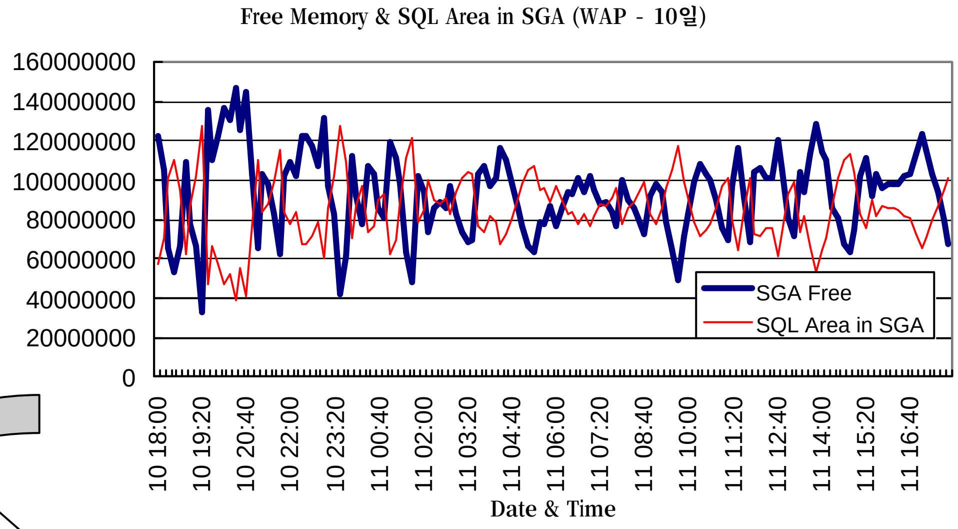
	SQL 유형	Shared Pool의 Memory사용	Parsing 수	Exec 수	Parsing CPU Usage
공유 SQL	select ename from emptest where empno = :1	9807	1	9999	0.01 sec
비공유SQL	select ename from emptest where empno = 1 select ename from emptest where empno = 2 select ename from emptest where empno = 3 …… ……	93,219,148 (92 MB)	9999	9999	14.33 sec

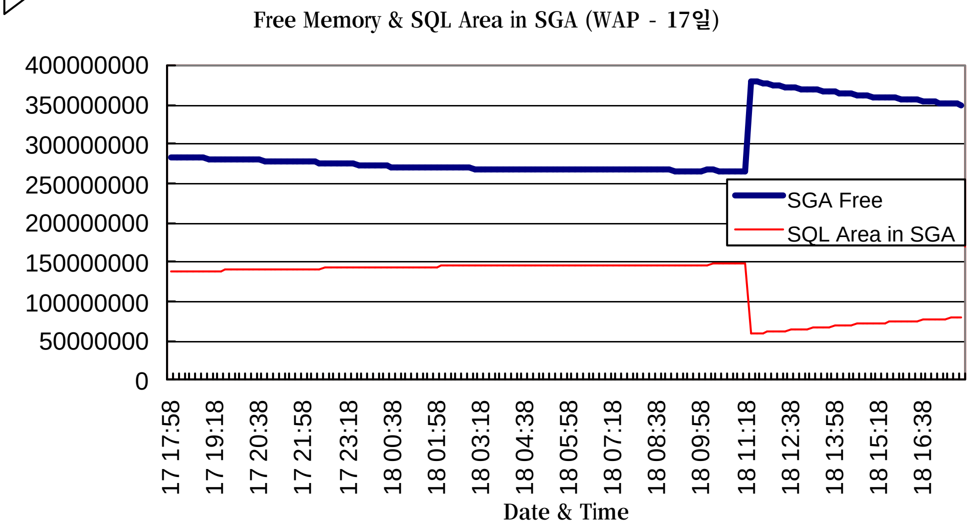
관련 사례

- ◆ Parsing율이 14%정도에서 거의 0%로 Parsing이 발생하지 않았다(WAP – 붉은 Line)

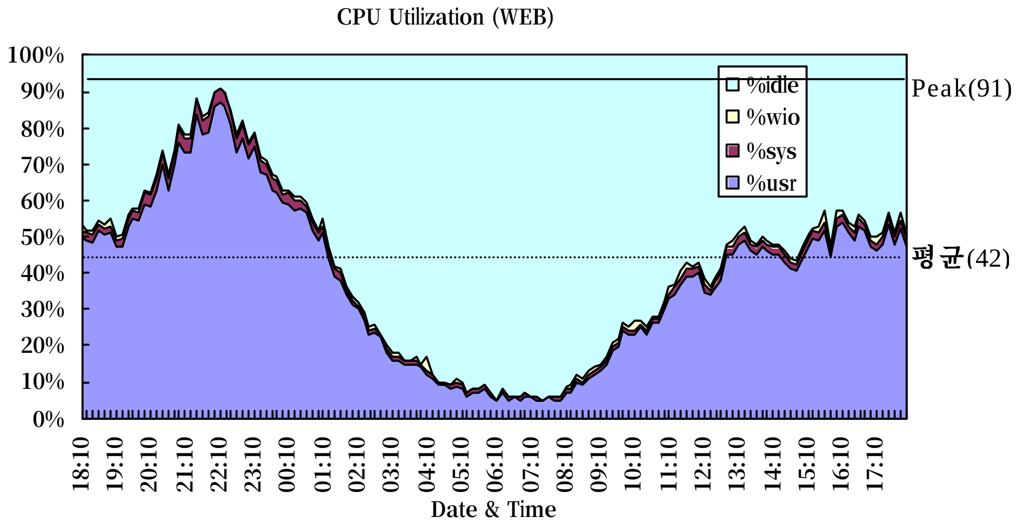


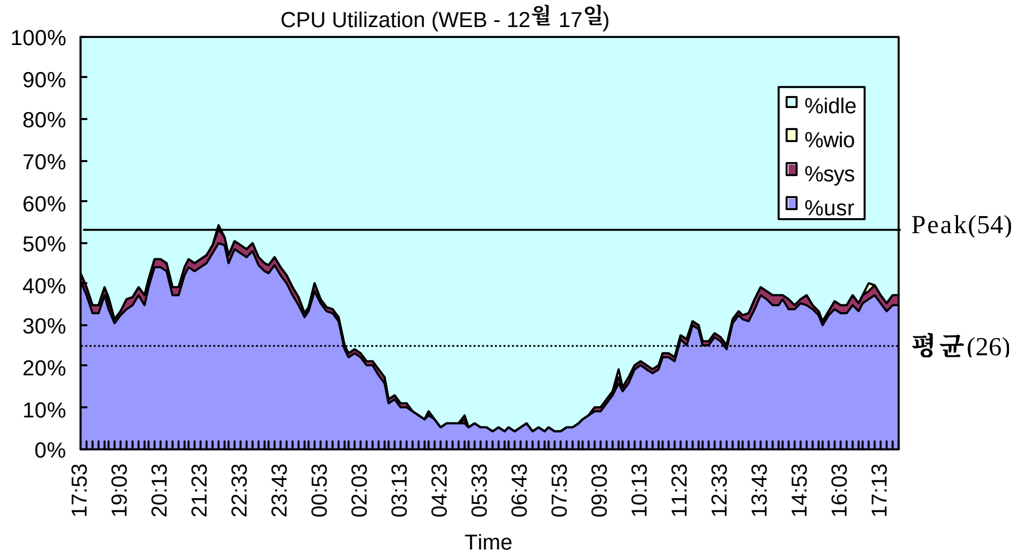
- SGA내에 Cache화 되어 있는 SQL문장이 짧은 시간에 채워지고 밀려나는 형태에서 거의 고정적으로 Cache화 되었으며, Memory의 절약 및 사용측면에서 Memory Fragmentation(Free memory는 있으나 연속된 실제 사용 가능한 연속된 memory가 없는 현상)이 없어졌다.





- CPU사용률이 Peak 91, 평균 42에서 54, 26으로 낮아졌다.





SQL Tuning대상을 찾기 위한 Monitoring 방법

비효율적인 SQL List 보기 SQL Script (get_sqllist.sql)

- **get_sqllist.sql**

```
select sql_text
     , round(decode(executions,null,0,0,0,(nvl(buffer_gets,0) / executions)),1)  BUFGETSPEREXEC
    ,EXECUTIONS,PARSE_CALLS,DISK_READS,BUFFER_GETS,ROWS_PROCESSED
     ,SHARABLE_MEM,PERSISTENT_MEM,RUNTIME_MEM,MODULE,USERS_EXECUTING
     ,SORTS,LOADED_VERSIONS,OPEN_VERSIONS,USERS_OPENING,LOADS
      ,FIRST_LOAD_TIME,INVALIDATIONS,COMMAND_TYPE,OPTIMIZER_MODE
      ,OPTIMIZER_COST, PARSING_USER_ID
from v$sql
where decode(executions,null,0,0,0,(nvl(buffer_gets,0) / executions))  > 10000
      and PARSING_USER_ID <> 0
order by BUFGETSPEREXEC  DESC
```

위의 SQL문장은 Execution마다 평균 몇 개의 Oracle Block을 읽었는지(Buffer로부터. 만일 Buffer에 오러클 Block이 없었다면 Disk로부터 가져온 값도 포함됨)의 역순으로 보여주므로 위의 Top순서 위주로 비효율적인 SQL로 판단하면 됨. 특히 ROWS_PROCESSED(처리된 Row수)가 작으면서 BUFGETSPEREXEC(평균 Exec당 Buffer Block Read Count)가 큰 SQL위주로 Tuning.

또한 EXECUTIONS가 높은 SQL문장이 자주 실행되므로 이들 SQL문장은 반드시 Tuning. 그러나 1회성 SQL들은 EXECUTIONS이 작지만 SQL로 자주 실행되는 형태를 판단해야 함.

위의 SQL을 TOAD와 같은 Tool을 이용하면 효과적

- **Display형태 및 Column 설명(일부)**

FULL_SQLTEXT	BUFGETSPEREXEC	EXECUTIONS	PARSE_CALLS	DISK_READS	BUFFER_GETS	ROWS_PROCESSED	…
SELECT B_.I_TYPE FROM patent.dmi_object B_ WHERE ((B_.I_TAG=:tag AND B_.R_OBJECT_ID_I=:handle) AND B_.I_IS_BASE_TYPE=1)	4	29651	29651	10550	118114	200	…
SELECT B_.I_TYPE FROM dmadmin.dmi_object B_ WHERE ((B_.I_TAG=:tag AND B_.R_OBJECT_ID_I=:handle) AND B_.I_IS_BASE_TYPE=1)	4	29178	29178	6288	116805	1200	…
. …..	. …..	. …..	. …..	. …..	. …..	. …..	. …..

✓ SQL_TEXT　　　: SQL 의 Text(V$SQL.SQL_TEXT는 최대 1KB만 보여줌)
✓ EXECUTIONS　　: SQL문장이 실행된 횟수 (Instance Startup이후 의 누적치).
　　　　　　　　　비 공유 SQL일 경우 대부분 1. 공유 SQL일 경우 이 값이 높다.
✓ FIRST_LOAD_TIME: SQL이 처음으로 Cache에 Loading된 시간
✓ PARSE_CALLS　　: Parse Request를 요청한 수(대부분 < executions)

- ✓ DISK_READS : SQL을 실행하기 위해 Disk로부터 읽어온 Oracle Block수
  (읽은 Size는 이 값 * db_block_size)
- ✓ BUFFER_GETS : SQL을 실행하기 위해 Buffer로부터 읽어온 Oracle Block수.
  주로 이 Column의 값을 exections으로 나누어 높은 값의 SQL이 비효율적인 SQL임.
  (Disk로 부터 읽어 온 값도 여기에 포함되어 있음)
- ✓ ROWS_PROCESSED : SQL로 실행되어 처리된(Select Low수 또는 Transaction대상 Row수)
- ✓ OPTIMIZER_MODE : 이 SQL문장이 실행될 때의 OPTIMIZER_MODE
- ✓ OPTIMIZER_COST : 이 SQL문장이 실행될 때의 Optimizer(Cost Base Optimizer)에 의해 계산된 Cost수
  이며, 큰 값일수록 비 효율적이나 이 값은 다른 SQL의 Cost와는 비교대상이 아님)
- ✓ PARSING_USER_ID: 이 SQL문장을 실행했던 User ID(0은 SYS user로 대부분 Recursive SQL,5는 SYSTEM)
- ✓ MODULE : 이 SQL문장을 실행했던 APP의 Module명 (예. SQL*Plus,T.O.A.D).
  DBMS_APPLICATION_INFO.SET_MODULE 로 Application에서 실행하게 되면 나타나며
  어떤 module에서 들어온 SQL인지 확인할 수 있다.

- **응용형태 (가장 많이 실행되는 SQL유형 보기 10000번 이상, 단 Recursive SQL은 제외)**

```
select sql_text
     , round(decode(executions,null,0,0,0,(nvl(buffer_gets,0) / executions)),1)  BUFGETSPEREXEC
    ,EXECUTIONS,PARSE_CALLS,DISK_READS,BUFFER_GETS,ROWS_PROCESSED
     ,SHARABLE_MEM,PERSISTENT_MEM,RUNTIME_MEM,MODULE,USERS_EXECUTING
     ,SORTS,LOADED_VERSIONS,OPEN_VERSIONS,USERS_OPENING,LOADS
      ,FIRST_LOAD_TIME,INVALIDATIONS,COMMAND_TYPE,OPTIMIZER_MODE
      ,OPTIMIZER_COST, PARSING_USER_ID
from v$sql
where executions  > 10000
      and PARSING_USER_ID <> 0
order by executions DESC
```

- **응용형태 (비공유 1회성 SQL유형 보기 => executions = 1)**

```
select sql_text
     , round(decode(executions,null,0,0,0,(nvl(buffer_gets,0) / executions)),1)  BUFGETSPEREXEC
    ,EXECUTIONS,PARSE_CALLS,DISK_READS,BUFFER_GETS,ROWS_PROCESSED
     ,SHARABLE_MEM,PERSISTENT_MEM,RUNTIME_MEM,MODULE,USERS_EXECUTING
     ,SORTS,LOADED_VERSIONS,OPEN_VERSIONS,USERS_OPENING,LOADS
      ,FIRST_LOAD_TIME,INVALIDATIONS,COMMAND_TYPE,OPTIMIZER_MODE
      ,OPTIMIZER_COST, PARSING_USER_ID
from v$sql
where executions  = 1
      and PARSING_USER_ID <> 0
```

- **응용형태 (최근 실행한 SQL 유형 보기 (1분전)=>** first_load_time >= to_char((sysdate - 1 / 1440),'YYYY-MM-DD / HH24:MI:SS')**)**

```
select sql_text
    , round(decode(executions,null,0,0,0,(nvl(buffer_gets,0) / executions)),1)  BUFGETSPEREXEC
    ,EXECUTIONS,PARSE_CALLS,DIS K_READS,BUFFER_GETS,ROWS_PROCESSED
    ,SHARABLE_MEM,PERSISTENT_MEM,RUNTIME_MEM,MODULE,USERS_EXECUTING
 ,SORTS, LOADED_VERSIONS, OPEN_VERSIONS, USERS_OPENING, LOADS,
FIRST_LOAD_TIME
 ,INVALIDATIONS, COMMAND_TYPE, OPTIMIZER_MODE, OPTIMIZER_COST,
PARSING_USER_ID
  -- ,PARSING_SCHEMA_ID, KEPT_VERSIONS, ADDRESS, TYPE_CHK_HEAP, HASH_VALUE,
CHILD_NUMBER
  -- ,MODULE_HASH, ACTION, ACTION_HASH, SERIALIZABLE_ABORTS,
OUTLINE_CATEGORY
 from v$sql
where first_load_time >= to_char((sysdate - 1 / 1440),'YYYY-MM-DD / HH24:MI:SS')
```

- **응용형태 (SQL ElapseTime이 1초 이상 SQL유형 보기,단 Recursive SQL은 제외)**

```
select sql_text
    , round(decode(executions,null,0,0,0,(nvl(buffer_gets,0) / executions)),1)  BUFGETSPEREXEC
    , round(decode(executions,null,0,0,0,(nvl(ELAPSED_TIME,0) / executions)),1)  ElapsedTimePerExec
    ,EXECUTIONS,PARSE_CALLS,DISK_READS,BUFFER_GETS,ROWS_PROCESSED
    ,SHARABLE_MEM,PERSISTENT_MEM,RUNTIME_MEM,MODULE,USERS_EXECUTING
    ,SORTS, LOADED_VERSIONS, OPEN_VERSIONS, USERS_OPENING,
    LOADS, FIRST_LOAD_TIME ,INVALIDATIONS, COMMAND_TYPE, OPTIMIZER_MODE,
       OPTIMIZER_COST, PARSING_USER_ID ,PARSING_SCHEMA_ID, KEPT_VERSIONS,
        ADDRESS, TYPE_CHK_HEAP, HASH_VALUE, CHILD_NUMBER    ,
       MODULE_HASH, ACTION, ACTION_HASH, SERIALIZABLE_ABORTS,
        OUTLINE_CATEGORY
from v$sql
where round(decode(executions,null,0,0,0,(nvl(ELAPSED_TIME,0) / executions)),1)  >= 1000000
       and PARSING_USER_ID <> 0
order by executions DESC
```

과다한 Sorting유발 SQL 찾기 및 TEMP Tablespace의 Sort Space 현황 보기

- **과다한 TEMPORARY Tablespace의 I/O유발 SQL 보기 (sort_usg.sql)**

```
select /*+ ORDERED */
        se.username ,
       session_num ,
       se.process ,
       segfile# ,
       segblk#,
       segtype,
       extents ,
       blocks ,
        getfullsql(hash_value) full_sqltext
from v$sort_usage so, v$session se, v$sql sq
where so.session_addr = se.saddr
and   se.sql_address = sq.address
--and   se.audsid != userenv('sessionid')
```

위의 SQL문장은 Temp Tablespace를 현재시점에 사용되는 현황을 확인하고, 또한 어떤형태의 Operation(SORT,HASH,…)에 의해 발생되며, 어떤 SQL문장인지를 확인하는 SQL문장이다. 여기서 extents(Extent 개수) 및 Blocks(Oracle Block수=> Size는 BLOCKS * DB+BLOCK_SIZE)이 큰 SQL을 잡아서 Tuning및 필요시 Session Level에 충분한 Memory(Sort 또는 Hash)를 할당해야 한다.

위의 SQL을 TOAD와 같은 Tool을 이용하면 효과적

- **Display형태 및 Column 설명(일부)**

USERNAME	SESSION_NUM	PROCESS	SEGFILE#	SEGBLK#	SEGTYPE	EXTENTS	BLOCKS	FULL_SQLTEXT
WEBLOGIC	61617	1464:1460	4	100866	SORT	10	5120	SELECT CHRG_BIZ_OFCE_CD,YEAR, COUNT(*) FROM (SELECT A.CHRG_BIZ_OFCE_CD CHRG_BIZ_OFCE_CD, SUBSTR(B.JUMIN_BIZ_NO,1,2) YEAR FROM CMBB01T01 A, CMAA01T01 B, CMBB02T01 C WHERE A.WK_STAT_CD NOT IN ('02','09') AND A.CHG_KIND_CD ='101' AND C.ENTR_CL_CD = '1' AND A.RECV_NO = C.RECV_NO AND A.CUST_NO = B.CUST_NO AND B.CUST_TYPE_CD = '1') GROUP BY CHRG_BIZ_OFCE_CD,YEAR

- **Column 설명(일부)**

✓ USERNAME : Disk Sort를 유발시킨 SQL실행 User
✓ SESSION_NUM : V$SESSION.SID
✓ PROCESS : V$PROCESS.PROCESS
✓ SEGFILE# : TEMPORARY Segment(SORT 또는 HASH, Temp Table,..) 의 Start File #
✓ SEGBLK# : TEMPORARY Segment(SORT 또는 HASH, Temp Table,..) 의 Start Block #
✓ SEGTYPE : SORT(Sort에 의한 Sort Segment), HASH(Hash Join에 의한 Segment)
✓ EXTENTS : 해당 Operation(SEGTYPE 형태별)의 발생시킨 Extent 개수
✓ BLOCKS : 해당 Operation(SEGTYPE 형태별)의 발생시킨 Block 개수
✓ FULL_SQLTEXT : 관련 SQL의 Full Text

(참고사항) SYS.DBA_SEGMENTS에서 SEGMENT_TYPE은 'TEMPORARY'로 나타나며 SEGMENT_NAME은 SEGFILE#.SEGBLK# 의 숫자형태로 나타남.

select * from dba_segments where segment_type = 'TEMPORARY';

- **TEMPORARY('TEMP') Tablespace의 Space 현황 보기 (sort_segs.sql)**

```
select segment_file ,
       segment_block,
       extent_size ,
       current_users ,
       total_extents,
       total_blocks,
       used_extents,
       used_blocks,
       free_extents,
       free_blocks,
       (max_sort_blocks * 8192)  ms_bytes
from v$sort_segment ;
```

위의 SQL문장은 Sort Storage의 현황을 확인 Sort의 Monitoring 과 적절한 Sort Size의 유지를 위해 사용한다.

- **Display형태 및 Column 설명(일부)**

SEGMENT_FILE	SEGMENT_BLOCK	EXTENT_SIZE	CURRENT_USERS	TOTAL_EXTENTS	TOTAL_BLOCKS	USED_EXTENTS	USED_BLOCKS	FREE_EXTENTS	FREE_BLOCKS	MS_BYTES

4	101378	512	0	135	69120	0	0	135	69120	562036736

- **Column 설명 (일부)**

✓ SEGMENT_FILE : Sort Segment의 Segment Start File #
✓ SEGMENT_BLOCK : Sort Segment의 Segment Start Block #
✓ EXTENT_SIZE   : Sort Segment의 전체 Extent 확장 Size(TEMP의 NEXT Size)
✓ CURRENT_USERS : 0 (SYS) , Sort Segment의 Owner는 SYS
✓ TOTAL_EXTENTS/TOTAL_BLOCKS   : segment의 전체 Extent & Block Size(Sort Pool)
✓ USED_EXTENTS/USED_BLOCKS   : 전체 중에서 현재 사용중인 Size (이 값이 클수록 현재 사용 중. 사용중인 것이 없을 경우 0)
✓ FREE_EXTENTS/FREE_BLOCKS   : segment중에 Free Extent & Block Size
✓ MS_BYTES   : MAX Sort Block의 Byte Size

(참고사항) 전체 Temp Tablespace중에 Extent로 확장된 영역이 TOTAL_BLOCKS * DB_BLOCK_SIZE이며, 외대Temp Tablespace Size까지 확장 됨. TOTAL_BLOCKS 이 Temp Tablespace의 전체 Block에 도달하였고 USED_BLOCKS도 이 값에 근접할 경우 Temp Space의 Extent Error가 발생되므로 주기적으로 monitoring필요.

1. Forced Rewrite

- ✓ QUERY_REWRITE_ENABLED기능은 Function-Based INdex와 Materialized VIew에 대해서 Query의 Rewrite(Index Column에 Function을 사용한 경우 Function-based Index를 사용하게 하고, Query가 MV와 조건절이 같은 경우 Master를 읽지 않고 MV를 읽도록 하는 내부적인 처리)에 대한 Session level에서 FORCE지정이 가능해졌다.
- ✓ QUERY_REWRITE_ENABLED이 FORCE로 설정되면 cost-base로 cost를 계산한 결과로 Rewrite를 결정하지 않고 항상 Rewrite를 적용한다. Oracle 8i에서는 TRUE/FALSE만으로 Cost-base로 enable 또는 Disable기능만 제공되었다.
- ✓ 이 기능은 항상 Query Rewrite기능이 적용되는 것이 항상 효과적인 곳에 사용될 수 있다.
- ✓ 그렇게 하므로서 Optimizer가 Cost계산에 의한 Rewrite의 결정여부를 판단하는데 대한compile time을 줄일 수 있는 이점이 있다.
- ✓ function-based Index를 사용하기 위해서는 QUERY_REWRITE_ENABLED이 TRUE 또는 FORCE로 설정되어야 한다.

- QUERY_REWRITE_ENABLED = {force | true | false}
  . TRUE:cost -based rewrite
  . FALSE:no rewrite ,function-based index는 사용되지 않는다. 단 실제 Column에 있는 값을 얻기 위해서 사용되어질 수 있다.
  . FORCE:forced rewrite (10g New). cost evaluation을 하지 않고 rewrite를 강제수행한다.

- For example:

```
CREATE INDEX idx ON table_1 (a + b * (c - 1), a, b);

SELECT a
FROM table_1
WHERE a + b * (c - 1) < 100;
```

2. Union-All Rewrite of Queries with Grouping Sets

- ✓ 새로운 Hint로 EXPAND_GSET_TO_UNION을 제공하며 Query에 function-based indexes가 사용될 경우 보다 낳은 최적의 Plan을 찾기 위해서 Query Rewrite기능이 수행되며, 특히 OLAP에서 중요하다.
- ✓ EXPAND_GSET_TO_UNION hint는 grouping sets (GROUP BY GROUPING SET 또는 GROUP BY ROLLUP)을 사용하는 곳에 이용될 수 있다. 이 Hint는 관련된 Query를 각 Grouping의 UNION ALL Query로의 Compound Query로 변환하여 각 Query Block별로 더 낳은(Meterialized View사용 중) 방법이 없는지를 찾기 위한 목적이다.

예제

```
SELECT /*+ EXPAND_GSET_TO_UNION */ year, quarter, month, sum(sales)
```

```
FROM T
GROUP BY year, rollup(quarter, month)

===>  tranformed to
SELECT year, quarter, month, sum(sales)
FROM T
GROUP BY year, quarter, month
UNION ALL
SELECT year, quarter, null, sum(sales)
FROM T
GROUP BY year, quarter
UNION ALL
SELECT year, null, null, sum(sales)
FROM T
GROUP BY year

===>  UNION ALL 의 Compound Query Block들은 각각 Query Rewrite가 가능한지 판단하여
      다음과 같이 Materialized View를 사용하기도 한다.

SELECT year, quarter, month, sum(sales)
FROM T
GROUP BY grouping set ( (year, quarter, month), (year, quarter) )
UNION ALL
SELECT year, null, null, sum_sales
FROM MV
```

3. Dynamic Sampling for the Optimizer

- ✓ Optimizer가 보다 정확한 Plan을 결정하기 위해서 더 정확한 selectivity,cardinality등을 결정하기 위해 Parsing시 통계정보를 돌리듯이 dynamic sampling하여 통계정보를 구성하는 방법이다. 주요 목적은 보다 낳은 최적의 Plan을 결정하기 위해 사용하는 방식으로 주로 Table의 통계정보가 없거나 또는 현실 Data와는 맞지 않게 너무 오래된 정보를 가지고 있는 경우 자동으로(level에 따라서) 결정되어 Sampling하게 된다.
- ✓ DYNAMIC_SAMPLING Parameter로 0~10 Level까지 설정 가능하며 DYNAMIC_SAMPLING hint로도 해당 SQL에 대한 보다 낳은 Plan결정을 위한 Dynamic Sampling기능으로 제공된다.
- ✓ Level이 높을 수록 Dynamic Sampling을 위한 통계정보를 만들기 위해 Recursive SQL을 통해 많은 Sample Block을 읽어서 Sampling하게 된다.

4. Locally Managed SYSTEM Tablespace

- ✓ Oracle9i R2(9.2)부터 SYSTEM tablespace에 대한 locally managed tablespac의 적용이 가능하다.
- ✓ 즉 CREATE DATABASE시 locally managed SYSTEM tablespace를 위해 "EXTENT MANAGEMENT LOCAL"을 사용할 수 있다.

5. Data Segment Compression

- ✓ Data segment compression기능은 Disk사용량과 Memory사용량(특히 Buffer Cache)을 줄여 준다.
- ✓ 주로 read-only operations이 주된 Table에 scaleup하기 위한 좋은 방법이며, 또한 Query의 실행시간을 단축시켜 준다.
- ✓ Oracle9i R2(9.2)에서는 특별한 Tuning없이도 많은 경우에 있어서 compression율이 개선되었다. 물론 보다 낳은 compression율을 위해서는 경우에 따라서 정교한 조정으로 효과를 볼 수 있다.

6.Shared Pool Advisory Statistics

- ✓ Oracle library cache 가용 Memory에 따라서 Parsing율을 크게 좌우한다.
- ✓ Oracle9i R2(9.2)에서부터 Memory의 Size의 변화에 따른 parse Rate와 같은 성능변화을 예측하는 shared pool advisory statistics을 제공한다.
  - library cache로 사용되고 있는 Memory량
  - 현재 Pinned되어 있는 Memory량
  - shared pool의 LRU list에 이는 Memory량
  - shared pool의 size변화량에 따라 얼마만큼의 Time의 득과 실에 대한 정보

7.PGA Aggregate Target Advisory

- ✓ Oracle이 모든 Memory관련 SQL Operator들의 Performance를 최대화 하기 위하여 가능한 최대한의 SQL Workarea(PGA memory의 cache memory: *_AREA_SIZE)를 사용하게 하는 방법. Oracle은 PGA_AGGREGATE_TARGET에서 지정한 한 Instance내에서 사용할 모든 process들의 PGA Memory의 Limit내에서 최대한의 Memory를 사용하도록 한다.

8.FILESYSTEMIO_OPTIONS

- ✓ Oracle9i R2(9.2)부터 Oracle File System File들에 대한 asynchronous I/O 또는 direct I/O에 대한 enable 또는 disable기능을 제공한다.
- ✓ 관련 Parameter는 FILESYSTEMIO_OPTIONS으로 Platform에 종속적이며 Platform별로 Default값이 최적인 상태로 설정되어 있다. 이 Parameter는 동적으로 변경 가능하다.

9.MTTR Advisory

- ✓ Oracle9i R2(9.2)부터 MTTR목적의 advisory기능을 제공한다.
  - STATISTICS_LEVEL = ALL 또는 TIPICAL
  - FAST_START_MTTR_TARGET :single instance의 Crash Recovery에 대한 MTTR(Mean Time To Recover)의 초 단위(0 to 3600 seconds)를 지정한다.
  - V$MTTR_TARGET_ADVICE : 관련 View

10. Statistics Collection Level

- ✓ Oracle9i R2(9.2)부터 주요 Statistics정보의 수집 및 Database의 Advisory기능을 제어할 수 있는 STATISTICS_LEVEL(=BASIC | TYPICAL | ALL)이 제공된다.
- ✓ 이값은 Database에 대한 Statistics정보의 수집 Level을 지정하는 것으로 default값은 TYPICAL이다.

11. Segment-Level Statistics

- ✓ Oracle9i R2(9.2)부터 segment-level(Table / Index등)의 개별 Segment들에 대한 statistics정보를 제공하여 Performance문제가 Segement별로 어느곳에서 발생하고 있는지를 쉽게 Monitoring할 수 있다.
- ✓ 즉 wait events와 system statistics과 병행해서 해당 Instance의 주요 Contention이 발생되고 있는 hot table 또는 index를 확인 가능하다. (관련 View : V$SEGMENT_STATISTICS, V$SEGSTAT, V$SEGSTAT_NAME)

12. Runtime Row Source Statistics

- ✓ 현재 Cursor Cache에 있는 SQL문장들의 Execution Plan을 확인할 수 있는 V$SQL_PLAN이 Oracle 9iR1(9.0.1)에 새로 제공되었으며. Oracle 9iR2(9.2)에서는 각 Operation의 Statistics을 확인할 수 있는 V$SQL_PLAN_STATISTICS을 제공한다. 이 정보를 보기 위해서는 STATISTICS_LEVEL=ALL로 설정을 해야 한다.
- ✓ 또한 VSQL_PLAN, VSQL_PLAN_STATISTICS,V$SQL_WORKAREA 를 이용하여 PLAN,Statistics,Memory사용량등을 이용하여 SQL Tuning을 위한 효과적인 정보로 활용될 수 있다.
- ✓ (관련 View : VSQL_PLAN,VSQL_WORKAREA,V$SQL_PLAN_STATISTICS, V$SQL_PLAN_STATISTICS_ALL)