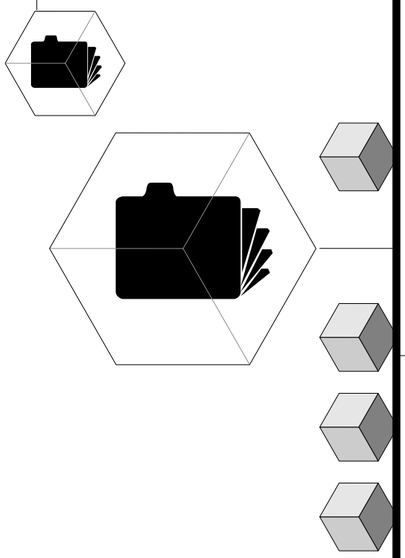

데이터베이스 총정리 및 실전 사례

지난 4회 동안 오라클 관련 애플리케이션 프로그램을 개발하는 개발자를 위해 꼭 알아둬야 할 문제를 짚어봤다. 이번 글은 마지막 편으로 이제까지 다룬 내용 중 반드시 기억해야 할 사항을 바탕으로 실제 프로젝트에서 성능에 대한 문제를 어떤 방식으로 해결했는지를 살펴보고자 한다.

이번 글은 실전에서 나왔던 SQL문 튜닝 사례를 소개하고자 한다. 사실 개인적인 의견으로는 업무를 모르는 상태에서 다른 프로젝트의 SQL문에 대한 튜닝 사례로 튜닝 공부를 하는 것은 상당히 힘들며 오랜 시간이 걸린다고 생각한다. 왜냐하면 테이블의 성격을 몰라 SQL문 자체를 이해하기 어렵고 특히 데이터 분포도를 모르는 상황이기 때문에 가슴에 와 닿지 않을 뿐더러 직접 고친 후 테스트를 해보고 시간을 앞당기는 쾌감을 얻기 힘들기 때문이다. 따라서 아직 경험이 부족한 개발자는 스스로 튜닝 환경을 만들어 테스트해 보길 바란다. DBA_USERS, DBA_TABLES, DBA_TAB_COLUMNS, DBA_INDEXES, DBA_IND_COLUMNS, DBA_TABLESPACES 6개 딕셔너리 뷰를 물리 테이블로 만들어 놓고 직접 연관되는 것끼리 조인하고 출력하는 연습이 많은 도움이 될 것이다.

필자가 실전 사례를 소개하고자 함은 실제 프로젝트에서는 여러 개의 테이블을 복잡다단하게 조인하며 결과를 내지만 문제를 일으키는 원인은 단순한 튜닝 원리를 적용하지 않았음을 상기시키고자 함이다. 실제 프로젝트에서 컨설턴트들이 문제를 일으킨 SQL문에 대한 튜닝 사례 중 업무 설명없이 독자들이 쉽게 이해할 수 있는 것만 추출했으니 기법을 잘 익혀두기 바란다.

박성우 | dont4get@chol.com

현엔지스템즈와 LG-CNS를 거쳐 현재는 수로텍 연구소 팀장으로 일하고 있다. 대형양 화재물류 개발, KT-ICIS 요금관리시스템 프로젝트 DBA, KTF 차세대 빌링 프로젝트 DBA, 통합 하수관거 유지관리 SW 개발 팀장으로 활동했다.

사례 1 : 조인 순서 변경

설명

첫 번째 읽을 테이블은 중첩 루프 조인이나 해시 조인이 중요하다. 중첩 루프 조인에서 첫 번째 테이블 선정은 뒤에 읽혀질 테이블의 인덱스 구조에 따라 선정이 달라진다. 조인될 대상 데이터가 최대한 중복으로 읽히지 않도록 첫 번째 테이블을 선정해 주는 것이 좋다. 적게 읽고, 읽은 데이터는 최대한 읽지 말라는 공식이 적용된다. 해시 조인은 해시 테이블을 되도록 적게 만들어 주는 것이 좋다. 가장 좋은 경우는 해시 테이블이 메모리 상에서 만들어지는 경우이다.

문제점 및 개선 사항

앞에서 보면 알 수 있듯이 select절의 두 테이블을 해시 조인하는데 VB_CCPISVCNOHS 테이블 용량은 1GB이고 VB_MPRDCD 테이블 용량은 4MB이다. 이러한 경우에 VB_MPRDCD 테이블 데이터를 해시 테이블로 사용하는 것이 훨씬 효율적이다. 변경 전 예문의 SQL을 ordered 힌트를 사용하고 from절의 테이블 순서를 변경하고 ORDERED 힌트문을 추가해 VB_MPRDCD 테이블부터 해시 조인될 수 있도록 했다. 또한 PARALLEL-TO-PARALLEL 효과를 볼 수 있도록 PARALLEL 수를 동일하게 주되 2개로 줄였다. PARALLEL 수를 줄임에도 불구하고 SQL의 변경을 통해 Inset 수행 성능이 약 3배 빨라졌다

변경 전 SQL문

```
Insert into VB_CCPISVCNOHS1
(
   SA_ID,
   START_DATE,
   END_DATE,
   SA_CD,
   SVC_NO,
   RS_CD,
   CHKFIELD,
   ERROR_FLAG,
   REG_DATE
)
select  /*+ USE_HASH(a b)
            FULL(a) parallel(a 10)
            FULL(b) parallel(b 10) */
       a.sa_id,
       a.start_date,
       a.end_date,
       a.prod_cd,
       a.svc_no,
       '1',
       null,
       null,
       sysdate
  from  VB_CCPISVCNOHS a,  /* 1GB */
        VB_MPRDCD b      /* 4MB */
  where  a.end_date is null
    and  a.prod_cd = b.sa_cd
    and  b.use_resource_type_cd in ( '01', '03', '04', '05');
```

개선된 SQL문

```
insert into  VB_CCPISVCNOHS1
(
  SA_ID,
  START_DATE,
  END_DATE,
  SA_CD,
  SVC_NO,
  RS_CD,
  CHKFIELD,
  ERROR_FLAG,
  REG_DATE
)
select  /*+ ORDERED USE_HASH(a b)
            FULL(a) parallel(a 2)
            FULL(b) parallel(b 2) */
       a.sa_id,
       a.start_date,
       a.end_date,
       a.prod_cd,
       a.svc_no,
       '1',
       null,
       null,
       sysdate
from  VB_MPRDCD b,         /* 4MB */
      VB_CCPISVCNOHS a   /* 1GB */
where  a.end_date is null
   and  a.prod_cd = b.sa_cd
   and  b.use_resource_type_cd in ( '01', '03', '04', '05');
```

사례 2 : 중첩 루프 조인에서 해시 조인으로 변경

설명

변경 전의 SQL문은 개발자가 USE_NL이란 힌트를 사용해 중첩 루프 방식으로 유도했으나 속도가 개선되지 않자 PARALLEL 옵션을 사용한 경우이다. USE_NL은 인덱스 방식의 READ를 좋아하기 때문에 블럭 READ를 잘 하지 않는다. PARALLEL 옵션은 풀 스캔일 경우에만 적용되는 옵션이다. 이 SQL문은 VB_CCPICARD란 VIEW가 200만 건 정도 나오므로 TB_CDBPKEY, VB_MPRDCD를 200만 번씩 인덱스와 데이터 읽기를 시도한다. 해시 조인의 유용성을 모르는 개발자들의 전형적인 코딩 방식이다.

문제점 및 개선 사항

이 SQL을 A, B, C 테이블의 조인 연산을 해시 조인으로 변경했다. 이렇게 중첩 루프 조인을 하던 A, B, C 테이블에 대한 조인을 해시 조인으로 변경함으로써 성능이 10배 좋아졌다.

변경 전 SQL문

```
SELECT /*+ ORDERED   USE_NL(A,B,C)  PARALLEL(A,10)  PARALLEL(B,10)
PARALLEL(C,10) */
        ......
FROM VB_CCPICARD A,       /* 200만 건, no proper index */
       TB_CDBPKEY  B,       /* 190만 건, PK:(KTCARDNO) */
       VB_MPRDCD   C        /* 544건,   PK:(SA_CD)    */
WHERE A.EDIT_SVC_NO = B.KTCARDNO
   AND A.SA_ID > ' '                              /* 99% TRUE */
   AND (A.LIFE_CYCLE = 'A' OR A.END_PROC_TYPE = 'D' ) /* 99% TRUE */ --최종 건만
   AND A.SA_CD = C.SA_CD(+)
```

개선된 SQL문

```
SELECT /*+ ORDERED  USE_HASH(A B C) full(a) full(b) full(c)
           PARALLEL(A,10) PARALLEL(B, 10) PARALLEL(C, 10) */
        ......
FROM VB_CCPICARD A,    /* 200만 건, no proper index */    — 서비스 계약(카드)
      TB_CDBPKEY  B,       /* 190만 건, PK:(KTCARDNO) */    -- CARD KEY TABLE
      VB_MPRDCD   C        /* 544건,   PK:(SA_CD)    */    -- 상품 코드 TABLE
WHERE A.EDIT_SVC_NO = B.KTCARDNO
  AND A.SA_ID > ' '                          /* 99% TRUE */
  AND (A.LIFE_CYCLE = 'A' OR A.END_PROC_TYPE = 'D' ) /* 99% TRUE */ — 최종 건만
  AND A.SA_CD = C.SA_CD(+)
```

사례 3 : 힌트의 적용

설명

개발자는 SQL문을 정확하게 구사했지만 비용 기반 옵티마이저에서는 테이블 정보를 수시로 UPDATE하지 않으면 원하는 경로로 읽히지 않을 경우가 있다. 이 SQL문은 :vcSaId라는 상수 조건을 통해 첫 번째 읽혀질 테이블을 명확하게 범위를 좁혀 놓았지만 정렬 병합이라는 엉뚱한 실행 계획으로 택해져 성능이 저하돼 끝내 중간에 프로그램을 중단시켰다. 이는 개발자가 PLAN을 작성해 보지 않고 그냥 실행한 경우이다.

문제점 및 개선 사항

SQL을 실행했을 시 정렬 병합 조인을 실행 계획으로 택하고 있어 성능이 저하된다. 개선된 SQL과 같이 테이블의 읽는 순서와 조인 방식에 대한 힌트를 추가해 중첩 루프 방식을 선택하도록 강제로 조정해 2초 내로 줄였다.

변경 전 SQL문

```
SELECT
       DECODE(N.START_ORD_NO, NULL, '0', '1'),
       B.NODE_NAME, B.SA_ID
FROM  TB_CCPIPRODHIER A, TB_CCPIPRODHIER B,
       TB_MPRDCD M, TB_CCPIDANODE N
WHERE A.SA_ID = :vcSaId AND            -- NODE_ID
       A.END_DATE IS NULL AND
       B.IA_ID = A.IA_ID AND
       B.END_DATE IS NULL AND
       M.SA_CD = B.SA_CD AND
       M.MNG_VAL_13 = '2' AND
       N.NODE_ID(+) = B.SA_ID AND
       N.NODE_TYPE(+) = 'S' AND
       N.END_DATE(+) IS NULL
```

개선된 SQL문

```
SELECT  /* ORDERED USE_NL(A B M N) */
        DECODE(N.START_ORD_NO, NULL, '0', '1'),
        B.NODE_NAME, B.SA_ID
FROM   TB_CCPIPRODHIER A, TB_CCPIPRODHIER B,
        TB_MPRDCD M, TB_CCPIDANODE N
WHERE  A.SA_ID = :vcSaId AND           -- NODE_ID
        A.END_DATE IS NULL AND
        B.IA_ID = A.IA_ID AND
        B.END_DATE IS NULL AND
        M.SA_CD = B.SA_CD AND
        M.MNG_VAL_13 = '2' AND
        N.NODE_ID(+) = B.SA_ID AND
        N.NODE_TYPE(+) = 'S' AND
        N.END_DATE(+) IS NULL
```

사례 4 : SQL 문장의 분리를 통한 READ 수 줄이기

설명

입력 인자에 따라 수행되는 SQL이 달라지고 이 SQL이 큰 SQL의 서브 쿼리로 들어가는 경우에 개발자들은 UNION ALL 문장 사용을 선호한다. 왜냐하면 서브 쿼리에 쓰이는 입력 인자 값이 달라질 때마다 if문으로 나누어 큰 SQL을 각각 작성해야 하기 때문이다. 이 UNION절의 사용은 입력 인자 구분 값의 개수와 외부 쿼리가 어느 정도 큰 SQL인가에 따라 사용여부가 달라져야 할 것이다. 이 업무는 고객이 청구 테이블이나 체납 테이블한 경우만 있는 경우임에도 불구하고 SQL문은 두 개의 쿼리를 UNION해 MNG_TEL_OFC_CD 1 row를 가져오도록 하였다. 즉 어느 한쪽은 쓸데없는 접근을 하고 있는 것이다.

문제점 및 개선 사항

SQL은 무조건 한 행의 MNG_TEL_OFC_CD만 가져오면서 UNION을 사용하고 있다. 랜덤하게 하나의 MNG_TEL_OFC_CD만 조회한다면 굳이 두 테이블을 UNION할 필요가 없다. 한 테이블만 접근해 MNG_TEL_OFC_CD만 조회하도록 한다. 또한 UNION은 UNION ALL과 달리 정렬해 중복된 row를 배제하는데 변경 전 SQL은 불필요한 정렬 작업까지 수행하고 있다. 실제로 개발자의 의도를 파악한 결과 업무적으로 청구의 경우는 UNION 위의 SQL을 사용하고 체납의 경우는 UNION 아래의 SQL을 작성해야 하는데 이를 분리하지 않아 잘못된 데이터 조회가 이뤄질 수 있는 오류까지 포함하고 있다. 그래서 개선된 SQL은 청구와 체납의 경우에 구분해 각각의 SQL을 수행하도록 프로그램을 수정했다. 이 SQL문이 1건 읽으러 가는 노력을 줄이는 것으로 보이지만 LOOP문 안에서 SQL문이 수행되고 이것을 수천 만 번 호출한다면 들어가는 노력이 훨씬 줄어들 것임을 기억하자.

변경 전 SQL문

```
select MNG_TEL_OFC_CD into :b0
from (select MNG_TEL_OFC_CD
  from 청구  where (INV_DATE=:b1 and IA_ID=:b2)
  union
  select MNG_TEL_OFC_CD
  from 체납  where (INV_DATE=:b1 and IA_ID=:b2))
where rownum =1
```

개선된 SQL문

```
If (청구) then
    select MNG_TEL_OFC_CD
    from 청구
    where INV_DATE=:b1 and IA_ID=:b2 and rownum = 1
else (체납) then
    select MNG_TEL_OFC_CD
    from 체납
    where INV_DATE=:b1 and IA_ID=:b2  and rownum =1
end if
```

사례 5 : SQL문 병합을 통한 READ 수 줄이기

설명

분리된 SQL문을 통합함으로써 프로그램의 성능을 향상시킨 튜닝 사례를 살펴보도록 하자. 예문은 커서로 선언된 SQL문이 있고 이 커서의 데이터를 배열 페치(fetch)한 후 루프 내에서 또 하나의 SQL문을 실행하는 구조이다. 이렇게 되면 loop문 내에서 커서로부터 읽은 건건의 데이터에 대해 SQL이 호출돼 성능 저하를 가져올 수 있다.

문제점 및 개선 사항

변경 전 SQL을 앞서 지적한 바와 같이 건건이 호출되는 SQL로 인한 성능 저하를 피하기 위해 하나의 SQL로 통합해 커서를 선언하고 페치하도록 수정했다. 메인 커서와 loop문 안의 SQL문을 통합하는 것은 대량 데이터 배치에서는 많은 효과를 거둔다. loop문 안에서 빈번한 SQL문의 호출은 바람직하지 않다. 만약 이런 경우가 발생한다면 테이블 간의 관계를 명확히 인식해 두 개의 SQL문을 통합하는 노력을 해야 한다.

변경 전 SQL문

```
.....
SELECT  NVL(P_NODE_ID,'*'), SA_ID, START_DATE, NVL(NODE_NAME,'*'), SA_CD, LEVEL
FROM    TB_CCPIPRODHIER
start with  P_NODE_TYPE = 'C'
   AND  P_NODE_ID = :vcCustomId
   AND      END_DATE IS NULL
connect by prior SA_ID = P_NODE_ID
   AND    P_NODE_TYPE = 'S'
   AND    END_DATE IS NULL order by LEVEL, SA_CD;
.....
   Fetch…
For (fetch count만큼) {.....
     SELECT 1, A.ORD_NO FROM TB_CORDORDSO a, TB_CORDORDINFO b
      WHERE a.SA_ID = :vcTempNodeId AND  a.SO_CANCEL_FLAG IS NULL
          AND a.ORD_NO = b.ORD_NO AND  b.ORG_ORD_NO IS NULL
          AND ROWNUM = 1;
.....}
```

개선된 SQL문

```
SELECT  /*+ use_hash(c d) */
        C.P_NODE_ID,
        C.SA_ID,
        C.START_DATE,
        C.NODE_NAME,
        C.SA_CD,
        C.LVL,
        D.CHKFLAG,
        D.ORD_NO
FROM (SELECT
             NVL(P_NODE_ID,'*') P_NODE_ID,
             SA_ID,
             START_DATE,
             NVL(NODE_NAME,'*') NODE_NAME,
             SA_CD,
             LEVEL  LVL
       FROM  TB_CCPIPRODHIER
       start with  P_NODE_TYPE = 'C'
             AND  P_NODE_ID = '&vcCustId'
             AND  END_DATE IS NULL
       connect by  prior SA_ID = P_NODE_ID
             AND  P_NODE_TYPE = 'S'
             AND  END_DATE IS NULL
     ) C,
     (SELECT  /*+ full(a) full(b) use_merge(a b) */
             a.SA_ID,
             1 CHKFLAG,
             MIN(A.ORD_NO)  ORD_NO
       FROM TB_CORDORDSO   a,
            TB_CORDORDINFO b
       WHERE a.SO_CANCEL_FLAG IS NULL
         AND a.ORD_NO = b.ORD_NO
         AND b.ORG_ORD_NO IS NULL
       GROUP BY  SA_ID
     ) D
 WHERE C.SA_ID = D.SA_ID(+)
 ORDER BY  C.LVL, C.SA_CD;
```

사례 6 : 정렬 줄이기(MAX 함수 예)

설명

실무에서 자주 나오는 SQL 유형 중 하나가 MAX ID를 찾는 것이다. 이러한 MAX 함수는 가장 큰 값을 찾기 위해 정렬 작업이 발생하고 SQL 성능은 저하된다. 이러한 정렬을 피할 수 있는 것이 INDEX_DESC 힌트이다.

문제점 및 개선 사항

변경 전 SQL은 전화번호 변동 이력을 조회하는 프로그램의 일부 SQL로 MAX 함수를 사용한 서브 쿼리의 내용은 다음과 같다. SA_ID를 조건으로 해당 SA_ID에 해당하는 가장 최근의 order 번호를 조회한다. 이 SQL은 (SA_ID, START_ORD_NO) 컬럼에 생성되어 있는 인덱스를 사용해 INDEX_DESC(내림차순으로 읽음)으로 읽은 후 order_no를 찾으면 그 값이 해당 sa_id에 대한 최대 오더 번

호가 된다. 그러면 SQL은 개선된 내용처럼 하나의 SQL로 묶일 수 있고 정렬 작업을 수행하지 않아도 된다. 이렇게 정렬 작업을 생략할 수 있는 경우가 튜닝을 하다보면 종종 나타난다. 굳이 DISTINCT 함수를 사용한다거나 UNION ALL 대신 UNION을 사용한다거나 인덱스 스캔으로 ORDER BY를 할 필요없이 데이터가 정렬되어 조회되는 경우에도 ORDER BY를 사용했거나 하는 식으로 정렬에 대한 불필요한 부담을 안고 있다. 이에 대한 인식을 바탕으로 SQL문을 유의해서 작성해야 할 것이다.

변경 전 SQL문

```
SELECT A.SVC_NO, A.NIC, B.SVC_NO_EDIT_RULE
   FROM TB_CCPISVCNOHRS A, TB_MPRDCD B
   WHERE  A.SA_ID = :vcData
   AND A.START_ORD_NO = (SELECT MAX(START_ORD_NO)
                         FROM TB_CCPISVCNOHRS
                         WHERE SA_ID = :vcData )
                         AND B.SA_CD = A.SA_CD
```

개선된 SQL문

```
SELECT  /*+index_desc(A  PK_CCPISVCNOHRS ) */
        A.SVC_NO, A.NIC, B.SVC_NO_EDIT_RULE
   INTO :vcSvcNo, :vcNic, :vcEditSvcNoRule
   FROM TB_CCPISVCNOHRS A, TB_MPRDCD B
   WHERE A.SA_ID = :vcData
   AND   B.SA_CD = A.SA_CD
   AND   ROWNUM < 2
```

사례 7 : 대용량 데이터 처리 필요시 힌트 추가

설명

많은 양의 데이터를 처리하는 SQL은 테이블을 풀 스캔함과 동시에 패러럴 프로세스가 작업을 나누어 수행해 전체 수행 시간을 단축하고 성능을 확보할 수 있도록 한다. 이러한 패러럴 프로세스의 사용은 테이블 생성시에 PARALLEL DEGREE를 지정한 경우 SQL 수행시에 패러럴하게 수행될 수 있다. 그러나 일반적으로 생성시에 PARALLEL DEGREE를 지정하지 않으므로 SQL문에 패러럴 힌트를 사용해 패러럴 쿼리가 가능하도록 한다. 앞에서도 간단히 설명한 바와 같이 패러럴 힌트를 줄 때에는 풀 스캔이 유리하며 작은 테이블을 해시 테이블로 먼저 만들어 주는 것이 중요하다.

큰 테이블에 대한 조인이 있는 경우에는 FULL 힌트와 USE_HASH 힌트, 그리고 테이블이 아주 큰 경우에 PARALLEL(테이블 또는 ALIAS명, 패러럴 DEGREE)를 주면 성능 향상에 효과적이다. 패러럴 쿼리와 마찬가지로 많은 양의 데이터를 처리하는 하나의 작업을 여러 프로세스가 나누어 작업하도록 하는 것 역시 성능을 향상시키는 방법이다. 하나의 프로그램 내에서 처리되는 쿼리 이외의 로직이 많거나 패러럴 DML이 불가능한 구조의 insert, update, delete문이 많은 경우는 패러럴 힌트 적용으로 얻는 성능만으로는 해결되지 않는다. 그러한 경우에는 작업 범위를 나누어 여러 개의 프로세스를 실행시켜 전체적인 작업을 패러럴화 하는 것이 더 효율적이다.

변경 전 SQL문

```
SELECT
    A.TABLE_ID
   ,A.SA_ID
   ,A.START_ORD_NO
   ..................
  FROM  VI_CCPIPSTN A,
        TB_MPRDCD B
 WHERE
          A.TOP_SA_ID BETWEEN :gszFromTopSaId AND :gszToTopSaId
   AND  ( A.START_REG_DATE >= TO_DATE(:gszStartDate, 'YYYYMMDDHH24MISS' )
          OR A.END_REG_DATE >= TO_DATE(:gszStartDate, 'YYYYMMDDHH24MISS' ))
   AND  A.LIFE_CYCLE <> 'C'
   AND  B.SA_CHNG_HIST_TYPE IS NOT NULL
   AND  B.SA_CD = A.SA_CD
   AND  NVL(A.TERMINAL_SUPPLY_TYPE_CD, '5' ) = '5'
   AND  A.SA_DTL_CD IS NOT NULL
ORDER BY A.TOP_SA_ID,DECODE(A.TOP_SA_ID,A.SA_ID, '1', '2' ),A.SA_ID,A.START_ORD_NO;
```

개선된 SQL문

```
SELECT /*+FULL(A) FULL(B) USE_HASH(B,A) PARALLEL(A,5) PARALLEL(B,5) */
    A.TABLE_ID
   ,A.SA_ID
   ,A.START_ORD_NO
   ..................
  FROM  VI_CCPIPSTN A,
        TB_MPRDCD B
 WHERE
          A.TOP_SA_ID BETWEEN :gszFromTopSaId AND :gszToTopSaId
   AND  ( A.START_REG_DATE >= TO_DATE(:gszStartDate, 'YYYYMMDDHH24MISS' )
          OR A.END_REG_DATE >= TO_DATE(:gszStartDate, 'YYYYMMDDHH24MISS' ))
   AND  A.LIFE_CYCLE <> 'C'
   AND  B.SA_CHNG_HIST_TYPE IS NOT NULL
   AND  B.SA_CD = A.SA_CD
   AND  NVL(A.TERMINAL_SUPPLY_TYPE_CD,'5' ) = '5'
   AND  A.SA_DTL_CD IS NOT NULL
ORDER BY A.TOP_SA_ID,DECODE(A.TOP_SA_ID,A.SA_ID, '1', '2' ),A.SA_ID,A.START_ORD_NO;
```

사례 8 : 인덱스의 잘못된 선택 예

설명

개발자가 CALLING_NODE_NO과 ERR_FLAG에 조건절이 있는 것을 보고 그것을 첫 번째 컬럼으로 한 인덱스를 선정한 경우이다. 독자들이 잘못된 SQL문 유형이라고 해서 금방 알아차리지만 상당히 많이 범하고 있는 오류이다. 인덱스를 구성하고 있는 컬럼에 조건절이 있으면 그냥 그것을 사용하는 오류를 범한다. 자세히 보면 PK_BRABHIDOMDTL이 더 범위를 좁혀 주는 컬럼에 '=' 조건으로 만들어져 있다. ERR_FLAG에 대한 조건이 범위를 좁혀 주는 조건이 아니라면 PK_BRABHIDOMDTL이 인덱스 구조상 합리적이다.

문제점 및 개선 사항

PK_BRABHIDOMDTL_01 인덱스를 사용하도록 힌트를 추가해 성능 개선을 보았다.

변경 전 SQL문

```
SELECT /*+INDEX (TB_BRABHIDOMDTL TX_BRABHIDOMDTL_01) */
     CALLING_NODE_NO||CALLING_T_NO||'  '||CALL_START_DATE
     ||'       '||CALL_START_TIME||'    '||CALL_END_TIME
     ||          ..................
  FROM TB_BRABHIDOMDTL
  WHERE PROD_CD='6148'
    AND   CALLING_DNIC_NO='4500'
    AND   INV_MONTH='200004'
    AND   ERR_FLAG IS NULL
    AND ((CALL_START_DATE ='19900101' AND
          CALL_START_TIME > '0000000' ) OR
         (CALL_START_DATE > '19900101' AND
          CALL_START_DATE <= '21000101' ))
    AND   CALLING_NODE_NO =  SUBSTR( '3364076',1,3)
    AND   CALLING_T_NO =  SUBSTR( '3364076',4,4)
ORDER BY CALL_START_DATE, CALL_START_TIME
```

〈테이블 및 인덱스 관련 정보〉

테이블 이름	인덱스 이름	인덱스 컬럼
TB_BRABHIDOMDTL	TX_BRABHIDOMDTL_01	CALLING_NODE_NO+ERR_FLAG+INV_MONTH
	PK_BRABHIDOMDTL	CALLING_NODE_NO+ERR_FLAG+INV_MONTHPK_BRABHIDOMDTLCALLING_DNIC_NO+CALLING_NODE_O+CALLING_T_NO+INV_MONTH+CALL_START_DATE+PROD_CD+CALL_START_TIME+CALLED_DNIC_NO+CALLED_NODE_NO+CALLED_T_NO+CALLING_SUB_NO+CALLED_SUB_NO+CALLING_LCN+CALLED_LCN

개선된 SQL문

```
SELECT /*+INDEX (TB_BRABHIDOMDTL TX_BRABHIDOMDTL_01) */
     CALLING_NODE_NO||CALLING_T_NO||'  '||CALL_START_DATE
     ||'       '||CALL_START_TIME||'    '||CALL_END_TIME
     ||          ..................
  FROM TB_BRABHIDOMDTL
  WHERE PROD_CD='6148'
    AND   CALLING_DNIC_NO='4500'
    AND   INV_MONTH='200004'
    AND   ERR_FLAG IS NULL
    AND ((CALL_START_DATE ='19900101' AND
          CALL_START_TIME > '0000000' ) OR
         (CALL_START_DATE > '19900101' AND
          CALL_START_DATE <= '21000101' ))
    AND   CALLING_NODE_NO =  SUBSTR( '3364076',1,3)
    AND   CALLING_T_NO =  SUBSTR( '3364076',4,4)
ORDER BY CALL_START_DATE, CALL_START_TIME
```

사례 9 : 인덱스 풀 스캔 활용

설명

이 SQL문은 먼저 테이블의 성격을 파악해야 하는데 TB_BIVG430B라는 테이블이 PROC_GRP_CD라는 컬럼을 기준으로 파티션이 되어 있다. 이 파티션된 테이블은 인덱스도 파티션에 따라 인덱스를 각각 생성할 수 있다. 즉 물리적으로 분리될 수 있으며 이를 LOCAL 인덱스라 한다. 이 테이블의 PK는 IA_ID, SA_ID, DC_PLAN_ID, PROC_GRP_CD로 조건절과 추출절에 다 있는 컬럼이다. 조건절이 범위를 많이 좁혀 주지 못한다 할지라도 굳이 테이블을 풀 스캔할 필요가 없다. 테이블보다 상대적으로 적은 용량의 인덱스 풀 스캔 기능을 활용해 블럭 READ를 할 수 있다.

문제점 및 개선 사항

개선 전 SQL에서는 TB_BIVG430B 파티션을 Index RANGE Scan 하고 있다. 인덱스 RANGE 스캔은 하나의 블럭씩 읽지만 Index Fast Full Scan은 db_file_multiblock_read_count 블럭만큼 읽기 때문에 많은 부분의 데이터를 읽을 때는 Index Fast Full Scan이 유리하다. 그러한 이유로 다음과 같이 힌트를 추가했으며 앞의 힌트를 해시 조인으로 변환했다.

변경 전 SQL문

```
SELECT /*+ INDEX_ASC  (A TX_BIVGUPCDR_MIN) */
       A.IA_ID,
       A.SA_ID,
       A.DC_PLAN_ID,
       A.PROD_CD,
       A.CHRG_ITEM_CD,
       A.USE_AMT,
       A.ROWID
  FROM TB_BIVGUPCDR A,
       ( SELECT
              DISTINCT B.IA_ID, B.SA_ID, B.DC_PLAN_ID
         FROM TB_BIVG430B B
```

```
          WHERE B.PROC_GRP_CD  = :gnProcGrpCd
            AND B.DC_PLAN_ID  = '0251'
       ) C
 WHERE A.IA_ID = C.IA_ID
   AND A.SA_ID = C.SA_ID
   AND A.DC_PLAN_ID = C.DC_PLAN_ID
```

개선된 SQL문

```
SELECT /*+ USE_HASH(C A) FULL(A) */
       A.IA_ID,
       A.SA_ID,
       A.DC_PLAN_ID,
       A.PROD_CD,
       A.CHRG_ITEM_CD,
       A.USE_AMT,
       A.ROWID
  FROM TB_BIVGUPCDR A,
       ( SELECT /*+ INDEX_FFS(B IND_BIVG430B) */
                DISTINCT B.IA_ID, B.SA_ID, B.DC_PLAN_ID
           FROM TB_BIVG430B B
          WHERE B.PROC_GRP_CD  = :gnProcGrpCd
            AND B.DC_PLAN_ID  = '0251'
       ) C
 WHERE A.IA_ID = C.IA_ID
   AND A.SA_ID = C.SA_ID
   AND A.DC_PLAN_ID = C.DC_PLAN_ID
```

사례 10 : 인덱스 추가 및 불필요 In-Line 뷰 제거

설명

조건절에 상수와의 비교 컬럼으로 인덱스가 필요하나 해당 컬럼을 leading 컬럼으로 하는 인덱스가 존재하지 않는 경우이다.

문제점 및 개선 사항

이 SQL문은 불필요한 In-Line 뷰를 포함하고 있고 TB_ CCAMPBXREPAIR 테이블의 RCV_DATE에 인덱스가 존재하지 않는다. In-Line 뷰를 조인문으로 바꾸고 RCV_DATE 컬럼으로 인덱스를 추가하면 다음의 SQL문이 가능하다.

변경 전 SQL문

```
SELECT /*+ INDEX(a TX_CCAMPBXREPAIR_01) */
       X.ORD_NO,
       X.SA_ID,
       B.SVC_NO,
       B.PROD_CD,
       C.CUST_NAME,
       X.REPAIR_FEE_FLAG
  FROM ( SELECT ORD_NO, SA_ID, REPAIR_FEE_FLAG
           FROM TB_CCAMPBXREPAIR
          WHERE RCV_DATE = :vcRcvDate
       ) X,
       TB_CORDORDINFO A,
       TB_CCPIPSTN B,
       TB_CCSTBASICINFO C
 WHERE A.ORD_NO = X.ORD_NO
   AND A.SA_ID = X.SA_ID
   AND A.SA_ID = B.SA_ID
   AND A.CUST_ID = C.CUST_ID
   AND A.COMPLETED_DATE_HH IS NULL
   AND B.END_DATE IS NULL
```

개선된 SQL문

```
SELECT
       X.ORD_NO,
       X.SA_ID,
       B.SVC_NO,
       B.PROD_CD,
       C.CUST_NAME,
       X.REPAIR_FEE_FLAG
  FROM
       TB_CCAMPBXREPAIR X,
       TB_CORDORDINFO A,
       TB_CCPIPSTN B,
       TB_CCSTBASICINFO C
 WHERE X.RCV_DATE = :vcRcvDate
   AND A.ORD_NO = X.ORD_NO
   AND A.SA_ID = X.SA_ID
   AND A.COMPLETED_DATE_HH IS NULL
   AND B.SA_ID = A.SA_ID
   AND B.END_DATE IS NULL
   AND C.CUST_ID = A.CUST_ID
```

사례 11 : 인덱스 추가 예 - 이론을 무조건 적용하지 말 것

설명

조건절에 사용된 컬럼이 사용할 인덱스가 존재하지 않아 인덱스를 추가한 예이다. 왜 이런 쉬운 예를 드는지 궁금해 하는 독자들이 있을 것이다. 하지만 이 개발자는 나름대로 튜닝에 대한 원칙을 지켰다. "한쪽 테이블에 인덱스가 있고 다른 쪽 테이블에 인덱스가 없을 때에는 인덱스가 없는 테이블을 먼저 읽어야 한다." 물론 인덱스를 생성하지 못할 경우라면 이 원칙은 고수돼야 하지만 중요한 프로그램에서 지금과 같이 절대적으로 범위를 줄이는 컬럼은 반드시 인덱스를 추가 생성해 주어야 한다. 튜닝 책을 나름대로 읽은 독자들이 오히려 너무 깊게 생각하여 단순히 인덱스만 추가하면 해결될 문제를 끙끙 앓는 경우가 의외로 많다.

문제점 및 개선 사항

TB_ BIVKBAHIST 테이블에 SA_ID 컬럼에 대한 인덱스 추가로 성능이 향상됐다.

변경 전 SQL문

```
SELECT  /*+ ordered use_nl(A B) */
        SUBSTR(A.INV_DATE,1,4)
               || '/' || SUBSTR(A.INV_DATE,5,2)
        .....................
        ,NVL(TO_CHAR(A.REG_DATE, 'YYYYMMDD'), '*' )
FROM    TB_BIVKBAHIST   A,
        TB_BIVRCHRGITEM B
WHERE   A.SA_ID         = :vcSaId           AND
        B.CHRG_ITEM_CD  = A.CHRG_ITEM_CD
```

사례 12 : 인덱스 조정 - 테이블 데이터 영역 읽지 않기

설명

WHERE절에 사용되고 있는 컬럼들이 사용하는 기존 인덱스에 컬럼 하나를 추가함으로서 테이블은 액세스하지 않고 인덱스 스캔만으로 처리가 가능하도록 하여 성능을 향상시킨 예이다.

문제점 및 개선 사항

인덱스 조정이 다음과 같이 수행됐다.

```
TB_CCPIPSTN 테이블 인덱스
  조정 전 TX_CCPIPSTN_01 : EXCH_OFC_CD+LIFE_CYCLE+ SA_CD+SVC_NO+USE_PURPOSE_CD
  조정 후 TX_CCPIPSTN_01 : EXCH_OFC_CD+LIFE_CYCLE+SA_CD+SVC_NO+USE_PURPOSE_CD+
                           CONNECT_TERMINAL_TYPE_CD
```

이와 같이 인덱스를 조정함으로써 TB_CCPIPSTN 테이블을 액세스하는 서브 쿼리는 테이블을 액세스하지 않고 인덱스 스캔만으로 처리가 됨으로써 속도가 약 10배 빨라졌다. 그러나 여기서 유의할 것은 모든 테이블의 인덱스와 쿼리에 대해 이렇게 인덱스 스캔만으로 처리할 수 있는 것은 아니라는 것이다. 테이블에 인덱스가 너무 많고 인덱스 구성 컬럼이 많으면 insert, update, delete의 속도는 상당히 저하된다는 점을 유의해야 한다. 쿼리의 수행 속도와 조정해서 얻을 수 있는 효율 정도가 높아야 하며, 조정시 추가돼야 하는 컬럼이 많을 경우에는 적용으로 인한 부담이 크므로 피하는 것이 좋다.

실행 SQL문

```
SELECT /*+ ordered use_nl(B C D) */
       NVL(D.NAME,'*'),
       NVL(B.K_no,'*'),
       .............
       NVL(B.MTwo + B.RTwo,'0')
  FROM (
        SELECT /* ordered use_nl(Z A) */
           substr(A.SVC_NO,-8,4) K_no,
           count(decode(A.SA_CD||A.CONNECT_TERMINAL_TYPE_CD,'050731',1)) MCoin,
           .............
           count(decode(A.SA_CD||A.CONNECT_TERMINAL_TYPE_CD,'050833',1)) RTwo
         FROM  (SELECT  SA_CD
                  FROM  TB_MPRDCD
                 WHERE  SA_CD IN  ('0507','0508')
                ) Z,
                TB_CCPIPSTN     A
        WHERE  A.EXCH_OFC_CD = '&vcOfcCode'
          AND  A.SA_CD = Z.SA_CD
          AND  A.LIFE_CYCLE = 'A'
        GROUP BY  substr(A.SVC_NO,-8,4)
       )  B,
       TB_CFACNXX  C,
       TB_CSYSCD   D
 WHERE C.K_NO = B.K_no
   AND C.ACPT_OFC_CD = '&vcOfcCode'
   AND D.CD     = C.SWITCH_CLASS_CD
   AND D.GRP_ID = '0132'
   AND ROWNUM <= 500;
```

사례 13 : 불필요한 IN문 사용

설명

IN을 사용해 서브 쿼리를 할 경우에는 IN 안에 있는 서브 쿼리가 UNIQUE한 값을 가지고 오기 위해 소트 작업이 일어난다. 중복 값이 없는 SQL문이라면 굳이 불필요한 소트를 유발하는 IN을 사용하지 말고 FROM 절의 IN-LINE 뷰로 유도하는 것이 좋다.

문제점 및 개선 사항

서브 쿼리로 사용된 구문을 In-Line 뷰로 전환해 조인으로 처리한 SQL은 개선된 SQL문을 통해 확인할 수 있다.

변경 전 SQL문

```
SELECT
       A.P_SA_ID,
       A.START_ORD_NO,
       SUBSTR(A.START_DATE,1,4) || '/' ||
       .....................
       C.SA_NAME,
       C.SA_CD
  FROM
       TB_CCPIPSTNA         A,
       TB_CCPIPRODHIER      B,
       TB_MPRDCD            C
 WHERE
       A.CNTRCT_END_RESV_DATE BETWEEN :vcIStartDate AND :vcIEndDate
 AND A.SVC_OFC_CD IN
         (    SELECT /*+ INDEX(TB_MSYSOFC PK_MSYSOFC) */
                     TEL_OFC_CD
```

```
 FROM  TB_MSYSOFC
WHERE  :vcISvcCdFlag = '1'
CONNECT BY PRIOR TEL_OFC_CD = P_TEL_OFC_CD
START WITH TEL_OFC_CD = :vcISvcOfcCd
UNION ALL
SELECT  TEL_OFC_CD
  FROM  TB_MSYSOFC
 WHERE  :vcISvcCdFlag = '0'
   AND TEL_OFC_CD = :vcISvcOfcCd
UNION ALL
SELECT  /*+ INDEX(TB_MSYSOFC PK_MSYSOFC) */
        TEL_OFC_CD
  FROM TB_MSYSOFC
 WHERE :vcISvcCdFlag = '2'
START WITH TEL_OFC_CD = :vcISvcOfcCd
CONNECT BY PRIOR TEL_OFC_CD = P_TEL_OFC_CD
   AND TEL_OFC_TYPE_CD IN ( '02' ,' 07' ,' 08' )
)
AND A.P_SA_ID = B.SA_ID
AND A.LIFE_CYCLE = 'A'
AND A.END_ORD_NO IS NULL
AND B.END_DATE IS NULL
AND B.SA_CD = C.SA_CD
AND A.SA_CD = '2008'
```

개선된 SQL문

```
SELECT /*+ ORDERED USE_NL(Z A B C) */
       A.P_SA_ID,
       A.START_ORD_NO,
       SUBSTR(A.START_DATE,1,4) || '/' ||
       SUBSTR(A.START_DATE,5,2) || '/' ||
       SUBSTR(A.START_DATE,7,2),
       SUBSTR(A.CNTRCT_END_RESV_DATE,1,4) || '/' ||
       SUBSTR(A.CNTRCT_END_RESV_DATE,5,2) || '/' ||
       SUBSTR(A.CNTRCT_END_RESV_DATE,7,2),
       DECODE(A.SVC_NO,NULL,'*',
        '0' ||  TO_CHAR(TO_NUMBER(SUBSTR(A.SVC_NO,1,4))) || '-' ||
       TO_CHAR(TO_NUMBER(SUBSTR(A.SVC_NO,5,4))) || '-' ||
       SUBSTR(A.SVC_NO,9,4)),
       DECODE(A.CONNECT_LINE_NO,NULL,'*',
        '0' ||  TO_CHAR(TO_NUMBER(SUBSTR(A.CONNECT_LINE_NO,1,4))) || '-' ||
       TO_CHAR(TO_NUMBER(SUBSTR(A.CONNECT_LINE_NO,5,4))) || '-' ||
       SUBSTR(A.CONNECT_LINE_NO,9,4)),
       C.SA_NAME,
       C.SA_CD
  FROM
       ( SELECT TEL_OFC_CD
           FROM TB_MSYSOFC
          WHERE :vcISvcCdFlag = '1'
         CONNECT BY PRIOR TEL_OFC_CD = P_TEL_OFC_CD
           START WITH TEL_OFC_CD = :vcISvcOfcCd
          UNION ALL
         SELECT TEL_OFC_CD
           FROM TB_MSYSOFC
          WHERE :vcISvcCdFlag = '0'
            AND TEL_OFC_CD = :vcISvcOfcCd
         UNION ALL
         SELECT TEL_OFC_CD
           FROM TB_MSYSOFC
          WHERE :vcISvcCdFlag = '2'
        CONNECT BY PRIOR TEL_OFC_CD = P_TEL_OFC_CD
            AND TEL_OFC_TYPE_CD IN ( '02' ,' 07' ,' 08' )
          START WITH TEL_OFC_CD = :vcISvcOfcCd
       ) Z,
       TB_CCPIPSTNA A,
       TB_CCPIPRODHIER B,
       TB_MPRDCD C
 WHERE A.CNTRCT_END_RESV_DATE BETWEEN :vcIStartDate AND :vcIEndDate
   AND A.SVC_OFC_CD = Z.TEL_OFC_CD
   AND A.LIFE_CYCLE = 'A'
   AND A.END_ORD_NO IS NULL
   AND A.SA_CD = '2008'
   AND B.SA_ID = A.P_SA_ID
   AND B.END_DATE IS NULL
   AND C.SA_CD = B.SA_CD
```

사례 14 : 불필요한 IN-LINE 뷰 사용

설명

In-Line 뷰를 사용하는 것은 조인에 의해 데이터를 액세스하는 작업보다 자원 소요가 더 많으므로 조인에 의해 처리될 수 있는 SQL문을 불필요하게 In-Line 뷰를 사용해 수행하지 않도록 한다. 다음의 SQL문 역시 조인으로 처리될 수 있는 테이블에 대하여 별도의 In-Line view를 사용한 예이다.

문제점 및 개선 사항

변경 전 SQL문에서 In-Line 뷰로 작성된 부분은 외부 쿼리에서 조인으로 이뤄질 수 있는 구문으로 개선된 내용처럼 SQL문을 간단히 할 수 있다.

변경 전 SQL문

```
SELECT /*+ ORDERED USE_NL(X Y) INDEX(X TK_CCPIKORNET_02)
                          INDEX(Y PK_CCPIPRODHIER) */
                    DISTINCT(Y.P_NODE_ID)
  FROM (SELECT /*+ ORDERED USE_NL(V W) INDEX(V TX_CBOMPRDCLAS_01)
                                       INDEX(W PK_MPRDCD) */
                                  V.SA_CD
                             FROM  TB_CBOMPRDCLAS V,
                                   TB_MPRDCD W
                            WHERE  V.P_SA_CD = ':g_vcBoSaCd'
                                   AND  W.SA_CD = V.SA_CD ) F,
                       TB_CCPIKORNET    X,
                       TB_CCPIPRODHIER  Y
 WHERE X.SA_CD = F.SA_CD
   AND X.SVC_OFC_CD = ':g_vcSvcOfcCd'
   AND X.SVC_NO BETWEEN ':g_vcStartSvcNo' AND ':g_vcEndSvcNo'
   AND X.END_ORD_NO  IS  NULL
   AND Y.SA_ID = X.SA_ID
   AND Y.P_NODE_TYPE = 'C'
   AND Y.END_DATE IS NULL
```

개선된 SQL문

```
SELECT /*+ ORDERED  USE_NL(A B C D) INDEX((C  TK_CCPIKORNET_02)*/
        DISTINCT(D.P_NODE_ID)
  FROM  TB_CBOMPRDCLAS A, TB_MPRDCD B,
       TB_CCPIKORNET  C, TB_CCPIPRODHIER  D
 WHERE  A.P_SA_CD = ':g_vcBoSaCd'
   AND  A.SA_CD = B.SA_CD
   AND  C.SA_CD = B.SA_CD
   AND  C.SA_ID = D.SA_ID
   AND  C.SVC_OFC_CD = ':g_vcSvcOfcCd'
   AND  C.SVC_NO  BETWEEN ':g_vcStartSvcNo' AND ':g_vcEndSvcNo'
   AND  C.END_ORD_NO  IS  NULL
   AND  D.P_NODE_TYPE = 'C'
   AND  D.END_DATE IS NULL
```

사례 15 : 불필요한 DUAL 문장

설명

데이터의 존재 유무만을 체크하는 SQL로 다음의 SQL을 사용하고 있다. DUAL도 하나의 테이블이다. 불필요하게 loop문 안에서 낭비의 요소를 가질 필요가 없다.

문제점 및 개선 사항

DUAL 테이블을 읽지 않도록 서브 쿼리를 없앴다.

변경 전 SQL문

```
SELECT 1
  INTO :vcDone
  FROM DUAL
 WHERE EXISTS(
    SELECT *
      FROM TB_CCPIDANODE
     WHERE NODE_ID = :vcSaId
       AND NODE_TYPE = 'S'
       AND END_ORD_NO IS NULL
       AND END_DATE IS NULL
       AND ROWNUM = 1)
```

개선된 SQL문

```
SELECT 1
  INTO :vcDone
  FROM TB_CCPIDANODE
 WHERE NODE_ID = :vcSaId
   AND NODE_TYPE = 'S'
   AND END_ORD_NO IS NULL
   AND END_DATE IS NULL
   AND ROWNUM = 1
```

사례 16 : 어레이 프로세싱

설명

데이터를 select하거나 insert, update, delete할 때 발생시마다 처리하는 것보다 특정 건의 어레이(array) 단위로 처리하는 것이 오라클과 응용 프로그램 사이의 데이터를 주고받는 데 발생하는 부하를 줄여주므로 효율적이다. 다음의 SQL문은 loop를 통해 건건이 데이터를 업데이트하던 구조에서 어레이 프로세싱으로 전환해 상당히 성능을 향상시킨 경우이다. 변수에 array 단위 만큼 여러 행의 로우(row)를 담은 후 해당 어레이 처리 단위만큼 업데이트문을 수행하도록 다음과 같이 기술한다.

어레이 프로세싱은 어레이 처리 단위를 무조건 크게 한다고 해서 성능이 향상되는 것은 아니므로 어레이 처리 단위를 적정하게 하여 SQL문을 수행하는 것이 바람직하다.

업데이트시의 어레이 프로세싱 예

```
EXEC SQL FOR :nFetchCnt
UPDATE CDBS151T
SET   SAID1 = DECODE(:gvcGubun, '01', LTRIM(:gccpilease.vcSaId), SAID1),
      SAID2 = DECODE(:gvcGubun, '02', LTRIM(:gccpilease.vcSaId), SAID2),
      SAID3 = DECODE(:gvcGubun, '03', LTRIM(:gccpilease.vcSaId), SAID3),
      ORDNO = DECODE(LEAST(:gccpilease.vcStartOrdNo, '00000000001'),
                   '00000000001', LTRIM(:gccpilease.vcStartOrdNo),
            DECODE(LEAST(:gccpilease.vcEndOrdNo, '00000000001'),
                   '00000000001', LTRIM(:gccpilease.vcEndOrdNo), ORDNO)),
      INSTL_CHNG_TYPE_CD = :gcInstlType,
      SACD = LTRIM(:gccpilease.vcProdCd)
WHERE ROWID = :gvcRowId
  AND SAID1 IS NULL;
```

사례 17 : GROUP BY 시 성능 개선 예

설명

Group by를 먼저 하고 조인을 하느냐 또는 조인을 먼저 하고 Group by를 하느냐의 선택 사례이다.

문제점 및 개선 사항

TB_BPMBOLDPMT TABLE의 ROW 수가 약 1300만 건, TB_BPMBAPLIMPOS TABLE의 ROW 수가 약 4000건인데 TB_BPMBOLDPMT TABLE은 GROUP BY 처리를 하더라도 건수가 크게 줄지 않기 때문에 TB_BPMBAPLIMPOS와 조인하는 과정에서 많은 로우(row)가 배제되고 최종 select되는 것은 몇천 건 이내가 된다. 이것은 상당히 비효율적이다. TB_BPMBOLDPMT를 모두 대상으로 GROUP BY 처리할 것이 아니라 TB_BPMBAPLIMPOS와의 JOIN을 통해 대상 건수를 줄여 GROUP BY 처리한 후 이것을 다시 TB_BPMBAPLIMPOS와 조인하는 방식으로 처리한다. 이런 방식으로 SQL을 수정했을 경우 처리시간이 2~4시간에서 2분으로 획기적으로 단축됐다.

변경 전 SQL문

```
SELECT  /*+ ordered use_hash( IMP PMT ) */
            PROC_DATE,
            CORR_SVC_CLASS_CD,
            CORR_SERVER_ID,
            CORR_INV_NO,
            PMT_OFC_CD,
            RCV_DATA_SEQ_NO,
            PMT_DATE,
            FINANCE_DATE,
            TOT_MONTH,
            NVL(PMT_SETTLE_AGNC_CD,'  '),
            NVL(CORR_PMT_AMT,0 ),
            NVL(CORR_OCR_TYPE_CD,'0'),
            NVL(APPLY_IMPOS_CORR_CD,'0'),
            CORR_INV_DATE,
            DECODE(old_inv_date,NULL,'X','0'),
            NVL(old_ia_id,'  ')
FROM    TB_BPMBAPLIMPOS IMP,
      ( SELECT  /*+ ordered use_nl( IMP PMT )
                  index ( PMT pk_bpmboldpmt ) */
                PMT.INV_DATE                old_inv_date,
                PMT.INV_NO                  old_inv_no,
                NVL(MAX(IMP.IA_ID),'  ')    old_ia_id
        FROM    TB_BPMBAPLIMPOS IMP,
                TB_BPMBOLDPMT   PMT
        WHERE   IMP.APPLY_IMPOS_TYPE_CD   != '04'
        AND     IMP.APPLY_IMPOS_STATUS_CD  = '01'
        AND     IMP.CORR_INV_DATE          = PMT.INV_DATE
        AND     IMP.CORR_INV_NO            = PMT.INV_NO
        GROUP BY PMT.INV_DATE,
                 PMT.INV_NO ) PMT
WHERE   IMP.APPLY_IMPOS_TYPE_CD   != '04'
AND     IMP.APPLY_IMPOS_STATUS_CD  = '01'
AND     IMP.CORR_INV_DATE          = old_inv_date(+)
AND     IMP.CORR_INV_NO            = old_inv_no(+) ;
```

```
COST OPERATION                      OPTIONS              OBJECT_NAME
------ ---------------------------- -------------------- ------------------
   847 SELECT STATEMENT
   847   HASH JOIN                   OUTER
     4   TABLE ACCESS                FULL                 TB_BPMBAPLIMPOS
   826   VIEW
   826   SORT                        GROUP BY
         PARTITION                   CONCATENATED
   826   TABLE ACCESS                BY LOCAL INDEX ROWID TB_BPMBOLDPMT
    26   INDEX                       FULL SCAN            PK_BPMBOLDPMT
```

개선된 SQL문

```
SELECT  /*+ ordered use_hash( IMP PMT ) */
            PROC_DATE,
            CORR_SVC_CLASS_CD,
            CORR_SERVER_ID,
            CORR_INV_NO,
            PMT_OFC_CD,
            RCV_DATA_SEQ_NO,
            PMT_DATE,
            FINANCE_DATE,
            TOT_MONTH,
            NVL(PMT_SETTLE_AGNC_CD,'  '),
            NVL(CORR_PMT_AMT,0 ),
            NVL(CORR_OCR_TYPE_CD,'0'),
            NVL(APPLY_IMPOS_CORR_CD,'0'),
            CORR_INV_DATE,
            DECODE(old_inv_date,NULL,'X','0'),
            NVL(old_ia_id,'  ')
FROM    TB_BPMBAPLIMPOS IMP,
      ( SELECT  /*+ full(TB_BPMBOLDPMT) */
                OLDPMT.INV_DATE                   old_inv_date,
                OLDPMT.INV_NO                     old_inv_no,
                MIN(OLDPMT.IA_ID)                 old_ia_id
        FROM  TB_BPMBAPLIMPOS LIMPOS,
              TB_BPMBOLDPMT OLDPMT
        WHERE LIMPOS.APPLY_IMPOS_TYPE_CD   != '04'
          AND LIMPOS.APPLY_IMPOS_STATUS_CD  = '01'
          AND LIMPOS.CORR_INV_DATE = OLDPMT.inv_date(+)
          AND LIMPOS.CORR_INV_NO   = OLDPMT.inv_no(+)
        GROUP BY OLDPMT.INV_DATE,
                 OLDPMT.INV_NO ) PMT
WHERE   IMP.APPLY_IMPOS_TYPE_CD   != '04'
AND     IMP.APPLY_IMPOS_STATUS_CD  = '01'
AND     IMP.CORR_INV_DATE          = old_inv_date(+)
AND     IMP.CORR_INV_NO            = old_inv_no(+) ;
```

```
COST OPERATION                      OPTIONS              OBJECT_NAME
------ ---------------------------- -------------------- ------------------
    79 SELECT STATEMENT
    79   HASH JOIN                   OUTER
     4     TABLE ACCESS              FULL                 TB_BPMBAPLIMPOS
    74     VIEW
    74       SORT                    GROUP BY
    19         NESTED LOOPS          OUTER
     4           TABLE ACCESS        FULL                 TB_BPMBAPLIMPOS
                 PARTITION           SINGLE
     3             TABLE ACCESS      BY LOCAL INDEX ROWID TB_BPMBOLDPMT
     2               INDEX           RANGE SCAN           PK_BPMBOLDPMT
```

사례 18 : 사소하지만 성능에 무시될 수 없는 사항

UNION과 UNION ALL

복합 SQL문에서 UNION과 UNION ALL의 차이를 제대로 인식하지 못하고 UNION ALL을 써도 무방한 SQL문에 대해 UNION을 사용하는 SQL문이 종종 나타난다. UNION은 각 쿼리문에 의해 리턴된 로우들 중 중복된 로우를 배제하고 최종 리턴하기 위해 소팅 작업이 발생한다. UNION ALL은 각 쿼리문에 의해 리턴된 로우들을 함께 리턴하는 것으로 작업이 끝나므로 UNION 작업에 비하여 훨씬 성능이 좋다. SQL문이 목표로 하는 로우가 반드시 중복을 배제하고 리턴해야 하는 경우에만 UNION을 사용하고 중복 로우가 존재해도 문제되지 않은 경우에는 UNION을 사용한다.

NVL 함수의 사용

Null Value Return에 대한 두려움으로 인해 개발자들은 NVL 함수를 남용하는 사례가 많다. 우선 조건절에 사용되는 NVL 함수의 경우에 해당 컬럼이 null을 포함하지 않는 컬럼이 명확할 때도(Not Null Constraints) NVL 함수를 사용하는 경우가 있다. 해당 컬럼이 not null임이 명확할 때에는 NVL 함수를 사용하지 않도록 한다.

또한 select절에서 NVL 함수를 모든 컬럼에 사용한 경우가 있다. 이것은 null value fetch error(ORA-1405)를 방지하고자 하는 의도에서 사용된 것이라고 한다. 이것은 null value error가 발생되지 않도록 하기 위해 사용되는 precompile option에 대한 공유가 안 되어 발생한 것이라고 볼 수 있다. precompile option UNSAFE_NULL=yes 옵션을 precompile 옵션에 추가해 컴파일하면 null value fetch error를 유발하지 않는다. 단, PL/SQL 구문에는 이 옵션이 적용되지 않으므로 PL/SQL 구문에 대해서는 indicator 변수를 사용한다.

그리고 SUM 함수 사용시 SUM(NVL(필드명,0))과 같이 필드명이 null인 경우에 0을 sum하도록 기술했으나 Null value가 sum하는 데이터 set에 포함돼도 null은 sum 대상에서 제외되고 정상으로 sum되므로 nvl 함수를 사용하지 않아도 된다. 단, sum한 결과가 null일 경우 0를 얻고자 하는 경우에는 NVL(SUM(필드명), 0)과 같이 기술해 사용한다.

입력 인자 value에 따라 다른 조건식

입력 인자 v1의 값이 '1'이면 필드명1이 value1과 같고 v1의 값이 '2'이면 필드명2가 value1과 같은 데이터를 조회하는 경우에 다음과 같이 SQL을 구사하고 있다.

```
Select …… where ((:v1 = '1' and 필드명1 = value1) or (:v1 = '2' and 필드명2 = value1))
```

이렇게 함으로써 SQL문을 별도로 분리하여 작성하지 않고 하나의 SQL문으로 처리가 가능하기 때문에 자주 사용된다. 그러나 SQL문의 복잡 정도나 조건의 적고 많음의 정도에 따라 이러한 유형의 SQL은 사용여부를 결정해야 한다. 우선은 입력 인자를 If문 조건으로 하여 SQL문을 분리하는 것이 가장 효율적이나 그렇게 해서 생성되는 SQL문이 아주 많아진다거나 SQL문 자체가 복잡한 경우에 위의 유형을 사용하는 것이 바람직할 수 있다.

불필요한 dual의 사용

C로 처리 가능한 로직들 중 여러 부문이 SQL로 작성되고 있다. C로 처리 가능한 로직은 select…dual 구조를 사용하지 않고 C로 로직 구사를 할 수 있도록 한다.

다이내믹 메쏘드 3의 사용

전체적인 SQL의 구조는 같지만 입력 조건에 따라 액세스하는 테이블이 달라지거나 입력 값이 달라지는 경우 상수 값을 SQL문에 대입하는 형태의 메쏘드 1 대신 value를 인자로 전달하는 메쏘드 3를 사용해 SQL문의 전체적인 파싱 시간을 단축하고 효율적인 공유 메모리 사용이 가능하도록 한다.

대장정을 마치며

이번을 마지막으로 5회에 걸쳐 '데이터베이스 : 오라클 애플리케이션 튜닝'에 대한 기고를 마치겠다. 솔직히 좀더 많은 시간과 노력을 들였어야 하는데 아쉬운 생각이 많이 든다. 어쨌든 많은 개발자들과 접촉하면서 해주고 싶었던 얘기를 내 나름대로 마음껏 펼쳐 보았다. 그동안 관심을 가져준 독자들에게 감사를 전한다.