Advanced SQL

실무에서 바로 사용하는 SQL 실전 사례

Edition 3.0
May 2018

Author : Chongha Ryu

목 차

1. 기본적인 SELECT 명령문 작성

1. EMPLOYEES 테이블에서, 이름(FIRST_NAME)이 Peter인 사원을 검색하시오.

검색 결과

```
EMPLOYEE_ID FIRST_NAME           LAST_NAME                 SALARY
----------- -------------------- ------------------------- ----------
        144 Peter                Vargas                      2500

1 row selected.
```

문자열을 비교할 때, 영문자는 대소문자를 구분한다. 때문에 저장된 문자가 대소문자 규칙을 가지고 있지 않다면 원하는 결과를 제대로 검색할 수 없을 수 있다. 한글이 저장된 컬럼은 상관없겠으나 영문자를 저장할 때는 대소문자 규칙을 지정하는 것이 검색을 편하게 할 수 있으며, 그렇지 않다면 UPPER, LOWER, INITCAP 함수를 활용한다.

답안 1. 저장된 문자열 일치

```
SQL> SELECT employee_id, first_name, last_name, salary
       FROM employees
      WHERE first_name = 'Peter' ;
```

답안 2. 비교 값 가공

```
SQL> SELECT employee_id, first_name, last_name, salary
       FROM employees
      WHERE first_name = INITCAP('PETER') ;
```

답안 3. 컬럼 가공

```
SQL> SELECT employee_id, first_name, last_name, salary
       FROM employees
      WHERE UPPER(first_name) = 'PETER' ;
```

답안 4. 비교 값, 컬럼 동시 가공

```
SQL> SELECT employee_id, first_name, last_name, salary
       FROM employees
      WHERE UPPER(first_name) = UPPER('peter') ;
```

2. EMP 테이블에서, 이름(ENAME)을 알파벳 순으로 정렬했을 때 'JONES' 이후에 나올 수 있는 사원 정보를 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7839 KING             5000         10
      7654 MARTIN           1250         30
      7934 MILLER           1300         10
      7788 SCOTT            3000         20
      7369 SMITH             800         20
      7844 TURNER           1500         30
      7521 WARD             1250         30

7 rows selected.
```

문자열에서 대소 비교가 가능한 것을 확인한다. 영문자에서는 의미가 없을 수도 있지만 한글이 저장된 컬럼에서 초성을 검색해야 한다면 대소 비교를 통해 원하는 검색을 할 수 있다.

답안. 문자열의 대소 비교

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE ename > 'JONES' ;
```

추가 실습

```
SQL> INSERT INTO emp (empno, ename, deptno)
     VALUES (1111, '류청하', 30) ;

SQL> SELECT empno, ename, deptno
       FROM emp
      WHERE ename BETWEEN '라' AND '마' ;
     EMPNO ENAME          DEPTNO
---------- ---------- ----------
      1111 류청하             30

SQL> ROLLBACK ;
```

3. EMP 테이블에서, 이름(ENAME)이 'TT'로 끝나는 사원을 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7788 SCOTT            3000         20

1 row selected.
```

문자의 패턴을 비교해야 할 때 LIKE 비교를 한다. 이때 사용되는 %를 앞에 붙이면 결과는 검색될 수 있으나 인덱스를 제대로 사용하지 못할 수 있다. INDEX RANGE SCAN의 실행 계획을 사용할 수 있도록 하려면 함수 기반 인덱스를 활용한다.

답안 1. LIKE 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE ename LIKE '%TT' ;
```

답안 2. 인덱스를 사용하는 조건절 (함수 기반 인덱스 활용)

```
SQL> SELECT ename, REVERSE(ename)
       FROM emp ;
ENAME      REVERSE(EN
---------- ----------
SCOTT      TTOCS
JONES      SENOJ
SMITH      HTIMS
ADAMS      SMADA
ALLEN      NELLA
WARD       DRAW
MARTIN     NITRAM
BLAKE      EKALB
CLARK      KRALC
TURNER     RENRUT
JAMES      SEMAJ
FORD       DROF
MILLER     RELLIM
KING       GNIK

14 rows selected.

SQL> CREATE INDEX ename_fbi ON emp(REVERSE(ename)) ;
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE REVERSE(ename) LIKE 'TT%' ;

SQL> DROP INDEX ename_fbi ;
```

4. EMP 테이블에서, 이름(ENAME)이 'S' 또는 'A'로 시작하는 사원을 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7788 SCOTT            3000         20
      7369 SMITH             800         20
      7876 ADAMS            1100         20
      7499 ALLEN            1600         30

4 rows selected.
```

LIKE 비교는 간단한 패턴을 비교할 때 사용한다. 만약 복잡한 패턴을 비교해야 한다면 정규식을 활용하는 것이 훨씬 쉽게 원하는 결과를 검색할 수 있다.

답안 1. LIKE 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE ename LIKE 'S%'
         OR ename LIKE 'A%' ;
```

답안 2. 정규식 사용 (REGEXP_LIKE)

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE REGEXP_LIKE(ename, '^(S¦A)') ;
```

5. EMP 테이블에서, 입사일자(HIREDATE)가 '1980/12/17'인 사원 정보를 검색하시오.

검색 결과

```
     EMPNO ENAME      HIREDATE     DEPTNO
---------- ---------- -------- ----------
      7369 SMITH      80/12/17         20

1 row selected.
```

오라클은 DB는 DATE 타입을 저장할 때 시, 분, 초를 함께 저장한다. 그리고 그 내용을 검색할 때는 NLS_DATE_FORMAT의 설정에 따라 일부분만 검색될 수 있다. 즉, 검색된 날짜 이외에도 시간이 함께 저장되어 있다면 날짜 비교만으로는 검색 결과를 보장할 수 없게 된다. 이를 보완하기 위한 방법을 확인한다.

주의 사항

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate = TO_DATE('1980/12/17','YYYY/MM/DD') ;
no rows selected

SQL> SELECT empno, ename, TO_CHAR(hiredate,'YYYY/MM/DD HH24:MI:SS') AS HIREDATE
       FROM emp
      WHERE empno = 7369 ;
     EMPNO ENAME      HIREDATE
---------- ---------- -------------------
      7369 SMITH      1980/12/17 13:20:19
```

답안 1. 저장된 날짜 값 일치

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate = TO_DATE('1980/12/17 13:20:19','YYYY/MM/DD HH24:MI:SS') ;
```

답안 2. 컬럼 가공

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE TO_CHAR(hiredate,'YYYY/MM/DD') = '1980/12/17' ;
```

답안 3. BETWEEN 사용

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate BETWEEN TO_DATE('1980/12/17 00:00:00','YYYY/MM/DD HH24:MI:SS')
                         AND TO_DATE('1980/12/17 23:59:59','YYYY/MM/DD HH24:MI:SS') ;
```

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate BETWEEN TO_DATE('1980/12/17','YYYY/MM/DD')
                         AND TO_DATE('1980/12/18','YYYY/MM/DD') - 1/86400 ;
```

6. EMP 테이블에서, 1981년에 입사한 사원 정보를 검색하시오. (HIREDATE : 입사일자)

검색 결과

```
     EMPNO ENAME      HIREDATE     DEPTNO
---------- ---------- -------- ----------
      7499 ALLEN      81/02/20         30
      7521 WARD       81/02/22         30
      7566 JONES      81/04/02         20
      7698 BLAKE      81/05/01         30
      7782 CLARK      81/06/09         10
      7844 TURNER     81/09/08         30
      7654 MARTIN     81/09/28         30
      7839 KING       81/11/17         10
      7900 JAMES      81/12/03         30
      7902 FORD       81/12/03         20

10 rows selected.
```

DATE 타입의 컬럼에서 범위 검색이 필요한 상황이면 컬럼의 가공보다는 인덱스를 활용할 수 있도록 BETWEEN 조건식을 사용한다.

답안 1. 컬럼 가공

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE TO_CHAR(hiredate, 'YYYY') = '1981' ;
```

답안 2. LIKE 사용

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate LIKE '81%' ;
```

답안 3. BETWEEN 사용

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate BETWEEN TO_DATE('19810101','YYYYMMDD')
                         AND TO_DATE('19811231','YYYYMMDD') ;
```

7. EMP 테이블에서, 4월에 입사한 사원을 검색하시오. (HIREDATE : 입사일자)

검색 결과

```
     EMPNO ENAME      HIREDATE     DEPTNO
---------- ---------- -------- ----------
      7788 SCOTT      87/04/19         20
      7566 JONES      81/04/02         20

2 rows selected.
```

DATE 타입에서 시작 연도를 한정 지을 수 없다면, 컬럼의 가공은 필요한다. 하지만 인덱스를 사용하지 못한다는 단점이 있으므로 테이블 설계시 연, 월, 일을 따로 저장하거나 날짜와 연관된 모든 형식 모델을 갖는 별도의 테이블(또는 집합)을 활용한다.

답안 1. 컬럼 가공

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE TO_CHAR(hiredate,'MM') = '04' ;
```

답안 2. LIKE 사용

```
SQL> SELECT empno, ename, hiredate, deptno
       FROM emp
      WHERE hiredate LIKE '___04___';
```

추가 실습

```
SQL> CREATE TABLE tab_date
     AS
     SELECT dt
           ,TO_CHAR(dt,'YYYY') AS YYYY
           ,TO_CHAR(dt,'MM')   AS MM
           ,TO_CHAR(dt,'DD')   AS DD
           ,TO_CHAR(dt,'DAY','NLS_DATE_LANGUAGE=AMERICAN') AS DAY
       FROM (SELECT TO_DATE('1980/01/01','YYYY/MM/DD') + LEVEL - 1 AS DT
               FROM dual
             CONNECT BY LEVEL <= 10000) ;

SQL> SELECT *
       FROM tab_date ;
DT       YYYY MM DD DAY
-------- ---- -- -- ---------------
80/01/01 1980 01 01 TUESDAY
80/01/02 1980 01 02 WEDNESDAY
80/01/03 1980 01 03 THURSDAY
80/01/04 1980 01 04 FRIDAY
80/01/05 1980 01 05 SATURDAY
...
07/05/16 2007 05 16 WEDNESDAY
07/05/17 2007 05 17 THURSDAY
07/05/18 2007 05 18 FRIDAY

10000 rows selected.

SQL> SELECT e.empno, e.ename, e.hiredate, e.deptno
       FROM emp e, tab_date d
      WHERE e.hiredate = d.dt
        AND d.mm       = '04' ;
     EMPNO ENAME      HIREDATE     DEPTNO
---------- ---------- -------- ----------
      7566 JONES      81/04/02         20
      7788 SCOTT      87/04/19         20

2 rows selected.

SQL> DROP TABLE tab_date PURGE ;
```

8. EMP 테이블에서, 급여(SAL)보다 커미션(COMM)을 많이 받는 사원을 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL       COMM     DEPTNO
---------- ---------- ---------- ---------- ----------
      7654 MARTIN           1250       1400         30

1 row selected.
```

하나의 테이블에서 두 컬럼의 조건 비교가 가능하다. 단, 같은 행에 있는 컬럼 값을 기준으로 조건이 비교되므로 다른 행의 값과 비교가 필요하면 조인이나 서브 쿼리를 사용한다.

답안.

```
SQL> SELECT empno, ename, sal, comm, deptno
       FROM emp
      WHERE sal < comm ;
```

Q. 'JONES' 보다 급여를 더 많이 받는 사원을 검색하시오.

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE sal > (SELECT sal
                     FROM emp
                    WHERE ename = 'JONES') ;
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7788 SCOTT            3000         20
      7902 FORD             3000         20
      7839 KING             5000         10

3 rows selected.
```

9. EMP 테이블에서, 커미션(COMM)을 받지 않는 모든 사원을 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL       COMM     DEPTNO
---------- ---------- ---------- ---------- ----------
      7788 SCOTT            3000                    20
      7566 JONES            2975                    20
      7369 SMITH             800                    20
      7876 ADAMS            1100                    20
      7698 BLAKE            2850                    30
      7782 CLARK            2450                    10
      7844 TURNER           1500          0         30
      7900 JAMES             950                    30
      7902 FORD             3000                    20
      7934 MILLER           1300                    10
      7839 KING             5000                    10

11 rows selected.
```

NULL은 필요한 값이지만, 조건 비교는 IS NULL, IS NOT NULL만 가능하다. 인덱스를 사용하지 못할 수도 있고, 복잡한 조건식을 만들 때는 오히려 불편할 수도 있다. 때문에 컬럼의 NULL이 반드시 필요한 값인지를 확인하고, 불필요한 NULL은 저장되지 않도록 한다.

답안 1. NVL 함수 사용

```
SQL> SELECT empno, ename, sal, comm, deptno
       FROM emp
      WHERE NVL(comm,0) = 0 ;
```

답안 2. IS NULL 사용

```
SQL> SELECT empno, ename, sal, comm, deptno
       FROM emp
      WHERE comm = 0
         OR comm IS NULL ;
```

추가 실습

```
SQL> UPDATE emp
     SET comm = 0
     WHERE comm IS NULL ;
SQL> SELECT empno, ename, sal, comm, deptno
       FROM emp
      WHERE comm = 0 ;

SQL> ROLLBACK ;
```

10. EMP 테이블에서, 급여(SAL)가 2000 보다 작고, 3000 보다 많은 사원을 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7369 SMITH             800         20
      7876 ADAMS            1100         20
      7499 ALLEN            1600         30
      7521 WARD             1250         30
      7654 MARTIN           1250         30
      7844 TURNER           1500         30
      7900 JAMES             950         30
      7934 MILLER           1300         10
      7839 KING             5000         10

9 rows selected.
```

배타적인 조건을 비교할 때 인덱스를 사용하지 못하는 경우가 존재한다. 결과는 검색되더라도 성능에 문제가 생긴다면 UNION ALL을 활용한다.

답안 1. BETWEEN 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE sal NOT BETWEEN 2000 AND 3000 ;

SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE sal < 2000
         OR sal > 3000 ;
```

답안 2. UNION ALL 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE sal < 2000
      UNION ALL
     SELECT empno, ename, sal, deptno
       FROM emp
      WHERE sal > 3000 ;
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7369 SMITH             800         20
      7900 JAMES             950         30
      7876 ADAMS            1100         20
      7521 WARD             1250         30
      7654 MARTIN           1250         30
...

9 rows selected.
```

11. EMP 테이블에서, 부서번호(DEPTNO)가 10,30 이면서 급여(SAL)가 2000 이상인 사원 정보를 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7782 CLARK            2450         10
      7698 BLAKE            2850         30
      7839 KING             5000         10

3 rows selected.
```

조건절의 둘 이상의 조건식은 AND, OR를 이용하여 연결한다. 이때 AND, OR가 함께 사용된다면 우선순위에 따라 AND 연산이 먼저 처리된다. 우선순위를 조정하려면 ( )를 이용한다.

답안.

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE deptno IN (10,30)
        AND sal    >= 2000 ;
```

주의 사항

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE deptno = 10
         OR deptno = 30
        AND sal   >= 2000 ;
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7698 BLAKE            2850         30
      7782 CLARK            2450         10
      7934 MILLER           1300         10
      7839 KING             5000         10

4 rows selected.
```

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE (deptno = 10
         OR deptno = 30)
        AND sal   >= 2000 ;
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7782 CLARK            2450         10
      7698 BLAKE            2850         30
      7839 KING             5000         10

3 rows selected.
```

Q. 다음 문장의 검색 결과는?

```
SQL> SELECT empno, ename, hiredate, sal, deptno
       FROM emp
      WHERE sal > 1500
         OR TO_CHAR(hiredate,'YYYY') = '1981'
        AND deptno = 20 ;
```

12. EMP 테이블에서, 연봉(SAL*12)이 35000 이상인 사원 정보를 검색하시오.
단, ANN_SAL의 별칭을 이용한 조건식 이용 (WHERE ANN_SAL >= 35000)

검색 결과

```
     EMPNO ENAME         ANN_SAL     DEPTNO
---------- ---------- ---------- ----------
      7788 SCOTT           36000         20
      7566 JONES           35700         20
      7902 FORD            36000         20
      7839 KING            60000         10

4 rows selected.
```

컬럼의 별칭은 ORDER BY 절에서만 사용 가능하다. 만약 다른 절에서 컬럼의 별칭을 이용하여 한다면 Inline View를 사용한다.

주의 사항

```
SQL> SELECT empno, ename, sal * 12 AS ANN_SAL, deptno
       FROM emp
      WHERE ann_sal >= 35000 ;
ERROR at line 3:
ORA-00904: "ANN_SAL": invalid identifier
```

답안 1. Inline View 사용

```
SQL> SELECT *
       FROM ( SELECT empno, ename, sal*12 AS ANN_SAL, deptno
                FROM emp )
      WHERE ann_sal >= 35000 ;
```

추가 실습. 인덱스를 사용하는 조건절

```
SQL> SELECT empno, ename, sal*12 AS ANN_SAL, deptno
       FROM emp
      WHERE sal >= 35000/12 ;
     EMPNO ENAME         ANN_SAL     DEPTNO
---------- ---------- ---------- ----------
      7566 JONES           35700         20
      7788 SCOTT           36000         20
      7902 FORD            36000         20
      7839 KING            60000         10

4 rows selected.
```

- 컬럼의 별칭은 ORDER BY 절에서만 사용 가능

```
SQL> SELECT deptno AS deptid, SUM(sal) AS sum_sal
       FROM emp
     GROUP BY deptid ;
ERROR at line 3:
ORA-00904: "DEPTID": invalid identifier

SQL> SELECT deptno AS deptid, SUM(sal) AS sum_sal
       FROM emp
     GROUP BY deptno
     HAVING sum_sal > 9000 ;
ERROR at line 4:
ORA-00904: "SUM_SAL": invalid identifier

SQL> SELECT empno, ename, sal, deptno AS deptid
       FROM emp
     ORDER BY deptid ;
     EMPNO ENAME             SAL     DEPTID
---------- ---------- ---------- ----------
      7934 MILLER           1300         10
      7782 CLARK            2450         10
      7839 KING             5000         10
      7902 FORD             3000         20
      7788 SCOTT            3000         20
      7876 ADAMS            1100         20
      7566 JONES            2975         20
      7369 SMITH             800         20
      7900 JAMES             950         30
      7844 TURNER           1500         30
      7698 BLAKE            2850         30
      7521 WARD             1250         30
      7499 ALLEN            1600         30
      7654 MARTIN           1250         30

14 rows selected.
```

13. EMP 테이블에서, 커미션(COMM)을 기준으로 내림차순 정렬된 결과를 검색하시오.
단, 커미션이 NULL인 행은 마지막에 검색되도록 한다.

검색 결과

```
     EMPNO ENAME             SAL       COMM
---------- ---------- ---------- ----------
      7654 MARTIN           1250       1400
      7521 WARD             1250        500
      7499 ALLEN            1600        300
      7844 TURNER           1500          0
      7839 KING             5000
      7788 SCOTT            3000
      7782 CLARK            2450
      7900 JAMES             950
      7902 FORD             3000
      7934 MILLER           1300
      7876 ADAMS            1100
      7369 SMITH             800
      7566 JONES            2975
      7698 BLAKE            2850

14 rows selected.
```

NULL은 정렬 수행 시 가장 큰 값으로 설정된다. 하지만 실제로 대소 비교가 가능한 것은 아니다. 때문에 정렬을 수행할 경우 NULL에 순서를 조정하려면 NVL 함수를 이용하거나 NULLS FIRST¦LAST를 사용한다.

답안 1. NVL 함수 사용

```
SQL> SELECT empno, ename, sal, comm
       FROM emp
     ORDER BY NVL(comm, -1) DESC ;
```

답안 2. NULLS FIRST¦LAST 사용

```
SQL> SELECT empno, ename, sal, comm
       FROM emp
     ORDER BY comm DESC NULLS LAST ;
```

추가 실습

```
SQL> SELECT empno, ename, sal, comm
       FROM emp
     ORDER BY comm ASC NULLS FIRST ;
```

14. EMP 테이블에서, 매니저(MGR)가 없는 사원은 'NO MANAGER'의 문자를 다음과 같이 검색하시오.

검색 결과

```
     EMPNO ENAME      JOB             MANAGER
---------- ---------- --------------- ---------------
      7788 SCOTT      ANALYST         7566
      7566 JONES      MANAGER         7839
      7369 SMITH      CLERK           7902
      7876 ADAMS      CLERK           7788
      7499 ALLEN      SALESMAN        7698
      7521 WARD       SALESMAN        7698
      7654 MARTIN     SALESMAN        7698
      7698 BLAKE      MANAGER         7839
      7782 CLARK      MANAGER         7839
      7844 TURNER     SALESMAN        7698
      7900 JAMES      CLERK           7698
      7902 FORD       ANALYST         7566
      7934 MILLER     CLERK           7782
      7839 KING       PRESIDENT       NO Manager

14 rows selected.
```

하나의 컬럼의 모든 행은 동일한 데이터 타입을 가져야 검색될 수 있다. 때문에 'NO Manager' 문자를 검색하려면 형 변환이 필요하고 그 방법을 확인한다.

주의 사항.

```
SQL> SELECT empno, ename, job, NVL(mgr, 'NO Manager') AS MANAGER
       FROM emp ;
ERROR at line 1:
ORA-01722: invalid number

SQL> SELECT empno, ename, job, CASE WHEN mgr IS NULL
                                    THEN 'NO Manager' ELSE mgr END AS MANAGER
       FROM emp ;
ERROR at line 1:
ORA-00932: inconsistent datatypes: expected CHAR got NUMBER
```

답안. 명시적, 암시적 형 변환

```
SQL> SELECT empno, ename, job, NVL(TO_CHAR(mgr), 'NO Manager') AS MANAGER
       FROM emp ;

SQL> SELECT empno, ename, job, CASE WHEN mgr IS NULL
                                    THEN 'NO Manager' ELSE TO_CHAR(mgr) END AS MANAGER
       FROM emp ;

SQL> SELECT empno, ename, job, DECODE(mgr, NULL, 'NO Manager', mgr) AS MANAGER
       FROM emp ;
```

2. INDEX를 사용하는 SQL

INDEX를 사용하는 SQL

SQL 명령문을 실행하는 도중 인덱스의 사용이 필요한 경우가 있다. 단, 인덱스는 존재한다고 사용되는 것이 아니라 인덱스를 사용할 수 있도록 작성된 SQL이 필요하다. 때문에 우선 인덱스의 사용이 가능한 SQL의 문장 작성 방법을 확인한다.

테이블은 업무에 필요한 정규화 된 데이터를 저장한다. 이러한 테이블은 데이터가 입력될 때 랜덤하게 저장된다는 특징을 가지고 있다. 즉, 특정 컬럼의 값이 어디에 저장되어 있는지는 그 누구도 모른다. 이럴 때 필요한 객체가 인덱스이다.

인덱스는 컬럼의 값과 각 행이 저장된 주소(ROWID)를 저장하고 있는 객체이다. 이러한 인덱스는 WHERE 절의 조건식에 만족하는 행을 보다 빠르게 찾기 위해 사용되며, SQL 문장의 성능 향상을 위해서 사용된다. 하지만 컬럼에 인덱스가 없거나, 인덱스를 사용할 수 있는 식을 정의하지 않았다면 하나의 행을 찾기 위해 테이블의 모든 행을 액세스하여 조건식을 비교하는 "FULL TABLE SCAN" 실행 계획이 사용된다.

```
SQL> ALTER INDEX emp_ename_ix INVISIBLE ;
SQL> SET AUTOTRACE ON EXPLAIN
SQL> column rowid new_value rid

SQL> SELECT empno, ename, rowid
       FROM emp
      WHERE ename = 'SCOTT' ;
     EMPNO ENAME      ROWID
---------- ---------- ------------------
      7788 SCOTT      AAASCIAAAAAAASnAAA

------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    32 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    32 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='SCOTT')
```

실행 계획은 SQL 문장을 수행하기 위한 순서도이며, "TABLE ACCESS FULL"의 의미는 테이블 가지고 있는 모든 저장 영역을 전수 조사했다는 의미를 갖는다. 대량의 데이터를 저장하고 있는 테이블에서 소량의 데이터를 검색할 때 FULL TABLE SCAN이 사용되면, 불필요한 작업의 증가로 SQL 문장의 성능은 감소하게 될 것이다. (한 권의 책을 찾고자 도서관 전체를 검색하는 행동은 효율적인 행위가 될 수 없다.)

```
SQL> SELECT empno, ename, rowid
       FROM emp
      WHERE rowid = '&rid' ;
old   3:       WHERE rowid = '&rid'
new   3:       WHERE rowid = 'AAASCIAAAAAAASnAAA'

     EMPNO ENAME      ROWID
---------- ---------- ------------------
      7788 SCOTT      AAASCIAAAAAAASnAAA

-------------------------------------------------------------------------------
| Id  | Operation                   | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |     1 |    32 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY USER ROWID| EMP  |     1 |    32 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

만약, 검색하려는 행의 주소를 알고 있다면? ROWID는 하나의 행이 저장되어 있는 주소를 의미한다. 위의 예문처럼 조건절에 ROWID를 이용하면 동일한 결과를 보다 빠르게 검색할 수 있다. 오라클 데이터베이스는 하나의 행이 저장된 주소를 알지 못하기 때문에 인덱스와 같은 보조적인 객체의 도움을 받으려고 한다. 하지만 사용자가 검색하고자 하는 행의 주소를 직접 정의해 준다면 인덱스의 도움을 받지 않아도 하나의 행을 찾아올 수 있다.

하나의 행을 검색할 때 ROWID를 이용한 조건식은 가장 빠른 접근 방법을 제공해 준다. 하지만 이러한 ROWID를 이용한 조건식은 배치 업무와 같은 상황이 아니라면 현실적으로 불가능하다. 때문에 ROWID를 저장하고 있고, 컬럼의 값을 기준으로 정렬된 상태를 보장하는 인덱스가 필요하다.

```
SQL> ALTER INDEX emp_ename_ix VISIBLE ;
SQL> SELECT empno, ename, rowid
       FROM emp
      WHERE ename = 'SCOTT' ;
     EMPNO ENAME      ROWID
---------- ---------- ------------------
      7788 SCOTT      AAASCIAAAAAAASnAAA

-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              |     1 |    32 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP          |     1 |    32 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | EMP_ENAME_IX |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ENAME"='SCOTT')
```

ENAME 컬럼에 인덱스가 생성되면, 인덱스 스캔을 통해 필요한 ROWID를 확보할 수 있고, 해당 ROWID의 주소만 액세스하여 조건에 만족하는 행을 검색하게 된다. 이는 불필요한 테이블의 저장 영역에 대한 액세스를 줄여 문장의 성능을 향상화 시킨다. 단, 인덱스는 경우에 따라 사용이 불가능한 경우가 존재하고 다음 예제를 통해 그 상황들을 정리한다.

사례 1. 잘못 사용된 조건식 또는 조건식의 부재

```
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT deptno, SUM(sal)
     FROM emp
     GROUP BY deptno
     HAVING deptno IN (10,20) ;
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    14 |   364 |     4  (25)| 00:00:01 |
|*  1 |  FILTER             |      |       |       |            |          |
|   2 |   HASH GROUP BY     |      |    14 |   364 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| EMP  |    14 |   364 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("DEPTNO"=10 OR "DEPTNO"=20)
```

```
SQL> SELECT deptno, SUM(sal)
     FROM emp
     WHERE deptno IN (10,20)
     GROUP BY deptno ;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     8 |   208 |     2   (0)| 00:00:01 |
|   1 |  SORT GROUP BY NOSORT         |               |     8 |   208 |     2   (0)| 00:00:01 |
|   2 |   INLIST ITERATOR             |               |       |       |            |          |
|   3 |    TABLE ACCESS BY INDEX ROWID| EMP           |     8 |   208 |     2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | EMP_DEPTNO_IX |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("DEPTNO"=10 OR "DEPTNO"=20)
```

두 문장은 동일한 결과를 검색하는 문장이다. 하지만 GROUP BY 절에 정의된 조건식은 인덱스의 사용이 불가능하다. GROUP BY 절은 그룹 함수의 조건식을 정의할 경우에만 사용하고 일반 컬럼의 조건식은 WHERE 절에 정의한다. (WHERE 절이 없어도 경우에 따라 인덱스를 사용할 수 있는 경우도 존재한다. 또한 START WITH, CONNECT BY와 같은 절은 인덱스를 사용할 수도 있다. 인덱스의 사용 유/무는 반드시 실행 계획을 통해 확인한다.)

사례 2. 컬럼의 변형 (표현식의 일부로 사용된 컬럼)

```
SQL> SELECT *
       FROM emp
      WHERE SUBSTR(ename, 2, 1) = 'C' ;
```

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    87 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    87 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(SUBSTR("ENAME",2,1)='C')
```

```
SQL> SELECT *
       FROM emp
      WHERE sal * 12 > 30000 ;
```

```
-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     5 |   435 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     5 |   435 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SAL"*12>30000)
```

인덱스는 컬럼의 값을 동일하게 저장하고 있다. 또한 검색 속도를 높이기 위해 항상 정렬된 상태를 유지한다. 때문에 컬럼의 값이 변형된 형태의 조건식이 사용되면 인덱스의 사용은 불가능하다. (예를 들어 종이 사전에서 "SEL"로 시작되는 단어를 찾는 것은 어렵지 않지만 "ELE"가 포함된 단어를 찾으려면 사전의 첫 장부터 마지막 장까지 모두 검색을 해야 한다.) 데이터베이스에서의 인덱스는 테이블의 데이터를 빠르게 탐색하고자 사용하는 것인데, 인덱스를 전체 검색해야 하는 상황이 생긴다면 비용 대비 효과가 감소할 수 있다. 때문에 위와 같이 컬럼의 값이 변형되어 조건식에 사용되면 해당 컬럼의 인덱스 사용은 포기한다. 다음과 같이 문장을 수정하면 인덱스는 사용 가능하다.

```
SQL> SELECT *
       FROM emp
      WHERE sal > 30000 / 12 ;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                           | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |            |     5 |   435 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP        |     5 |   435 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | EMP_SAL_IX |     5 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("SAL">2500)
```

첫 번째 문장처럼 수정이 불가능한 문장은 Function Based Index(함수 기반 인덱스)를 생성하여 인덱스를 사용 가능하도록 한다.

```
SQL> CREATE INDEX emp_ename_fbi ON emp(SUBSTR(ename,2,1));
SQL> SELECT *
       FROM emp
      WHERE SUBSTR(ename, 2, 1) = 'C' ;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     1 |    91 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP           |     1 |    91 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | EMP_ENAME_FBI |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(SUBSTR("ENAME",2,1)='C')
```

```
SQL> DROP INDEX emp_ename_fbi ;
```

함수 기반 인덱스는 Oracle Database 8i부터 생성 가능하며, 계산된 결과를(표현식) 저장하고 있다. 때문에 인덱스를 사용하지 못하는 다양한 상황에서 인덱스를 사용할 수 있도록 도움을 줄 수 있다. 하지만 추가적으로 생성되는 객체이므로 추가적인 저장 공간이 필요하고, 주기적으로 관리를(통계 수집 및 인덱스 재구성) 해야 하므로 최후의 방법으로 사용해야 한다.

사례 3. IS NULL, IS NOT NULL 비교

```
SQL> SELECT *
       FROM emp
      WHERE comm IS NULL ;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    10 |   870 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |    10 |   870 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("COMM" IS NULL)
```

B*Tree 인덱스는 NULL을 제외한 컬럼의 값을 저장한다. 때문에 IS NULL의 조건식이 사용된 컬럼은 인덱스의 유무와 상관없이 해당 컬럼의 인덱스를 사용할 수 없다. (NULL의 주소가 저장되어 있지 않으므로) 만약 인덱스를 사용할 수 있도록 해야 한다면 NULL이 아닌 값을 저장하여 IS NULL의 조건식이 아닌 리터럴(상수) 값을 비교할 수 있도록 조건식을 변경한다.

```
SQL> SELECT *
       FROM emp
      WHERE comm IS NOT NULL ;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |             |     4 |   152 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP         |     4 |   152 |     2   (0)| 00:00:01 |
|*  2 |   INDEX FULL SCAN                   | EMP_COMM_IX |     4 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("COMM" IS NOT NULL)
```

IS NULL의 반대인 IS NOT NULL은 경우에 따라 인덱스를 사용할 수도 있다. 단, NULL이 아닌 값이 전체 데이터에 비해 적은 양일 때 가능하다. NULL이 아닌 모든 값을 비교해야 하는 상황이므로 인덱스의 특정 부분만(INDEX RANGE SCAN) 액세스하는 것이 아니라 INDEX FULL SCAN이 사용되는 것이 특징이다. 때문에 IS NOT NULL의 조건에 만족하는 범위가 넓으면 FULL TABLE SCAN이 오히려 성능상 유리할 수 있다. 이러한 인덱스 사용 유무의 선택은 오라클 데이터베이스가 판단하며 옵티마이저 통계 정보를 근거로 한다. (교통 정보를 이용하여 최적의 경로를 탐색하는 내비게이션과 비슷하다.)

```
SQL> SELECT /*+ full(emp) */ *
       FROM emp
      WHERE comm IS NOT NULL ;
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     4 |   152 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     4 |   152 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("COMM" IS NOT NULL)
```

IS NULL의 조건식은 인덱스를 사용하지 않는다. IS NOT NULL의 조건식은 경우에 따라 인덱스를 사용할 수도 있다. 다만, INDEX FULL SCAN 실행 계획은 항상 최적의 실행 계획이라고 할 수는 없으므로 성능상 유리하지 않을 수도 있음을 확인한다. (성능과 관련된 보다 자세한 상황은 SQL 튜닝 과정을 참고한다.)

사례 4. 잘못 사용된 LIKE 조건식

```
SQL> SELECT *
       FROM emp
      WHERE ename LIKE 'S%' ;
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              |     2 |    76 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP          |     2 |    76 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | EMP_ENAME_IX |     2 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("ENAME" LIKE 'S%')
       filter("ENAME" LIKE 'S%')
```

```
SQL> SELECT *
       FROM emp
      WHERE ename LIKE '%S' ;
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              |     1 |    38 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP          |     1 |    38 |     2   (0)| 00:00:01 |
|*  2 |   INDEX FULL SCAN                   | EMP_ENAME_IX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("ENAME" LIKE '%S' AND "ENAME" IS NOT NULL)
```

LIKE 조건식은 문자열의 부분적인 패턴을 비교할 때 많이 사용된다. 하지만 와일드카드로 사용되는 '%, _' 특수 문자가 앞에 사용되면 INDEX RANGE SCAN을 사용할 수 없다. 앞서 설명했듯이 시작 문자를 알지 못하면 사전의 특정 부분만을 검색할 수 없기 때문이다. 힌트를 사용하거나, 데이터의 양에 따라 인덱스를 사용하는 실행 계획이 나올 수도 있지만 INDEX FULL SCAN을 이용하기 때문에 성능이 최적이라고 할 수는 없다.

LIKE 조건식을 작성할 때는 와일드카드가 검색하는 문자열의 뒤에 정의될 수 있도록 한다. (첫 번째 예제 참고) 만약 와일드카드가 앞에 등장할 경우에는 인덱스의 사용이 불필요하거나, INDEX FULL SCAN을 사용해도 성능상 문제가 없는 경우에만 사용한다.

INDEX RANGE SCAN을 사용할 수 있는 또 다른 방법은 다음과 같이 함수 기반 인덱스를 사용하는 방법이다. 단, 와일드카드가 앞뒤로 정의되는 경우에는 사용이 불가능하다.

```
SQL> CREATE INDEX ename_fbi ON emp(REVERSE(ename)) ;

SQL> SELECT *
       FROM emp
      WHERE REVERSE(ename) LIKE 'TT%' ;
-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |           |     1 |    38 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP       |     1 |    38 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | ENAME_FBI |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access(REVERSE("ENAME") LIKE 'TT%')
       filter(REVERSE("ENAME") LIKE 'TT%')

SQL> DROP INDEX ename_fbi ;
```

사례 5. 부정형 비교

```
SQL> SELECT *
       FROM emp
      WHERE deptno != 20 ;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     9 |   342 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     9 |   342 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("DEPTNO"<>20)
```

```
SQL> SELECT /*+ index(emp(deptno)) */ *
       FROM emp
      WHERE deptno != 20 ;
```

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     9 |   342 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP           |     9 |   342 |     2   (0)| 00:00:01 |
|*  2 |   INDEX FULL SCAN                   | EMP_DEPTNO_IX |     9 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("DEPTNO"<>20)
```

부정형 비교를 하는 조건식은 인덱스를 사용하지 못하거나, 사용한다면 INDEX FULL SCAN이 사용된다. INDEX RANGE SCAN은 조건에 만족하는 시작점에서 끝점까지의 한 번의 스캔을 통해 액세스 가능할 때 사용된다. 하지만 부정형 비교를 하면 시작점이 서로 다른 범위를 읽어야 하는 경우가 발생하고 이러한 부분을 피하기 위해 INDEX FULL SCAN을 사용하거나 인덱스 사용을 아예 안 하고 FULL TABLE SCAN이 사용된다. 조건에 만족하는 범위가 넓지 않다면 INDEX RANGE SCAN이 가능하도록 조건식을 변경하는 것도 답안 중 하나이다.

```
SQL> SELECT *
       FROM emp
      WHERE deptno IN (10,30) ;
```

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |               |     9 |   342 |     2   (0)| 00:00:01 |
|   1 |  INLIST ITERATOR                    |               |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMP           |     9 |   342 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN                 | EMP_DEPTNO_IX |     9 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("DEPTNO"=10 OR "DEPTNO"=30)
```

사례 6. 암시적 형 변환

```
SQL> SELECT *
       FROM emp
      WHERE empno LIKE '77%' ;
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    38 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    38 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TO_CHAR("EMPNO") LIKE '77%')
```

조건식에 정의된 컬럼은 필요에 따라 암시적인 데이터 타입의 변환 작업이 수행될 수 있다. 위의 예제처럼 LIKE 비교는 문자 타입에서만 가능한 부분이기 때문에 TO_CHAR("EMPNO")의 컬럼 변형이 발생한다. 이는 컬럼의 변형에 의한 인덱스를 사용하지 못하는 상황으로 연결되고, 성능 저하로도 이어질 수 있다. 함수 기반 인덱스를 생성해서 인덱스를 사용하는 실행 계획을 만들 수도 있지만, 실행 계획을 확인하여 암시적인 형 변환이 발생하지 않도록 문장을 수정하는 것이 가장 좋은 방법이다.

```
SQL> SELECT *
       FROM emp
      WHERE empno BETWEEN 7700 AND 7799 ;
-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |              |     4 |   152 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMP          |     4 |   152 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | EMP_EMPNO_IX |     4 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO">=7700 AND "EMPNO"<=7799)

SQL> SET AUTOTRACE OFF
```

지금까지 확인한 몇 가지 사례들은 인덱스를 사용하지 못하는 경우와 사용을 하더라도 INDEX RANGE SCAN을 이용하지 못하는 상황들을 정리한 것이다. 성능을 고려할 때 인덱스는 무조건 사용해야 하는 객체는 아니다. 하지만 어떻게 문장이 작성되었느냐에 따라 아예 사용을 못하는 상황이 발생하면 힌트를 사용해도 인덱스 사용이 불가능한 경우가 존재한다. 때문에 인덱스가 사용 가능한 상황들을 이해하고, 올바른 문장의 작성을 통해서 필요에 따라 인덱스를 사용하도록 한다. 성능과 관련된 자세한 사항은 SQL Tuning 과정을 참고한다.

3. 날짜 함수 활용

1. EMP 테이블에서, 입사 일자(HIREDATE) 컬럼을 이용하여 연도, 월/일, 요일, 분기를 검색하시오. 단, 입사일자는 월요일부터 일요일 순으로 정렬합니다.

검색 결과

```
     EMPNO ENAME      Year       Date               Day            Quarter
---------- ---------- ---------- ------------------ -------------- ----------
      7654 MARTIN     1981       28, September      Monday         3
      7782 CLARK      1981       09, June           Tuesday        2
      7839 KING       1981       17, November       Tuesday        4
      7844 TURNER     1981       08, September      Tuesday        3
      7369 SMITH      1980       17, December       Wednesday      4
      7902 FORD       1981       03, December       Thursday       4
      7566 JONES      1981       02, April          Thursday       2
      7900 JAMES      1981       03, December       Thursday       4
      7499 ALLEN      1981       20, February       Friday         1
      7698 BLAKE      1981       01, May            Friday         2
      7876 ADAMS      1987       23, May            Saturday       2
      7934 MILLER     1982       23, January        Saturday       1
      7521 WARD       1981       22, February       Sunday         1
      7788 SCOTT      1987       19, April          Sunday         2

14 rows selected.
```

답안.

```
SQL> SELECT empno, ename,
            TO_CHAR(hiredate, 'YYYY') AS "Year",
            TO_CHAR(hiredate, 'DD, Month') AS "Date",
            TO_CHAR(hiredate, 'Day') AS "Day",
            TO_CHAR(hiredate, 'Q')   AS "Quarter"
     FROM emp
     ORDER BY TO_CHAR(hiredate - 1, 'D') ;
```

DATE 타입에서 특정 형식을 추출할 경우 TO_CHAR 함수를 이용한다. 다양한 형식 모델이 있으므로 필요한 형식의 조합을 진행할 수 있도록 연습한다. 사용할 수 있는 형식 모델은 오라클 문서를 참고한다.

Datetime Format Models

http://docs.oracle.com/database/121/SQLRF/sql_elements004.htm#SQLRF00212

2. EMP 테이블에서 20번 부서에 근무하는 사원들을 입사 일자를 기준으로 정렬하여 다음과 같이 검색하시오.
   START_DATE : 입사 일자가 포함된 한 주의 시작일 (일요일)
   END_DATE : 입사 일자가 포함된 한 주의 종료일 (토요일)

검색 결과

```
     EMPNO ENAME      HIREDATE        DAY             START_DATE      END_DATE
---------- ---------- --------------- --------------- --------------- ---------------
      7369 SMITH      80/12/17        WED             80/12/14        80/12/20
      7566 JONES      81/04/02        THU             81/03/29        81/04/04
      7902 FORD       81/12/03        THU             81/11/29        81/12/05
      7788 SCOTT      87/04/19        SUN             87/04/19        87/04/25
      7876 ADAMS      87/05/23        SAT             87/05/17        87/05/23

5 rows selected.
```

답안

```
SQL> SELECT empno, ename, hiredate,
            TO_CHAR(hiredate, 'DY')    AS "DAY",
            TRUNC(hiredate, 'DAY')     AS START_DATE,
            TRUNC(hiredate, 'DAY') + 6 AS END_DATE
     FROM emp
     WHERE deptno = 20
     ORDER BY hiredate ;
```

TRUNC 함수는 NUMBER 타입에서 절삭을 할 때 주로 사용된다. 하지만 위의 사용 방법처럼 DATE 타입에서도 사용이 가능하고, 지정된 형식의 하위 형식을 절삭하여 DATE 타입으로 결과를 생성할 수 있다. 다음의 예제를 수행하면 그 결과를 보다 정확히 이해할 수 있다.

```
SQL> ALTER SESSION SET nls_date_format = 'YYYY/MM/DD HH24:MI:SS' ;
SQL> SELECT SYSDATE
           ,TRUNC(SYSDATE, 'YYYY') AS YYYY
           ,TRUNC(SYSDATE,'MM')    AS MM
           ,TRUNC(SYSDATE,'DD')    AS DD
           ,TRUNC(SYSDATE,'HH')    AS HH
      FROM dual ;
SYSDATE             YYYY                 MM                   DD                   HH
------------------- -------------------- -------------------- -------------------- --------------------
2017/05/12 11:41:36 2017/01/01 00:00:00 2017/05/01 00:00:00 2017/05/12 00:00:00 2017/05/12 11:00:00

SQL> ALTER SESSION SET nls_date_format = 'RR/MM/DD' ;
```

3. SALES 테이블에서, TIME_ID 컬럼의 값이 '1998/05/01' 일을 포함한 한 주(일요일-토요일)의 판매 내역을 요일 별로 금액(AMOUNT_SOLD) 합계를 검색하시오.
단, 검색 결과는 일요일부터 토요일까지 정렬합니다.

검색 결과

```
DAY             SUM(AMOUNT_SOLD)
--------------- ----------------
Sunday                213986.56
Monday                 86039.97
Tuesday                26093.72
Wednesday              17584.96
Thursday               99296.19
Friday                 11996.05
Saturday               25852.07

7 rows selected.
```

답안

```
SQL> SELECT TO_CHAR(time_id, 'Day') AS day, SUM(amount_sold)
     FROM sales
     WHERE time_id BETWEEN TRUNC(TO_DATE('1998/05/01','YYYY/MM/DD'),'D')
                       AND TRUNC(TO_DATE('1998/05/01','YYYY/MM/DD'),'D') + 7 - 1/86400
     GROUP BY time_id
     ORDER BY time_id ;
```

TRUNC 함수를 이용하여 필요한 한 주의 데이터를 검색한다. 이때 한가지 주의 사항은 지정한 특정 날짜를 TO_DATE 함수를 이용하여 DATE 타입으로 형 변환이 필요하다. TRUNC 함수는 NUMBER 타입에서도 사용이 가능하기 때문에 경우에 따라 실행이 불가능한 경우가 존재한다.

```
SQL> SELECT TRUNC('98/05/01','D') FROM dual ;
ERROR at line 1:
ORA-01722: invalid number
```

Data Type이 명확하게 정의된 컬럼 이름이나 SYSDATE를 사용하는 경우를 제외하고, 특정 날짜를 직접 정의할 경우에는 TO_DATE 함수를 통해서 명시적으로 DATE 타입으로 형 변환을 수행하는 것이 에러 발생 확률을 줄여 준다.

추가 실습

```
SQL> CREATE TABLE t1 (c1    NUMBER, c2    DATE) ;
SQL> INSERT INTO t1
     SELECT level, ADD_MONTHS(SYSDATE,-3) + level - 1
       FROM dual
     CONNECT BY level <= 200 ;
SQL> COMMIT ;
SQL> SELECT c1, TO_CHAR(c2, 'YYYY/MM/DD HH24:MI:SS') AS DT
       FROM t1
     ORDER BY c1 ;
        C1 DT
---------- -------------------
         1 2017/02/12 12:29:46
         2 2017/02/13 12:29:46
         3 2017/02/14 12:29:46
         4 2017/02/15 12:29:46
...
```

WHERE 절의 조건식을 변경하여 원하는 범위의 결과를 검색한다.

	SELECT * FROM t1
오늘	WHERE c2 BETWEEN TRUNC(SYSDATE) AND TRUNC(SYSDATE+1) - 1/86400 ;
어제	WHERE c2 BETWEEN TRUNC(SYSDATE-1) AND TRUNC(SYSDATE) - 1/86400 ;
내일	WHERE c2 BETWEEN TRUNC(SYSDATE+1) AND TRUNC(SYSDATE+2) - 1/86400 ;
이번 주	WHERE c2 BETWEEN TRUNC(SYSDATE,'D') AND TRUNC(SYSDATE,'D') + 7 - 1/86400 ;
지난 주	WHERE c2 BETWEEN TRUNC(SYSDATE-7,'D') AND TRUNC(SYSDATE-7,'D') + 7 - 1/86400 ;
다음 주	WHERE c2 BETWEEN TRUNC(SYSDATE+7,'D') AND TRUNC(SYSDATE+7,'D') + 7 - 1/86400 ;
이번 달	WHERE c2 BETWEEN TRUNC(SYSDATE,'MM') AND TRUNC(ADD_MONTHS(SYSDATE,1),'MM') - 1/86400 ;
지난 달	WHERE c2 BETWEEN TRUNC(ADD_MONTHS(SYSDATE,-1),'MM') AND TRUNC(SYSDATE,'MM') - 1/86400 ;
다음 달	WHERE c2 BETWEEN TRUNC(ADD_MONTHS(SYSDATE,1),'MM') AND TRUNC(ADD_MONTHS(SYSDATE,2),'MM') - 1/86400 ;

```
SQL> DROP TABLE t1 PURGE ;
```

4. EMP 테이블에서, 입사 일자(HIREDATE)의 주차를 검색하시오.
   한 주의 시작일은 일요일이며, 달력을 기준으로 주차를 검색합니다.

예) 2015/01/06 일은 2주차 입니다.

검색 결과

EMPNO	ENAME	HIREDATE	Week
7369	SMITH	80/12/17	3
7499	ALLEN	81/02/20	3
7521	WARD	81/02/22	4
7566	JONES	81/04/02	1
7698	BLAKE	81/05/01	1
7782	CLARK	81/06/09	2
7844	TURNER	81/09/08	2
7654	MARTIN	81/09/28	5
7839	KING	81/11/17	3
7900	JAMES	81/12/03	1
7902	FORD	81/12/03	1
7934	MILLER	82/01/23	4
7788	SCOTT	87/04/19	4
7876	ADAMS	87/05/23	4

14 rows selected.

주의 사항

```
SQL> SELECT empno, ename, hiredate, TO_CHAR(hiredate, 'W') AS "Week"
       FROM emp
      ORDER BY hiredate ;
     EMPNO ENAME      HIREDATE        Week
---------- ---------- --------------- ----------
      7369 SMITH      80/12/17        3
      7499 ALLEN      81/02/20        3
      7521 WARD       81/02/22        4
      7566 JONES      81/04/02        1
      7698 BLAKE      81/05/01        1
      7782 CLARK      81/06/09        2
      7844 TURNER     81/09/08        2
      7654 MARTIN     81/09/28        4           -- 검색 결과에서는 5주차
      7839 KING       81/11/17        3
      7900 JAMES      81/12/03        1
      7902 FORD       81/12/03        1
      7934 MILLER     82/01/23        4
      7788 SCOTT      87/04/19        3           -- 검색 결과에서는 4주차
      7876 ADAMS      87/05/23        4

14 rows selected.
```

TO_CHAR 함수에서 'W' 형식 모델을 사용하면 주차 계산이 가능하다. 하지만 'W' 형식 모델은 해당 월의 1일을 기준으로 7일씩 한 주로 계산하므로 문제의 요구 사항과는 다른 결과가 검색된다.


```
SQL> SELECT TO_CHAR(TO_DATE('2015/01/06', 'YYYY/MM/DD'), 'W') AS week
       FROM dual ;
WEEK
------
1
```

SQL 명령문은 표현식을 어떻게 정의했느냐에 따라 결과가 달라질 수 있다. 위와 같이 전체가 아닌 일부 행이 잘못된 검색 결과를 가져오면 그 차이점의 구분이 쉽지 않다. 때문에 원하는 문장의 결과가 무엇인지를 정확하게 정의하고 작성된 문장이 신뢰받을 수 있도록 정확한 문장 정의가 필요하다. 'W' 형식 모델이 항상 잘못되었다는 것은 아니다. 하지만 '2015/01/06' 일을 2주차로 계산해야 하는 경우라면 다음과 같이 작성할 수도 있다.

```
SQL> SELECT TRUNC(base, 'D')              AS DATE1
          , TRUNC(base, 'MM')             AS DATE2
          , TRUNC(TRUNC(base,'MM'),'D')   AS DATE3
          , (TRUNC(base,'D') - TRUNC(TRUNC(base,'MM'),'D'))/7 + 1 AS "Week"
      FROM ( SELECT TO_DATE('2015/01/06', 'YYYY/MM/DD') AS base
               FROM dual ) ;
DATE1    DATE2    DATE3       Week
-------- -------- -------- -------
15/01/04 15/01/01 14/12/28       2
```

답안

```
SQL> SELECT empno, ename, hiredate,
            (TRUNC(hiredate,'D') - TRUNC(TRUNC(hiredate,'MM'),'D'))/7 + 1 AS "Week"
     FROM emp
     ORDER BY hiredate ;
```

TRUNC 함수는 DATE 타입에서 다양하게 사용될 수 있으며, 지금과 같은 주차 계산에서도 활용될 수 있다.

5. EMP_DATE 테이블에서, 잘못된 입사일자(HIREDATE)를 갖는 사원을 검색하시오.

```
SQL> DESCRIBE emp_date
 Name                      Null?    Type
 ------------------------- -------- ----------------
 EMPNO                              NUMBER(4)
 ENAME                              VARCHAR2(10)
 SAL                                NUMBER(7,2)
 DEPTNO                             NUMBER(2)
 HIREDATE                           VARCHAR2(8)
```

```
SQL> SELECT * FROM emp_date ;
     EMPNO ENAME             SAL     DEPTNO HIREDATE
---------- ---------- ---------- ---------- ---------------
      7788 SCOTT            3000         20 19870426
      7566 JONES            2975         20 19810409
      7369 SMITH             800         20 19801224
      7876 ADAMS            1100         20 19870530
      7499 ALLEN            1600         30 19810227
      7521 WARD             1250         30 19810229
      7654 MARTIN           1250         30 19810935
      7698 BLAKE            2850         30 19810508
      7782 CLARK            2450         10 19810616
      7844 TURNER           1500         30 19810915
      7900 JAMES             950         30 19811210
      7902 FORD             3000         20 19811210
      7934 MILLER           1300         10 19820130
      7839 KING             5000         10 19811124

14 rows selected.
```

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO HIREDATE
---------- ---------- ---------- ---------- ---------------
      7521 WARD             1250         30 19810229
      7654 MARTIN           1250         30 19810935

2 rows selected.
```

날짜 값을 저장할 때 DATE 타입이 아닌 문자 타입을 사용하는 경우가 있다. LIKE 조건식에서 인덱스 사용이 가능하도록 문자 타입을 사용하거나, TO_DATE 같은 함수의 사용 없이 특정 날짜를 보다 쉽게 조작하기 위해서 사용하는 경우가 존재한다. 가급적 날짜를 저장할 경우 DATE 타입을 이용하는 것이 좋겠지만 필요에 의해 문자 타입을 사용하면 문제와 같이 잘못된 날짜의 입력이 발생할 수 있다. 이때 유효하지 않은 날짜를 검색하기 위한 방법을 연구한다.

답안 1. 사용자 정의 함수 사용

```
SQL> SELECT empno, ename, hiredate, TO_DATE(hiredate, 'YYYYMMDD')
       FROM emp_date ;
ERROR:
ORA-01839: date not valid for month specified
```

SQL 명령문은 EXCEPTION 처리가 불가능하다. 때문에 TO_DATE 함수를 이용하여 날짜의 유효성을 체크할 때 발생하는 에러는 PL/SQL 함수를 이용하여 처리한다.

```
SQL> CREATE OR REPLACE FUNCTION ck_date (p_val VARCHAR2)
     RETURN VARCHAR2 IS
       v_ck     DATE ;
     BEGIN
      v_ck := to_date(p_val,'YYYYMMDD') ;
      RETURN 'Valid' ;
     EXCEPTION
       WHEN OTHERS THEN
         RETURN 'Invalid' ;
     END;
/

SQL> SELECT empno, ename, hiredate, ck_date(hiredate)  AS CK_DATE
       FROM emp_date
      WHERE ck_date(hiredate) = 'Invalid' ;
     EMPNO ENAME      HIREDATE        CK_DATE
---------- ---------- --------------- ---------------
      7521 WARD       19810229        Invalid
      7654 MARTIN     19810935        Invalid

2 rows selected.
```

답안 2. 유효한 날짜 범위 비교

```
SQL> SELECT TO_CHAR(base,'YYYYMMDD')           AS START_DATE,
            TO_CHAR(last_day(base),'YYYYMMDD') AS END_DATE
      FROM (SELECT ADD_MONTHS(TO_DATE('1980/01/01','YYYY/MM/DD'),level - 1) AS base
              FROM dual
            CONNECT BY level <= 500) ;
START_DATE      END_DATE
--------------- ---------------
19800101        19800131
19800201        19800229
19800301        19800331
...
20210601        20210630
20210701        20210731
20210801        20210831

500 rows selected.
```

날짜 범위의 설정하여 각각의 월별 시작 일자와 종료 일자를 별도로 계산한다. 위의 검색 결과는 VIEW 또는 TABLE로 저장할 수도 있고, Subquery를 이용하여 처리할 수도 있다.

```
SQL> SELECT *
       FROM emp_date c
      WHERE NOT EXISTS (SELECT * FROM
                         (SELECT TO_CHAR(base,'YYYYMMDD')           AS START_DATE,
                                 TO_CHAR(last_day(base),'YYYYMMDD') AS END_DATE
                          FROM (SELECT ADD_MONTHS(TO_DATE('1980/01/01','YYYY/MM/DD'),
                                                  level - 1) AS base
                                FROM dual
                                CONNECT BY level <= 500))
                        WHERE c.hiredate BETWEEN start_date AND end_date) ;
     EMPNO ENAME             SAL     DEPTNO HIREDATE
---------- ---------- ---------- ---------- ---------------
      7521 WARD             1250         30 19810229
      7654 MARTIN           1250         30 19810935

2 rows selected.
```

답안 3. VALIDATE_CONVERSION 함수 사용 (New Features From 12cR2)

```
SQL> SELECT *
       FROM emp_date
      WHERE VALIDATE_CONVERSION(hiredate AS DATE, 'YYYYMMDD') = 0 ;
     EMPNO ENAME             SAL     DEPTNO HIREDATE
---------- ---------- ---------- ---------- ---------------
      7521 WARD             1250         30 19810229
      7654 MARTIN           1250         30 19810935

2 rows selected.

SQL> SELECT empno, ename, hiredate,
            VALIDATE_CONVERSION(hiredate AS DATE, 'YYYYMMDD') AS VALIDATION
       FROM emp_date ;
     EMPNO ENAME      HIREDATE        VALIDATION
---------- ---------- --------------- ----------
      7788 SCOTT      19870426                 1
      7566 JONES      19810409                 1
      7369 SMITH      19801224                 1
      7876 ADAMS      19870530                 1
      7499 ALLEN      19810227                 1
      7521 WARD       19810229                 0
      7654 MARTIN     19810935                 0
      7698 BLAKE      19810508                 1
      7782 CLARK      19810616                 1
      7844 TURNER     19810915                 1
      7900 JAMES      19811210                 1
      7902 FORD       19811210                 1
      7934 MILLER     19820130                 1
      7839 KING       19811124                 1

14 rows selected.
```

4. TOP-n 질의 활용

1. EMP 테이블에서, 급여(SAL)를 가장 많이 받는 3명을 다음과 같이 검색하시오. (ROWNUM 활용)

검색 결과

```
     EMPNO ENAME      JOB              MGR HIREDATE              SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------------- ---------- ---------- ----------
      7839 KING       PRESIDENT            81/11/17              5000                    10
      7788 SCOTT      ANALYST          7566 87/04/19              3000                    20
      7902 FORD       ANALYST          7566 81/12/03              3000                    20

3 rows selected.
```

답안.

```
SQL> SELECT *
       FROM ( SELECT *
                FROM emp
               ORDER BY sal DESC )
     WHERE rownum <= 3 ;
```

ROWNUM은 행 번호를 의미하며 데이터 행에 접근된 순서대로 번호를 부여한다. 때문에 다음과 같이 문장을 실행하면 3개의 행이 검색은 되지만 결과가 틀려진다.

```
SQL> SELECT *
       FROM emp
      WHERE rownum <= 3
     ORDER BY sal DESC ;
     EMPNO ENAME      JOB              MGR HIREDATE              SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------------- ---------- ---------- ----------
      7788 SCOTT      ANALYST          7566 87/04/19              3000                    20
      7566 JONES      MANAGER          7839 81/04/02              2975                    20
      7369 SMITH      CLERK            7902 80/12/17               800                    20

3 rows selected.
```

```
SQL> SELECT rownum, e.*
       FROM ( SELECT rownum sub_rnum, empno, ename, sal, deptno
                FROM emp
               ORDER BY sal DESC ) e ;
    ROWNUM   SUB_RNUM      EMPNO ENAME             SAL      DEPTNO
---------- ---------- ---------- ---------- ---------- ----------
         1         14       7839 KING             5000         10
         2         12       7902 FORD             3000         20
         3          1       7788 SCOTT            3000         20
         4          2       7566 JONES            2975         20
         5          8       7698 BLAKE            2850         30
         6          9       7782 CLARK            2450         10
         7          5       7499 ALLEN            1600         30
         8         10       7844 TURNER           1500         30
         9         13       7934 MILLER           1300         10
        10          6       7521 WARD             1250         30
        11          7       7654 MARTIN           1250         30
        12          4       7876 ADAMS            1100         20
        13         11       7900 JAMES             950         30
        14          3       7369 SMITH             800         20

14 rows selected.
```

ROWNUM (Pseudo Column)은 하나의 Query Block에 하나씩 정의될 수 있으며, 예제와 같이 Inline View를 이용하고 있다면 Main Query에 리턴되는 순서와 Subquery에서 접근한 데이터의 순서를 따로 정의할 수 있다. 문장 작성 시 ROWNUM은 활용도가 크기 때문에 그 특성 및 사용 방법을 정확하게 이해하고 있는 것이 필요하다.

추가 실습

```
SQL> SELECT rownum, empno, ename, sal
       FROM emp
      WHERE rownum <= 2 ;
    ROWNUM      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
         1       7788 SCOTT            3000
         2       7566 JONES            2975

2 rows selected.

SQL> SELECT rownum, empno, ename, sal
       FROM emp
      WHERE rownum = 1 ;
    ROWNUM      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
         1       7788 SCOTT            3000

1 row selected.
```

```
SQL> SELECT rownum, empno, ename, sal
       FROM emp
      WHERE rownum > 2 ;
no rows selected

SQL> SELECT rownum, empno, ename, sal
       FROM emp
      WHERE rownum = 2 ;
no rows selected
```

ROWNUM은 실제 저장된 값을 검색하는 것이 아니다. 각 Query Block에 접근되는 순서대로 행 번호를 생성하게 되는데, 1번 행이 없는 상태에서 2번 행부터 검색할 수는 없다. 때문에 ROWNUM을 이용한 조건식은 모든 비교 연산을 지원하지 않는다.

ROWNUM을 이용한 조건식은 다음의 조건만 가능하다.

- **WHERE ROWNUM = 1**
- **WHERE ROWNUM < or <= n**

2. EMP 테이블에서, 급여를 가장 많이 받는 사원 순으로 5 ~ 10등의 사원 정보를 다음과 같이 검색하시오. (ROWNUM 활용)

검색 결과

```
      RANK      EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ---------- ----------
         5       7698 BLAKE            2850         30
         6       7782 CLARK            2450         10
         7       7499 ALLEN            1600         30
         8       7844 TURNER           1500         30
         9       7934 MILLER           1300         10
        10       7521 WARD             1250         30

6 rows selected.
```

답안

```
SQL> SELECT rank, empno, ename, sal, deptno
       FROM (SELECT rownum AS rank, empno, ename, sal, deptno
               FROM ( SELECT *
                        FROM emp
                      ORDER BY sal DESC ) )
      WHERE rank BETWEEN 5 AND 10 ;
```

앞서 확인했듯이 ROWNUM은 조건식 생성에 제약이 존재한다. 하지만 필요에 따라 BETWEEN 같은 조건식을 생성해야 하는 경우 Inline View를 활용할 수 있다. FROM 절의 서브 쿼리를 사용하여 ROWNUM의 결과를 컬럼처럼 정의하면 ROWNUM을 이용한 조건식의 제약은 사라진다. 때문에 다양한 조건식을 생성할 수 있다.

3. EMP 테이블에서 급여를 가장 많이 받는 2명을 검색하시오.
단, 동일한 급여를 받는 사원이 둘 이상 있다면 함께 검색한다.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7839 KING             5000         10
      7902 FORD             3000         20
      7788 SCOTT            3000         20

3 rows selected.
```

답안 1.

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE sal IN (SELECT sal
                      FROM (SELECT DISTINCT sal
                              FROM emp
                             ORDER BY sal DESC)
                     WHERE rownum <= 2)
     ORDER BY sal DESC ;
```

ROWNUM을 이용할 경우 값을 기반으로 동일 순위를 정의할 수는 없다. 때문에 3000의 급여를 받는 두 명을 검색하기 위해, 중복을 제거한 상태에서의 순위를 정의하고 동일한 급여를 받는 사원들을 검색하도록 문장을 작성하였다. 실행 시 결과는 맞게 나오지만 동일한 테이블의 반복적인 접근으로 인해 성능이 저하된다.

답안 2. 분석 함수 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM (SELECT empno, ename, sal, deptno, RANK() OVER (ORDER BY sal DESC) AS rank
               FROM emp)
      WHERE rank <= 2 ;
```

추가 실습

```
SQL> SELECT empno, ename, sal, deptno, RANK() OVER (ORDER BY sal DESC) AS rank
       FROM emp ;
     EMPNO ENAME             SAL     DEPTNO       RANK
---------- ---------- ---------- ---------- ----------
      7839 KING             5000         10          1
      7902 FORD             3000         20          2
      7788 SCOTT            3000         20          2
      7566 JONES            2975         20          4
      7698 BLAKE            2850         30          5
      7782 CLARK            2450         10          6
      7499 ALLEN            1600         30          7
      7844 TURNER           1500         30          8
      7934 MILLER           1300         10          9
      7521 WARD             1250         30         10
      7654 MARTIN           1250         30         10
      7876 ADAMS            1100         20         12
      7900 JAMES             950         30         13
      7369 SMITH             800         20         14

14 rows selected.
```

RANK 함수는 분석 함수의 한 종류이며 값을 기반으로 동일 순위를 정의할 수 있다.

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp
      WHERE RANK() OVER (ORDER BY sal DESC) <= 2 ;
ERROR at line 3:
ORA-30483: window functions are not allowed here
```

단, 분석 함수는 WHERE 절에서 바로 사용할 수는 없다. 분석 함수는 WHERE 절의 수행 이후에 작동하기 때문에 분석 함수의 결과를 이용하여 조건식을 정의해야 한다면 Inline View를 이용한다.

4. EMP 테이블에서, 부서별(DEPTNO) 가장 많은 급여(SAL)를 받는 사원을 한 명씩 검색하시오. 단, 동일한 급여를 받는 사원이 존재할 경우 임의의 한 명을 검색한다.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7839 KING             5000         10
      7788 SCOTT            3000         20
      7698 BLAKE            2850         30

3 rows selected.
```

답안 1. UNION ALL 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM ( SELECT *
                FROM emp
               WHERE deptno = 10
              ORDER BY sal DESC )
      WHERE rownum = 1
     UNION ALL
     SELECT empno, ename, sal, deptno
       FROM ( SELECT *
                FROM emp
               WHERE deptno = 20
              ORDER BY sal DESC )
      WHERE rownum = 1
     UNION ALL
     SELECT empno, ename, sal, deptno
       FROM ( SELECT *
                FROM emp
               WHERE deptno = 30
              ORDER BY sal DESC )
      WHERE rownum = 1 ;
```

ROWNUM은 특정 컬럼의 값을 기준으로 행 번호를 새로 시작할 수 없다. 때문에 각각의 부서별로 TOP-n 질의 문장을 이용하여 UNION ALL으로 묶어주면 원하는 결과를 검색할 수 있으나, 부서 번호의 개수가 많을 경우 문장이 복잡해질 것이고 성능 역시 저하된다.

현재 요구 조건에는 만족하지 않지만, 부서별 최대 급여를 받는 사원을 검색하는 경우라면 다음과 같은 문장을 이용할 수도 있다.

```
SQL> SELECT empno, ename, sal, deptno
       FROM emp a
      WHERE sal = ( SELECT MAX(sal)
                      FROM emp
                     WHERE deptno = a.deptno )
      ORDER BY deptno ;
     EMPNO ENAME             SAL     DEPTNO
---------- ---------- ---------- ----------
      7839 KING             5000         10
      7788 SCOTT            3000         20
      7902 FORD             3000         20
      7698 BLAKE            2850         30

4 rows selected.
```

답안 2. 분석 함수 사용

```
SQL> SELECT empno, ename, sal, deptno
       FROM (SELECT empno, ename, sal, deptno,
                    ROW_NUMBER() OVER(PARTITION BY deptno ORDER BY sal DESC) AS rank
               FROM emp )
      WHERE rank = 1 ;
```

분석 함수는 PARTITION BY 절을 통해 순위를 초기화할 수 있다. 또한 RANK, DENSE_RANK, ROW_NUMBER의 함수를 이용하여 원하는 결과를 다양하게 생성할 수 있다. 각각의 함수가 어떠한 결과를 검색하는지 정확히 구분하고 필요한 함수를 사용한다.

```
SQL> SELECT empno, ename, sal,
            ROW_NUMBER() OVER(ORDER BY sal DESC) AS row_num,
            RANK()       OVER(ORDER BY sal DESC) AS rank,
            DENSE_RANK() OVER(ORDER BY sal DESC) AS drank
       FROM emp ;
     EMPNO ENAME             SAL    ROW_NUM       RANK      DRANK
---------- ---------- ---------- ---------- ---------- ----------
      7839 KING             5000          1          1          1
      7902 FORD             3000          2          2          2
      7788 SCOTT            3000          3          2          2
      7566 JONES            2975          4          4          3
      7698 BLAKE            2850          5          5          4
      7782 CLARK            2450          6          6          5
      7499 ALLEN            1600          7          7          6
      7844 TURNER           1500          8          8          7
      7934 MILLER           1300          9          9          8
      7521 WARD             1250         10         10          9
      7654 MARTIN           1250         11         10          9
      7876 ADAMS            1100         12         12         10
      7900 JAMES             950         13         13         11
      7369 SMITH             800         14         14         12

14 rows selected.
```

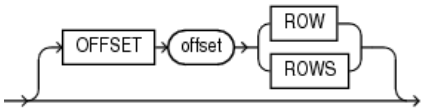
ROW_NUMBER 함수는 ROWNUM과 비슷하다. 정렬 컬럼의 값이 동일하더라도 행 번호는 증가되며, RANK, DENSE_RANK 함수는 정렬 컬럼의 값이 동일할 경우 동일한 순위를 할당한다. 단, 동일 순위가 만들어졌을 때 다음 순위를 정의하는 기준이 서로 다르다. 또한 3개의 함수 모두 PARTITION BY 절을 이용하여 원하는 컬럼을 기준으로 순위의 초기화가 가능하다.

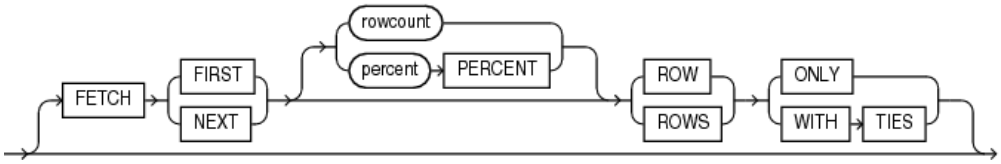
```
SQL> SELECT empno, ename, comm,
            RANK() OVER(ORDER BY comm DESC)           AS rank1,
            RANK() OVER(ORDER BY comm DESC NULLS LAST) AS rank2
      FROM emp ;
     EMPNO ENAME            COMM      RANK1      RANK2
---------- ---------- ---------- ---------- ----------
      7654 MARTIN           1400         11          1
      7521 WARD              500         12          2
      7499 ALLEN             300         13          3
      7844 TURNER              0         14          4
      7900 JAMES                         1          5
      7902 FORD                          1          5
      7698 BLAKE                         1          5
      7782 CLARK                         1          5
      7934 MILLER                        1          5
      7876 ADAMS                         1          5
      7369 SMITH                         1          5
      7566 JONES                         1          5
      7788 SCOTT                         1          5
      7839 KING                          1          5

14 rows selected.
```

정렬 작업 시 ASC, DESC 키워드를 통해 오름차순, 내림차순 정렬이 가능하고, NULL을 포함하는 컬럼에서 NULL 값의 위치를 기본 설정에 반대가 될 수 있도록 설정이 가능하다.

※ Oracle 12c Database New Features : row_limiting_clause





주. FIRST¦NEXT, ROW¦ROWS 키워드는 차이점이 없으므로 구분 없이 사용 가능하다.

EMP 테이블에서 급여를 기준으로 정렬된 결과는 다음과 같다.

```
SQL> SELECT empno, ename, sal
       FROM emp
     ORDER BY sal DESC ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7839 KING             5000
      7902 FORD             3000
      7788 SCOTT            3000
      7566 JONES            2975
      7698 BLAKE            2850
      7782 CLARK            2450
      7499 ALLEN            1600
      7844 TURNER           1500
      7934 MILLER           1300
      7521 WARD             1250
      7654 MARTIN           1250
      7876 ADAMS            1100
      7900 JAMES             950
      7369 SMITH             800

14 rows selected.
```

FETCH 절을 이용하면 TOP-n 질의를 보다 손 쉽게 검색할 수 있다. 이때 ONLY ¦ WITH TIES 옵션을 이용하여 동일 순위의 결과를 함께 검색할지 말지를 정의할 수 있다.

```
SQL> SELECT empno, ename, sal
       FROM emp
     ORDER BY sal DESC
     FETCH FIRST 2 ROWS ONLY ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7839 KING             5000
      7788 SCOTT            3000

2 rows selected.

SQL> SELECT empno, ename, sal
       FROM emp
     ORDER BY sal DESC
     FETCH FIRST 2 ROWS WITH TIES ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7839 KING             5000
      7788 SCOTT            3000
      7902 FORD             3000

3 rows selected.
```

OFFSET 절을 이용하여 원하는 행의 개수를 건너뛸 수 있다.

```
SQL> SELECT empno, ename, sal
       FROM emp
     ORDER BY sal DESC
     OFFSET 10 ROWS ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7654 MARTIN           1250
      7876 ADAMS            1100
      7900 JAMES             950
      7369 SMITH             800

4 rows selected.
```

OFFSET과 FETCH를 함께 사용하면 특정 행은 스킵하고 TOP-n 질의의 결과를 생성할 수 있다.

```
SQL> SELECT empno, ename, sal
       FROM emp
     ORDER BY sal DESC
     OFFSET 8 ROWS FETCH FIRST 2 ROWS ONLY ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7934 MILLER           1300
      7521 WARD             1250

2 rows selected.

SQL> SELECT empno, ename, sal
       FROM emp ORDER BY sal DESC
     OFFSET 8 ROWS FETCH FIRST 2 ROWS WITH TIES ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7934 MILLER           1300
      7521 WARD             1250
      7654 MARTIN           1250

3 rows selected.
```

FETCH 되는 행은 개수가 아닌 PERCENT 키워드를 통해 백분율로 제한할 수도 있다.

```
SQL> SELECT empno, ename, sal
       FROM emp
     ORDER BY sal DESC
     FETCH FIRST 10 PERCENT ROWS ONLY ;
     EMPNO ENAME             SAL
---------- ---------- ----------
      7839 KING             5000
      7902 FORD             3000

2 rows selected.
```

5. 조인, 서브쿼리 활용

1. DEPT, EMP 테이블을 사용하여 SALES 부서에 근무하고, 1981년 상반기(1월-6월)에 입사한 사원 정보를 검색하시오. 이때 커미션을 합한 급여(SAL + COMM)를 함께 검색하고 입사 일자 순으로 정렬합니다.

검색 결과

```
     EMPNO ENAME      HIREDATE     INCOME
---------- ---------- -------- ----------
      7499 ALLEN      81/02/20       1900
      7521 WARD       81/02/22       1750
      7698 BLAKE      81/05/01       2850

3 rows selected.
```

답안.

```
SQL> SELECT empno, ename, hiredate, sal+NVL(comm,0) AS INCOME
       FROM emp
      WHERE hiredate BETWEEN TO_DATE('19810101','YYYYMMDD')
                         AND TO_DATE('19810630','YYYYMMDD')
        AND deptno = ( SELECT deptno
                         FROM dept
                        WHERE dname = 'SALES' )
     ORDER BY hiredate ;
```

2. DEPT, EMP 테이블을 사용하여 JOB이 'MANAGER'인 사원들의 부서 정보 및 사원 정보를 검색하시오.

검색 결과

```
    DEPTNO DNAME          ENAME             SAL
---------- -------------- ---------- ----------
        10 ACCOUNTING     CLARK            2450
        20 RESEARCH       JONES            2975
        30 SALES          BLAKE            2850

3 rows selected.
```

답안.

```
SQL> SELECT d.deptno, d.dname, e.ename, e.sal
       FROM dept d, emp e
      WHERE d.deptno = e.deptno
        AND e.job    = 'MANAGER' ;
```

3. DEPT, EMP 테이블을 사용하여, 소속 부서의 평균 급여보다 많은 급여를 받는 'MANAGER' 들의 부서 번호, 사원 번호, 사원 이름, 급여를 검색하시오.

검색 결과

```
    DEPTNO DNAME              EMPNO ENAME             SAL
---------- -------------- ---------- ---------- ----------
        20 RESEARCH             7566 JONES            2975
        30 SALES                7698 BLAKE            2850

2 rows selected.
```

답안.

```
SQL> SELECT d.deptno, d.dname, e.empno, e.ename, e.sal
     FROM emp e, dept d
     WHERE e.deptno = d.deptno
       AND e.job    = 'MANAGER'
       AND e.sal    > (SELECT AVG(sal)
                       FROM emp
                       WHERE deptno = d.deptno) ;
```

4. DEPT, EMP 테이블에서, 2000 이상의 급여(SAL)를 받는 사원들의 소속 부서의 이름(DNAME)을 함께 검색하시오. 단, 근무하는 사원이 없는 부서이름도 검색합니다.

검색 결과

```
DEPT_DEPTNO DNAME          EMP_DEPTNO      EMPNO ENAME             SAL
----------- -------------- ---------- ---------- ---------- ----------
         10 ACCOUNTING             10       7782 CLARK            2450
         10 ACCOUNTING             10       7839 KING             5000
         20 RESEARCH               20       7566 JONES            2975
         20 RESEARCH               20       7788 SCOTT            3000
         20 RESEARCH               20       7902 FORD             3000
         30 SALES                  30       7698 BLAKE            2850
         40 OPERATIONS

7 rows selected.
```

OUTER JOIN 수행 시 일반 조건이 추가된다면 OUTER JOIN의 결과가 제대로 검색되는지 확인해야 한다. 어느 집합에 조건이 추가되느냐에 따라 INNER JOIN의 결과만 출력될 수도 있기 때문이다.

주의 사항

```
SQL> SELECT d.deptno AS dept_deptno, d.dname
           ,e.deptno AS emp_deptno, e.empno, e.ename, e.sal
       FROM dept d, emp e
      WHERE e.deptno (+)  = d.deptno
        AND e.sal        >= 2000 ;
DEPT_DEPTNO DNAME          EMP_DEPTNO      EMPNO ENAME             SAL
----------- -------------- ---------- ---------- ---------- ----------
         10 ACCOUNTING             10       7782 CLARK            2450
         10 ACCOUNTING             10       7839 KING             5000
         20 RESEARCH               20       7566 JONES            2975
         20 RESEARCH               20       7788 SCOTT            3000
         20 RESEARCH               20       7902 FORD             3000
         30 SALES                  30       7698 BLAKE            2850

6 rows selected.

SQL> SELECT d.deptno AS dept_deptno, d.dname
           ,e.deptno AS emp_deptno, e.empno, e.ename, e.sal
       FROM dept d LEFT OUTER JOIN emp e
         ON d.deptno = e.deptno
      WHERE e.sal   >= 2000 ;
DEPT_DEPTNO DNAME          EMP_DEPTNO      EMPNO ENAME             SAL
----------- -------------- ---------- ---------- ---------- ----------
         10 ACCOUNTING             10       7782 CLARK            2450
         10 ACCOUNTING             10       7839 KING             5000
         20 RESEARCH               20       7566 JONES            2975
         20 RESEARCH               20       7788 SCOTT            3000
         20 RESEARCH               20       7902 FORD             3000
         30 SALES                  30       7698 BLAKE            2850
6 rows selected.
```

답안1. 비조인 술어 비교 후 OUTER JOIN

```
SQL> SELECT d.deptno AS dept_deptno, d.dname
           ,e.deptno AS emp_deptno, e.empno, e.ename, e.sal
       FROM dept d, emp e
      WHERE e.deptno (+)  = d.deptno
        AND e.sal    (+) >= 2000 ;

SQL> SELECT d.deptno AS dept_deptno, d.dname
           ,e.deptno AS emp_deptno, e.empno, e.ename, e.sal
       FROM dept d LEFT OUTER JOIN emp e
         ON d.deptno = e.deptno
        AND e.sal   >= 2000 ;
```

답안2. OUTER JOIN 후 비조인 술어 비교

```
SQL> SELECT d.deptno AS dept_deptno, d.dname
           ,e.deptno AS emp_deptno, e.empno, e.ename, e.sal
       FROM dept d, emp e
      WHERE e.deptno (+)  = d.deptno
        AND (e.sal >= 2000 OR e.sal IS NULL);

SQL> SELECT d.deptno AS dept_deptno, d.dname
           ,e.deptno AS emp_deptno, e.empno, e.ename, e.sal
       FROM dept d LEFT OUTER JOIN emp e
         ON d.deptno = e.deptno
      WHERE (e.sal >= 2000 OR e.sal IS NULL);
```

5. DEPARTMENTS, EMPLOYEES 테이블을 조인하여 다음과 같이 검색하시오.
단, 근무하는 사원이 없는 부서 정보 및 소속된 부서가 없는 사원 정보도 함께 검색합니다.

검색 결과

```
DEPARTMENT_ID DEPARTMENT_NAME      EMPLOYEE_ID LAST_NAME           SALARY DEPARTMENT_ID
------------- -------------------- ----------- --------------- ---------- -------------
           10 Administration              200 Whalen               4400            10
           20 Marketing                   201 Hartstein           13000            20
           20 Marketing                   202 Fay                  6000            20
           50 Shipping                    124 Mourgos              5800            50
           50 Shipping                    141 Rajs                 3500            50
           50 Shipping                    142 Davies               3100            50
           50 Shipping                    143 Matos                2600            50
           50 Shipping                    144 Vargas               2500            50
           60 IT                          103 Hunold               9000            60
           60 IT                          104 Ernst                6000            60
           60 IT                          107 Lorentz              4200            60
           80 Sales                       149 Zlotkey             10500            80
           80 Sales                       174 Abel                11000            80
           80 Sales                       176 Taylor               8600            80
           90 Executive                   100 King                24000            90
           90 Executive                   101 Kochhar             17000            90
           90 Executive                   102 De Haan             17000            90
          110 Accounting                  205 Higgins             12008           110
          110 Accounting                  206 Gietz                8300           110
          190 Contracting
                                          178 Grant                7000

21 rows selected.
```

오라클 조인 구문에서는 FULL OUTER JOIN을 지원하지 않는다. 때문에 ANSI 구문을 이용한 조인 방법을 사용하고, 만약 ANSI 구문을 사용하지 못하는 경우라면 FULL OUTER JOIN 결과를 만드는 방법을 확인한다.

답안 1. FULL OUTER JOIN 사용

```
SQL> SELECT d.department_id, d.department_name,
            e.employee_id, e.last_name, e.salary, e.department_id
       FROM departments d FULL OUTER JOIN employees e
         ON d.department_id = e.department_id
     ORDER BY d.department_id, e.employee_id ;
```

답안 2. 오라클 조인 사용

```
SQL> SELECT d.department_id, d.department_name,
            e.employee_id, e.last_name, e.salary, e.department_id
       FROM departments d , employees e
      WHERE d.department_id (+) = e.department_id (+)
     ORDER BY d.department_id, e.employee_id ;
ERROR at line 4:
ORA-01468: a predicate may reference only one outer-joined table

SQL> SELECT d.department_id, d.department_name,
            e.employee_id, e.last_name, e.salary, e.department_id
       FROM departments d , employees e
      WHERE d.department_id = e.department_id (+)
      UNION
     SELECT d.department_id, d.department_name,
            e.employee_id, e.last_name, e.salary, e.department_id
       FROM departments d , employees e
      WHERE d.department_id (+) = e.department_id
     ORDER BY 1, 3 ;
```

6. EMP 테이블에서, 'JONES' (ENAME)보다 더 많은 급여(SAL)를 받는 사원을 검색하시오.
단, JONES의 급여도 함께 검색합니다.

검색 결과

```
     EMPNO ENAME             SAL Jones's Salary
---------- ---------- ---------- --------------
      7788 SCOTT            3000           2975
      7902 FORD             3000           2975
      7839 KING             5000           2975

3 rows selected.
```

조건절의 서브 쿼리는 조건을 비교할 때만 사용할 수 있고, 그 결과를 출력할 수 없다. 때문에 검색 결과로 출력되어야 한다면 조인문을 사용한다.

답안. Self Join 사용

```
SQL> SELECT e.empno, e.ename, e.sal, j.sal AS "Jones's Salary"
       FROM emp e, emp j
      WHERE j.ename = 'JONES'
        AND j.sal   < e.sal ;
```

주의 사항

```
SQL> SELECT empno, ename, sal,
            (SELECT sal
               FROM emp
              WHERE ename = 'JONES') AS "Jones's Salary"
       FROM emp
      WHERE sal > (SELECT sal
                     FROM emp
                    WHERE ename = 'JONES') ;
```

7. DEPARTMENTS, EMPLOYEES 테이블에서, 근무하는 사원이 없는 부서 정보를 검색하시오.

검색 결과

```
DEPARTMENT_ID DEPARTMENT_NAME               MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          190 Contracting                                      1700

1 row selected.
```

서브 쿼리에서 NULL이 리턴될 때 NOT IN 연산을 사용하면 검색 결과는 항상 나올 수 없다. 때문에 NOT IN 연산을 사용할 때는 NULL이 검색되지 않도록 하거나 NOT IN 대신 NOT EXISTS를 사용한다.

주의 사항. NOT IN과 Subquery의 NULL

```
SQL> SELECT *
       FROM departments
      WHERE department_id NOT IN (SELECT department_id
                                  FROM employees ) ;
no rows selected
```

답안 1. Subquery의 NULL 제거

```
SQL> SELECT *
       FROM departments
      WHERE department_id NOT IN (SELECT department_id
                                  FROM employees
                                 WHERE department_id IS NOT NULL) ;
DEPARTMENT_ID DEPARTMENT_NAME               MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          190 Contracting                                      1700

1 row selected.

SQL> SELECT *
       FROM departments
      WHERE department_id NOT IN (SELECT NVL(department_id, 9999)
                                  FROM employees ) ;
```

답안 2. NOT EXISTS 사용

```
SQL> SELECT *
       FROM departments d
      WHERE NOT EXISTS (SELECT 1
                        FROM employees
                       WHERE department_id = d.department_id) ;
```

8. DEPT, EMP 테이블을 사용하여 각 부서의 소속 사원 유무를 확인하는 검색 결과를 만드시오. EMP 컬럼은 소속 사원이 존재할 때 'YES', 아니면 'NO'를 검색합니다.

검색 결과

```
    DEPTNO DNAME          LOC           EMP
---------- -------------- ------------- ---
        10 ACCOUNTING     NEW YORK      YES
        20 RESEARCH       DALLAS        YES
        30 SALES          CHICAGO       YES
        40 OPERATIONS     BOSTON        NO

4 rows selected.
```

답안 1. Join 사용

```
SQL> SELECT d.deptno, d.dname, d.loc, DECODE(COUNT(e.deptno), 0, 'NO', 'YES') AS EMP
     FROM dept d, emp e
     WHERE d.deptno = e.deptno (+)
     GROUP BY d.deptno, d.dname, d.loc
     ORDER BY d.deptno ;
```

DEPT, EMP 테이블에 OUTER JOIN을 수행하여 DEPT의 모든 행을 검색될 수 있도록 했고, 동일한 부서 번호를 갖는 EMP 테이블의 행의 개수를 확인하기 위해 COUNT를 수행했다. 이때 EMP에 동일 부서 번호가 존재하는지 확인하기 위해 DEPTNO 이외의 다른 컬럼을 사용하면, 성능에 유리하지 않으므로 가급적 조인 컬럼을 이용하는 것이 좋다.

```
SQL> SELECT d.*, DECODE(e.deptno, NULL, 'NO', 'YES') AS EMP
       FROM dept d, (SELECT DISTINCT deptno
                     FROM emp) e
     WHERE d.deptno = e.deptno (+) ;
```

GROUP BY, DISTINCT의 결과는 중복을 값을 제거할 수 있다. 때문에 조인 작업 시 1:1의 연결이 가능하고, 그로 인해 성능이 향상화될 수 있다.

답안 2. Correlated Subquery 사용

```
SQL> SELECT d.*, NVL((SELECT 'YES' FROM emp
                      WHERE deptno = d.deptno
                        AND rownum = 1 ), 'NO') AS EMP
     FROM dept d ;
```

FROM 절에 정의되는 집합은 그 결과를 화면에 출력해야 하는 집합만을 정의하는 것이 좋다. 현재 문제는 부서 정보와 사원의 존재 유무를 확인하는 것이다. 'YES','NO'의 값은 테이블에 저장되어 있는 값이 아니다. EMP 테이블의 접근이 필요한 것은 사실이지만, 모든 테이블을 반드시 FROM 절에 정의할 필요는 없다. 서브 쿼리는 다양한 곳에서 사용될 수 있으며 위와 같이 SELECT 리스트에도 정의 가능하다. 또한 근무 여부를 확인하기 위해 모든 사원을 확인할 필요도 없다.

```
SQL> SELECT d.*, NVL((SELECT 'YES'
                       FROM dual
                      WHERE EXISTS ( SELECT 1
                                       FROM emp
                                      WHERE deptno = d.deptno)), 'NO') AS EMP
       FROM dept d ;
```

EXISTS 연산을 사용해도 동일한 결과를 확인할 수 있다. 존재 유무를 확인해야 하는 경우가 있다면 COUNT나 DISTINCT 등을 이용하지 않는다. 불필요한 스캔이 필요할 수 있으므로 필요한 부분까지의 스캔만 수행할 수 있도록 ROWNUM을 활용하거나 EXISTS를 사용한다. 또한 모든 문장을 조인, 서브 쿼리 어느 한쪽을 우선적으로 고려하지 않는다. 상황에 맞는 문장의 작성 능력을 습득해야 한다.

9. COUNTRIES, EMPLOYEES 테이블을 이용하여, 'Canada'에서 근무 중인 사원 정보를 다음과 같이 검색하시오. 만약 추가적으로 필요한 테이블이 더 있다면 함께 사용합니다.

검색 결과

```
FIRST_NAME      LAST_NAME           SALARY JOB_ID     COUNTRY_NAME
--------------- --------------- ---------- ---------- ---------------
Michael         Hartstein            13000 MK_MAN     Canada
Pat             Fay                   6000 MK_REP     Canada

2 rows selected.
```

답안.

```
SQL> SELECT e.first_name, e.last_name, e.salary, e.job_id, c.country_name
       FROM employees e,
            departments d,
            locations l,
            countries c
      WHERE e.department_id = d.department_id
        AND d.location_id   = l.location_id
        AND l.country_id    = c.country_id
        AND c.country_name  = 'Canada';
```

EMPLOYEES 테이블과 COUNTRIES 테이블은 직접적인 관계가 없다. 때문에 관계를 가지고 있는 DEPARTMENTS, LOCATIONS 테이블을 징검다리 역할로 함께 조인을 수행해야 한다. DEPARTMENTS, LOCATIONS 테이블의 컬럼도 함께 검색해야 하는 경우라면 상관없겠지만 지금과 같은 경우에는 불필요한 조인 때문에 오히려 성능이 저하될 수도 있다.

업무의 특성을 고려하여 자주 조인이 되는 테이블은 직접적으로 연결이 가능하도록 설계 단계에서 고려해야 한다.

10. 'Asten'(CUSTOMERS.CITY)에 거주하는 고객이 관심 상품(WISHLIST)에 등록한 상품과 실제 주문한 상품(ORDERS, ORDER_ITEMS)의 제품별 금액의 합(SUM(UNIT_PRICE*QUANTITY))을 검색하시오. 이때 관심 상품에만 등록 된 금액과, 관심 상품 등록 없이 주문한 제품의 금액 합도 함께 검색한다.

검색 결과

CUST_ID	CUST_LNAME	PRODUCT_ID	WISH_TOT	ORDER_TOT
148	Steenburgen	1910	0	117
148	Steenburgen	1948	16035.6	10357.6
148	Steenburgen	2289	0	4752
148	Steenburgen	2302	0	4704
148	Steenburgen	2308	0	2106
148	Steenburgen	2322	0	990
148	Steenburgen	2326	0	52.8
148	Steenburgen	2330	0	64.9
148	Steenburgen	2334	0	16.5
148	Steenburgen	2335	0	4930.2
148	Steenburgen	2340	0	994
148	Steenburgen	2350	0	126462.6
148	Steenburgen	2365	0	2079
148	Steenburgen	2370	0	2520
148	Steenburgen	2375	0	2336
148	Steenburgen	2378	0	8966.1
148	Steenburgen	2394	0	4197.6
148	Steenburgen	2631	24	0
148	Steenburgen	2721	0	425
148	Steenburgen	2725	0	13.2
148	Steenburgen	2761	0	494
148	Steenburgen	2782	1210	1922
148	Steenburgen	3193	0	13.2
148	Steenburgen	3216	0	330
148	Steenburgen	3234	0	612
148	Steenburgen	3248	0	5519.8
148	Steenburgen	3252	0	725
148	Steenburgen	3334	1224	0
153	Sheen	1787	0	505
153	Sheen	1791	717	788.7
153	Sheen	1797	0	2436
153	Sheen	1799	0	9614
153	Sheen	1808	0	825
153	Sheen	1820	0	936
153	Sheen	1822	0	32965.9
153	Sheen	3077	711	0
170	Fonda	3106	0	7820
...				
326	Olin	1806	50	0
345	Weaver	2384	599	0
349	Glenn	3139	78.2	0

52 rows selected.

답안 1. FULL OUTER JOIN 사용

```
SQL> SELECT c.cust_id, c.cust_lname, x.product_id, x.wish_tot, x.order_tot
       FROM (SELECT cust_id, cust_lname
               FROM customers
              WHERE city      = 'Asten') c
            ,(SELECT NVL(w.cust_id, o.cust_id)       AS cust_id
                    ,NVL(w.product_id, o.product_id) AS product_id
                    ,NVL(w.wish_tot,0)               AS wish_tot
                    ,NVL(o.order_tot,0)              AS order_tot
              FROM (SELECT cust_id
                           ,product_id
                           ,SUM(unit_price * quantity) AS wish_tot
                      FROM wishlist
                     WHERE deleted = 'N'
                     GROUP BY cust_id, product_id) w
             FULL OUTER JOIN
                   (SELECT o.cust_id
                          ,i.product_id
                          ,SUM(i.unit_price * i.quantity) AS order_tot
                     FROM orders o
                         ,order_items i
                    WHERE o.order_id = i.order_id
                    GROUP BY o.cust_id, i.product_id) o
             ON  o.cust_id    = w.cust_id
             AND o.product_id = w.product_id) x
      WHERE c.cust_id = x.cust_id
     ORDER BY c.cust_id, x.product_id ;
```

답안 2. UNION ALL 사용

```
SQL> SELECT c.cust_id, c.cust_lname, x.product_id, x.wish_tot, x.order_tot
       FROM (SELECT cust_id, cust_lname
               FROM customers
              WHERE city = 'Asten') c ,
            (SELECT cust_id
                  ,product_id
                  ,SUM(wish_tot)  AS wish_tot,
                  ,SUM(order_tot) AS order_tot
               FROM (SELECT cust_id
                           ,product_id
                           ,unit_price * quantity AS wish_tot
                           ,0                     AS order_tot
                      FROM wishlist
                     WHERE deleted = 'N'
                     UNION ALL
                     SELECT o.cust_id
                           ,i.product_id
                           ,0
                           ,i.unit_price * i.quantity AS order_tot
                      FROM orders o,
                           order_items i
                     WHERE o.order_id = i.order_id)
            GROUP BY cust_id, product_id) x
     WHERE c.cust_id = x.cust_id
    ORDER BY c.cust_id, x.product_id ;
```

11. PRODS, SALES 테이블을 이용하여 제품별 판매 수량(QUANTITY_SOLD)의 합계를 다음과 같이 검색하시오. 단, 판매되지 않은 제품이 존재한다면 해당 제품도 함께 표시

검색 결과

```
   PROD_ID PROD_NAME                                          SOLD_SUM
---------- -------------------------------------------------- ----------
        13 5MP Telephoto Digital Camera                             6002
        14 17" LCD w/built-in HDTV Tuner                            5998
        15 Envoy 256MB - 40GB                                       5766
        16 Y Box                                                    6929
        17 Mini DV Camcorder with 3.5" Swivel LCD                   6160
        18 Envoy Ambassador                                         9591
        19 Laptop carrying case                                    10430
        20 Home Theatre Package with DVD-Audio/Video Play          10826
        21 18" Flat Panel Graphics Monitor                          5202
        22 Envoy External Keyboard                                  3441
        23 External 101-key keyboard                               19642
...

72 rows selected.
```

답안

```
SQL> SELECT p.prod_id, p.prod_name, SUM(s.quantity_sold) AS sold_sum
       FROM prods p, sales s
      WHERE p.prod_id = s.prod_id(+)
     GROUP BY p.prod_id, p.prod_name ;
```

그룹 함수의 결과를 생성할 때 조인의 결과를 이용하여 그룹 함수의 결과를 생성하는 경우가 있다. 위의 문장이 틀리지는 않았지만 실행 계획을 확인하면 성능상 취약점이 존재한다. 다음 문장의 실행 계획과 함께 비교하면 그 차이를 구분할 수 있다.

```
SQL> SELECT p.prod_id, p.prod_name, s.sold_sum
       FROM prods p, ( SELECT prod_id, SUM(quantity_sold) AS sold_sum
                         FROM sales
                        GROUP BY prod_id ) s
      WHERE p.prod_id = s.prod_id (+) ;
```

두 문장의 결과는 동일하다. 하지만 수행 속도의 차이는 분명히 확인될 수 있다. 이는 조인에 대한 부담이 줄어들기 때문에 발생하는 현상이다. SALES 테이블의 PROD_ID를 기준으로 그룹을 생성하면 SALES의 중복되는 PROD_ID는 고유한 키 값으로 Main Query에 리턴된다. 때문에 위의 문장은 1:1의 조인을 수행할 수 있게 되며, 앞서 확인한 문장은 1:M의 결과를 생성하고 그룹 함수의 결과를 생성하기 때문에 수행 속도의 차이가 발생한다.

추가 실습

```
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT p.prod_id, p.prod_name, SUM(s.quantity_sold) AS sold_sum
       FROM prods p, sales s
      WHERE p.prod_id = s.prod_id
     GROUP BY p.prod_id, p.prod_name ;

Elapsed: 00:00:00.10
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |           |    71 |  3976 |  1256   (3)| 00:00:01 |
|   1 |  MERGE JOIN                   |           |    71 |  3976 |  1256   (3)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| PRODS     |    72 |  2160 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | PROD_IX01 |    72 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |           |    71 |  1846 |  1254   (3)| 00:00:01 |
|   5 |    VIEW                       | VW_GBC_5  |    71 |  1846 |  1253   (3)| 00:00:01 |
|   6 |     HASH GROUP BY             |           |    71 |   497 |  1253   (3)| 00:00:01 |
|   7 |      TABLE ACCESS FULL        | SALES     |   913K|  6241K|  1230   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("P"."PROD_ID"="ITEM_1")
       filter("P"."PROD_ID"="ITEM_1")


SQL> SET AUTOTRACE OFF
```

INNER JOIN의 결과를 생성하는 문장으로 수정하였다. 한가지 놀라운 사실은, 작성된 문장은 조인 후 그룹 함수를 적용하도록 했지만, 실행 계획은 그룹을 먼저 생성하고 조인을 수행했다. 때문에 더 빠른 수행 시간을 확인할 수 있다.

Oracle Database의 버전이 올라갈수록 사용자가 작성한 문장은 지금처럼 최적화된 실행 계획을 생성할 수 있도록 변경된다. 이를 Query Transformation이라 하며 자세한 사항은 튜닝 과정에서 소개한다.

12. EMP 테이블에서 1981년도에 입사한 사원들을 입사 월별로 인원수를 검색하시오.
단, 사원이 없는 월도 함께 출력

검색 결과

```
HIRE                CNT
------------- ----------
1981/01               0
1981/02               2
1981/03               0
1981/04               1
1981/05               1
1981/06               1
1981/07               0
1981/08               0
1981/09               2
1981/10               0
1981/11               1
1981/12               2

12 rows selected.
```

답안.

```
SQL> SELECT b.hire, NVL(a.cnt,0) CNT
       FROM (SELECT TO_CHAR(hiredate,'YYYY/MM') hire, count(*) cnt
               FROM emp
              WHERE hiredate BETWEEN TO_DATE('81/01/01','RR/MM/DD')
                                 AND TO_DATE('81/12/31','RR/MM/DD')
             GROUP BY TO_CHAR(hiredate,'YYYY/MM')) a,
            (SELECT '1981/'||LPAD(LEVEL,2,0) hire
               FROM dual
             CONNECT BY LEVEL <= 12) b
      WHERE a.hire (+) = b.hire
     ORDER BY 1 ;
```

13. EMPLOYEES 테이블을 이용하여 부서별 최대 급여를 받는 사원 정보를 검색하시오.

검색 결과

```
LAST_NAME                     SALARY JOB_ID     DEPARTMENT_ID
------------------------- ---------- ---------- -------------
Whalen                          4400 AD_ASST               10
Hartstein                      13000 MK_MAN                20
Higgins                        12008 AC_MGR               110
King                           24000 AD_PRES               90
Hunold                          9000 IT_PROG               60
Mourgos                         5800 ST_MAN                50
Abel                           11000 SA_REP                80

7 rows selected.
```

답안 1. MAX 함수를 대체하는 인덱스 사용

```
SQL> CREATE INDEX empl_x01 ON employees(department_id, salary) ;

SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT last_name, salary, job_id, department_id
       FROM employees e
      WHERE salary = ( SELECT /*+ index_rs_desc(se empl_x01) */ salary
                         FROM employees se
                        WHERE department_id = e.department_id
                          AND ROWNUM        = 1 ) ;
```

```
-----------------------------------------------------------------------------------------------
¦ Id  ¦ Operation                     ¦ Name      ¦ Rows  ¦ Bytes ¦ Cost (%CPU)¦ Time     ¦
-----------------------------------------------------------------------------------------------
¦   0 ¦ SELECT STATEMENT              ¦           ¦     1 ¦    22 ¦    10   (0)¦ 00:00:01 ¦
¦*  1 ¦  FILTER                      ¦           ¦       ¦       ¦           ¦          ¦
¦   2 ¦   TABLE ACCESS FULL          ¦ EMPLOYEES ¦    20 ¦   440 ¦     3   (0)¦ 00:00:01 ¦
¦*  3 ¦   COUNT STOPKEY              ¦           ¦       ¦       ¦           ¦          ¦
¦*  4 ¦    INDEX RANGE SCAN DESCENDING¦ EMPL_X01  ¦     1 ¦     7 ¦     1   (0)¦ 00:00:01 ¦
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SALARY"= (SELECT /*+ INDEX_RS_DESC ("SE" "EMPL_X01") */ "SALARY"
              FROM "EMPLOYEES" "SE" WHERE ROWNUM=1 AND "DEPARTMENT_ID"=:B1))
   3 - filter(ROWNUM=1)
   4 - access("DEPARTMENT_ID"=:B1)
```

필요한 인덱스를 생성하여 인덱스의 스캔 방향을 조정하면, MIN/MAX 함수를 사용하지 않고 최소/최댓값을 찾을 수 있다. 하지만 Subquery에서의 ROWNUM의 사용은 FILTER 방식을 사용하게 되고, 후보행이 많은 경우 성능은 저하될 수 있다.

답안 2. MAX 함수 사용

```
SQL> SELECT e.last_name, e.salary, e.job_id, e.department_id
       FROM employees e
      WHERE e.salary = ( SELECT MAX(salary)
                           FROM employees se
                          WHERE se.department_id = e.department_id ) ;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |    52 |  2132 |     5  (40)| 00:00:01 |
|*  1 |  FILTER                       |                |       |       |            |          |
|   2 |   HASH GROUP BY               |                |    52 |  2132 |     5  (40)| 00:00:01 |
|   3 |    MERGE JOIN                 |                |    52 |  2132 |     4  (25)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES      |    20 |   680 |     2   (0)| 00:00:01 |
|   5 |      INDEX FULL SCAN          | EMPLOYEES_IX04 |    19 |       |     1   (0)| 00:00:01 |
|*  6 |     SORT JOIN                 |                |    20 |   140 |     2  (50)| 00:00:01 |
|   7 |      INDEX FULL SCAN          | EMPL_X01       |    20 |   140 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("E"."SALARY"=MAX("SALARY"))
   6 - access("SE"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
       filter("SE"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
```

MAX 함수를 사용하여 동일 결과를 검색할 수 있도록 수정하였다. 해당 실행 계획은 아래와 같이 Query Transformation이 진행되면서 실행된 것을 확인할 수 있다.

```
SQL> SELECT e.last_name, e.salary, e.job_id, e.department_id
     FROM employees e, employees se
     WHERE e.department_id = se.department_id
     GROUP BY se.department_id, e.rowid, e.department_id, e.job_id, e.salary, e.last_name
     HAVING e.salary = MAX(se.salary) ;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |    52 |  2132 |     5  (40)| 00:00:01 |
|*  1 |  FILTER                       |                |       |       |            |          |
|   2 |   HASH GROUP BY               |                |    52 |  2132 |     5  (40)| 00:00:01 |
|   3 |    MERGE JOIN                 |                |    52 |  2132 |     4  (25)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES      |    20 |   680 |     2   (0)| 00:00:01 |
|   5 |      INDEX FULL SCAN          | EMPLOYEES_IX04 |    19 |       |     1   (0)| 00:00:01 |
|*  6 |     SORT JOIN                 |                |    20 |   140 |     2  (50)| 00:00:01 |
|   7 |      INDEX FULL SCAN          | EMPL_X01       |    20 |   140 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("E"."SALARY"=MAX("SE"."SALARY"))
   6 - access("E"."DEPARTMENT_ID"="SE"."DEPARTMENT_ID")
       filter("E"."DEPARTMENT_ID"="SE"."DEPARTMENT_ID")
```

Query Transformation의 결과를 확인한 것이지 직접 위와 같은 문장을 작성할 필요는 없다. 경우에 따라 Unnesting이 수행되면 조인의 방식으로 수행될 수도 있고, FILTER 방식으로 수행될 수도 있다. 단, 위의 실행 계획에 아쉬운 부분은 EMPLOYEES 테이블을 FULL SCAN 했는데, 인덱스의 접근이 다시 필요한 부분이다.

답안 3. 분석 함수 사용

```
SQL> SELECT last_name,
            salary,
            job_id,
            department_id
      FROM ( SELECT last_name,
                    salary,
                    job_id,
                    department_id,
                    MAX(salary) OVER(PARTITION BY department_id) AS max
               FROM employees e
              WHERE department_id IS NOT NULL)
     WHERE salary = max ;
```

```
-----------------------------------------------------------------------------------------------
¦ Id  ¦ Operation                     ¦ Name           ¦ Rows  ¦ Bytes ¦ Cost (%CPU)¦ Time     ¦
-----------------------------------------------------------------------------------------------
¦   0 ¦ SELECT STATEMENT              ¦                ¦    19 ¦  1140 ¦     2   (0)¦ 00:00:01 ¦
¦*  1 ¦  VIEW                        ¦                ¦    19 ¦  1140 ¦     2   (0)¦ 00:00:01 ¦
¦   2 ¦   WINDOW BUFFER              ¦                ¦    19 ¦   418 ¦     2   (0)¦ 00:00:01 ¦
¦   3 ¦    TABLE ACCESS BY INDEX ROWID¦ EMPLOYEES     ¦    19 ¦   418 ¦     2   (0)¦ 00:00:01 ¦
¦*  4 ¦     INDEX FULL SCAN          ¦ EMPLOYEES_IX04 ¦    19 ¦       ¦     1   (0)¦ 00:00:01 ¦
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SALARY"="MAX")
   4 - filter("DEPARTMENT_ID" IS NOT NULL)
```

분석 함수를 사용하면 FULL TABLE SCAN을 통해 필요한 결과를 모두 생성할 수 있다. 때문에 불필요한 인덱스의 접근은 필요 없다. 오히려 앞서 생성한 인덱스를 삭제해도 무방하다. 하지만 테이블의 크기가 크다면 전체 테이블의 접근이 진행될 때까지 조건에 만족하는 행은 검색이 불가능할 수 있다. 즉, ALL_ROWS 상황에 최적화된 실행 계획이다.

답안 4. 분석 함수 사용 (부분 범위 처리 상황)

```
SQL> SELECT /*+ use_nl(m e) */
            e.last_name, e.salary, e.job_id, e.department_id
       FROM employees e,
            (SELECT *
             FROM (SELECT /*+ index_ffs(e(department_id,salary)) */
                          rowid  AS rid,
                          salary,
                          MAX(salary) OVER(PARTITION BY department_id) AS max
                     FROM employees e
                    WHERE department_id IS NOT NULL)
            WHERE salary = max) m
      WHERE e.rowid = m.rid ;
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    19 |  1368 |    22   (5)| 00:00:01 |
|   1 |  NESTED LOOPS                |           |    19 |  1368 |    22   (5)| 00:00:01 |
|*  2 |   VIEW                       |           |    19 |   722 |     3  (34)| 00:00:01 |
|   3 |    WINDOW SORT               |           |    19 |   361 |     3  (34)| 00:00:01 |
|*  4 |     INDEX FAST FULL SCAN     | EMPL_X01  |    19 |   361 |     2   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY USER ROWID| EMPLOYEES |     1 |    34 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("SALARY"="MAX")
   4 - filter("DEPARTMENT_ID" IS NOT NULL)
```

일반적으로 인덱스는 테이블보다는 크기가 작다. 때문에 테이블의 접근 대신, 인덱스를 스캔하여 조건에 만족하는 행을 찾고, ROWID를 이용하여 테이블에 접근하는 방식이다. 항상 위와 같이 문장을 작성할 수는 없다. 또한 성능과 관련된 자세한 사항은 SQL Tuning 과정에서 확인할 것이다. 여기서 확인하는 부분은 필요에 따라 위와 같이 작성해야 하는 경우도 있다는 것을 확인한다.

```
SQL> DROP INDEX empl_x01 ;
SQL> SET AUTOTRACE OFF
```

6. WITH 절

WITH 절

Oracle Database 10g부터 지원하는 WITH 절은 하나의 쿼리 내에서 반복되는 쿼리 집합을 미리 정의하여 임시 테이블에 저장된 데이터로 사용한다. 즉, 하나의 쿼리 내에서만 접근 가능한 임시 테이블이 WITH 절이다.

```
SQL> SELECT deptno, SUM(sal) AS SUM
       FROM emp
     GROUP BY deptno
     HAVING SUM(sal) > ( SELECT AVG(SUM(sal))
                           FROM emp
                         GROUP BY deptno ) ;
    DEPTNO        SUM
---------- ----------
        20      10875

1 row selected.
```

• 발견 사항

부서별 급여의 합계 계산 작업이 두 번 실행된다.

```
SQL> WITH sum_sal AS ( SELECT deptno, SUM(sal) AS SUM
                       FROM emp
                       GROUP BY deptno )
     SELECT *
     FROM sum_sal
     WHERE sum > ( SELECT AVG(sum) FROM sum_sal ) ;
    DEPTNO        SUM
---------- ----------
        20      10875

1 row selected.
```

• 발견 사항

중복될 수 있는 Subquery를 WITH 절에서 한 번만 실행할 수 있다. WITH 절에 정의된 집합은 해당 문장 내에서만 사용 가능한 임시 테이블의 이름을 가지므로 문장 어디에서든 해당 집합에 대한 접근이 가능하다. 때문에 하나의 쿼리 안에서 반복되는 동일 쿼리 집합이 있다면 이를 WITH 절에 먼저 정의해 놓음으로써 반복적인 쿼리의 재 실행을 줄일 수 있다.

그렇다면 성능은?

```
SQL> SET AUTOTRACE ON EXPLAIN
SQL> WITH sum_sal AS ( SELECT deptno, SUM(sal) AS SUM
                       FROM emp
                       GROUP BY deptno )
     SELECT *
     FROM sum_sal
     WHERE sum > ( SELECT AVG(sum) FROM sum_sal ) ;
```

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                            |     3 |    78 |     8  (13)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION |                            |       |       |            |          |
|   2 |   LOAD AS SELECT           | SYS_TEMP_0FD9D6640_C58506  |       |       |            |          |
|   3 |    HASH GROUP BY           |                            |     3 |    21 |     4  (25)| 00:00:01 |
|   4 |     TABLE ACCESS FULL      | EMP                        |    14 |    98 |     3   (0)| 00:00:01 |
|*  5 |   VIEW                     |                            |     3 |    78 |     2   (0)| 00:00:01 |
|   6 |    TABLE ACCESS FULL       | SYS_TEMP_0FD9D6640_C58506  |     3 |    21 |     2   (0)| 00:00:01 |
|   7 |    SORT AGGREGATE          |                            |     1 |    13 |            |          |
|   8 |     VIEW                   |                            |     3 |    39 |     2   (0)| 00:00:01 |
|   9 |      TABLE ACCESS FULL     | SYS_TEMP_0FD9D6640_C58506  |     3 |    21 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - filter("SUM"> (SELECT AVG("SUM") FROM  (SELECT /*+ CACHE_TEMP_TABLE ("T1") */ "C0"
              "DEPTNO","C1" "SUM" FROM "SYS"."SYS_TEMP_0FD9D6640_C58506" "T1") "SUM_SAL"))
```

- **발견 사항**

WITH 절의 집합은 문장 실행 시 가장 먼저 실행되며 해당 결과를 임시 데이터로 저장한다. 문제는 이러한 임시 데이터의 반복 접근이 일어날 때마다 물리적인 I/O 가 증가되어 오히려 성능이 저하될 가능성도 있다.

※ WITH 절의 Query Transformation

```
SQL> WITH sum_sal AS ( SELECT deptno, SUM(sal) AS SUM
                       FROM emp
                       GROUP BY deptno )
     SELECT *
     FROM dept d, sum_sal s
     WHERE d.deptno = s.deptno ;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |     3 |   138 |     7  (29)| 00:00:01 |
|   1 |  MERGE JOIN                   |                |     3 |   138 |     7  (29)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPT           |     4 |    80 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | DEPT_DEPTNO_IX |     4 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |                |     3 |    78 |     5  (40)| 00:00:01 |
|   5 |    VIEW                       |                |     3 |    78 |     4  (25)| 00:00:01 |
|   6 |     HASH GROUP BY             |                |     3 |    21 |     4  (25)| 00:00:01 |
|   7 |      TABLE ACCESS FULL        | EMP            |    14 |    98 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("D"."DEPTNO"="S"."DEPTNO")
       filter("D"."DEPTNO"="S"."DEPTNO")
```

• 발견 사항

WITH 절에 정의된 문장이 한 번만 사용되면 Inline View 형식으로 변형되어 사용된다. 즉, 임시 데이터의 집합이 생성되지 않는다.

• 결론

1. WITH 절은 반복되는 쿼리의 결과를 문장 시작 전 액세스하여 임시 테이블로 저장, 재 사용을 목적으로 한다.
2. 문장 안에 WITH 절의 Subquery 가 두 번 이상 사용되면 임시 테이블이 생성되며, 한 번 사용되면 Inline View 형식으로 실행된다.
3. WITH 절의 임시 테이블 형식의 사용은 Physical I/O를 동반할 수 있으므로 그 크기가 크지 않은 경우에 유리하다.
4. MATERIALIZE, INLINE 힌트를 이용하여 형식 지정 가능 (11gNF)
5. "_with_subquery" 파라미터로 제어 가능 (11gNF)

• 힌트 사용 방법

```
SQL> WITH sum_sal AS ( SELECT /*+ materialize */ deptno, SUM(sal) AS SUM
                       FROM emp
                       GROUP BY deptno )
     SELECT *
     FROM dept d, sum_sal s
     WHERE d.deptno = s.deptno ;
```

```
-------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                            |     3 |   138 |     9  (23)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION               |                            |       |       |            |          |
|   2 |   LOAD AS SELECT (CURSOR DURATION MEMORY)| SYS_TEMP_0FD9D668E_15C723  |       |       |            |          |
|   3 |    HASH GROUP BY                         |                            |     3 |    21 |     4  (25)| 00:00:01 |
|   4 |     TABLE ACCESS FULL                    | EMP                        |    14 |    98 |     3   (0)| 00:00:01 |
|   5 |   MERGE JOIN                             |                            |     3 |   138 |     5  (20)| 00:00:01 |
|   6 |    TABLE ACCESS BY INDEX ROWID           | DEPT                       |     4 |    80 |     2   (0)| 00:00:01 |
|   7 |     INDEX FULL SCAN                      | DEPT_DEPTNO_IX             |     4 |       |     1   (0)| 00:00:01 |
|*  8 |    SORT JOIN                             |                            |     3 |    78 |     3  (34)| 00:00:01 |
|   9 |     VIEW                                 |                            |     3 |    78 |     2   (0)| 00:00:01 |
|  10 |      TABLE ACCESS FULL                   | SYS_TEMP_0FD9D668E_15C723  |     3 |    21 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   8 - access("D"."DEPTNO"="S"."DEPTNO")
       filter("D"."DEPTNO"="S"."DEPTNO")
```

```
SQL> WITH sum_sal AS ( SELECT /*+ inline */ deptno, SUM(sal) AS SUM
                       FROM emp
                       GROUP BY deptno )
     SELECT *
     FROM sum_sal
     WHERE sum > ( SELECT AVG(sum) FROM sum_sal ) ;
```

```
-----------------------------------------------------------------------------
| Id  | Operation             | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |      |     1 |     7 |     4  (25)| 00:00:01 |
|*  1 |  FILTER               |      |       |       |            |          |
|   2 |   HASH GROUP BY       |      |     1 |     7 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL  | EMP  |    14 |    98 |     3   (0)| 00:00:01 |
|   4 |   SORT AGGREGATE      |      |     1 |    13 |            |          |
|   5 |    VIEW               |      |     3 |    39 |     4  (25)| 00:00:01 |
|   6 |     SORT GROUP BY     |      |     3 |    21 |     4  (25)| 00:00:01 |
|   7 |      TABLE ACCESS FULL| EMP  |    14 |    98 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(SUM("SAL")> (SELECT SUM("SUM")/COUNT("SUM") FROM  (SELECT
              "DEPTNO" "DEPTNO",SUM("SAL") "SUM" FROM "EMP" "EMP" GROUP BY "DEPTNO")
              "SUM_SAL"))
```

7. 그룹 함수 활용

1. EMP 테이블에서, 부서별 급여의 합계를 검색하면서 전체 급여의 합계도 함께 검색하시오.

검색 결과

```
    DEPTNO    SUM_SAL
---------- ----------
        10       8750
        20      10875
        30       9400
                29025

4 rows selected.
```

답안 1. UNION ALL 사용

```
SQL> SELECT deptno, SUM(sal) AS sum_sal
       FROM emp
     GROUP BY deptno
     UNION ALL
     SELECT NULL, SUM(sal)
       FROM emp
     ORDER BY deptno ;
```

다양한 Grouping이 필요할 때 가장 손쉬운 방법이 UNION ALL을 이용하는 문장이지만 테이블의 반복 접근이 성능을 저하시킬 수도 있다. 하지만 경우에 따라 각 집합을 개별적으로 최적화하는 것도 가능하므로 성능에 문제가 없다면 사용해도 된다.

답안 2. ROLLUP 사용

```
SQL> SELECT deptno, SUM(sal) AS sum_sal
       FROM emp
     GROUP BY ROLLUP(deptno) ;
```

GROUP BY의 확장된 사용 방법이며 ROLLUP, CUBE를 이용하면 하나의 쿼리 집합을 통해 다양한 Grouping 결과를 검색 가능하다. 테이블의 반복 접근은 줄기 때문에 성능 향상이 있을 수도 있지만 아닌 경우도 존재한다. 때문에 실제 사용된 실행 계획 및 성능 측정이 필요하다.

추가 실습. CUBE 사용

```
SQL> SELECT deptno, job, SUM(sal)
       FROM emp
     GROUP BY CUBE(deptno, job)
     ORDER BY deptno, job ;
    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 ANALYST         6000
        20 CLERK           1900
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400
           ANALYST         6000
           CLERK           4150
           MANAGER         8275
           PRESIDENT       5000
           SALESMAN        5600
                          29025

18 rows selected.
```

ROLLUP은 목록의 오른쪽 끝부터 하나의 단위씩 컬럼을 제거하면서 Grouping을 한다. 때문에 일부 컬럼은 Grouping에 참여하지 못하지만 CUBE는 모든 Grouping 식을 사용하므로 모든 Grouping 결과를 다 만들고자 할 때 사용할 수 있다.

2. EMP 테이블에서 DEPTNO, JOB 컬럼으로 그룹화된 급여의 합계, DEPTNO 컬럼만으로 그룹화된 급여의 합계를 검색하시오. (즉, 전체 급여의 합계는 출력하지 않는다.)

검색 결과

```
    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 CLERK           1900
        20 ANALYST         6000
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400

12 rows selected.
```

답안

```
SQL> SELECT deptno, job, SUM(sal)
       FROM emp
     GROUP BY deptno, ROLLUP(job) ;
```

특정 컬럼이 항상 Grouping에 참여해야 할 경우 GROUP BY 절에 ROLLUP, CUBE 연산 없이 컬럼 정의가 가능하다.

추가 실습

```
SQL> SELECT TO_CHAR(hiredate,'YYYY') AS YEAR, deptno, job, SUM(sal),
            GROUPING_ID(TO_CHAR(hiredate,'YYYY'), deptno, job) AS EXPRESSION
       FROM emp
     GROUP BY ROLLUP(TO_CHAR(hiredate,'YYYY')), CUBE(deptno, job)
     ORDER BY expression, year, deptno, job ;
YEAR     DEPTNO JOB         SUM(SAL) EXPRESSION
---- ---------- --------- ---------- ----------
1980         20 CLERK            800          0          -- GROUP BY YEAR, DEPTNO, JOB
1981         10 MANAGER         2450          0
1981         10 PRESIDENT       5000          0
...
1980         20                  800          1          -- GROUP BY YEAR, DEPTNO
1981         10                 7450          1
1981         20                 5975          1
...
1980            CLERK            800          2          -- GROUP BY YEAR, JOB
1981            ANALYST         3000          2
1981            CLERK            950          2
...
1980                             800          3          -- GROUP BY YEAR
1981                           22825          3
1982                            1300          3
1987                            4100          3
             10 CLERK           1300          4          -- GROUP BY DEPTNO, JOB
             10 MANAGER         2450          4
             10 PRESIDENT       5000          4
...
             10                 8750          5          -- GROUP BY DEPTNO
             20                10875          5
             30                 9400          5
                ANALYST         6000          6          -- GROUP BY JOB
                CLERK           4150          6
                MANAGER         8275          6
                PRESIDENT       5000          6
                SALESMAN        5600          6
                                29025          7          -- GROUP BY ()

48 rows selected.
```

3. EMP 테이블의 사원 정보를 검색하면서 각 부서의 급여 합계를 함께 출력하시오.

검색 결과

```
    DEPTNO      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
        10       7782 CLARK            2450
        10       7839 KING             5000
        10       7934 MILLER           1300
        10                             8750
        20       7369 SMITH             800
        20       7566 JONES            2975
        20       7788 SCOTT            3000
        20       7876 ADAMS            1100
        20       7902 FORD             3000
        20                            10875
        30       7900 JAMES             950
        30       7499 ALLEN            1600
        30       7521 WARD             1250
        30       7654 MARTIN           1250
        30       7698 BLAKE            2850
        30       7844 TURNER           1500
        30                             9400

17 rows selected.
```

답안

```
SQL> SELECT deptno, empno, ename, SUM(sal) AS sal
       FROM emp
     GROUP BY deptno, ROLLUP((empno,ename)) ;
```

ROLLUP, CUBE 진행 시 Grouping 후 제거할 단위를 둘 이상의 컬럼으로 지정 가능하다.

추가 실습

```
SQL> SELECT deptno, empno, ename, SUM(sal) AS sal
       FROM emp
     GROUP BY ROLLUP(deptno, (empno,ename)) ;
    DEPTNO      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
        10       7782 CLARK            2450
        10       7839 KING             5000
        10       7934 MILLER           1300
...
                                      29025

18 rows selected.
```

DEPTNO 컬럼을 ROLLUP 연산에 포함 시키면 전체 급여의 합계도 함께 출력 가능하다.

4. EMP 테이블에서 30번 부서에 근무하는 사원 정보를 검색하면서, 급여 합계와 평균을 다음과 같이 검색하시오.

검색 결과

```
    DEPTNO      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
        30       7900 JAMES             950
        30       7499 ALLEN            1600
        30       7521 WARD             1250
        30       7654 MARTIN           1250
        30       7698 BLAKE            2850
        30       7844 TURNER           1500
                                       9400            -- 급여 합계
                                    1566.67            -- 급여 평균

8 rows selected.
```

답안

```
SQL> SELECT deptno, empno, ename,
            DECODE(grouping_id(1,deptno),3,ROUND(AVG(sal),2),SUM(sal)) AS sal
       FROM emp
      WHERE deptno = 30
     GROUP BY ROLLUP(1,(deptno,empno,ename));
```

추가 실습

SELECT 문장에 포함된 리터럴 값은 모든 행에 동일한 값을 생성한다. 때문에 다음과 같이 문장을 실행하면 모든 행이 1의 값을 가질 수 있다. 그럼 이 첫 번째 행으로 GROUP BY 절을 사용하면?

```
SQL> SELECT 1, sal FROM emp ;
         1        SAL
---------- ----------
         1       3000
         1       2975
         1        800
...
14 rows selected.

SQL> SELECT 1, SUM(sal)
       FROM emp
     GROUP BY 1 ;
         1   SUM(SAL)
---------- ----------
         1      29025       -- 전체 급여의 합계 계산 가능

1 row selected.
```

ROLLUP 연산을 사용하면?

```
SQL> SELECT SUM(sal)
       FROM emp
     GROUP BY ROLLUP(1) ;
  SUM(SAL)
----------
     29025
     29025

2 rows selected.
```

1로 그룹을 생성하고, 1이 제외되면서 SUM(sal)의 결과가 두 번 검색되었다. 여기서 GROUP BY 절에 포함 여부만 확인할 수 있다면 하나의 쿼리 내에서 합계와 평균을 함께 검색할 수 있다.

ROLLUP, CUBE 등을 이용하여 하나 이상의 그룹을 생성하고 있을 때, 특정 컬럼의 Grouping 참여 여부를 확인할 수 있는 함수가 GROUPING, GROUPING_ID 함수이다. GROUPING 함수는 하나의 컬럼에 대해 Grouping 참여 여부를 확인하며, GROUPING_ID 함수는 하나 이상의 컬럼을 이용하여 각 컬럼의 Grouping 여부를 확인할 수 있게 한다.

```
SQL> SELECT deptno, job, SUM(sal),
            GROUPING(deptno), GROUPING(job), GROUPING_ID(deptno,job)
       FROM emp
      GROUP BY CUBE(deptno,job)
      ORDER BY 6 ;
    DEPTNO JOB         SUM(SAL) GROUPING(DEPTNO) GROUPING(JOB) GROUPING_ID(DEPTNO,JOB)
---------- --------- ---------- ---------------- -------------- ----------------------
        10 MANAGER         2450                0              0                      0
        30 MANAGER         2850                0              0                      0
        30 CLERK            950                0              0                      0
        20 MANAGER         2975                0              0                      0
        20 ANALYST         6000                0              0                      0
        20 CLERK           1900                0              0                      0
        10 PRESIDENT       5000                0              0                      0
        30 SALESMAN        5600                0              0                      0
        10 CLERK           1300                0              0                      0
        20                10875                0              1                      1
        30                 9400                0              1                      1
        10                 8750                0              1                      1
           PRESIDENT       5000                1              0                      2
           SALESMAN        5600                1              0                      2
           MANAGER         8275                1              0                      2
           ANALYST         6000                1              0                      2
           CLERK           4150                1              0                      2
                          29025                1              1                      3

18 rows selected.
```

GROUPING, GROUPING_ID 함수를 이용하면 필요한 둘 이상의 함수를 실행 시킬 수 있다.

```
SQL> SELECT DECODE(GROUPING(1), 0, SUM(sal)
                                , AVG(sal)) AS COMPUTE
       FROM emp
     GROUP BY ROLLUP(1) ;
   COMPUTE
----------
     29025
2073.21429

2 rows selected.
```

5. EMP 테이블의 사원 정보를 검색하면서 각 부서별 급여 합계와 평균, 전체 급여 합계와 평균을 다음과 같이 검색하시오.

검색 결과

```
    DEPTNO      EMPNO ENAME             SAL
---------- ---------- ---------- ----------
        10       7782 CLARK            2450
        10       7839 KING             5000
        10       7934 MILLER           1300
        10            DEPT_SUM         8750
        10            DEPT_AVG       2916.7
        20       7369 SMITH             800
        20       7566 JONES            2975
        20       7788 SCOTT            3000
        20       7876 ADAMS            1100
        20       7902 FORD             3000
        20            DEPT_SUM        10875
        20            DEPT_AVG         2175
        30       7900 JAMES             950
        30       7499 ALLEN            1600
        30       7521 WARD             1250
        30       7654 MARTIN           1250
        30       7698 BLAKE            2850
        30       7844 TURNER           1500
        30            DEPT_SUM         9400
        30            DEPT_AVG       1566.7
                      TOTAL_SUM       29025
                      TOTAL_AVG      2073.2

22 rows selected.
```

답안

```
SQL> SELECT deptno, empno,
            DECODE(grouping_id(1,deptno,2,empno), 1,'DEPT_SUM',   3,'DEPT_AVG',
                                                  7,'TOTAL_SUM', 15,'TOTAL_AVG',
                                                     ename) AS ename,
            DECODE(grouping_id(1,deptno,2,empno), 1,SUM(sal),     3,ROUND(AVG(sal),1),
                                                  7,SUM(sal),    15,ROUND(AVG(sal),1),
                                                    SUM(sal)) AS sal
       FROM emp
      GROUP BY ROLLUP(1,deptno,2,(empno,ename));
```

6. EMP 테이블에서 DEPTNO, JOB 컬럼으로 그룹화된 급여의 합계와 DEPTNO, MGR 컬럼으로 그룹화된 급여의 합계를 함께 출력하시오.

검색 결과

```
    DEPTNO JOB              MGR    SUM_SAL
---------- --------- ---------- ----------
        20                7839       2975
        10                7839       2450
        20                7566       6000
        30                7698       6550
        10                7782       1300
        20                7902        800
        10                           5000
        20                7788       1100
        30                7839       2850
        20 MANAGER                    2975
        20 CLERK                      1900
        30 SALESMAN                   5600
        30 CLERK                       950
        10 PRESIDENT                  5000
        30 MANAGER                    2850
        10 CLERK                      1300
        20 ANALYST                    6000
        10 MANAGER                    2450

18 rows selected.
```

답안 1. UNION ALL 사용

```
SQL> SELECT deptno, TO_CHAR(NULL) AS job, mgr, SUM(sal) AS sum_sal
       FROM emp
     GROUP BY deptno, mgr
     UNION ALL
     SELECT deptno, job, NULL, SUM(sal)
       FROM emp
     GROUP BY deptno, job ;
```

답안 2. CUBE 활용

```
SQL> SELECT deptno, job, mgr, SUM(sal) AS sum_sal
     FROM emp
     GROUP BY CUBE(deptno,job,mgr)
     HAVING GROUPING_ID(deptno,job,mgr) = 1
         OR GROUPING_ID(deptno,job,mgr) = 2 ;
```

답안 3. GROUPING SETS 사용

```
SQL> SELECT deptno, job, mgr, SUM(sal) AS sum_sal
       FROM emp
     GROUP BY GROUPING SETS ((deptno,job), (deptno,mgr)) ;
```

GROUPING SETS 은 복잡한 UNION ALL을 대체할 수 있으며 필요한 계산식만 표시함으로써 다양한 그룹화 작업에 간결한 문장을 작성할 수 있다. 단, 필요한 후보 집합을 검색하여 임시 테이블로 저장하기 때문에 후보 집합의 크기가 크면 물리적인 I/O가 증가되는 경우도 발생한다. 때문에 성능이 항상 최적이라고 할 수는 없다.

7. EMP 테이블에서 DEPTNO, JOB 컬럼으로 Grouping 된 급여의 합계를 다음과 같이 검색하시오.

검색 결과

```
    DEPTNO    ANALYST      CLERK    MANAGER  PRESIDENT   SALESMAN
---------- ---------- ---------- ---------- ---------- ----------
        10                  1300       2450       5000
        20       6000       1900       2975
        30                   950       2850                  5600

3 rows selected.
```

답안 1. SUM(DECODE( )) 사용

```
SQL> SELECT deptno, SUM(DECODE(job, 'ANALYST', sal))  AS analyst,
                    SUM(DECODE(job,'CLERK', sal))     AS clerk,
                    SUM(DECODE(job,'MANAGER', sal))   AS manager,
                    SUM(DECODE(job,'PRESIDENT', sal)) AS president,
                    SUM(DECODE(job,'SALESMAN', sal))  AS salesman
      FROM emp
     GROUP BY deptno
     ORDER BY deptno ;
```

DECODE 함수는 WHERE 절의 조건에 만족하는 행을 대상으로 IF 문의 처리를 가능하게 한다. 때문에 각 JOB 별로 원하는 표현식을 정의하면 다양한 형식의 Pivoting 결과를 생성할 수 있다. 단, 복잡한 형식의 DECODE 함수는 반복적인 함수 호출로 인한 성능이 저하될 수도 있다.

답안 2. Inline View를 이용하여 Grouping 후 SUM(DECODE( )) 사용

```
SQL> SELECT deptno, SUM(DECODE(job, 'ANALYST', sal))  AS analyst,
                    SUM(DECODE(job,'CLERK', sal))     AS clerk,
                    SUM(DECODE(job,'MANAGER', sal))   AS manager,
                    SUM(DECODE(job,'PRESIDENT', sal)) AS president,
                    SUM(DECODE(job,'SALESMAN', sal))  AS salesman
      FROM ( SELECT deptno, job, SUM(sal) AS sal
               FROM emp
              GROUP BY deptno, job )
     GROUP BY deptno
     ORDER BY deptno ;
```

Inline View를 이용하여 필요한 Grouping의 결과를 미리 생성하면 DECODE의 반복적인 호출 작업이 줄어들기 때문에 성능 개선이 가능하다.

8. 분석 함수 활용

분석 함수 활용

SQL은 Relationship 이 존재하는 테이블 구조의 데이터를 제어하기 위한 언어이다. 이러한 SQL은 DB에 저장된 데이터를 제어하기 위한 매우 강력한 언어이지만, 다양한 Business Intelligence Calculation 을 수행하기에는 부족함이 존재한다. 때문에 복잡한 형태의 분석 작업을 진행하려면 과도한 프로그래밍이 사용되고, 그로 인해 성능은 저하되기도 한다. Oracle Database 8i부터는 이러한 요구 사항들을 해결하기 위해 새로운 함수를 제공한다. 이 함수들은 분석 작업에 유용하기 때문에 Analytic Functions이라고 하며 DB 버전이 올라갈수록 계속 추가되고 있다.

```
SQL> SELECT empno, ename, sal, deptno, SUM(sal) OVER(PARTITION BY deptno) AS dept_tot
       FROM emp;
```

EMPNO	ENAME	SAL	DEPTNO	DEPT_TOT
7934	MILLER	1300	10	8750
7782	CLARK	2450	10	8750
7839	KING	5000	10	8750
7902	FORD	3000	20	10875
7788	SCOTT	3000	20	10875
7876	ADAMS	1100	20	10875
7566	JONES	2975	20	10875
7369	SMITH	800	20	10875
7900	JAMES	950	30	9400
7844	TURNER	1500	30	9400
7698	BLAKE	2850	30	9400
7521	WARD	1250	30	9400
7499	ALLEN	1600	30	9400
7654	MARTIN	1250	30	9400

14 rows selected.

분석 함수는 Aggregate Function 뒤에 Analytic Clause(OVER 절)을 통해 행 그룹의 정의를 지정하고 각 그룹당 결과 값을 반복하여 출력한다. 여기서 행 그룹의 범위를 WINDOW라 부르며 하나의 WINDOW가 계산을 수행하는데 사용되는 행들의 집합을 결정하게 되며 PARTITION BY, ORDER BY, WINDOWING을 통하여 조절하게 된다.

분석 함수는 Join 문장, WHERE, GROUP BY, HAVING 등과 함께 쓰일 때 가장 마지막에 연산(집계)을 진행하며 SELECT 절과 ORDER BY 절에서만 사용이 가능하다.

PARTITION BY 절은 GROUP BY 절과 동일한 작업 수행한다. 단, GROUP BY 절을 사용하지 않고 필요한 집합으로 (WINDOW) 행들을 그룹화 시킬 수 있다.

```
SQL> SELECT empno, ename, sal, AVG(sal) OVER()                   AS avg_overall,
                               AVG(sal) OVER(PARTITION BY deptno) AS avg_deptno
       FROM emp ;
     EMPNO ENAME             SAL AVG_OVERALL AVG_DEPTNO
---------- ---------- ---------- ----------- ----------
      7934 MILLER           1300  2073.21429 2916.66667
      7782 CLARK            2450  2073.21429 2916.66667
      7839 KING             5000  2073.21429 2916.66667
      7902 FORD             3000  2073.21429       2175
      7788 SCOTT            3000  2073.21429       2175
      7876 ADAMS            1100  2073.21429       2175
      7566 JONES            2975  2073.21429       2175
      7369 SMITH             800  2073.21429       2175
      7900 JAMES             950  2073.21429 1566.66667
      7844 TURNER           1500  2073.21429 1566.66667
      7698 BLAKE            2850  2073.21429 1566.66667
      7521 WARD             1250  2073.21429 1566.66667
      7499 ALLEN            1600  2073.21429 1566.66667
      7654 MARTIN           1250  2073.21429 1566.66667

14 rows selected.
```

ORDER BY 절은 PARTITION BY로 정의된 WINDOW 내에서의 행들의 정렬 순서를 정의한다.

```
SQL> SELECT empno, ename, sal, deptno,
            row_number( ) over ( ORDER BY sal ASC ) AS rnum
       FROM emp ;
     EMPNO ENAME             SAL     DEPTNO       RNUM
---------- ---------- ---------- ---------- ----------
      7369 SMITH             800         20          1
      7900 JAMES             950         30          2
      7876 ADAMS            1100         20          3
      7654 MARTIN           1250         30          4
      7521 WARD             1250         30          5
      7934 MILLER           1300         10          6
      7844 TURNER           1500         30          7
      7499 ALLEN            1600         30          8
      7782 CLARK            2450         10          9
      7698 BLAKE            2850         30         10
      7566 JONES            2975         20         11
      7788 SCOTT            3000         20         12
      7902 FORD             3000         20         13
      7839 KING             5000         10         14

14 rows selected.
```

```
SQL> SELECT empno, ename, sal, deptno,
            row_number( ) over ( PARTITION BY deptno ORDER BY sal DESC ) AS rnum
       FROM emp ;
     EMPNO ENAME             SAL     DEPTNO       RNUM
---------- ---------- ---------- ---------- ----------
      7839 KING             5000         10          1
      7782 CLARK            2450         10          2
      7934 MILLER           1300         10          3
      7902 FORD             3000         20          1
      7788 SCOTT            3000         20          2
      7566 JONES            2975         20          3
      7876 ADAMS            1100         20          4
      7369 SMITH             800         20          5
      7698 BLAKE            2850         30          1
      7499 ALLEN            1600         30          2
      7844 TURNER           1500         30          3
      7521 WARD             1250         30          4
      7654 MARTIN           1250         30          5
      7900 JAMES             950         30          6

14 rows selected.

SQL> SELECT empno, ename, sal, comm,
             DENSE_RANK( ) over ( ORDER BY comm ASC )             AS no1,
             DENSE_RANK( ) over ( ORDER BY comm ASC NULLS FIRST ) AS no2
       FROM emp
      WHERE deptno = 30 ;
     EMPNO ENAME             SAL       COMM        NO1        NO2
---------- ---------- ---------- ---------- ---------- ----------
      7698 BLAKE            2850                     5          1
      7900 JAMES             950                     5          1
      7844 TURNER           1500          0          1          2
      7499 ALLEN            1600        300          2          3
      7521 WARD             1250        500          3          4
      7654 MARTIN           1250       1400          4          5

6 rows selected.
```

ORDER BY 절은 PARTITION BY에 의해 그룹화된 행들의 정렬 순서를 결정하며 NULL 값을 가지고 있는 행이 있을 경우 NULL에 대한 값을 FIRST, LAST로 보낼 수 있도록 조절 가능하다.

WINDOWING 절은 일부 Aggregate Function과 함께 쓰일 수 있으며 행들의 그룹을 물리적, 논리적으로 조절하여 Function이 적용될 WINDOW를 정의한다. 즉, PARTITON BY 절은 컬럼에 같은 값을 기준으로만 그룹화하지만 WINDOWING 절은 ROWS와 RANGE를 이용하여 하나의 WINDOW를 결정하는 범위를 보다 자유롭게 조정할 수 있다.

```
SQL> SELECT empno, ename, sal,
            SUM(sal) over ( ORDER BY empno
                            ROWS BETWEEN 1 PRECEDING
                                     AND 1 FOLLOWING ) AS physical,
            SUM(sal) over ( ORDER BY empno
                            RANGE BETWEEN 100 PRECEDING
                                      AND 100 FOLLOWING ) AS logical
       FROM emp
      WHERE deptno IN (10,20)
      ORDER BY empno ;
     EMPNO ENAME             SAL   PHYSICAL    LOGICAL
---------- ---------- ---------- ---------- ----------
      7369 SMITH             800       3775        800
      7566 JONES            2975       6225       2975
      7782 CLARK            2450       8425      11550      : EMPNO BETWEEN 7682 AND 7882
      7788 SCOTT            3000      10450      11550
      7839 KING             5000       9100      15850
      7876 ADAMS            1100       9100      15850
      7902 FORD             3000       5400      10400
      7934 MILLER           1300       4300      10400

8 rows selected.
```

ROWS는 WINDOW의 범위를 정의할 때 물리적인 행을 지정한다. 어떤 행에서 시작해서 어떤 행 까지가 하나의 WINDOW 영역으로 정의 할지 범위를 BETWEEN을 통하여 정의 할 수 있다. 그리고 추가적으로 UNBOUNDED PRECEDING는 첫 번째 행, UNBOUNDED FOLLOWING은 마지막 행, CURRENT ROW는 현재 행 참조하게 할 수 있다.

RANGE는 논리적인 값을 근거로 WINDOW 범위를 설정할 수 있다. 모든 함수가 WINDOWING절을 사용할 수 있는 것은 아니다. (매뉴얼 참조)

1. EMP 테이블에서 사원 정보와 소속 부서의 평균 급여를 다음과 같이 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL     DEPTNO    AVG_SAL
---------- ---------- ---------- ---------- ----------
      7782 CLARK            2450         10    2916.67
      7934 MILLER           1300         10    2916.67
      7839 KING             5000         10    2916.67
      7788 SCOTT            3000         20       2175
      7566 JONES            2975         20       2175
      7369 SMITH             800         20       2175
      7876 ADAMS            1100         20       2175
      7902 FORD             3000         20       2175
      7499 ALLEN            1600         30    1566.67
      7521 WARD             1250         30    1566.67
      7654 MARTIN           1250         30    1566.67
      7698 BLAKE            2850         30    1566.67
      7844 TURNER           1500         30    1566.67
      7900 JAMES             950         30    1566.67

14 rows selected.
```

답안 1. Inline View 활용

```
SQL> SELECT e.empno, e.ename, e.sal, e.deptno, a.avg_sal
       FROM emp e, ( SELECT deptno, ROUND(AVG(sal),2) AS AVG_SAL
                       FROM emp
                     GROUP BY deptno ) a
      WHERE e.deptno = a.deptno
     ORDER BY e.deptno ;
```

답안 2. Correlated Subquery 이용

```
SQL> SELECT empno, ename, sal, deptno, (SELECT ROUND(AVG(sal),2)
                                          FROM emp
                                         WHERE deptno = e.deptno) AS AVG_SAL
       FROM emp e
     ORDER BY deptno ;
```

두 가지의 Subquery의 사용 방법을 통해 문제를 해결할 수 있다. 경우에 따라서는 위와 같은 문장의 사용이 필요할 수도 있으나 동일 테이블의 반복적인 접근이 오히려 성능을 저하시키는 경우도 있다.

답안 3. 분석 함수 사용

```
SQL> SELECT empno, ename, sal, deptno,
            ROUND(AVG(sal) OVER(PARTITION BY deptno),2) AS AVG_SAL
       FROM emp
     ORDER BY deptno ;
```

분석 함수의 사용이 항상 정답은 아니다. 경우에 따라 일반적인 Subquery를 이용하는 것이 필요할 수도 있다. 단, 다양한 문장을 통해 동일한 결과를 검색하도록 문장을 작성할 수 있다면 경우에 따라 필요한 문장의 선택을 보다 손쉽게 할 수 있다. 성능과 관련된 자세한 사항은 SQL Tuning 과정을 참고한다.

2. EMP 테이블에서 1981년에 입사한 사원 정보와 소속 부서의 평균 급여를 분석 함수를 이용하여 검색하시오.

검색 결과

```
     EMPNO ENAME             SAL HIREDATE     DEPTNO    AVG_SAL
---------- ---------- ---------- -------- ---------- ----------
      7782 CLARK            2450 81/06/09         10    2916.67
      7839 KING             5000 81/11/17         10    2916.67
      7566 JONES            2975 81/04/02         20       2175
      7902 FORD             3000 81/12/03         20       2175
      7499 ALLEN            1600 81/02/20         30    1566.67
      7521 WARD             1250 81/02/22         30    1566.67
      7654 MARTIN           1250 81/09/28         30    1566.67
      7698 BLAKE            2850 81/05/01         30    1566.67
      7844 TURNER           1500 81/09/08         30    1566.67
      7900 JAMES             950 81/12/03         30    1566.67

10 rows selected.
```

답안 1. Correlated Subquery 이용

```
SQL> SELECT empno, ename, sal, hiredate, deptno, (SELECT ROUND(AVG(sal),2)
                                                    FROM emp
                                                   WHERE deptno = e.deptno) AS AVG_SAL
      FROM emp e
     WHERE hiredate BETWEEN TO_DATE('1981/01/01','YYYY/MM/DD')
                        AND TO_DATE('1981/12/31','YYYY/MM/DD')
     ORDER BY deptno ;
```

답안 2. Inline View 이용

```
SQL> SELECT e.empno, e.ename, e.sal, e.hiredate, e.deptno, a.avg_sal
       FROM emp e, ( SELECT deptno, ROUND(AVG(sal),2) AS AVG_SAL
                       FROM emp
                      GROUP BY deptno ) a
     WHERE e.deptno = a.deptno
       AND hiredate BETWEEN TO_DATE('1981/01/01','YYYY/MM/DD')
                        AND TO_DATE('1981/12/31','YYYY/MM/DD')
     ORDER BY e.deptno ;
```

답안 3. 분석 함수 이용

```
SQL> SELECT empno, ename, sal, hiredate, deptno,
            ROUND(AVG(sal) OVER(PARTITION BY deptno),2) AS AVG_SAL
       FROM emp
      WHERE hiredate BETWEEN TO_DATE('1981/01/01','YYYY/MM/DD')
                         AND TO_DATE('1981/12/31','YYYY/MM/DD') ;
     EMPNO ENAME             SAL HIREDATE     DEPTNO    AVG_SAL
---------- ---------- ---------- -------- ---------- ----------
      7782 CLARK            2450 81/06/09         10       3725
      7839 KING             5000 81/11/17         10       3725
      7566 JONES            2975 81/04/02         20     2987.5
      7902 FORD             3000 81/12/03         20     2987.5
      7698 BLAKE            2850 81/05/01         30    1566.67
      7844 TURNER           1500 81/09/08         30    1566.67
      7654 MARTIN           1250 81/09/28         30    1566.67
      7900 JAMES             950 81/12/03         30    1566.67
      7521 WARD             1250 81/02/22         30    1566.67
      7499 ALLEN            1600 81/02/20         30    1566.67

10 rows selected.
```

분석 함수는 WHERE절의 조건식 비교 후 결과 값을 생성한다. 때문에 위와 같은 문장은 조건절이 먼저 비교되면서 부서별 평균 급여는 잘못된 값을 보여줄 수 있다.

```
SQL> SELECT empno, ename, sal, hiredate, deptno, avg_sal
       FROM (SELECT empno, ename, sal, hiredate, deptno,
                    ROUND(AVG(sal) OVER(PARTITION BY deptno),2) AS AVG_SAL
               FROM emp)
      WHERE hiredate BETWEEN TO_DATE('1981/01/01','YYYY/MM/DD')
                         AND TO_DATE('1981/12/31','YYYY/MM/DD') ;
```

3. EMP 테이블에서 사원들의 정보를 EMPNO 컬럼으로 정렬하고, 각 사원의 급여를 행 별로 누적하여 TOTAL 컬럼을 검색하시오.

검색 결과

EMPNO	ENAME	SAL	TOTAL
7369	SMITH	800	800
7499	ALLEN	1600	2400
7521	WARD	1250	3650
7566	JONES	2975	6625
7654	MARTIN	1250	7875
7698	BLAKE	2850	10725
7782	CLARK	2450	13175
7788	SCOTT	3000	16175
7839	KING	5000	21175
7844	TURNER	1500	22675
7876	ADAMS	1100	23775
7900	JAMES	950	24725
7902	FORD	3000	27725
7934	MILLER	1300	29025

14 rows selected.

답안 1. Correlated Subquery 이용

```
SQL> SELECT a.empno, a.ename, a.sal, (SELECT SUM(sal)
                                      FROM emp
                                     WHERE empno <= a.empno) AS TOTAL
      FROM emp a
     ORDER BY a.empno ;
```

답안 2. Self Join 이용

```
SQL> SELECT a.empno, a.ename, a.sal , SUM(b.sal) AS TOTAL
     FROM emp a , emp b
     WHERE a.empno >= b.empno
     GROUP BY a.empno, a.ename, a.sal
     ORDER BY a.empno ;
```

답안 3. 분석 함수 이용

```
SQL> SELECT empno, ename, sal,
            SUM(sal) OVER(ORDER BY empno ROWS BETWEEN UNBOUNDED PRECEDING
                                                  AND CURRENT ROW) AS TOTAL
       FROM emp ;
```

JOIN, Subquery 등을 이용하여도 결과는 생성 가능하지만 동일 집합의 반복 접근은 I/O를 증가시키며 성능을 악화 시킬 수 있다. 필요에 따라 Join, Subquery를 이용해야 할 수도 있지만 분석 함수의 사용을 고려한다.

추가 실습

```
SQL> SELECT empno, ename, sal, SUM(sal) OVER (ORDER BY empno) AS TOTAL
       FROM emp ;
     EMPNO ENAME             SAL      TOTAL
---------- ---------- ---------- ----------
      7369 SMITH             800        800
      7499 ALLEN            1600       2400
      7521 WARD             1250       3650
      7566 JONES            2975       6625
      7654 MARTIN           1250       7875
      7698 BLAKE            2850      10725
      7782 CLARK            2450      13175
...

14 rows selected.

SQL> SELECT empno, ename, sal,
            SUM(sal) OVER(ORDER BY empno ROWS BETWEEN CURRENT ROW
                                                  AND UNBOUNDED FOLLOWING) AS TOTAL
       FROM emp ;
     EMPNO ENAME             SAL      TOTAL
---------- ---------- ---------- ----------
      7369 SMITH             800      29025
      7499 ALLEN            1600      28225
      7521 WARD             1250      26625
      7566 JONES            2975      25375
      7654 MARTIN           1250      22400
      7698 BLAKE            2850      21150
      7782 CLARK            2450      18300
...

14 rows selected.
```

```
SQL> SELECT empno, ename, sal,
            SUM(sal) OVER(ORDER BY empno ROWS BETWEEN 1 PRECEDING
                                                 AND 1 FOLLOWING) AS TOTAL
       FROM emp ;
     EMPNO ENAME             SAL      TOTAL
---------- ---------- ---------- ----------
      7369 SMITH             800       2400
      7499 ALLEN            1600       3650
      7521 WARD             1250       5825
      7566 JONES            2975       5475
      7654 MARTIN           1250       7075
      7698 BLAKE            2850       6550
      7782 CLARK            2450       8300
      7788 SCOTT            3000      10450
      7839 KING             5000       9500
      7844 TURNER           1500       7600
      7876 ADAMS            1100       3550
      7900 JAMES             950       5050
      7902 FORD             3000       5250
      7934 MILLER           1300       4300

14 rows selected.
```

4. EMP 테이블에서, 각 사원의 급여가 최대 급여, 최소 급여와의 차이가 얼마인지 검색하시오.

검색 결과

EMPNO	ENAME	SAL	MAXSAL	MINSAL	MAXSAL-SAL	SAL-MINSAL
7788	SCOTT	3000	5000	800	2000	2200
7566	JONES	2975	5000	800	2025	2175
7369	SMITH	800	5000	800	4200	0
7876	ADAMS	1100	5000	800	3900	300
7499	ALLEN	1600	5000	800	3400	800
7521	WARD	1250	5000	800	3750	450
7654	MARTIN	1250	5000	800	3750	450
7698	BLAKE	2850	5000	800	2150	2050
7782	CLARK	2450	5000	800	2550	1650
7844	TURNER	1500	5000	800	3500	700
7900	JAMES	950	5000	800	4050	150
7902	FORD	3000	5000	800	2000	2200
7934	MILLER	1300	5000	800	3700	500
7839	KING	5000	5000	800	0	4200

14 rows selected.

답안 1. Inline view 사용

```
SQL> SELECT e1.empno, e1.ename, e1.sal, e2.maxsal, e2.minsal,
            e2.maxsal-e1.sal, e1.sal-e2.minsal
     FROM emp e1, (SELECT MAX(sal) AS MAXSAL, MIN(sal) AS MINSAL
                   FROM emp) e2 ;
```

답안 3. 분석 함수 이용

```
SQL> SELECT empno, ename, sal, maxsal, minsal,
            maxsal - sal , sal - minsal
      FROM (SELECT empno, ename, sal, MAX(sal) OVER() AS MAXSAL, MIN(sal) OVER() AS MINSAL
            FROM emp) ;
```

5. EMP 테이블에서 HIREDATE, EMPNO 컬럼으로 정렬된 사원 정보를 검색하시오.
이때 해당 사원보다 위, 아래(앞, 뒤)의 입사 일자를 함께 출력합니다.

검색 결과

```
     EMPNO ENAME      HIREDATE        PREV_HIRE       NEXT_HIRE
---------- ---------- --------------- --------------- ---------------
      7369 SMITH      80/12/17                        81/02/20
      7499 ALLEN      81/02/20        80/12/17        81/02/22
      7521 WARD       81/02/22        81/02/20        81/04/02
      7566 JONES      81/04/02        81/02/22        81/05/01
      7698 BLAKE      81/05/01        81/04/02        81/06/09
      7782 CLARK      81/06/09        81/05/01        81/09/08
      7844 TURNER     81/09/08        81/06/09        81/09/28
      7654 MARTIN     81/09/28        81/09/08        81/11/17
      7839 KING       81/11/17        81/09/28        81/12/03
      7900 JAMES      81/12/03        81/11/17        81/12/03
      7902 FORD       81/12/03        81/12/03        82/01/23
      7934 MILLER     82/01/23        81/12/03        87/04/19
      7788 SCOTT      87/04/19        82/01/23        87/05/23
      7876 ADAMS      87/05/23        87/04/19

14 rows selected.
```

답안 1. ROWNUM을 이용한 Outer Join 사용

```
SQL> SELECT a.empno, a.ename, a.hiredate,b.hiredate AS PREV_HIRE,c.hiredate AS NEXT_HIRE
       FROM (SELECT ROWNUM no1, empno, ename, hiredate
               FROM (SELECT empno, ename, hiredate
                       FROM emp
                      ORDER BY hiredate,empno)) a,
            (SELECT ROWNUM+1 no2, empno, ename, hiredate
               FROM (SELECT empno, ename, hiredate
                       FROM emp
                      ORDER BY hiredate,empno)) b,
            (SELECT ROWNUM-1 no3, empno, ename, hiredate
               FROM (SELECT empno, ename, hiredate
                       FROM emp
                      ORDER BY hiredate,empno)) c
      WHERE a.no1 = b.no2 (+)
        AND a.no1 = c.no3 (+)
     ORDER BY a.hiredate, a.empno ;
```

답안 2. 분석 함수 이용

```
SQL> SELECT empno, ename, hiredate,
            LAG(hiredate)  OVER (ORDER BY hiredate,empno) AS PREV_HIRE,
            LEAD(hiredate) OVER (ORDER BY hiredate,empno) AS NEXT_HIRE
       FROM emp ;
```

함수 소개: LAG / LEAD

지정된 개수의 이전, 이후 행의 값 가져오기. WINDOWING 절을 지정하지 못하며 NULL 값을 대체하는 값을 지정할 수 있음. (NVL 불필요)

Q. 30번 부서의 사원을 이름순으로 정렬하여 검색하며 이전, 다음 행의 급여를 함께 표시

```
SQL> SELECT empno, ename, sal,
            LAG (sal,1,0) over ( order by ename ) prev_sal,
            LEAD (sal,1,0) over ( order by ename ) next_sal
      FROM emp
     WHERE deptno = 30 ;
     EMPNO ENAME             SAL   PREV_SAL   NEXT_SAL
---------- ---------- ---------- ---------- ----------
      7499 ALLEN            1600          0       2850
      7698 BLAKE            2850       1600        950
      7900 JAMES             950       2850       1250
      7654 MARTIN           1250        950       1500
      7844 TURNER           1500       1250       1250
      7521 WARD             1250       1500          0

6 rows selected.
```

6. EMP 테이블에서 다음과 같이 부서별 사원 이름을 검색하시오.

검색 결과

```
    DEPTNO EMPLOYEES
---------- ------------------------------------------------
        10 CLARK,KING,MILLER
        20 ADAMS,FORD,JONES,SCOTT,SMITH
        30 ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
```

답안 1. XMLAGG 사용 (9i)

```
SQL> SELECT deptno,
            SUBSTR(XMLAGG(XMLELEMENT(x, ',', ename) ORDER BY ename).EXTRACT('//text()'),2)
            AS employee
       FROM emp
       GROUP BY deptno ;
```

답안 2. LISTAGG 사용 (v11gR2)

```
SQL> SELECT deptno,
            LISTAGG(ename,',') WITHIN GROUP (ORDER BY ename) AS employee
       FROM emp
      GROUP BY deptno ;
```

추가 실습

```
SQL> SELECT d.dname, LISTAGG(e.ename||'('||e.sal||')',',')
                     WITHIN GROUP (ORDER BY ename) AS employee
       FROM emp e, dept d
      WHERE e.deptno = d.deptno
     GROUP BY d.dname ;
DNAME          EMPLOYEE
-------------- --------------------------------------------------------------------------
ACCOUNTING     CLARK(2450),KING(5000),MILLER(1300)
RESEARCH       ADAMS(1100),FORD(3000),JONES(2975),SCOTT(3000),SMITH(800)
SALES          ALLEN(1600),BLAKE(2850),JAMES(950),MARTIN(1250),TURNER(1500),WARD(1250)

3 rows selected.
```

7. EMP 테이블에서 DEPTNO, SAL, ENAME 컬럼을 기준으로 정렬된 정보를 검색하면서 부서별 누적된 급여, 부서별 급여의 백분율, 전체 사원의 급여 합계에서의 백분율을 검색한다.

검색 결과

DEPTNO	ENAME	SAL	CUM_SAL	PCT_DEPT	PCT_OVERALL
10	MILLER	1300	1300	14.9	4.5
10	CLARK	2450	3750	28	8.4
10	KING	5000	8750	57.1	17.2
20	SMITH	800	800	7.4	2.8
20	ADAMS	1100	1900	10.1	3.8
20	JONES	2975	4875	27.4	10.2
20	FORD	3000	7875	27.6	10.3
20	SCOTT	3000	10875	27.6	10.3
30	JAMES	950	950	10.1	3.3
30	MARTIN	1250	2200	13.3	4.3
30	WARD	1250	3450	13.3	4.3
30	TURNER	1500	4950	16	5.2
30	ALLEN	1600	6550	17	5.5
30	BLAKE	2850	9400	30.3	9.8

14 rows selected.

답안. 분석 함수 이용

```
SQL> SELECT deptno, ename, sal,
            SUM(sal) OVER(PARTITION BY deptno ORDER BY sal, ename)       AS cum_sal,
            ROUND(100*RATIO_TO_REPORT(sal) OVER(PARTITION BY deptno),1)  AS pct_dept,
            ROUND(100*RATIO_TO_REPORT(sal) OVER(),1)                     AS pct_overall
     FROM emp ;
```

함수 소개 : RATIO_TO_REPORT

: WINDOW 영역의 합계 내에서 현재 값이 차지하는 백분율. 별도의 WINDOWING 절을 설정하는 것은 불가능하다.

Q. 사원 정보를 출력하면서 부서별 급여의 합계 중 해당 사원이 받는 급여의 백분율을 표시하고 부서별 급여의 합계 출력

```
SQL> break on deptno skip 1
SQL> compute sum label 'TOTAL' of sal on deptno
SQL> SELECT deptno, ename, sal,
            ROUND ( RATIO_TO_REPORT (sal) over ( partition BY deptno ) , 2) AS ratio
       FROM emp ;
    DEPTNO ENAME             SAL      RATIO
---------- ---------- ---------- ----------
        10 CLARK            2450        .28
           KING             5000        .57
           MILLER           1300        .15
********** ----------
TOTAL                       8750

        20 JONES            2975        .27
           FORD             3000        .28
           ADAMS            1100         .1
           SMITH             800        .07
           SCOTT            3000        .28
********** ----------
TOTAL                      10875

        30 WARD             1250        .13
           TURNER           1500        .16
           ALLEN            1600        .17
           JAMES             950         .1
           BLAKE            2850         .3
           MARTIN           1250        .13
********** ----------
TOTAL                       9400

SQL> clear compute break
```

추가 함수 소개

Hypothetical Functions
: 가정에 근거하여 각 함수에 맞는 값을 확인

```
SQL> SELECT ename, deptno, sal
           ,RANK()       OVER(PARTITION BY deptno ORDER BY sal DESC) AS RK
           ,DENSE_RANK() OVER(PARTITION BY deptno ORDER BY sal DESC) AS DENSE_RK
      FROM emp ;
ENAME          DEPTNO        SAL         RK   DENSE_RK
---------- ---------- ---------- ---------- ----------
KING               10       5000          1          1
CLARK              10       2450          2          2
MILLER             10       1300          3          3
FORD               20       3000          1          1
SCOTT              20       3000          1          1
JONES              20       2975          3          2
ADAMS              20       1100          4          3
SMITH              20        800          5          4
BLAKE              30       2850          1          1
ALLEN              30       1600          2          2
TURNER             30       1500          3          3
WARD               30       1250          4          4
MARTIN             30       1250          4          4
JAMES              30        950          6          5

14 rows selected.
```

Q. 부서별 2500 급여를 갖는 사원이 있다면 순위는?

```
SQL> SELECT deptno,
           RANK(2500)       WITHIN GROUP ( ORDER BY sal DESC)       AS RANK,
           DENSE_RANK(2500) WITHIN GROUP ( ORDER BY sal DESC) AS DENSE_RANK
       FROM emp
     GROUP BY deptno;
    DEPTNO       RANK DENSE_RANK
---------- ---------- ----------
        10          2          2
        20          4          3
        30          2          2

3 rows selected.
```

NTILE

: WINDOW 그룹의 행을 정렬 후 지정한 개수의 범위(등급)으로 나눈 후 각 값이 가지고 있는 등급 값을 보여준다.

Q. EMP 테이블에서 급여를 많이 받는 순서대로 정렬하였을 때 5개의 범위로 나누어 각 등급을 다음과 같이 출력하시오.

```
SQL> SELECT empno, ename, sal, NTILE (5) OVER (ORDER BY sal DESC) grade
       FROM emp ;
     EMPNO ENAME             SAL      GRADE
---------- ---------- ---------- ----------
      7839 KING             5000          1
      7902 FORD             3000          1
      7788 SCOTT            3000          1
      7566 JONES            2975          2
      7698 BLAKE            2850          2
      7782 CLARK            2450          2
      7499 ALLEN            1600          3
      7844 TURNER           1500          3
      7934 MILLER           1300          3
      7521 WARD             1250          4
      7654 MARTIN           1250          4
      7876 ADAMS            1100          4
      7900 JAMES             950          5
      7369 SMITH             800          5

14 rows selected.
```

CUME_DIST (cumulative distribution) / PERCENT_RANK

: 둘 모두 계산식에 차이가 있으나 WINDOW 그룹 내에서 누적 분포를 계산할 때 사용
값의 범위는 0 ~ 1까지 사용되며 PERCENT_RANK는 항상 시작 값이 0부터 시작

Q. 급여를 내림차순 기준으로 정렬하였을 경우 각 사원의 누적 분포를 계산

```
SQL> SELECT ename, sal,
            ROUND ( PERCENT_RANK ( ) over ( order by sal DESC ),2)     AS per_rank,
            ROUND ( CUME_DIST( )     over ( order by sal DESC ),2)     AS cume_dist,
                    RANK( )          over ( order by sal DESC )        AS rank,
                    ROW_NUMBER ( )   over ( order by sal DESC )        AS row_num
     FROM emp ;
```

ENAME	SAL	PER_RANK	CUME_DIST	RANK	ROW_NUM
KING	5000	0	.07	1	1
FORD	3000	.08	.21	2	2
SCOTT	3000	.08	.21	2	3
JONES	2975	.23	.29	4	4
BLAKE	2850	.31	.36	5	5
CLARK	2450	.38	.43	6	6
ALLEN	1600	.46	.5	7	7
TURNER	1500	.54	.57	8	8
MILLER	1300	.62	.64	9	9
WARD	1250	.69	.79	10	10
MARTIN	1250	.69	.79	10	11
ADAMS	1100	.85	.86	12	12
JAMES	950	.92	.93	13	13
SMITH	800	1	1	14	14

14 rows selected.

PERCENT_RANK : (RANK - 1) / (COUNT(*) - 1)
CUME_DIST : (RANK or ROW_NUMBER) / COUNT(*)

동등 순위의 RANK 발생 시 해당 RANK의 마지막 ROW_NUMBER 사용

PERCENTILE_CONT / PERCENTILE_DISC

: cume_dist, percent_rank의 결과가 누적 분포도를 계산 한다면 PERCENTILE_CONT, PERCENTILE_DISC는 분포도 값(지정되는 백분율)을 역으로 계산하여 실제의 값을 가져온다.

```
SQL> SELECT empno, ename, sal, deptno,
            ROUND(cume_dist ( ) over ( order by sal DESC ),2) AS cume_dist
       FROM emp
      WHERE deptno = 30 ;
     EMPNO ENAME             SAL     DEPTNO  CUME_DIST
---------- ---------- ---------- ---------- ----------
      7698 BLAKE            2850         30        .17
      7499 ALLEN            1600         30        .33
      7844 TURNER           1500         30         .5
      7521 WARD             1250         30        .83
      7654 MARTIN           1250         30        .83
      7900 JAMES             950         30          1

6 rows selected.
```

30번 부서 번호의 사원 중 급여를 기준으로 내림차순 정렬 했을 때 1500의 급여가 전체 WINDOW 중 50% 범위에 해당하는 것을 확인할 수 있다.

Q. 30번 부서 사원 중 급여를 기준으로 내림차순 정렬을 하였을 때 50% 범위에 해당하는 급여는 얼마인가?

```
SQL> SELECT PERCENTILE_CONT(0.5) within GROUP ( ORDER BY sal DESC ) AS CONT ,
            PERCENTILE_DISC(0.5) within GROUP ( ORDER BY sal DESC ) AS DISC
      FROM emp
     WHERE deptno = 30 ;
      CONT       DISC
---------- ----------
      1375       1500
```

PERCENTILE_CONT는 선형 보간법을 이용하여 평균에 근거하는 결과를 보여주므로 실제의 값을 보여주는 PERCENTILE_DISC보다는 정확한 값을 보여주지는 않을 수 있다. 보다 자세한 계산 공식은 매뉴얼 참고 가능 (MEDIAN 함수도 같은 결과 확인 가능하다.)

```
SQL> SELECT MEDIAN(sal)
       FROM emp
      WHERE deptno = 30 ;
MEDIAN(SAL)
-----------
       1375
```

9. 계층 질의 활용

1. EMP 테이블에서 EMPNO와 MGR 컬럼을 이용하여, KING 아래에 근무하는 사원들을 계층적으로 검색하시오.

검색 결과

```
NAME                LEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
KING                    1       7839
  JONES                 2       7566       7839
    SCOTT               3       7788       7566
      ADAMS             4       7876       7788
    FORD                3       7902       7566
      SMITH             4       7369       7902
  BLAKE                 2       7698       7839
    ALLEN               3       7499       7698
    WARD                3       7521       7698
    MARTIN              3       7654       7698
    TURNER              3       7844       7698
    JAMES               3       7900       7698
  CLARK                 2       7782       7839
    MILLER              3       7934       7782

14 rows selected.
```

답안. 계층 질의 사용

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME, LEVEL, empno, mgr
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
```

Oracle Database에서 사용 가능한 구문이며, START WITH 절을 이용하여 계층 질의의 시작 행을 식별하고 CONNECT BY 절을 이용하여 트리의 조건식을 정의한다. 이러한 계층 질의 구문은 ANSI-SQL의 포맷은 아니다.

2. EMP 테이블에서 계층 질의를 검색하면서, SCOTT 사원을 포함한 하위 직원은 제외하시오.

검색 결과

```
NAME                LEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
KING                    1       7839
  JONES                 2       7566       7839
    FORD                3       7902       7566
      SMITH             4       7369       7902
  BLAKE                 2       7698       7839
    ALLEN               3       7499       7698
    WARD                3       7521       7698
    MARTIN              3       7654       7698
    TURNER              3       7844       7698
    JAMES               3       7900       7698
  CLARK                 2       7782       7839
    MILLER              3       7934       7782

12 rows selected.
```

주의 사항. WHERE절의 부정형 비교

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME, LEVEL, empno, mgr
       FROM emp
      WHERE ename != 'SCOTT'
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
NAME                LEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
KING                    1       7839
  JONES                 2       7566       7839
      ADAMS             4       7876       7788
    FORD                3       7902       7566
      SMITH             4       7369       7902
  BLAKE                 2       7698       7839
    ALLEN               3       7499       7698
    WARD                3       7521       7698
    MARTIN              3       7654       7698
    TURNER              3       7844       7698
    JAMES               3       7900       7698
  CLARK                 2       7782       7839
    MILLER              3       7934       7782

13 rows selected.
```

답안. CONNECT BY 절의 부정형 비교

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME, LEVEL, empno, mgr
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr
         AND ename != 'SCOTT' ;
```

WHERE절은 트리 구조를 생성한 이후에 조건 비교를 진행한다. 때문에 WHERE절의 부정형 비교는 전체 가지(Branch) 제거를 수행할 수 없다. 하위 계층에 대한 제거도 함께 해야 한다면 CONNECT BY절에서의 조건 비교를 사용한다.

3. EMP 테이블에서, 계층 질의를 검색하면서 현재 레벨과 상위 레벨의 이름(ENAME)을 함께 검색하시오. Worker는 현재 노드의 ENAME 컬럼이며, Manager는 상위 노드의 ENAME 컬럼을 출력한다.

검색 결과

```
Tree                 LEVEL Worker     Manager
--------------- ---------- ---------- ----------
KING                     1 KING
  JONES                  2 JONES      KING
    SCOTT                3 SCOTT      JONES
      ADAMS              4 ADAMS      SCOTT
    FORD                 3 FORD       JONES
      SMITH              4 SMITH      FORD
  BLAKE                  2 BLAKE      KING
    ALLEN                3 ALLEN      BLAKE
    WARD                 3 WARD       BLAKE
    MARTIN               3 MARTIN     BLAKE
    TURNER               3 TURNER     BLAKE
    JAMES                3 JAMES      BLAKE
  CLARK                  2 CLARK      KING
    MILLER               3 MILLER     CLARK

14 rows selected.
```

답안. PRIOR 키워드 사용

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS "Tree",
            level,
            ename AS "Worker",
            prior ename AS "Manager"
      FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
```

상위 노드의 컬럼이 필요할 때 PRIOR 키워드를 이용하여 참조가 가능하다.

4. EMP 테이블에서 계층 질의를 검색하면서, ENAME 컬럼을 기준으로 정렬된 결과를 검색하시오. 단, 계층 구조를 훼손하지 않도록 한다.

검색 결과

```
NAME                LEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
KING                    1       7839
  BLAKE                 2       7698       7839
    ALLEN               3       7499       7698
    JAMES               3       7900       7698
    MARTIN              3       7654       7698
    TURNER              3       7844       7698
    WARD                3       7521       7698
  CLARK                 2       7782       7839
    MILLER              3       7934       7782
  JONES                 2       7566       7839
    FORD                3       7902       7566
      SMITH             4       7369       7902
    SCOTT               3       7788       7566
      ADAMS             4       7876       7788

14 rows selected.
```

주의 사항

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME, LEVEL, empno, mgr
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr
     ORDER BY ename ;
NAME                LEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
      ADAMS             4       7876       7788
    ALLEN               3       7499       7698
  BLAKE                 2       7698       7839
  CLARK                 2       7782       7839
    FORD                3       7902       7566
    JAMES               3       7900       7698
  JONES                 2       7566       7839
KING                    1       7839
    MARTIN              3       7654       7698
    MILLER              3       7934       7782
    SCOTT               3       7788       7566
      SMITH             4       7369       7902
    TURNER              3       7844       7698
    WARD                3       7521       7698

14 rows selected.
```

답안. SIBLINGS 키워드 사용

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME, LEVEL, empno, mgr
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr
     ORDER SIBLINGS BY ename ;
```

추가 실습

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME, LEVEL, empno, mgr, deptno
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr
     ORDER SIBLINGS BY deptno ;
```

NAME	LEVEL	EMPNO	MGR	DEPTNO
KING	1	7839		10
CLARK	2	7782	7839	10
MILLER	3	7934	7782	10
JONES	2	7566	7839	20
SCOTT	3	7788	7566	20
ADAMS	4	7876	7788	20
FORD	3	7902	7566	20
SMITH	4	7369	7902	20
BLAKE	2	7698	7839	30
ALLEN	3	7499	7698	30
WARD	3	7521	7698	30
MARTIN	3	7654	7698	30
TURNER	3	7844	7698	30
JAMES	3	7900	7698	30

14 rows selected.

5. 계층 질의 문장을 활용하여 현재 LEVEL까지 상위 이름을 함께 출력하며, 구분자로 "/" 사용하는 문장을 작성하시오.

검색 결과

```
NAME           PATH
-------------- ------------------------------
KING           /KING
  JONES        /KING/JONES
    SCOTT      /KING/JONES/SCOTT
      ADAMS    /KING/JONES/SCOTT/ADAMS
    FORD       /KING/JONES/FORD
      SMITH    /KING/JONES/FORD/SMITH
  BLAKE        /KING/BLAKE
    ALLEN      /KING/BLAKE/ALLEN
    WARD       /KING/BLAKE/WARD
    MARTIN     /KING/BLAKE/MARTIN
    TURNER     /KING/BLAKE/TURNER
    JAMES      /KING/BLAKE/JAMES
  CLARK        /KING/CLARK
    MILLER     /KING/CLARK/MILLER

14 rows selected.
```

답안. SYS_CONNECT_BY_PATH 함수 사용

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            SYS_CONNECT_BY_PATH(ENAME,'/') AS PATH
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
```

추가 실습

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            LTRIM(SYS_CONNECT_BY_PATH(ENAME,'/'),'/') AS PATH
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
NAME           PATH
-------------- ------------------------------
KING           KING
  JONES        KING/JONES
    SCOTT      KING/JONES/SCOTT
      ADAMS    KING/JONES/SCOTT/ADAMS
...

14 rows selected.
```

6. 계층 질의 결과를 검색하면서 최상의 레벨의 급여와의 현재 레벨 급여의 차이를 계산하시오.

검색 결과

```
NAME                LEVEL      EMPNO        MGR        SAL       DIFF
--------------- ---------- ---------- ---------- ---------- ----------
KING                    1       7839                  5000          0
  JONES                 2       7566       7839       2975       2025
    SCOTT               3       7788       7566       3000       2000
      ADAMS             4       7876       7788       1100       3900
    FORD                3       7902       7566       3000       2000
      SMITH             4       7369       7902        800       4200
  BLAKE                 2       7698       7839       2850       2150
    ALLEN               3       7499       7698       1600       3400
    WARD                3       7521       7698       1250       3750
    MARTIN              3       7654       7698       1250       3750
    TURNER              3       7844       7698       1500       3500
    JAMES               3       7900       7698        950       4050
  CLARK                 2       7782       7839       2450       2550
    MILLER              3       7934       7782       1300       3700

14 rows selected.
```

답안. CONNECT_BY_ROOT 사용

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            level, empno, mgr, sal,
            CONNECT_BY_ROOT sal - sal AS diff
      FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
```

CONNECT_BY_ROOT를 이용하여 최상위 루트의 컬럼을 검색할 수 있다. START WITH의 조건식을 통해 하나의 루트만이 정의되고 있으므로 KING의 급여를 이용하여 계산이 가능하다.

7. EMP 테이블에서 계층 질의를 검색하면서 부하 직원이 없는 사원은 다음과 같이 표시하시오.

검색 결과

```
NAME                LEVEL      EMPNO        MGR CK_LAST
--------------- ---------- ---------- ---------- ----------
KING                    1       7839
  JONES                 2       7566       7839
    SCOTT               3       7788       7566
      ADAMS             4       7876       7788 LAST
    FORD                3       7902       7566
      SMITH             4       7369       7902 LAST
  BLAKE                 2       7698       7839
    ALLEN               3       7499       7698 LAST
    WARD                3       7521       7698 LAST
    MARTIN              3       7654       7698 LAST
    TURNER              3       7844       7698 LAST
    JAMES               3       7900       7698 LAST
  CLARK                 2       7782       7839
    MILLER              3       7934       7782 LAST

14 rows selected.
```

답안. CONNECT_BY_ISLEAF 사용

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            level, empno, mgr,
            DECODE(CONNECT_BY_ISLEAF,1,'LAST') AS CK_LAST
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;

SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            level, empno, mgr,
            CONNECT_BY_ISLEAF
       FROM emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
NAME                LEVEL      EMPNO        MGR CONNECT_BY_ISLEAF
--------------- ---------- ---------- ---------- -----------------
KING                    1       7839                          0
  JONES                 2       7566       7839               0
    SCOTT               3       7788       7566               0
      ADAMS             4       7876       7788               1
    FORD                3       7902       7566               0
      SMITH             4       7369       7902               1
...

14 rows selected.
```

추가 실습 (계층 질의에서의 무한 루프)

```
SQL> DROP TABLE copy_emp PURGE ;
SQL> CREATE TABLE copy_emp
     AS SELECT * FROM emp ;

SQL> UPDATE copy_emp
     SET empno = 7788
     WHERE empno = 7876 ;
SQL> COMMIT ;

SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            LEVEL, empno, mgr
       FROM copy_emp
     START WITH mgr is null
     CONNECT BY prior empno = mgr ;
ERROR:
ORA-01436: CONNECT BY loop in user data

no rows selected

SQL> SELECT ename, empno, mgr
       FROM copy_emp
      WHERE empno = 7788 ;
ENAME           EMPNO        MGR
---------- ---------- ----------
SCOTT            7788       7566
ADAMS            7788       7788

2 rows selected.
```

COPY_EMP 테이블은 UPDATE를 통해 ADAMS의 EMPNO가 7788로 수정되었다. 때문에 CONNECT BY의 조건식이 ADAMS의 행까지 오면 무한 루프를 수행하는 상황이 발생하기 때문에 문장은 실행되지 못한다. 이러한 무한 루프의 수행을 막기 위해 NOCYCLE 키워드를 사용하면 문제가 되는 노드는 제외한 계층 질의를 수행한다. CONNECT_BY_ISCYCLE 컬럼(Pseudo column)은 무한 루프 유무를 확인하는데 사용된다.

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME,
            LEVEL, empno, mgr, CONNECT_BY_ISCYCLE
       FROM copy_emp
     START WITH mgr is null
     CONNECT BY NOCYCLE prior empno = mgr ;
```

NAME	LEVEL	EMPNO	MGR	CONNECT_BY_ISCYCLE
KING	1	7839		0
JONES	2	7566	7839	0
SCOTT	3	7788	7566	1
FORD	3	7902	7566	0
SMITH	4	7369	7902	0
BLAKE	2	7698	7839	0
ALLEN	3	7499	7698	0
WARD	3	7521	7698	0
MARTIN	3	7654	7698	0
TURNER	3	7844	7698	0
JAMES	3	7900	7698	0
CLARK	2	7782	7839	0
MILLER	3	7934	7782	0

```
13 rows selected.

SQL> DROP TABLE copy_emp PURGE ;
```

8. EMP 테이블에서 계층 질의 결과를 Recursive WITH 절을 이용하여 검색하시오.

검색 결과

```
NAME               HLEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
KING                    1       7839
  JONES                 2       7566       7839
    SCOTT               3       7788       7566
      ADAMS             4       7876       7788
    FORD                3       7902       7566
      SMITH             4       7369       7902
  BLAKE                 2       7698       7839
    ALLEN               3       7499       7698
    WARD                3       7521       7698
    MARTIN              3       7654       7698
    TURNER              3       7844       7698
    JAMES               3       7900       7698
  CLARK                 2       7782       7839
    MILLER              3       7934       7782

14 rows selected.
```

주의 사항

Oracle 11gR2부터는 ANSI-SQL에서 지원하는 Recursive WITH 절을 사용할 수 있다.
WITH 절에 UNION ALL로 분기된 두 개의 Query Block이 존재한다. 이때 위에 정의된 Query Block을 Anchor 멤버, 아래 정의된 Query Block을 Recursive 멤버라 한다. Anchor 멤버는 START WITH 절의 역할을 수행하고, Recursive 멤버는 CONNECT BY 절의 역할을 수행한다. 그리고 Recursive WITH 절은 LEVEL이나 SYS_CONNECT_BY_PATH와 같은 함수가 없으므로 필요한 식을 직접 만들어서 사용한다.

```
SQL> WITH htree(hlevel, ename, empno, mgr)
     AS ( SELECT 1, ename, empno, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT hlevel + 1, e.ename, e.empno, e.mgr
            FROM emp e, htree h
           WHERE e.mgr = h.empno )
    SELECT LPAD(' ',hlevel * 2 - 2)||ename AS name, hlevel, empno, mgr
      FROM htree ;
NAME               HLEVEL      EMPNO        MGR
--------------- ---------- ---------- ----------
KING                    1       7839
  JONES                 2       7566       7839
  BLAKE                 2       7698       7839
  CLARK                 2       7782       7839
    SCOTT               3       7788       7566
    FORD                3       7902       7566
    ALLEN               3       7499       7698
    WARD                3       7521       7698
    MARTIN              3       7654       7698
    TURNER              3       7844       7698
    JAMES               3       7900       7698
    MILLER              3       7934       7782
      ADAMS             4       7876       7788
      SMITH             4       7369       7902

14 rows selected.
```

검색 결과는 분명 계층적으로 보이긴 하나, 원하는 결과는 아니다. Recursive WITH 절은 Recursive 멤버의 조건에 만족하는 행들을 하나씩 탐색하는 것이 아니라 조건(E.MGR = H.EMPNO)에 만족하는 모든 행을 탐색하므로 CONNECT BY 절을 사용하는 것과는 다르게 출력된다. 또한 ORDER BY 절을 이용한 정렬도 소용없다.

```
SQL> WITH htree(hlevel, ename, empno, mgr)
     AS ( SELECT 1, ename, empno, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT hlevel + 1, e.ename, e.empno, e.mgr
            FROM emp e, htree h
           WHERE e.mgr = h.empno )
    SELECT LPAD(' ',hlevel * 2 - 2)||ename AS name, hlevel, empno, mgr
      FROM htree
    ORDER BY name;
```

NAME	HLEVEL	EMPNO	MGR
ADAMS	4	7876	7788
SMITH	4	7369	7902
ALLEN	3	7499	7698
FORD	3	7902	7566
JAMES	3	7900	7698
MARTIN	3	7654	7698
MILLER	3	7934	7782
SCOTT	3	7788	7566
TURNER	3	7844	7698
WARD	3	7521	7698
BLAKE	2	7698	7839
CLARK	2	7782	7839
JONES	2	7566	7839
KING	1	7839	

14 rows selected.

Recursive WITH절에 SEARCH 절을 이용하면 동일한 레벨에서 특정 컬럼을 기준으로 정렬 상태를 조정할 수 있다. (계층 질의에서 SIBLING 사용과 유사함)

답안.

```
SQL> WITH htree(hlevel, ename, empno, mgr)
     AS ( SELECT 1, ename, empno, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT hlevel + 1, e.ename, e.empno, e.mgr
            FROM emp e, htree h
           WHERE e.mgr = h.empno )
    SEARCH DEPTH FIRST BY empno ASC SET idx
    SELECT LPAD(' ',hlevel * 2 - 2)||ename AS name, hlevel, empno, mgr
      FROM htree ;
```

IDX 확인

```
SQL> WITH htree(hlevel, ename, empno, mgr)
     AS ( SELECT 1, ename, empno, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT hlevel + 1, e.ename, e.empno, e.mgr
            FROM emp e, htree h
           WHERE e.mgr = h.empno )
    SEARCH DEPTH FIRST BY empno ASC SET idx
    SELECT LPAD(' ',hlevel * 2 - 2)||ename AS name, hlevel, empno, mgr, idx
      FROM htree ;
NAME               HLEVEL      EMPNO        MGR        IDX
-------------- ---------- ---------- ---------- ----------
KING                    1       7839                    1
  JONES                 2       7566       7839         2
    SCOTT               3       7788       7566         3
      ADAMS             4       7876       7788         4
    FORD                3       7902       7566         5
      SMITH             4       7369       7902         6
  BLAKE                 2       7698       7839         7
    ALLEN               3       7499       7698         8
    WARD                3       7521       7698         9
    MARTIN              3       7654       7698        10
    TURNER              3       7844       7698        11
    JAMES               3       7900       7698        12
  CLARK                 2       7782       7839        13
    MILLER              3       7934       7782        14

14 rows selected.
```

오라클의 계층 질의 구문과 동일한 결과를 검색했다. 하지만 위와 같은 결과를 검색할 때는 Recursive WITH 절보다는 오라클의 계층 질의가 훨씬 간결하고 문장 작성이 편리하다. 또한 계층 질의는 여러 함수들이 제공되기 때문에 경우에 따라 원하는 결과를 보다 손쉽게 작성할 수 있다. 그렇다면 Recursive WITH 절은 언제 필요할까?

추가 실습

```
SQL> SELECT LPAD(' ',LEVEL*2-2)||ename AS NAME
           ,level
           ,empno
           ,mgr
           ,sal
           ,LTRIM(SYS_CONNECT_BY_PATH(sal,'+'),'+') AS FORMULA
      FROM emp
      START WITH mgr IS NULL
      CONNECT BY prior empno = mgr ;
```

NAME	LEVEL	EMPNO	MGR	SAL	FORMULA
KING	1	7839		5000	5000
JONES	2	7566	7839	2975	5000+2975
SCOTT	3	7788	7566	3000	5000+2975+3000
ADAMS	4	7876	7788	1100	5000+2975+3000+1100
FORD	3	7902	7566	3000	5000+2975+3000
SMITH	4	7369	7902	800	5000+2975+3000+800
BLAKE	2	7698	7839	2850	5000+2850
ALLEN	3	7499	7698	1600	5000+2850+1600
WARD	3	7521	7698	1250	5000+2850+1250
MARTIN	3	7654	7698	1250	5000+2850+1250
TURNER	3	7844	7698	1500	5000+2850+1500
JAMES	3	7900	7698	950	5000+2850+950
CLARK	2	7782	7839	2450	5000+2450
MILLER	3	7934	7782	1300	5000+2450+1300

14 rows selected.

계층 질의 문장을 이용하여 위와 같은 결과를 검색했을 때, FORMULA 컬럼의 계산식 결과를 함께 검색하려면?

계층 질의 문장은 CONNECT_BY_ROOT, PRIOR 을 이용하여 최상위 노드와 상위 노드의 컬럼 값은 참조가 가능하지만 현재 노드까지의 값 전체를 접근할 수는 없다. 때문에 계산식은 손쉽게 만들었어도 그 결과를 함께 검색하기는 까다롭다. (기준 되는 값이 있다면 분석 함수 등을 활용할 수는 있다.)

하지만 Recursive WITH 절은?

```
SQL> WITH htree(hlevel, ename, empno, mgr, sal, formula, sum_sal)
     AS ( SELECT 1, ename, empno, mgr, sal, TO_CHAR(sal), sal
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT hlevel + 1, e.ename, e.empno, e.mgr, e.sal
                ,h.formula||'+'||TO_CHAR(e.sal)
                ,h.sum_sal + e.sal
           FROM emp e, htree h
          WHERE e.mgr = h.empno )
    SEARCH DEPTH FIRST BY empno ASC SET idx
    SELECT LPAD(' ',hlevel * 2 - 2)||ename AS name, hlevel, empno, mgr, sal, formula, sum_sal
      FROM htree ;
```

NAME	HLEVEL	EMPNO	MGR	SAL	FORMULA	SUM_SAL
KING	1	7839		5000	5000	5000
JONES	2	7566	7839	2975	5000+2975	7975
SCOTT	3	7788	7566	3000	5000+2975+3000	10975
ADAMS	4	7876	7788	1100	5000+2975+3000+1100	12075
FORD	3	7902	7566	3000	5000+2975+3000	10975
SMITH	4	7369	7902	800	5000+2975+3000+800	11775
BLAKE	2	7698	7839	2850	5000+2850	7850
ALLEN	3	7499	7698	1600	5000+2850+1600	9450
WARD	3	7521	7698	1250	5000+2850+1250	9100
MARTIN	3	7654	7698	1250	5000+2850+1250	9100
TURNER	3	7844	7698	1500	5000+2850+1500	9350
JAMES	3	7900	7698	950	5000+2850+950	8800
CLARK	2	7782	7839	2450	5000+2450	7450
MILLER	3	7934	7782	1300	5000+2450+1300	8750

14 rows selected.

10. 정규식 활용

정규식 (Regular Expression)

문자열 데이터의 간단한 패턴 및 복잡한 패턴을 검색할 수 있는 정규식은 기존의 LIKE 연산의 한계를 뛰어넘는 막강한 검색 도구이다. 다양한 Meta Character를 이용하여 Data Validation, ETL(Extract, Transform, Load), Data Cleansing, Data Mining 등의 작업에서 유용하게 사용될 수 있다. 제약조건으로 테이블의 Data 의 유효성을 검증할 때도 사용 가능하다.

Oracle 10g Database부터 지원하며 다음의 함수를 사용한다.

Function	Description
REGEXP_LIKE	Like 연산과 유사하며 정규식 패턴을 검색
REGEXP_REPLACE	정규식 패턴을 검색하여 대체 문자열로 변경
REGEXP_INSTR	정규식 패턴을 검색하여 위치 반환
REGEXP_SUBSTR	정규식 패턴을 검색하여 부분 문자 추출
REGEXP_COUNT (v11g)	정규식 패턴을 검색하여 발견된 횟수 반환

정규식에서는 여러 가지 Meta Character를 지원하며, 복잡한 표현식도 보다 손쉽게 정의할 수 있다.

Meta Character	Description
.	지원되는 Character set 에서 NULL 을 제외한 임의의 문자와 일치
+	한 번 이상 발생 수 일치
?	0 또는 1번 발생 수 일치
*	선행 하위식의 0번 이상 발생 수 일치
{m}	선행 표현식의 정확히 m번 발생 수 일치
{m , }	선행 하위식과 최소 m번 이상 발생 수 일치
{m , n}	선행 하위식의 최소 m번 이상, 최대 n번 이하 발생 수 일치
[...]	괄호 안의 리스트에 있는 임의의 단일 문자와 일치
¦	여러 대안 중 하나와 일치 (OR)
(. . .)	괄호로 묶인 표현식을 한 단위로 취급함. 하위식은 리터럴의 문자열이나 연산자를 포함한 복잡한 표현식 가능
^	문자열 시작 부분과 일치
$	문자열 끝 부분과 일치
\	표현식에서 후속 메타 문자를 리터럴로 처리 (ESCAPE)
\n	괄호 안의 그룹화된 n번째 (1~9) 선행 하위식과 일치. 괄호는 표현식이 기억되도록 만들고 backreference에서 표현식 참조
\d	숫자 문자
[:class:]	지정된 POSIX 문자 클래스에 속한 임의의 문자와 일치 [:alpha:] 알파벳 문자 [:digit:] 숫자 [:lower:] 소문자 알파벳 문자 [:upper:] 대문자 알파벳 문자 [:alnum:] 알파벳/숫자 [:space:] 공백 문자 [:punct:] 구두점 기호 [:cntrl:] 컨트롤 문자 [:print:] 출력 가능한 문자

• REGEXP_LIKE

```
SQL> SELECT cust_id, cust_fname, phone_number
       FROM customers
      WHERE REGEXP_LIKE(phone_number, '[[:alpha:]]' ) ;
   CUST_ID CUST_FNAME           PHONE_NUMBER
---------- -------------------- ------------------------
       119 Maurice              (528) 885-A458
       836 Alexander            (225) 62T-2354

2 rows selected.

SQL> SELECT cust_id, cust_fname, phone_number
       FROM customers ;
   CUST_ID CUST_FNAME           PHONE_NUMBER
---------- -------------------- ------------------------
       110 Charlie              (256) 605-4295
       260 Ellen                551-325-2586
       853 Amrish               (343) 476-4468
       227 Kathy                613-405-3254
       835 Alexander            (725) 201-9538
...

319 rows selected.

SQL> SELECT cust_id, cust_fname, phone_number
       FROM customers
      WHERE REGEXP_LIKE(phone_number,'^\d');
   CUST_ID CUST_FNAME           PHONE_NUMBER
---------- -------------------- ------------------------
       260 Ellen                551-325-2586
       227 Kathy                613-405-3254
       282 Kurt                 688-103-9628
...
34 rows selected.
```

• REGEXP_REPLACE

```
SQL> SELECT cust_id, cust_fname, phone_number,
            REGEXP_REPLACE(phone_number,'(\d{3})-(\d{3})-(\d{4})','(\1) \2-\3') AS NEW_PHONE
       FROM customers
      WHERE REGEXP_LIKE(phone_number,'^\d');
   CUST_ID CUST_FNAME           PHONE_NUMBER             NEW_PHONE
---------- -------------------- ------------------------ ------------------------------
       260 Ellen                551-325-2586             (551) 325-2586
       227 Kathy                613-405-3254             (613) 405-3254
       282 Kurt                 688-103-9628             (688) 103-9628
...
34 rows selected.
```

• REGEXP_INSTR

```
SQL> SELECT cust_id, cust_fname, phone_number, REGEXP_INSTR (phone_number, '[[:alpha:]]' ) pos
       FROM customers
      WHERE REGEXP_LIKE(phone_number, '[[:alpha:]]' ) ;
   CUST_ID CUST_FNAME           PHONE_NUMBER                    POS
---------- -------------------- ------------------------- ----------
       119 Maurice              (528) 885-A458                   11
       836 Alexander            (225) 62T-2354                    9

2 rows selected.
```

• REGEXP_SUBSTR

```
SQL> SELECT cust_id, cust_fname, street_address, REGEXP_SUBSTR (street_address, ' [^ ]+ ') road
       FROM customers ;
   CUST_ID CUST_FNAME           STREET_ADDRESS                          ROAD
---------- -------------------- --------------------------------------- ------------------------------
       110 Charlie              3045 Amos Lane                           Amos
       260 Ellen                37 Butler Street                         Butler
       853 Amrish               5123 Muzzles Street                      Muzzles
       227 Kathy                37 Butterfly Street                      Butterfly
...
319 rows selected.
```

• REGEXP_COUNT (from 11gNF)

```
SQL> SELECT cust_id, cust_fname, street_address
           ,REGEXP_COUNT (street_address, 'a',1,'i') AS cnt1
           ,REGEXP_COUNT (street_address, 'a',1,'c') AS cnt2
       FROM customers
      WHERE street_address LIKE '%A%';
   CUST_ID CUST_FNAME           STREET_ADDRESS                               CNT1       CNT2
---------- -------------------- --------------------------------------- ---------- ----------
       110 Charlie              3045 Amos Lane                                   2          1
       852 Amanda               7466 Cautions Avenue                             2          1
       195 Sean                 3923 Meat Avenue                                 2          1
       833 Alec                 10199 Yelps Avenue                               1          0
       253 Sally                11679 Apportion Street                           1          0
...

37 rows selected.
```

• **Subexpressions (하위식)**

하위식은 REGEXP_INSTR, REGEXP_SUBSTR에서 지원되며 정규식의 검색을 진행할 때 특정 문자열을 지정할 수 있다.

```
SQL> SELECT REGEXP_INSTR ('0123456789',          -- Source
                          '(123)(4(56)(78))',    -- Pattern
                           1,                    -- position
                           1,                    -- occurrence
                           0,                    -- return option
                          'i',                   -- match parameter
                           2 ) "Position"        -- Subexpression
       FROM dual ;
  Position
----------
         5

1 row selected.
```

위의 예제는 12345678의 문자 패턴을 비교하면서 45678의 문자열이 시작되는 위치를 찾아 준다.

```
SQL> SELECT REGEXP_SUBSTR ('0123456789', '(123)(4(56)(78))', 1, 1, 'i', 1 ) "Exp1" ,
            REGEXP_SUBSTR ('0123456789', '(123)(4(56)(78))', 1, 1, 'i', 2 ) "Exp2" ,
            REGEXP_SUBSTR ('0123456789', '(123)(4(56)(78))', 1, 1, 'i', 3 ) "Exp3" ,
            REGEXP_SUBSTR ('0123456789', '(123)(4(56)(78))', 1, 1, 'i', 4 ) "Exp4"
      FROM dual ;
Exp1       Exp2       Exp3       Exp4
---------- ---------- ---------- ----------
123        45678      56         78

1 row selected.
```

12345678의 문자열을 검색하며 ( )의 순서에 따라 하위식 순번을 결정한다. 첫 번째 하위식은 (123)이며 두 번째 하위식은 (45678)이 된다. 세 번째, 네 번째 하위식은 각각 (56), (78)이다. REGEXP_INSTR은 위치를 찾을 수 있으며 REGEXP_SUBSTR은 그 문자열에 해당되는 부분을 추출할 수 있다.

11. Data Manipulation Language (DML) 활용

다중 테이블의 INSERT

EMP 테이블에서 1983년 이전에 입사한 사원 정보와 1982년 이후에 입사한 사원 정보를 서로 다른 테이블에 입력하려면 다음과 같이 수행할 수 있다.

```
SQL> INSERT INTO sal_hist
     SELECT empno, ename, hiredate, sal
     FROM emp
     WHERE hiredate < TO_DATE('1982/12/31','YYYY/MM/DD') ;
13 rows created.

SQL> INSERT INTO mgr_hist
     SELECT empno, ename, hiredate, mgr
     FROM emp
     WHERE hiredate > TO_DATE('1982/01/01','YYYY/MM/DD') ;
3 rows created.

SQL> ROLLBACK ;
```

하지만 동일한 데이터의 반복적인 액세스와 명령문이 하나 이상으로 처리되는 것은 DBMS call 증가와 불필요한 액세스 증가로 성능에 악영향을 미칠 수 있다.

※ Unconditional INSERT ALL

```
SQL> INSERT ALL
       INTO sal_hist     VALUES (empno,ename,hiredate,sal)
       INTO mgr_hist    VALUES (empno,ename,hiredate,mgr)
     SELECT empno, ename, hiredate, sal, mgr
     FROM emp ;
28 rows created.

SQL> SELECT * FROM sal_hist ;
     EMPNO ENAME      HIREDATE          SAL
---------- ---------- ---------- ----------
      7788 SCOTT      1987/04/19       3000
      7566 JONES      1981/04/02       2975
      7369 SMITH      1980/12/17        800
      7876 ADAMS      1987/05/23       1100
      7499 ALLEN      1981/02/20       1600
      7521 WARD       1981/02/22       1250
      7654 MARTIN     1981/09/28       1250
      7698 BLAKE      1981/05/01       2850
      7782 CLARK      1981/06/09       2450
      7844 TURNER     1981/09/08       1500
      7900 JAMES      1981/12/03        950
      7902 FORD       1981/12/03       3000
      7934 MILLER     1982/01/23       1300
      7839 KING       1981/11/17       5000
14 rows selected.

SQL> SELECT * FROM mgr_hist ;
     EMPNO ENAME      HIREDATE          MGR
---------- ---------- ---------- ----------
      7788 SCOTT      1987/04/19       7566
      7566 JONES      1981/04/02       7839
      7369 SMITH      1980/12/17       7902
      7876 ADAMS      1987/05/23       7788
      7499 ALLEN      1981/02/20       7698
      7521 WARD       1981/02/22       7698
      7654 MARTIN     1981/09/28       7698
      7698 BLAKE      1981/05/01       7839
      7782 CLARK      1981/06/09       7839
      7844 TURNER     1981/09/08       7698
      7900 JAMES      1981/12/03       7698
      7902 FORD       1981/12/03       7566
      7934 MILLER     1982/01/23       7782
      7839 KING       1981/11/17
14 rows selected.

SQL> ROLLBACK ;
Rollback complete.
```

※ Conditional INSERT ALL

```
SQL> INSERT ALL
       WHEN hiredate < TO_DATE('1982/12/31','YYYY/MM/DD') THEN
         INTO sal_hist   VALUES (empno,ename,hiredate,sal)
       WHEN hiredate > TO_DATE('1982/01/01','YYYY/MM/DD') THEN
         INTO mgr_hist    VALUES (empno,ename,hiredate,mgr)
     SELECT empno, ename, hiredate, sal, mgr
     FROM emp ;
15 rows created.

SQL> SELECT * FROM sal_hist ;
     EMPNO ENAME      HIREDATE          SAL
---------- ---------- ---------- ----------
      7566 JONES      1981/04/02       2975
      7369 SMITH      1980/12/17        800
      7499 ALLEN      1981/02/20       1600
      7521 WARD       1981/02/22       1250
      7654 MARTIN     1981/09/28       1250
      7698 BLAKE      1981/05/01       2850
      7782 CLARK      1981/06/09       2450
      7844 TURNER     1981/09/08       1500
      7900 JAMES      1981/12/03        950
      7902 FORD       1981/12/03       3000
      7934 MILLER     1982/01/23       1300
      7839 KING       1981/11/17       5000
12 rows selected.

SQL> SELECT * FROM mgr_hist ;
     EMPNO ENAME      HIREDATE          MGR
---------- ---------- ---------- ----------
      7788 SCOTT      1987/04/19       7566
      7876 ADAMS      1987/05/23       7788
      7934 MILLER     1982/01/23       7782
3 rows selected.

SQL> ROLLBACK ;
Rollback complete.
```

※ Conditional INSERT FIRST

```
SQL> INSERT FIRST
       WHEN hiredate < TO_DATE('1982/12/31','YYYY/MM/DD') THEN
         INTO sal_hist   VALUES (empno,ename,hiredate,sal)
       WHEN hiredate > TO_DATE('1982/01/01','YYYY/MM/DD') THEN
         INTO mgr_hist    VALUES (empno,ename,hiredate,mgr)
     SELECT empno, ename, hiredate, sal, mgr
     FROM emp ;
14 rows selected.

SQL> SELECT * FROM sal_hist ;
     EMPNO ENAME      HIREDATE          SAL
---------- ---------- ---------- ----------
      7566 JONES      1981/04/02       2975
      7369 SMITH      1980/12/17        800
      7499 ALLEN      1981/02/20       1600
      7521 WARD       1981/02/22       1250
      7654 MARTIN     1981/09/28       1250
      7698 BLAKE      1981/05/01       2850
      7782 CLARK      1981/06/09       2450
      7844 TURNER     1981/09/08       1500
      7900 JAMES      1981/12/03        950
      7902 FORD       1981/12/03       3000
      7934 MILLER     1982/01/23       1300
      7839 KING       1981/11/17       5000
12 rows selected.

SQL> SELECT * FROM mgr_hist ;
     EMPNO ENAME      HIREDATE          MGR
---------- ---------- ---------- ----------
      7788 SCOTT      1987/04/19       7566
      7876 ADAMS      1987/05/23       7788
2 rows selected.

SQL> ROLLBACK ;
Rollback complete.
```

※ Pivot INSERT ALL

```
SQL> SELECT *
       FROM source_data
      ORDER BY empno, year, week_id;
     EMPNO       YEAR    WEEK_ID  SALES_MON  SALES_TUE  SALES_WED SALES_THUR  SALES_FRI
---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
      7499       2016          1        477        333        422        339        704
      7499       2016          2        396         77        642        167        401
      7499       2016          3        538        389        595        643        913
      7499       2016          4        201        238        517        763        802
      7499       2016          5        763        944        251        444        364
...

212 rows selected.

SQL> SELECT SUM(sales_mon), SUM(sales_tue), SUM(sales_wed), SUM(sales_thur), SUM(sales_fri)
       FROM source_data ;
SUM(SALES_MON) SUM(SALES_TUE) SUM(SALES_WED) SUM(SALES_THUR) SUM(SALES_FRI)
-------------- -------------- -------------- --------------- --------------
        102757         102274         106921           98849         105694

SQL> SELECT SUM(sales_mon), SUM(sales_tue), SUM(sales_wed), SUM(sales_thur), SUM(sales_fri),
            AVG(sales_mon), AVG(sales_tue), AVG(sales_wed), AVG(sales_thur), AVG(sales_fri)
       FROM source_data ;
...

SQL> desc sales_info
 Name                      Null?    Type
 ------------------------- -------- ----------------
 EMPNO                              NUMBER(4)
 YEAR                               NUMBER(4)
 WEEK_ID                            NUMBER(2)
 DAY                                VARCHAR2(4)
 SALES_QTY                          NUMBER(4)

SQL> SELECT *
       FROM sales_info
      ORDER BY empno, year, week_id ;
     EMPNO       YEAR    WEEK_ID DAY   SALES_QTY
---------- ---------- ---------- ---- ----------
      7499       2016          1 THUR        339
      7499       2016          1 FRI         704
      7499       2016          1 WED         422
      7499       2016          1 TUE         333
      7499       2016          1 MON         477
...
1060 rows selected.
```

```
SQL> SELECT day, SUM(sales_qty), AVG(sales_qty)
       FROM sales_info
       GROUP BY day ;
DAY  SUM(SALES_QTY) AVG(SALES_QTY)
---- -------------- --------------
TUE          102274     491.701923
THUR          98849     475.235577
FRI          105694     498.556604
MON          102757     494.024038
WED          106921     514.043269

5 rows selected.

SQL> TRUNCATE TABLE sales_info ;

SQL> INSERT ALL
       INTO sales_info VALUES (empno, year, week_id, 'MON', sales_mon)
       INTO sales_info VALUES (empno, year, week_id, 'TUE', sales_tue)
       INTO sales_info VALUES (empno, year, week_id, 'WED', sales_wed)
       INTO sales_info VALUES (empno, year, week_id, 'THUR', sales_thur)
       INTO sales_info VALUES (empno, year, week_id, 'FRI', sales_fri)
     SELECT * FROM source_data ;
1060 rows created.

SQL> commit ;

SQL> SELECT *
       FROM sales_info
       ORDER BY empno, year, week_id ;
     EMPNO       YEAR    WEEK_ID DAY   SALES_QTY
---------- ---------- ---------- ---- ----------
      7499       2016          1 THUR        339
      7499       2016          1 FRI         704
      7499       2016          1 WED         422
      7499       2016          1 TUE         333
      7499       2016          1 MON         477
...

1060 rows selected.
```

※ PIVOT 사용

```
SQL> SELECT *
       FROM ( SELECT day, sales_qty FROM sales_info )
      PIVOT ( SUM(sales_qty) FOR day IN ('MON'  AS SALES_MON,
                                         'TUE'  AS SALES_TUE,
                                         'WED'  AS SALES_WED,
                                         'THUR' AS SALES_THUR,
                                         'FRI'  AS SALES_FRI)) ;
 SALES_MON  SALES_TUE  SALES_WED SALES_THUR  SALES_FRI
---------- ---------- ---------- ---------- ----------
    102757     102274     106921      98849     105694
```

※ UNPIVOT 사용

```
SQL> SELECT empno, year, week_id, SUBSTR(day,7) AS day, sales
       FROM source_data
      UNPIVOT ( sales FOR day IN (SALES_MON,SALES_TUE,SALES_WED,SALES_THUR,SALES_FRI))
      ORDER BY empno, year, week_id ;
     EMPNO       YEAR    WEEK_ID DAY                 SALES
---------- ---------- ---------- ---------------- ----------
      7499       2016          1 MON                     477
      7499       2016          1 TUE                     333
      7499       2016          1 WED                     422
      7499       2016          1 THUR                    339
      7499       2016          1 FRI                     704
...

1044 rows selected.
```

MERGE 문 사용

```
SQL> CREATE TABLE copy_emp
     AS
     SELECT *
       FROM emp
      WHERE deptno = 30 ;
Table created.

SQL> UPDATE copy_emp
      SET sal = sal * 0.5
      WHERE job = 'SALESMAN' ;
4 rows updated.

SQL> COMMIT ;
Commit complete.

SQL> SELECT ename, sal, comm, deptno
       FROM emp
      ORDER BY deptno, sal ;
ENAME             SAL       COMM     DEPTNO
---------- ---------- ---------- ----------
MILLER          1300                    10
CLARK           2450                    10
KING            5000                    10
SMITH            800                    20
ADAMS           1100                    20
JONES           2975                    20
SCOTT           3000                    20
FORD            3000                    20
JAMES            950                    30
WARD            1250        500         30
MARTIN          1250       1400         30
TURNER          1500          0         30
ALLEN           1600        300         30
BLAKE           2850                    30

14 rows selected.
```

```
SQL> SELECT ename, sal, comm, deptno
       FROM copy_emp
      ORDER BY deptno, sal ;
ENAME             SAL       COMM     DEPTNO
---------- ---------- ---------- ----------
MARTIN            625       1400         30
WARD              625        500         30
TURNER            750          0         30
ALLEN             800        300         30
JAMES             950                    30
BLAKE            2850                    30

6 rows selected.

SQL> MERGE INTO copy_emp c
     USING emp e
     ON (c.empno = e.empno)
     WHEN MATCHED THEN
     UPDATE
     SET c.sal  = e.sal ,
         c.comm = e.comm
     WHEN NOT MATCHED THEN
     INSERT
     VALUES (e.empno,e.ename,e.job, e.mgr, e.hiredate, e.sal, e.comm, e.deptno) ;
14 rows merged.

SQL> SELECT ename, sal, comm, deptno
       FROM copy_emp
      ORDER BY deptno, sal ;
ENAME             SAL       COMM     DEPTNO
---------- ---------- ---------- ----------
MILLER           1300                    10
CLARK            2450                    10
KING             5000                    10
SMITH             800                    20
ADAMS            1100                    20
JONES            2975                    20
SCOTT            3000                    20
FORD             3000                    20
JAMES             950                    30
MARTIN           1250       1400         30
WARD             1250        500         30
TURNER           1500          0         30
ALLEN            1600        300         30
BLAKE            2850                    30

14 rows selected.

SQL> ROLLBACK ;
```

```
SQL> MERGE INTO copy_emp c
     USING emp e
      ON (c.empno = e.empno)
      WHEN MATCHED THEN
      UPDATE
      SET c.sal    = e.sal ,
          c.comm = e.comm
      DELETE WHERE (c.comm IS NOT NULL)
      WHEN NOT MATCHED THEN
      INSERT
      VALUES (e.empno,e.ename,e.job, e.mgr, e.hiredate, e.sal, e.comm, e.deptno) ;
14 rows merged.

SQL> SELECT ename, sal, comm, deptno
       FROM copy_emp
      ORDER BY deptno, sal ;
ENAME             SAL       COMM     DEPTNO
---------- ---------- ---------- ----------
MILLER          1300                    10
CLARK           2450                    10
KING            5000                    10
SMITH            800                    20
ADAMS           1100                    20
JONES           2975                    20
SCOTT           3000                    20
FORD            3000                    20
JAMES            950                    30
BLAKE           2850                    30

10 rows selected.

SQL> DROP TABLE copy_emp PURGE ;
Table dropped.
```

12. 읽기 일관성 (Read Consistency)

읽기 일관성

오라클 데이터베이스는 수 많은 동시 접속자가 하나의 테이블(데이터)에 액세스를 하고 있으며, 그 중에는 DML 문장도 함께 실행된다. 이때 DML 을 수행하는 세션과 검색을 수행하는 세션이 서로 다르면 COMMIT 된 결과만을 검색할 수 있다는 것은 기초적인 SQL 구문을 공부한 사용자라면 이해하고 있을 것이다. 그렇다면 다음 상황에서는 어떤 결과를 확인할 수 있을까?

# SESSION 1	# SESSION 2
SQL> set pagesize 10 **SQL> set arraysize 7** **SQL> set pause on** **SQL> SELECT ename, sal FROM emp;** ENAME SAL ---------- ---------- ... ALLEN 1600 WARD 1250 MARTIN 1250 *[PAUSE]*	
SESSION 1 의 검색이 끝나지 않은 상태에서 데이터가 수정되고, COMMIT 이 수행되면 그 결과가 반영될 수 있을까?	**SQL> SELECT ename, sal** **FROM emp** **WHERE ename = 'KING' ;** ENAME SAL ---------- ---------- KING 5000 **SQL> UPDATE emp** **SET sal = 6000** **WHERE ename = 'KING' ;** 1 row updated. **SQL> COMMIT ;**
[ENTER] ENAME SAL ---------- ---------- ... FORD 3000 MILLER 1300 **KING 5000** 14 rows selected. 또는 ERROR at line 1: ORA-01555: snapshot too old: rollback segment number 19 with name "_SYSSMU19_810267679$" too small	실제 결과를 확인하면 수정 전 값을 검색한다. 이는 Undo Data 를 참조할 수 있을때 가능하며, Undo Data 를 참조할 수 없으면 에러가 발생한다. (ORA-01555: shapshot too old)

SQL> SELECT ename, sal FROM emp; ENAME SAL ---------- ---------- ... MILLER 1300 KING 6000 14 rows selected.	문장이 새로 시작되면 수정 결과를 반영할 수 있다. 즉, 하나의 쿼리가 수행되는 도중에는 다른 세션의 DML 결과가 반영되지 않는다.
	SQL> UPDATE emp SET sal = 5000 WHERE ename = 'KING' ; 1 row updated. SQL> COMMIT ;

하지만 동일 SELECT 명령문이라도 그 실행 횟수가 2 번 이상이면, 다른 세션의 종료된 트랜잭션 결과는 반영될 수 있다.

```
SQL> DROP TABLE emp2 PURGE ;
SQL> CREATE TABLE emp2
     AS SELECT ROWNUM AS empno, ename, sal, deptno FROM emp ;

SQL> CREATE OR REPLACE FUNCTION GET_AVG (P_DEPTNO  NUMBER)
       RETURN NUMBER
     IS
       V_AVG  NUMBER ;
     BEGIN
        SELECT /*+ TEST */ ROUND(AVG(SAL)) INTO V_AVG
          FROM EMP2
         WHERE DEPTNO = P_DEPTNO ;

        RETURN V_AVG ;
     END ;
/
```

사용자 정의 함수에서 SQL 명령문을 포함하고 있다면 어떠한 문제가 발생하는지도 함께 확인하다.

```
SQL> SELECT EMPNO, ENAME, SAL, DEPTNO, GET_AVG(DEPTNO)
       FROM EMP2 ;
     EMPNO ENAME                      SAL     DEPTNO GET_AVG(DEPTNO)
---------- -------------------- ---------- ---------- ---------------
         1 SCOTT                      3000         20            2175
         2 JONES                      2975         20            2175
...
        11 JAMES                       950         30            1567
        12 FORD                       3000         20            2175
        13 MILLER                     1300         10            2917
        14 KING                       5000         10            2917

14 rows selected.

SQL> SELECT sql_text, executions
       FROM v$sql
      WHERE sql_text LIKE 'SELECT /*+ TEST */%';
SQL_TEXT                           EXECUTIONS
---------------------------------- ----------
SELECT /*+ TEST */ ROUND(AVG(SAL))         14
FROM EMP2 WHERE DEPTNO = :B1
```

SQL 명령문에 포함된 함수는 행의 개수에 따라 그 실행횟수가 증가될 수 있고, 함수 안에 SQL 명령문이 포함돼있다면 해당 SQL 명령문도 반복적으로 실행된다. 이때 발생하는 문제는 무엇일까?

```
SQL> INSERT INTO emp2
     SELECT ROWNUM+14, ename, sal, deptno
       FROM emp2 e
       CROSS JOIN
            (SELECT LEVEL AS NO
               FROM dual
            CONNECT BY LEVEL <= 1000) ;
SQL> COMMIT ;

SQL> set arraysize 100
SQL> set pagesize 100
SQL> set pause on
SQL> SELECT rownum, empno, ename, sal, deptno, GET_AVG(deptno)
      FROM emp2
     WHERE ename = 'SCOTT' ;
    ROWNUM      EMPNO ENAME                      SAL     DEPTNO GET_AVG(DEPTNO)
---------- ---------- -------------------- ---------- ---------- ---------------
         1          1 SCOTT                      3000         20            2175
         2         15 SCOTT                      3000         20            2175
...
        96       1331 SCOTT                      3000         20            2175
        97       1345 SCOTT                      3000         20            2175
[PAUSE]
```

```
# New Terminal

$ sqlplus test/test
SQL> UPDATE emp2
     SET sal = 1500
     WHERE ename = 'SCOTT' ;
SQL> COMMIT ;
SQL> EXIT

# Original Terminal

[ENTER]
    ROWNUM      EMPNO ENAME                     SAL     DEPTNO GET_AVG(DEPTNO)
---------- ---------- -------------------- ---------- ---------- ---------------
        98       1359 SCOTT                      3000         20            2175
        99       1373 SCOTT                      3000         20            2175
       100       2731 SCOTT                      3000         20            2175
       101       2745 SCOTT                      3000         20            2175
       102       2759 SCOTT                      3000         20            1875
       103       2773 SCOTT                      3000         20            1875
...
SQL> set pause off
SQL> DROP TABLE emp2 PURGE ;
```

첫 번째 페이지의 평균 급여와 두 번째 페이지의 평균 급여의 차이가 발생한다. 데이터는 계속 변하기 때문에 시차를 두고 검색되는 페이지마다 서로 다른 평균 급여를 보여줄 수는 있다. 하지만 평균 급여는 수정된 SCOTT 의 급여로 계산됐지만 SCOTT 의 급여는 수정 전의 급여가 검색된다. 이것이 일관된 검색결과라고 할 수 있을까?

또한 사용자 정의 함수를 생성할 때 SQL 명령이 포함되면, 일관성에 문제가 없는 상황에서만 함수를 호출하는 것이 필요하다.

13. 데이터 타입의 올바른 사용

1. 데이터 타입의 올바른 사용

데이터베이스 사용 경험이 있다면 기본적인 데이터 타입의 구분 및 특징은 알고 있을 것입니다. 하지만 저자가 강의를 하면서 만나 본 많은 사용자들은 데이터 타입에 대한 정확한 이해보다는 기초적인 데이터 타입의 특성만을 이해하고, 경험을 기준으로 데이터 타입을 사용하고 있었습니다. 이는 원하는 데이터를 가공할 때 데이터의 무결성을 훼손하거나, 불필요한 형 변환 함수의 사용으로 성능 저하를 초래할 수도 있습니다. 이번 과정에서는 데이터 타입에 대한 올바른 이해와 불필요한 함수 사용을 최소화하는 방법 및 데이터 가공시 유용한 함수의 사용 방법을 숙지합니다.

1.1. 데이터 타입 종류

데이터 타입은 여러 종류의 데이터를 식별하고 저장하는 방식을 정의한 것으로 오라클 데이터베이스에도 많은 데이터 타입이 존재합니다. 그 전체 목록은 오라클 매뉴얼에서 확인할 수 있습니다. 여기서는 일반적으로 자주 사용하고, 오류를 범하기 쉬운 데이터 타입의 특징 및 주의사항을 소개합니다.

Type	Size	Description
VARCHAR2	1 ~ 4000 byte	가변 길이 문자
CHAR	1 ~ 2000 byte	고정 길이 문자
NUMBER	NUMBER(p,s)	숫자
DATE	7byte	날짜 및 시,분,초 형식의 날짜
TIMESTAMP	11byte	DATE + 정밀화된 시간 (최대 9 자리까지 세분화된 초)

참고. Oracle DB 12c 부터는 VARCHAR2, CHAR 의 최대 길이가 32,767 byte 까지 설정 가능

경우에 따라 Large Object나 Binary 데이터를 저장하기 위해 LONG, CLOB, BLOB 등을 사용하기도 하지만 본 과정의 주제와 맞지 않는 부분이 있으므로 해당 데이터 타입에 대한 설명은 제외합니다.

1.1.1. CHARACTER 타입

문자 데이터를 저장할 경우 가장 많이 사용하는 타입이 CHAR, VARCHAR2입니다. 두 데이터 타입은 고정 길이와 가변 길이로 데이터를 저장하며, 데이터베이스 버전에 따라 최대 길이의 차이가 존재할 수 있습니다. 실습을 통해 고정 길이, 가변 길이 문자를 저장하기 위한 어떠한 방식을 사용하는지 확인하겠습니다.

```
SQL> CREATE TABLE chr_typ
     (c1  CHAR(5),                                    -- 고정 길이 5byte
      c2  VARCHAR2(5)) ;                              -- 가변 길이 5byte
SQL> INSERT INTO chr_typ VALUES ('ABC', 'ABC') ;      -- 3byte 문자 입력
SQL> COMMIT ;
SQL> SELECT c1, DUMP(c1), c2, DUMP(c2) FROM chr_typ;

C1    DUMP(C1)                       C2    DUMP(C2)
----- ------------------------------ ----- ------------------------------
ABC   Typ=96 Len=5: 65,66,67,32,32   ABC   Typ=1 Len=3: 65,66,67

14 rows selected.
```

동일한 길이의 문자를 입력하였지만 데이터 타입에 따라 실제 저장된 내용은 다른 것을 확인할 수 있습니다. 고정 길이 문자는 최댓값 미만의 문자가 입력될 때 공백 문자를(ASCII: 32) 뒤에 붙여서 최대 길이를 항상 만족시킵니다. 하지만 가변 길이 문자는 입력된 문자까지 만을 저장합니다. 때문에 가변 길이 문자를 사용하게 되면 불필요한 저장 공간의 낭비를 제거할 수 있다는 장점을 갖습니다. 반면 차후 컬럼의 길이가 길어지는 UPDATE가 수행될 경우 저장공간의 부족으로 Row Migration 현상이 발생할 수도 있습니다. 고정 길이 문자 타입은 저장공간의 낭비는 있지만 항상 최댓값까지의 저장 공간을 확보하고 있으므로 입력 시점보다 길이가 길어지는 UPDATE가 수행되더라도 Row Migration 현상은 없을 것입니다.

실제 현장에서는 CHAR보다 VARCHAR2를 많이 사용합니다. 성능을 위해서라면 고정 길이가 좋을듯싶지만 Row Migration 현상을 방지할 수 있는 오라클 데이터베이스의 설정도 존재하기 때문에 Row Migration 현상 때문에 고정 길이 문자를 사용하지는 않습니다. 오히려 저장 공간의 낭비를 줄일 수 있는 VARCHAR2 타입이 선호됩니다.

고정 길이와 가변 길이의 차이점은 구분하셨나요? 여기까지는 이미 알고 있던 사실을 확인한 수준일 수 있습니다. 실제로 더 중요한 것은 두 데이터 타입의 사용상 차이점과 주의 사항을 이해하는 것입니다.

다음과 같이 고정 길이 문자 컬럼에서 3개의 WHERE 절을 각각 사용했을 때 검색 결과가 나올 수 있는 것은 무엇일까요?

```
SQL> SELECT * FROM chr_typ
   ① WHERE c1 = 'ABC' ;          -- 공백이 없는 경우
   ② WHERE c1 = 'ABC  ' ;        -- 공백을 2 개 붙인 경우
   ③ WHERE c1 = 'ABC     ' ;     -- 공백을 5 개 붙인 경우
```

선뜻 답을 유추할 수 없다면 고정 길이 문자가 어떻게 비교되는지 이해가 부족하다는 의미입니다. 동일한 조건절을 사용하여 가변 길이 문자에서 비교한다면 어떨까요?

```
SQL> SELECT * FROM chr_typ
   ① WHERE c2 = 'ABC' ;          -- 공백이 없는 경우
   ② WHERE c2 = 'ABC  ' ;        -- 공백을 2 개 붙인 경우
   ③ WHERE c2 = 'ABC     ' ;     -- 공백을 5 개 붙인 경우
```

가변 길이 문자는 저장된 데이터가 'ABC'이므로 동일한 값을 비교하는 1번 조건절만 검색 결과를 가져올 수 있습니다. 나머지 2, 3번의 조건절은 "no rows selected". 즉, 조건에 만족하는 검색 결과가 없습니다. 하지만 고정 길이 문자는 다릅니다. 3개의 조건절 모두 동일한 검색 결과를 갖습니다. 즉, 'ABC ' 값을 검색할 수 있습니다.

```
SQL> SELECT * FROM chr_typ WHERE c1 = 'ABC' ;
SQL> SELECT * FROM chr_typ WHERE c1 = 'ABC  ' ;
SQL> SELECT * FROM chr_typ WHERE c1 = 'ABC     ' ;      -- 모두 동일한 결과
C1    C2
----- -----
ABC   ABC
1 row selected.

SQL> SELECT * FROM chr_typ WHERE c1 = 'ABC     A' ;
no rows selected
```

고정 길이 문자는 조건을 비교할 경우 공백 문자를 붙여 양쪽 값을 더 긴 쪽과 동일하게 만듭니다. 어차피 INSERT 시점에도 고정 길이를 유지하기 위해 공백 문자를 붙일 수 있듯이, 조건을 비교할 때도 동일한 작업을 수행합니다. 때문에 비교되는 값이 5byte 미만이더라도, 아니면 오히려 더 길더라도 더 짧은 값에 공백을 붙여 비교하므로 모두 동일한 검색 결과를 보여줍니다. 하지만 4번째 예제는 마지막 자리에 공백이 아닌 다른 문자가 비교되므로 검색되는 행이 존재할 수 없습니다.

CHAR 타입을 사용하는 컬럼이 위와 같이 비조인 술어에서 사용될 때는 별도의 함수 사용 없이도 정상적인 검색이 가능합니다. 하지만 문제는 조인 술어나, DECODE 같은 함수에서 발생합니다.

```
SQL> ALTER TABLE emp MODIFY job CHAR(9) ;          -- 고정 길이 9byte 로 변경
SQL> SELECT deptno, job, LENGTH(TRIM(job)), LENGTH(job), sal
     FROM emp ;                                     -- 문자열 길이 계산
    DEPTNO JOB       LENGTH(TRIM(JOB)) LENGTH(JOB)        SAL
---------- --------- ----------------- ----------- ----------
        20 CLERK                     5           9        800
        30 SALESMAN                  8           9       1600
        30 SALESMAN                  8           9       1250
        20 MANAGER                   7           9       2975
        30 SALESMAN                  8           9       1250
        30 MANAGER                   7           9       2850
        10 MANAGER                   7           9       2450
        20 ANALYST                   7           9       3000
        10 PRESIDENT                 9           9       5000
        30 SALESMAN                  8           9       1500
        20 CLERK                     5           9       1100
        30 CLERK                     5           9        950
        20 ANALYST                   7           9       3000
        10 CLERK                     5           9       1300

14 rows selected.
```

JOB 컬럼의 데이터 타입을 고정 길이 문자로 변경하였습니다. 컬럼 길이를 확인하면 실제 문자 개수와 상관없이 고정 길이 9byte를 갖습니다. 즉, 9byte 미만의 문자열은 공백을 붙여 모든 행이 9byte 길이만큼 저장되었습니다. 이때 다음과 같은 문장은 신뢰할 수 있을까요?

```
SQL> SELECT deptno,
            NVL(SUM(DECODE(job, 'ANALYST',   sal)),0)  AS ANALYST,
            NVL(SUM(DECODE(job, 'CLERK',     sal)) ,0) AS CLERK,
            NVL(SUM(DECODE(job, 'MANAGER',   sal)) ,0) AS MANAGER,
            NVL(SUM(DECODE(job, 'PRESIDENT', sal)) ,0) AS PRESIDENT,
            NVL(SUM(DECODE(job, 'SALESMAN',  sal)) ,0) AS SALESMAN
     FROM emp
     GROUP BY deptno
     ORDER BY deptno ;
    DEPTNO    ANALYST      CLERK    MANAGER  PRESIDENT   SALESMAN
---------- ---------- ---------- ---------- ---------- ----------
        10          0          0          0       5000          0
        20          0          0          0          0          0
        30          0          0          0          0          0
3 rows selected.
```

앞서 검색한 EMP 테이블에서는 부서별 여러 JOB 컬럼 값을 가지고 있었으나 검색 결과는 10번 부서의 "PRESIDENT" 값만 출력되었습니다. 에러가 발생하지 않았다고 위의 결과를 신뢰한다면 데이터 품질 관리가 제대로 이루어지지는 않을 것입니다. 물론 검색 결과 내용을 검토하면서 업무를 진행하니 그냥 넘어가지는 않을 수도 있습니다. 하지만 검색된 모든 결과를 확인할 수도 없습니다. 위의 예제는 쉽게 오류를 파악할 수 있는 상황이지만 전체 검색 결과에서 1~2개의 잘못된 행은 확인하기가 매우 까다롭습니다. 때문에 보다 신뢰받는 문장을 작성하고자 한다면 데이터 타입의 주의 사항을 이해하고 데이터 타입에 맞는 문장을 사용해야 합니다.

우선은 위와 같은 상황이 발생된 원인은 무엇일까요? DECODE 함수는 비교 컬럼의 데이터 타입이 고정 길이 문자일 경우 앞선 비조인 술어처럼 탐색 값에 자동으로 공백 문자를 붙여서 비교하지 않습니다. 동일한 상황은 조인 술어에서도 발생합니다.

```
SQL> SELECT employee_id, last_name, job_id, department_id
     FROM employees
     WHERE department_id IN (50,110) ;
EMPLOYEE_ID LAST_NAME       JOB_ID     DEPARTMENT_ID
----------- --------------- ---------- -------------
        124 Mourgos         ST_MAN                50
        141 Rajs            ST_CLERK              50
        142 Davies          ST_CLERK              50
        143 Matos           ST_CLERK              50
        144 Vargas          ST_CLERK              50
        205 Higgins         AC_MGR               110
        206 Gietz           AC_ACCOUNT           110
7 rows selected.

SQL> SELECT job_id, job_title
     FROM jobs ;
JOB_ID     JOB_TITLE
---------- -----------------------------------
AD_PRES    President
AD_VP      Administration Vice President
AD_ASST    Administration Assistant
AC_MGR     Accounting Manager
AC_ACCOUNT Public Accountant
SA_MAN     Sales Manager
SA_REP     Sales Representative
ST_MAN     Stock Manager
ST_CLERK   Stock Clerk
...
12 rows selected.
```

EMPLOYEES 테이블의 검색 결과는 7개입니다. JOB_TITLE 컬럼은 JOBS 테이블에 존재하므로 조인문을 작성해야 사원 정보와 JOB_TITLE을 함께 볼 수 있습니다. 그렇다면 조인을 수행하면 7개의 행 정보가 제대로 보일까요?

```
SQL> SELECT e.employee_id, e.last_name, e.job_id, j.job_title
     FROM employees e, jobs j
     WHERE e.job_id = j.job_id
       AND e.department_id IN (50,110) ;
EMPLOYEE_ID LAST_NAME       JOB_ID     JOB_TITLE
----------- --------------- ---------- -----------------------------------
        206 Gietz           AC_ACCOUNT Public Accountant
1 row selected.
```

검색 결과는 1개 행이 검색되었고, 'AC_ACCOUNT'를 제외한 나머지 행이 누락된 것을 확인할 수 있습니다. 그 이유는 무엇일까요?

```
SQL> DESCRIBE employees
 Name              Null?    Type
 ----------------- -------- ------------
...
 HIRE_DATE                  DATE
 JOB_ID                     VARCHAR2(10)
...
SQL> DESCRIBE jobs
 Name              Null?    Type
 ----------------- -------- ------------
 JOB_ID                     CHAR(10)
 JOB_TITLE                  VARCHAR2(35)
...
```

조인 컬럼의 데이터 타입이 서로 다른 것을 확인할 수 있습니다. JOBS 테이블의 JOB_ID 컬럼은 CHAR 타입이므로 10byte 미만의 문자열은 뒤에 공백이 붙어 있습니다.

조인 조건절에서는 컬럼과 컬럼이 비교되어야 합니다. 이때 두 컬럼의 데이터 타입이 갖지 않으면 암시적인 형 변환이 수행될 수도 있으며, 같은 계열일 경우에는 형 변환은 수행되지 않고 저장된 값을 기준으로 검색이 수행됩니다. 때문에 실제 저장된 데이터를 기준으로 비교 작업이 수행되면 길이가 같은 컬럼만 비교가 진행될 수 있습니다.

```
SQL> DESCRIBE SYS.STANDARD                -- STANDARD 패키지의 Subprogram 확인

FUNCTION DECODE RETURNS NUMBER
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 EXPR                           VARCHAR2                IN
 PAT                            VARCHAR2                IN
 RES                            NUMBER                  IN
...
```

함수는 각 파라미터마다 데이터 타입이 정의되어 있습니다. 그리고 DECODE 함수는 CHAR 타입을 사용하는 파라미터는 존재하지 않습니다. 즉, 고정 길이로 처리되는 정의 사항이 없기 때문에 자동으로 공백을 붙여서 비교를 할 수 없습니다. 이는 저장된 값을 그대로 사용해야 한다는 것을 의미합니다. 그래서 앞선 예제에서 컬럼과 비교 값의 길이가 같은 "PRESIDENT" 만이 처리될 수 있었던 것입니다.

그렇다면 지금과 같은 상황에서 정상적인 결과가 검색될 수 있도록 하려면 어떻게 해야 할까요? 맞습니다. TRIM 함수를 이용하여 컬럼의 공백 문자를 제거하여 비교하면 됩니다. 고정 길이 문자를 갖는 컬럼에서 공백을 제거하면 손쉽게 문제를 해결할 수 있습니다.

```
SQL> SELECT deptno,
            NVL(SUM(DECODE(TRIM(job),'ANALYST',  sal)),0) AS ANALYST,
            NVL(SUM(DECODE(TRIM(job),'CLERK',    sal)),0) AS CLERK,
            NVL(SUM(DECODE(TRIM(job),'MANAGER',  sal)),0) AS MANAGER,
            NVL(SUM(DECODE(TRIM(job),'PRESIDENT',sal)),0) AS PRESIDENT,
            NVL(SUM(DECODE(TRIM(job),'SALESMAN', sal)),0) AS SALESMAN
     FROM emp
     GROUP BY deptno
     ORDER BY deptno ;
    DEPTNO    ANALYST      CLERK    MANAGER  PRESIDENT   SALESMAN
---------- ---------- ---------- ---------- ---------- ----------
        10          0       1300       2450       5000          0
        20       6000       1900       2975          0          0
        30          0        950       2850          0       5600

3 rows selected.

SQL> SELECT e.employee_id, e.last_name, e.job_id, j.job_title
     FROM employees e, jobs j
     WHERE e.job_id = TRIM(j.job_id)
       AND e.department_id IN (50,110) ;
EMPLOYEE_ID LAST_NAME       JOB_ID     JOB_TITLE
----------- --------------- ---------- ----------------------------------
        205 Higgins         AC_MGR     Accounting Manager
        206 Gietz           AC_ACCOUNT Public Accountant
        124 Mourgos         ST_MAN     Stock Manager
        144 Vargas          ST_CLERK   Stock Clerk
        143 Matos           ST_CLERK   Stock Clerk
        142 Davies          ST_CLERK   Stock Clerk
        141 Rajs            ST_CLERK   Stock Clerk

7 rows selected.
```

DECODE 경우는 CASE 표현식으로 변경하는 것도 해결 방법이 됩니다. CASE 표현식은 함수가 아닙니다. 그 의미는 비교 값, 탐색 값들의 데이터 타입이 고정되어 있지 않기 때문에 같은 계열의 타입이라면 각 자리의 타입을 그대로 이용하여 비교할 수 있습니다. 즉, 비조인 술어를 비교할 때처럼 고정 길이 문자는 비교 값의 길이를 맞춰서 비교하므로 TRIM 함수를 사용하지 않고도 원하는 결과를 생성할 수 있습니다.

```
SQL> SELECT deptno,
            NVL(SUM(CASE job WHEN 'ANALYST'   THEN sal END),0) AS ANALYST,
            NVL(SUM(CASE job WHEN 'CLERK'     THEN sal END),0) AS CLERK,
            NVL(SUM(CASE job WHEN 'MANAGER'   THEN sal END),0) AS MANAGER,
            NVL(SUM(CASE job WHEN 'PRESIDENT' THEN sal END),0) AS PRESIDENT,
            NVL(SUM(CASE job WHEN 'SALESMAN'  THEN sal END),0) AS SALESMAN
     FROM emp
     GROUP BY deptno
     ORDER BY deptno ;
    DEPTNO    ANALYST      CLERK    MANAGER  PRESIDENT   SALESMAN
---------- ---------- ---------- ---------- ---------- ----------
        10          0       1300       2450       5000          0
        20       6000       1900       2975          0          0
        30          0        950       2850          0       5600
3 rows selected.
```

문자형 데이터를 저장할 경우 CHAR 타입을 사용하면 어떤 문제가 발생할 수 있는지 확인하셨나요? 또한 그에 대한 해결 방법도? 하지만 위의 해결 방법 중 TRIM 함수를 이용하는 것은 상황에 따라 성능에 악영향을 미칠 수 있습니다. 결과를 위해 추가적인 함수를 사용해야 한다면 어쩔 수 없이 사용해야겠지만 조인 조건절 컬럼에 함수를 사용하게 되면 해당 컬럼의 인덱스를 이용하지 못 합니다. 이는 SQL 실행에 매우 안 좋은 영향을 미칠 수 있습니다. 설사 조건절에 사용된 컬럼이 아니라서 인덱스와는 상관이 없다 하더라도 추가적인 함수 호출 작업이 많아지면, 처리하고자 하는 데이터의 양에 따라 처리 속도는 느려질 수 있습니다.

때문에 가장 올바른 선택은 테이블 설계 시점에 각 컬럼의 데이터 타입을 올바르게 정의하는 것입니다. 조인이 걸릴 수 있는 컬럼은 동일한 타입과 동일한 크기 설정을 필수로 하고, 조건절 내에서 절대 추가적인 가공이 발생하지 않도록 설계하는 것이 우선되어야 합니다. 또한 불필요한 CHAR 타입 사용을 최소화하고 VARCHAR2 타입을 사용하는 것을 고려해 볼 수 있습니다. 만약 테이블 설계가 잘 됐다면 문자 컬럼에서 사용되는 불필요한 TRIM 함수 사용을 최소화하는 것도 필요합니다.

이미 현업에서는 CHAR 타입보다 VARCHAR2 타입을 많이 사용하고 있습니다. 하지만 튜닝을 위해 SQL을 조사하다 보면 TRIM 함수가 빈번하게 사용되는 것을 볼 수 있었습니다. 이미 VARCHAR2 타입을 사용하고 있는데도 말입니다. 원인을 확인해보면,

첫째, 과거 시스템의 SQL 문장을 재활용하기 때문입니다. 개발이 진행될 때 기존의 업무가 그대로 이어지거나 또는 비슷한 부분이 존재할 수 있습니다. 이때 많은 개발자들은 활용 가능한 SQL 문장을 "copy&paste"를 하게 됩니다. 테이블 설계는 새로 했어도 SQL 문장은 기존 시스템 또는 유사 업무에서 사용했던 SQL을 복사하여 필요한 부분만 살짝 수정하여 사용합니다. 이때 기존 SQL을 완벽하게 분석하여 재사용하는 것이 아니라면 TRIM과 같은 함수를 제거했을 때 데이터의 정합성이 훼손될까 염려되어 그대로 두는 경우가 종종 발생합니다. 때문에 과거 시스템에서는 CHAR 타입을 이용하면서 어쩔 수 없이 사용했던 TRIM 함수가 VARCHAR2 타입으로 변경되어 더 이상 필요가 없더라도 현재 문장에서는 계속 사용되는 경우가 있습니다.

둘째, VARCHAR2 타입을 이용하더라도 공백 문자를 갖는 데이터가 실제 저장되어 있기 때문입니다. 이는 앞선 상황처럼 프로젝트가 진행되면서 기존 CHAR 타입의 데이터가 이관되어 VARCHAR2 타입으로 저장될 때 제대로 된 정제 작업이(Data Cleansing) 진행되지 않았거나 사용자가 데이터를 입력할 때 공백 문자를 입력했을 때 발생할 수 있습니다. 실제 데이터로 공백 문자가 저장되어 있다면 VARCHAR2 컬럼이라도 TRIM 함수 사용은 필수가 될 수 있습니다.

첫 번째 상황은 TRIM 함수를 제거해서 결과가 틀리지 않다면 손쉽게 제거할 수 있으므로 큰 문제가 되지 않습니다. 문제는 두 번째 상황입니다. 이미 VARCHAR2 타입으로 저장된 데이터에 공백 문자를 갖고 있다면 데이터의 정제 작업을 수행하기 전까지는 어쩔 수 없이 TRIM 함수의 사용이 필요합니다.

데이터의 정제 작업이 가능하다면 VARCHAR2 타입 중 공백 문자를 갖는 행이 존재하는지를 확인하고, 공백을 제거하여 UPDATE를 수행하면 됩니다. 공백 문자가 없다면 TRIM 함수 호출 없이도 동일한 결과를 처리할 수 있어질 것입니다. 다만, 차후 테이블에 입력되는 데이터에 공백 문자가 다시 입력되지 않도록 INSERT 시점에 공백이 저장될 수 있는 컬럼은 프로그램단에서 공백을 미리 제거하거나, INSERT 문의 VALUES 절에서 TRIM 함수를 사용하여 공백이 저장되지 못하도록 정의합니다. 가능하다면 제약조건을 정의하여 공백이 저장되는 것을 원천적으로 막는 것도 좋은 방법이 될 수 있습니다.

▣ Data Cleansing 수행

```
SQL> ALTER TABLE emp
     MODIFY job VARCHAR2(9) ;                    -- 가변 길이 9byte로 변경
SQL> SELECT job, LENGTH(TRIM(job)), LENGTH(job)
     FROM emp
     WHERE TRIM(job) != job ;                    -- 공백 문자를 포함한 행 검색
JOB       LENGTH(TRIM(JOB)) LENGTH(JOB)
--------- ----------------- -----------
CLERK                     5           9
SALESMAN                  8           9
...
13 rows selected.

SQL> UPDATE emp
     SET job = TRIM(job)
     WHERE TRIM(job) != job ;                   -- 공백 문자를 절삭하여 UPDATE
13 rows updated.

SQL> SELECT job, LENGTH(TRIM(job)), LENGTH(job)
     FROM emp
     WHERE TRIM(job) != job ;                   -- UPDATE 결과 확인
no rows selected
```

▣ 제약 조건을 이용하여 공백 문자 입력 방지

```
SQL> ALTER TABLE emp
     ADD CONSTRAINTS emp_ck CHECK ( job = TRIM(job) ) ;   -- 제약 조건 정의
SQL> UPDATE emp
      SET job = 'MANAGER  '
      WHERE empno = 7566 ;                      -- 공백 문자 입력 방지 가능
ERROR at line 1:
ORA-02290: check constraint (TEST.EMP_CK) violated
```

▣ INSERT 시 공백 문자를 제거하여 입력

```
SQL> VARIABLE job CHAR(9)
SQL> EXECUTE :job := 'CLERK    ' ;
SQL> SELECT LENGTH(:job) FROM dual ;               -- 공백 문자 포함 확인
LENGTH(:JOB)
------------
           9
1 row selected.
```

```
SQL> INSERT INTO emp (empno, ename, job)
     VALUES (1234, 'TEST', TRIM(:job)) ;            -- 공백 문자 절삭 후 입력
1 row created.

SQL> SELECT job, LENGTH(TRIM(job)), LENGTH(job)
     FROM emp
     WHERE empno = 1234 ;                           -- 입력된 결과 확인
JOB       LENGTH(TRIM(JOB)) LENGTH(JOB)
--------- ----------------- -----------
CLERK                     5           5
1 row selected.

SQL> ROLLBACK ;
```

입력되는 문자가 서로 다른 길이를 가질 수 있는 상황에서는 VARCHAR2 타입을 사용하도록 설계합니다. VARCHAR2 타입을 사용할 경우 저장된 공백 문자의 존재를 확인하여 불필요한 TRIM 함수 사용은 제거하고, 공백이 존재할 때는 정제 작업 및 제약 조건을 이용하여 추가 공백 문자의 입력을 제한하면 TRIM 함수 사용은 불필요할 것입니다. CHAR 타입을 사용한다면 공백 문자가 존재할 경우 TRIM 함수를 이용해야 결과를 보장할 수 있으며, 역시 공백 문자가 없는 경우에는 불필요한 TRIM 함수 사용을 최소화하는 것이 성능상 좀 더 유리할 수 있습니다.

1.1.2. NUMBER 타입

그동안 만나본 많은 사용자들이 NUMBER 타입을 지정할 때 (p, s) 의미를 전체 자릿수(p), 소수점 자릿수(s)로 이해하고 있었습니다. 그렇다면 다음의 명령문을 실행하고자 할 때 각 컬럼의 저장 가능한 최대, 최솟값은 무엇이며, 에러가 발생할 수 있는 정의는 무엇일까요?

```
SQL> CREATE TABLE num_typ                                    -- 실습 테이블 생성
     (c1     NUMBER,
      c2     NUMBER(4,2),
      c3     NUMBER(4),          -- or NUMBER(4,0)
      c4     NUMBER(2,4),
      c5     NUMBER(4,-1) ) ;
```

(p, s)를 전체 자릿수, 소수점 자릿수로 이해하고 있을 때 문제가 되는 부분은 C4 컬럼일 것입니다. "전체 자릿수는 2자리이고 소수점 자릿수는 4자리?" 정상적으로 생성되지 않을 것처럼 느껴지시나요? 하지만 위의 명령문을 실행하면 에러 없이 테이블은 생성됩니다. 즉, precision은 전체 자릿수를 의미하지 않습니다.

precision은 전체 자릿수가 아닌, 사용자 정의로 표현할 수 있는 자릿수로 보는 것이 보다 정확할 수 있습니다. 이해가 되셨나요? 어렵다고요? 다음의 설명을 확인하시면 보다 쉽게 이해할 수 있을 것입니다.

실수형 데이터는 크게 부동형 소수점 방식과 고정형 소수점 방식으로 나눌 수 있습니다. C1 컬럼의 정의처럼 사이즈를 제한하지 않은 컬럼은 부동형 소수점 방식으로 오라클에서 NUMBER 타입으로 저장할 수 있는 모든 표현 방식을 저장할 수 있습니다. 즉, 상황에 따라 원하는 자리의 소수점 데이터 및 정수형 등의 모든 데이터를 입력할 수 있습니다. 하지만 나머지 C2~C5 컬럼은 모두 크기를 제한합니다. 이때는 precision을 먼저 보는 것이 아니라 scale 값을 먼저 읽어야 합니다. 왜냐고요? 고정형 소수점 방식으로 NUMBER 타입을 저장하겠다는 의미이기 때문입니다. 고정형 소수점 방식에서는 소수점 몇 자리를 사용할 것인지가 정의됩니다.

C2 컬럼은 2자리, C3 컬럼은 소수점 자리를 사용하지 않는 정수형 데이터, C4 컬럼은 4자리, C5 컬럼은 정수 자리로 한자리 올라오라는 의미를 갖습니다. 그렇게 소수점 자릿수가 결정되면 precision 값을 확인하여 지정된 소수점 자리를 만족하며 사용자 정의의 값을 몇 자리를 사용할 수 있는지 정의합니다.

또한 해당 소수점 자리까지의 반올림을 합니다. 이렇게 고정형 소수점 방식을 이용할 때는 각 컬럼의 최소, 최댓값이 결정되게 됩니다. 표로 정리하면 다음과 같습니다.

정의	규칙	설명	범위
NUMBER(4,2)	p > s	소수점 자리를 포함한 데이터	±0.01 ~ 99.99
NUMBER(4)	s = 0	정수형 데이터	± 0 ~ 9999
NUMBER(2,4)	p <= s	실수형 데이터	± 0.0001 ~ 0.0099
NUMBER(4,-1)	s < 0	정수부의 지정 자릿수를 0 사용	± 10 ~ 99990

참고. precision 의 범위는 1~38, scale 의 범위는 -84~127 입니다.

NUMBER 타입을 사용할 경우 컬럼 값의 범위를 제한해야 할 경우에는 고정형 소수점 방식을 이용하고, 표현할 수 있는 모든 수를 처리하고자 할 때는 사이즈 제한 없는 부동형 소수점 방식을 사용하면 됩니다. 고정형 소수점 방식을 처리할 경우에도 실수형 데이터만 저장을 원하는 경우에는 p<=s 공식으로 크기를 제한합니다. NUMBER(4,4)의 설정은 0.9999 보다 큰 숫자를 저장하지 못합니다. scale 값을 음수로 사용하는 경우는 약간은 특수한 상황일 것입니다. 지정된 자리까지 정수부를 "0"으로 고정할 수 있습니다. 이렇듯 별도의 제약 조건을 이용하지 않아도 컬럼의 데이터 타입을 어떻게 정의했느냐에 따라 값의 범위를 제한 할 수 있습니다.

```
SQL> INSERT INTO num_typ (c2, c3, c4, c5) VALUES (0,0,0,0) ;
1 row created.

SQL> INSERT INTO num_typ (c2) VALUES (12.34567) ;
1 row created.

SQL> INSERT INTO num_typ (c2) VALUES (99.999) ;
ERROR at line 1:
ORA-01438: value larger than specified precision allowed for this column

SQL> SELECT c2 FROM num_typ ;
        C2
----------
         0
     12.35
```

모든 NUMBER 타입은 "0"의 저장을 허용하며, 두 번째 INSERT 문처럼 소수점 자리가 SIZE보다 길게 입력되면 지정된 소수점 자리까지 반올림을 진행하여 입력합니다. 단, 최댓값을 넘어가는 경우가 발생하면 에러가 발생합니다.

NUMBER 타입을 좀 더 자세히 이해하셨나요? 일반적으로 NUMBER 타입은 실수형 데이터만을 이용하기 위해 사용하는 경우는 거의 없습니다. (필요하다면 실수형 데이터만을 저장할 수 있는 FLOAT 타입이나, Oracle 10g DB부터 지원하는 BINARY_FLOAT, BINARY_DOUBLE 타입을 사용할 수도 있습니다.) 때문에 위와 같은 구분을 정확히 이해하지 못해도 사용상 어려움이 없었을 것입니다. 하지만 데이터 타입의 특성을 정확히 이해하고 있다면 제약 조건 없이도 값의 범위를 제한할 수 있고 유효한 데이터만을 저장할 수도 있으니 앞으로 NUMBER 타입이 필요할 때는 참고하시기 바랍니다.

1.1.3. DATE 타입

날짜 데이터를 저장할 때 사용되는 타입이 DATE입니다. DATE 타입은 고정 길이 형식으로 항상 7byte의 저장 공간을 사용합니다. 실제 날짜 데이터가 저장될 때는 숫자 형식으로 세기, 년, 월, 일, 시, 분, 초까지 저장하며 NLS_DATE_FORMAT 형식에 따라 출력 형식이 정의됩니다. 간단한 예제를 통해 확인해 보겠습니다.

```
SQL> CREATE TABLE dt_typ                                    -- 실습 테이블 생성
      (c1      DATE) ;
SQL> INSERT INTO dt_typ VALUES ( SYSDATE ) ;

SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY/MM/DD HH24:MI:SS' ;
SQL> SELECT * FROM dt_typ ;          -- 'YYYY/MM/DD HH24:MI:SS' 포맷으로 출력
C1
-------------------
2016/04/06 06:43:33

SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY/MM/DD' ;
SQL> SELECT *
     FROM dt_typ ;         -- 'YYYY/MM/DD' 포맷으로 출력
C1
----------
2016/04/06

SQL> SELECT DUMP(c1)
     FROM dt_typ ;
DUMP(C1)
----------------------------------------
Typ=12 Len=7: 120,116,4,6,7,44,34
```

참고. DUMP 함수의 마지막 3 자리는 시, 분, 초를 의미하며, 각 자리에 저장된 값에서 -1 을 수행하면 실제 시간 값이 저장된 것을 확인할 수 있습니다.

DUMP 함수 결과가 중요한 것은 아닙니다. 다만, DATE 컬럼에 값이 저장될 때는 시, 분, 초까지 시간 정보가 함께 저장되고 NLS_DATE_FORMAT 설정을 어떻게 지정했느냐에 따라 시간 정보는 출력되지 않을 수 있습니다. 초급 사용자들이 가장 혼동하는 것 중 하나가 이점입니다.

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY/MM/DD' ;
SQL> SELECT *
     FROM dt_typ
     WHERE c1 = TO_DATE('2016/04/06','YYYY/MM/DD') ;
no rows selected
SQL> SELECT *
     FROM dt_typ
     WHERE c1 = TO_DATE('2016/04/06 06:43:33','YYYY/MM/DD HH24:MI:SS');
C1
----------
2016/04/06
```

실제 저장된 값은 시간을 포함하여 저장되었지만 출력 포맷이 날짜만 출력하도록 설정하였습니다. 때문에 첫 번째 조건식처럼 WHERE 절을 작성하였다면 검색된 결과는 존재할 수 없습니다. 실제로는 시간 정보가 저장되어 있기 때문에 모든 요소가 일치해야만 검색 결과가 보장됩니다. 현재는 검색돼야 할 행이 있다는 사실을 알기 때문에 두 번째 문장처럼 저장된 시간 정보를 정확히 비교하여 검색할 수도 있습니다. 하지만 실제 업무에서는 위와 같은 방법이 결코 쉽지 않습니다. 많은 데이터가 저장된 테이블에서는 1~2개 행이 검색되지 않은 것은 쉽게 발견되지 않습니다. 때문에 일부 누락된 데이터가 발생하여 업무에 지장을 초래하는 경우도 발생합니다. 이러한 상황을 피하기 위해 시간 정보가 저장된 컬럼에서는 다음과 같은 조건식을 사용하기도 합니다.

```
SQL> SELECT * FROM dt_typ
     WHERE TO_CHAR(c1, 'YYYY/MM/DD') = '2016/04/06' ;
C1
----------
2016/04/06
1 row selected.
SQL> SELECT * FROM dt_typ
     WHERE c1 BETWEEN TO_DATE('2016/04/06 00:00:00','YYYY/MM/DD HH24:MI:SS')
                  AND        TO_DATE('2016/04/06        23:59:59','YYYY/MM/DD
HH24:MI:SS') ;
C1
----------
2016/04/06
1 row selected.
```

시간 정보를 갖는 컬럼에서 특정 일자 값을 비교할 때, 날짜 부분만 추출하여 비교를 하거나 BETWEEN 조건식을 이용하면 원하는 검색 결과를 확인할 수 있습니다. 이때 차라리 BETWEEN 조건식을 이용하면 성능 이슈는 줄어들 수 있는데, 조건식이 길어진다는 이유로 컬럼을 가공하는 경우가 종

종 있습니다. 이는 해당 컬럼의 인덱스를 사용하지 못하면서 성능 저하의 원인이 되기도 합니다.

시간 정보 없이 날짜 값만 필요할 때 근본적인 해결 방법은 "YYYY/MM/DD 12:00:00 AM" 값으로 저장하는 것입니다. 하지만 SYSDATE 함수를 잘못 사용하여 시간 정보를 입력했다면 BETWEEN 조건을 사용하는 것이 인덱스를 사용할 수 있는 조건절이 될 수 있습니다. 이러한 번거로움(?)을 피하고자 실제 업무에서는 DATE가 아닌 CHAR 또는 VARCHAR2 타입으로 "YYYYMMDD"까지만 저장하는 경우도 있습니다. 문자 타입으로 저장되는 게 좋다는 의미는 아닙니다. 날짜를 문자 타입으로 저장했을 때의 주의사항은 조금 뒤에 확인하고, 우선은 위의 예제와 같은 상황에서 불필요한 시간 정보의 입력은 제약 조건을 통해 방지하고, 잘못된 데이터는 Data Cleansing을 통해 시간 정보를 절삭합니다. 이때는 TRUNC 함수가 유용하게 사용됩니다.

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY/MM/DD HH24:MI:SS' ;
SQL> UPDATE dt_typ
     SET c1 = TRUNC(c1) ;                              -- Data Cleansing
1 row updated.
SQL> SELECT c1 FROM dt_typ ;
C1
-------------------
2016/04/06 00:00:00
1 row selected.

SQL> ALTER TABLE dt_typ                        -- 제약 조건을 통해 시간 입력 방지
     ADD CONSTRAINT dt_typ_ck CHECK ( c1 = TRUNC(c1) ) ;
SQL> INSERT INTO dt_typ VALUES ( SYSDATE ) ;          -- 시간 정보 입력 불가능
ERROR at line 1:
ORA-02290: check constraint (TEST.DT_TYP_CK) violated

SQL> INSERT INTO dt_typ VALUES ( TRUNC(SYSDATE) ) ;       -- 입력 시 시간 절삭
1 row created.
SQL> SELECT * FROM dt_typ ;
C1
-------------------
2016/04/06 00:00:00
2016/04/06 00:00:00

SQL> SELECT c1 FROM dt_typ                          -- 날짜 비교만으로 검색 가능
     WHERE c1 = TO_DATE('2016/04/06','YYYY/MM/DD') ;
C1
-------------------
2016/04/06 00:00:00
2016/04/06 00:00:00
```

어떤가요? 시간 정보가 자정 시각으로 저장되어 있다면 BETWEEN을 이용하지 않고도 검색이 가능한 것을 확인할 수 있습니다. 또한 제약 조건을 정의하여 불필요한 시간 정보의 입력도 방지할 수 있습니다.

1.2. 데이터 타입 선택

앞선 내용을 통해 기본적인 데이터 타입의 저장 방식과 주의 사항을 확인하셨습니다. 이제는 데이터 타입을 결정할 때 반드시 이해하고 있어야 하는 주의 사항을 확인하겠습니다.

1.2.1. DATE vs CHAR, VARCHAR2

업무상 날짜, 시간 정보를 저장해야 할 때 DATE를 사용하시나요? 제가 경험했던 많은 사이트에서는 문자 타입을 이용하여 날짜를 관리하는 경우가 많았습니다. 단, 시간 정보를 함께 관리해야 하는 경우는 DATE 또는 TIMESTAMP 타입을 사용합니다. 만약 문자 타입을 이용하여 "YYYYMMDDHHMISS"까지 값을 저장하려면 14byte가 필요하기 때문에 고정 길이 7byte로 시, 분, 초까지를 저장할 수 있는 DATE 타입을 사용하는 것은 저장 공간의 낭비를 줄이기 위해서라도 대부분 선택되는 사항입니다.

문제는 날짜만 저장해야 할 때 시간 정보가 함께 관리되는 DATE보다 문자 타입이 좀 더 사용이 편할 수 있고, LIKE와 같은 조건식에서 인덱스를 사용할 수 있기 때문에 문자 타입을 선호하는 경향이 있습니다. 하지만 날짜를 문자로 저장하면 사용의 편의성은 증대될 수 있어도 또 다른 문제가 발생할 수 있습니다.

첫째, 문자 타입으로 관리되는 컬럼에는 잘못된 날짜가 저장될 수 있습니다.

```
SQL> DESCRIBE dtvschar
 Name              Null?    Type
 ----------------- -------- ------------
 NO                         NUMBER
 C1                         DATE
 C2                         VARCHAR2(8)              -- 가변 길이 문자

SQL> SELECT * FROM dtvschar ;
        NO C1         C2
---------- ---------- --------
         1 2015/01/01 20150101                       -- '2015/01/01' 부터
...                                                     '2016/12/31' 까지 저장
      7310 2016/12/31 20161231

7310 rows selected.

SQL> SELECT * FROM dtvschar
     WHERE no IN (590, 1510, 2730, 3040, 4560) ;
        NO C1         C2
---------- ---------- --------
       590 2015/02/28 20150229                     -- C2 컬럼에는 유효하지
      1510 2015/05/31 20150532                        않은 날짜 저장되어 있음
      2730 2015/09/30 20150931
      3040 2015/10/31 20151032
      4560 2016/03/31 20160332

5 rows selected.
```

참고. DTVSCHAR 테이블은 미리 생성된 실습 테이블입니다.

실습 결과를 통해 확인 가능하듯 문자 타입의 컬럼에는 존재할 수 없는 날짜 값이 저장될 수 있습니다. 물론 입력 시점에 유효성을 체크하고 저장할 수도 있으나, 제약 조건이 정의되지 않는 이상 문자 타입의 컬럼에는 잘못된 날짜가 존재할 수 있다는 가정은 항상 성립됩니다.

둘째, 실행 계획 생성 시 정확한 비용 측정이 불가능할 수 있습니다.

```
SQL> SELECT COUNT(*) FROM dtvschar
     WHERE c1 BETWEEN TO_DATE('2016/04/30','YYYY/MM/DD')
                  AND TO_DATE('2016/05/01','YYYY/MM/DD') ;
  COUNT(*)
----------
        20

SQL> EXECUTE DBMS_STATS.GATHER_TABLE_STATS('TEST','DTVSCHAR', -
                                  method_opt=>'for all columns size 1') ;
PL/SQL procedure successfully completed.
```

조건에 만족하는 행은 20이고, 최적의 실행 계획 수립을 위해 옵티마이저 통계를 새로 수집하였습니다. C1, C2 컬럼은 동일한 날짜 값을 저장하였기 때문에 조건에 만족하는 행의 개수는 동일합니다. 그렇다면 실제 예상되는 행의 개수는 어떨까요?

```
SQL> EXPLAIN PLAN FOR
     SELECT * FROM dtvschar
     WHERE c1 BETWEEN TO_DATE('2016/04/30','YYYY/MM/DD')
                  AND TO_DATE('2016/05/01','YYYY/MM/DD') ;
Explained.
SQL> SELECT * FROM table(dbms_xplan.display(null,null,'typical -cost')) ;
-------------------------------------------------------------------------------
| Id  | Operation                   | Name       | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |    30 |   630 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DTVSCHAR   |    30 |   630 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | DTVSCH_IX1 |    30 |       | 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C1">=TO_DATE(' 2016-04-30 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
         AND "C1"<=TO_DATE(' 2016-05-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

SQL> EXPLAIN PLAN FOR
     SELECT * FROM dtvschar
     WHERE c2 BETWEEN '20160430' AND '20160501' ;
Explained.




SQL> SELECT * FROM table(dbms_xplan.display(null,null,'typical -cost')) ;
```

```
-------------------------------------------------------------------------------
| Id  | Operation                   | Name       | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |    66 |  1386 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DTVSCHAR   |    66 |  1386 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | DTVSCH_IX2 |    66 |       | 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C2">='20160430' AND "C2"<='20160501')
```

C1 컬럼(DATE)에서는 예상되는 Rows 개수가 30이며, C2 컬럼(VARCHAR2)에서는 66개인 것을 확인할 수 있습니다. 조건에 만족하는 행은 20개이므로 둘 모두 정확한 예측 결과를 보여준 것은 아닙니다. 하지만 예상되는 행의 개수가 실제 상황에 근접한 컬럼은 C1 컬럼인 것을 확인할 수 있습니다. 조건 범위를 좀 더 넓혀보면 어떨까요?

```
SQL> SELECT COUNT(*) FROM dtvschar
     WHERE c1 BETWEEN TO_DATE('2016/05/01','YYYY/MM/DD')
                  AND TO_DATE('2016/05/31','YYYY/MM/DD') ;
  COUNT(*)                                -- 조건에 만족하는 행의 개수
----------
       310

SQL> EXPLAIN PLAN FOR
     SELECT * FROM dtvschar
     WHERE c1 BETWEEN TO_DATE('2016/05/01','YYYY/MM/DD')
                  AND TO_DATE('2016/05/31','YYYY/MM/DD') ;
Explained.

SQL> SELECT * FROM table(dbms_xplan.display(null,null,'typical -cost')) ;
-------------------------------------------------------------------------------
| Id  | Operation                   | Name       | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |   320 |  6720 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DTVSCHAR   |   320 |  6720 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | DTVSCH_IX1 |   320 |       | 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C1">=TO_DATE(' 2016-05-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')
         AND "C1"<=TO_DATE(' 2016-05-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))

SQL> EXPLAIN PLAN FOR
     SELECT * FROM dtvschar
     WHERE c2 BETWEEN '20160501' AND '20160531' ;
Explained.
SQL> SELECT * FROM table(dbms_xplan.display(null,null,'typical -cost')) ;
```

```
-------------------------------------------------------------------------------
| Id  | Operation                   | Name       | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |            |    40 |   840 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| DTVSCHAR   |    40 |   840 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | DTVSCH_IX2 |    40 |       | 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C2">='20160501' AND "C2"<='20160531')
```

실제 행의 개수인 310개에 보다 근접한 결과를 보여주는 것은 역시 C1(DATE) 컬럼입니다. VARCHAR2 타입으로 관리되는 C2 컬럼은 실제 상황을 정확하게 예측하지 못하고 있습니다. 이는 문자 타입에서는 실제 날짜 값의 관계를 정확히 파악할 수 없기 때문입니다. 이렇듯 조건에 만족하는 예상 행의 개수가 틀리면 인덱스 선택이 잘못되어 오히려 성능이 저하되는 경우도 발생할 수 있습니다. 물론 히스토그램을 수집하면 문자 타입에서도 실제 조건에 만족하는 범위 값을 좀 더 정확히 예측할 수도 있습니다. 하지만 바인드 변수를 이용한 Static SQL 형식을 사용하고 있다면 히스토그램을 활용하지 못할 수도 있기 때문에 히스토그램 수집이 근본적인 해결책이 될 수는 없습니다.

날짜 형식 값을 문자 타입으로 저장하고 있다면 유효하지 않은 날짜를 저장할 수 있습니다. 이는 데이터 품질 관리에 매우 치명적인 악영향을 미칠 수 있습니다. 추가적인 Data Cleansing 작업을 주기적으로 수행해야 할 수도 있으며, 제약 조건을 사용하여 잘못된 날짜 값의 입력을 방지해야 할 수도 있습니다.

```
SQL> DROP TABLE t1 PURGE ;
SQL> CREATE TABLE t1 (c1 VARCHAR2(8)) ;

▣ 제약 조건을 이용하여 유효한 값만 입력 허용
SQL> ALTER TABLE t1
     ADD CHECK (c1=TO_CHAR(TO_DATE(c1,'YYYYMMDD'),'YYYYMMDD'));

SQL> INSERT INTO t1 VALUES ('20160431') ;          -- 유효하지 않은 날짜
ERROR at line 1:
ORA-01839: date not valid for month specified

SQL> INSERT INTO t1 VALUES ('20160430') ;          -- 유효한 날짜 입력 가능
1 row created.
```

위와 같은 경우 DATE 타입을 이용했다면 제약 조건을 이용하지 않고도 '20160431'의 잘못된 입력을 막을 수 있습니다. 하지만 문자 타입에서는 모든 형식의 문자가 입력 가능하기 때문에 추가적인 검증

작업을 사용해야 합니다. 설사 위와 같이 유효한 날짜만 저장 가능하도록 제약 조건을 사용했다 하더라도 정확한 예상 행의 개수를 예측하지 못한다면 성능상 유리하지 않습니다.

DATE 타입은 고정 길이 7byte로 세기, 년, 월, 일, 시, 분, 초까지 저장합니다. 때문에 시, 분, 초 값이 필요할 때는 문자 타입보다 DATE 타입이 저장 공간 크기를 줄일 수 있습니다. 일자(YYYY/MM/DD) 값만 필요하더라도 DATE 타입이 문자 타입보다 더 적은 공간을 사용할 수 있고, 데이터 품질 관리 및 성능상 유리합니다. 단지 사용자 편의성을 위해 문자 타입을 사용하는 것은 올바른 선택이 아닙니다. 그리고 문자 타입으로 저장된다고 항상 편의성이 증대되는 것도 아닙니다. 만약, 날짜 연산을 수행해야 하는 경우가 있다면 TO_DATE 함수를 추가적으로 사용해야 원하는 결과를 만들 수 있기 때문입니다.

```
SQL> SELECT SYSDATE - c1
     FROM dtvschar
     WHERE no = 1 ;                                    -- 직접 연산 가능
SYSDATE-C1
----------
461.552211

SQL> SELECT SYSDATE - c2
     FROM dtvschar
     WHERE no = 1 ;                                    -- 연산 불가능
ERROR at line 1:
ORA-01841: (full) year must be between -4713 and +9999, and not be 0

SQL> SELECT SYSDATE - TO_DATE(c2, 'YYYYMMDD')
     FROM dtvschar
     WHERE no = 1 ;                          -- TO_DATE 함수 사용 후 연산 가능
SYSDATE-TO_DATE(C2,'YYYYMMDD')
------------------------------
                   461.552558
```

그렇다면 날짜를 DATE가 아닌 문자 타입(또는 NUMBER)을 사용해야 하는 경우는 언제일까요?

```
SQL> SELECT COUNT(*) FROM dtvschar
     WHERE c1 BETWEEN TO_DATE('2016/01/01','YYYY/MM/DD')
                  AND TO_DATE('2016/12/31','YYYY/MM/DD') ;
  COUNT(*)
----------
      3660

SQL> SELECT COUNT(*) FROM dtvschar
     WHERE TO_CHAR(c1, 'MM') = '01' ;
  COUNT(*)
----------
       620
```

날짜 요소 전체를 비교할 경우에는 문자보다 DATE 타입이 유리합니다. 또한 BETWEEN 조건식을 이용 가능하다면 TO_CHAR와 같은 함수를 사용하지 않고도 원하는 결과를 조회할 수 있습니다. 다만, 두 번째 조건처럼 특정 월에 해당되는 값을 조회하려면 함수 사용은 필수가 되고, 이는 인덱스를 사용하지 못하는 경우가 됩니다. Function Based Index를 이용할 수도 있지만 다양한 패턴의 부분적인 날짜 요소 값을 비교하고 있다면 차라리 여러 개의 컬럼으로 분리하여 저장하는 것이 좋을 수도 있습니다.

```
SQL> ALTER TABLE dtvschar ADD c3  VARCHAR2(2) ;
SQL> UPDATE dtvschar SET c3 = TO_CHAR(c1, 'MM')  ;
7310 rows updated.
SQL> SELECT COUNT(*) FROM dtvschar WHERE c3 = '01' ;
  COUNT(*)
----------
       620
```

문장이 실행될 때 인덱스 사용이 반드시 필요한 것은 아닙니다. 다만, 인덱스를 이용할 수 있는 문장을 생성하고자 한다면 위와 같이 날짜 요소를 분리하여 저장하는 것도 고려할 수 있으며, 조건에 만족하는 범위가 넓어 어차피 인덱스 사용이 불필요하다면 DATE 타입 컬럼에서 TO_CHAR 함수를 이용한 조건을 사용해도 됩니다.

이제 DATE와 CHAR, VARCHAR2 선택이 가능하신가요? 모든 사항에 항상 동일한 정답이란 없습니다. 가급적 DATE 타입을 이용하는 것을 권장하지만 경우에 따라서는 다른 타입을 이용해야 할 수도 있습니다. 다만, 어떠한 타입을 사용하느냐에 따라 데이터 품질 및 관리 포인트를 잘 파악하고 사용해야 합니다.

1.2.2. NUMBER vs CHAR, VARCHAR2

NUMBER 타입과 문자 타입을 선택하는 것은 DATE 타입의 고민보다는 상대적으로 쉽습니다. 누가 봐도 숫자가 저장되어야 하는 컬럼을 문자 타입으로 저장하지는 않으니까요. 다만 계산식에 사용되지 않으며 숫자 형식을 갖는 경우라면 문자 타입을 이용할 수도 있습니다. 예를 든다면 Primary Key 값으로 저장되는 EMP 테이블의 EMPNO 컬럼 같은 경우입니다.

```
SQL> DROP TABLE emp2 PURGE ;
SQL> CREATE TABLE emp2
     (empno    VARCHAR2(5),                          -- 가변 길이 5byte
      ename    VARCHAR2(10)) ;
Table created.

SQL> INSERT INTO emp2 SELECT empno, ename FROM emp ;
14 rows created.

SQL> CREATE INDEX emp2_ix ON emp2(empno) ;
Index created.

SQL> SELECT * FROM emp2 ;
EMPNO ENAME
----- ----------
7369  SMITH
7499  ALLEN
7521  WARD
7566  JONES
7654  MARTIN
7698  BLAKE
...
14 rows selected.
```

EMPNO 컬럼을 이용하여 계산을 할 경우는 없습니다. 때문에 NUMBER 타입이 아닌 문자 타입을 이용하는 것이 가능합니다. 하지만 이 또한 성능상 이슈가 발생할 수 있습니다.

```
SQL> EXPLAIN PLAN FOR
     SELECT * FROM emp2 WHERE empno = 7788 ;
Explained.

SQL> SELECT * FROM table(dbms_xplan.display(null,null,'TYPICAL -cost')) ;
------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Time     |
------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    11 | 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP2 |     1 |    11 | 00:00:01 |
------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TO_NUMBER("EMPNO")=7788)
```

"Predicate Information"을 확인하면 조건식에 직접 입력하지 않은 TO_NUMBER 함수가 사용되는 것을 확인할 수 있습니다. 이는 암시적인 형 변환 작업이 수행된 것을 의미합니다. 때문에 해당 컬럼에 인덱스가 존재하더라도 컬럼 가공으로 인해 인덱스 사용은 불가능합니다. 물론 데이터 타입을 동일하게 비교하면 암시적 형 변환 작업은 일어나지 않습니다.

```
SQL> EXPLAIN PLAN FOR
     SELECT * FROM emp2 WHERE empno = '7788' ;
Explained.

SQL> SELECT * FROM table(dbms_xplan.display(null,null,'TYPICAL -cost')) ;
-------------------------------------------------------------------------
| Id  | Operation                   | Name    | Rows  | Bytes | Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |     1 |    11 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP2    |     1 |    11 | 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | EMP2_IX |     1 |       | 00:00:01 |
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"='7788')
```

하지만 호출 환경에서 실행 계획을 확인하면서 작업하지 않으면 간혹 데이터 타입을 동일하게 사용하지 않아 위와 같이 실행 계획에 영향을 줄 수도 있습니다. 만약 EMP 테이블에서 동일 상황이 발생하면 어떨까요?

```
SQL> EXPLAIN PLAN FOR
     SELECT * FROM emp WHERE empno = '7788' ;
Explained.

SQL> SELECT * FROM table(dbms_xplan.display(null,null,'TYPICAL -cost')) ;
-------------------------------------------------------------------------------
| Id  | Operation                   | Name         | Rows  | Bytes | Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |     1 |    39 | 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP          |     1 |    39 | 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | EMP_EMPNO_IX |     1 |       | 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=7788)
```

문자 타입으로 비교했지만 실제 사용된 조건은 숫자 형식으로 바뀐 것을 확인할 수 있습니다. 암시적인 형 변환은 문자를 숫자로 바꾸려고 합니다. 때문에 경우에 따라 컬럼이 가공될 수도 있고, 리터럴 값이 변경될 수도 있습니다.

그리고 문자 타입으로 관리되는 EMPNO에서는 다음과 같이 정렬 상태의 이슈가 발생할 수도 있습니다.

```
SQL> SELECT MIN(empno) AS MIN, MAX(empno) AS MAX FROM emp2 ;
MIN   MAX
----- -----
7369  7934

SQL> INSERT INTO emp2 VALUES ('12345', 'RYU') ;
1 row created.

SQL> SELECT MIN(empno) AS MIN, MAX(empno) AS MAX FROM emp2 ;
MIN   MAX                        -- 5 자리의 '12345'가 가장 작은 값
----- -----
12345 7934

SQL> SELECT * FROM emp2 ORDER BY empno ;
EMPNO ENAME
----- ----------
12345 RYU                       -- NUMBER 타입에서는 MAX 값이지만
7369  SMITH                        문자 타입에서는 MIN 값이다.
7499  ALLEN
7521  WARD
...
7900  JAMES
7902  FORD
7934  MILLER
15 rows selected.
```

문자 타입에서는 각 자리마다 저장된 값을 이용하여 Binary Sorting을 수행하므로 전체 자릿수가 다를 경우 정렬 상태가 어긋날 수 있습니다. 때문에 숫자 타입에서는 가장 큰 값이라도 문자 타입에서는 가장 작은 값이 될 수도 있습니다. 문자 타입으로 숫자 형식을 저장하고 있고, 경우에 따라 정렬 작업이나 MIN, MAX 함수 결과가 필요하다면 선행 자리에 '0'를 채워 자릿수를 일치 시켜야 합니다.

```
SQL> UPDATE emp2
     SET empno = LPAD(empno, 5, '0') ;
15 rows updated.

SQL> SELECT MIN(empno) AS MIN, MAX(empno) AS MAX
     FROM emp2 ;
MIN   MAX
----- -----
07369 12345

SQL> SELECT *
     FROM emp2
     ORDER BY empno ;
EMPNO ENAME
----- ----------
07369 SMITH
07499 ALLEN
07521 WARD
07566 JONES
07654 MARTIN
07698 BLAKE
07782 CLARK
07788 SCOTT
07839 KING
07844 TURNER
07876 ADAMS
07900 JAMES
07902 FORD
07934 MILLER
12345 RYU

15 rows selected.
```

계산식에 사용되는 컬럼은 NUMBER 타입을 사용하고, 그렇지 않은 경우는 경우에 따라 필요한 타입을 선택하면 됩니다. 무조건 NUMBER 타입을 사용해야 한다고 할 수는 없습니다. LIKE와 같은 조건식이 필요할 수도 있기 때문에 오히려 문자 타입으로 저장되는 것이 좋을 수도 있습니다. 하지만 문자 타입으로 숫자 형식을 저장한다면 암시적 형 변환에 의한 성능 저하가 발생하지 않도록 하고, 정렬과 관련된 이슈가 발생하지 않도록 추가적인 관리가 필요합니다.