```
-- 'CREATE TABLE' DDL을 통해 특정 테이블을 생성할 수 있다.
-- '()'사이에 컬럼을 쓰고, 마지막은 세미콜론으로 끝난다.
CREATE TABLE EMP
(
empno number(10) primary key,
ename varchar2(20),
sal number(6)
);
```

```
-- 제약조건으로 PRIMARY KEY를 설정(해당 방법이 정석이다.)
-- "CONSTRAINT 제약조건명 PRIMARY KEY(기본키 칼럼명)"
CREATE TABLE EMP
(
empno number(10),
ename varchar2(20),
sal number(10,2) default 0,
deptno varchar2(4) not null,
createdatae date default sysdate,
constraint emppk primary key(empno)
);
```

```
-- DEPT 테이블 생성
CREATE TABLE DEPT
(
deptno varchar(2) primary key,
deptname varchar2(20)
);

-- 'ALTER'로 EMP테이블에 외래키 설정
-- "CONSTRAINT 제약조건명 FOREIGN KEY(외래키 칼럼명) REFERENCES 참조테이블(참조테이블_PK)"
-- 'REFERENCES'에서 'S'붙는 것 명심
ALTER TABLE EMP ADD CONSTRAINTS deptfk foreign key (deptno) references DEPT (deptno);
```

-- FOREGIN KEY CONSTRAINT에서 'ON DELETE CASCADE' 설정하기
-- 'ON DELETE CASCADE'옵션은 자신이 참조하고 있는 테이블의 데이터가 삭제되면, 테이블 내 해당 데이터와 관련있는 인스턴스도 삭제되는 옵션이다.
-- 해당 옵션을 사용하면, '참조 무결성'을 준수할 수 있다. 참조 무결성이란, 마스터 테이블에는 해당 부서번호가 없는데, 슬레이브 테이블에는 해당 있는 경우를 참조 무결성 위배로 볼 수 있다.

```
CREATE TABLE DEPT
(
 deptno VARCHAR2(4),
 deptname VARCHAR2(20),
 CONSTRAINT dptnopk PRIMARY KEY(deptno)
);

INSERT INTO DEPT VALUES('1000', '인사팀');
INSERT INTO DEPT VALUES('1001', '총무팀');

CREATE TABLE EMP
(
    empno number(10),
    ename varchar2(20),
    sal number(10,2) default 0,
    deptno varchar2(4) not null,
    createdate date default sysdate,
    CONSTRAINT empnopk primary key(empno),
    CONSTRAINT deptfk FOREIGN key(deptno) REFERENCES DEPT(deptno) ON DELETE
CASCADE
);

INSERT INTO EMP VALUES(100, '임베스트', 1000, '1000', sysdate);
INSERT INTO EMP VALUES(101, '을지문덕', 2000, '1001', sysdate);
SELECT * FROM EMP;

DELETE FROM DEPT WHERE DEPTNO = '1000';
SELECT * FROM EMP;
```

-- 'ALTER TABLE 테이블명' DDL을 통해 테이블을 변경할 수 있으며, 테이블명 변경, 칼럼 추가, 칼럼 변경, 칼럼 삭제 등을 할 수 있다.

-- 'ALTER TABLE ~ RENAME TO ~'을 통해 테이블명을 변경할 수 있다.
ALTER TABLE EMP RENAME TO NEW_EMP;

-- 'ALTER TABLE 테이블명 ADD()'을 통해 칼럼을 추가할 수 있다.
ALTER TABLE NEW_EMP ADD(age NUMBER(2) default 1);

-- 'ALTER TABLE 테이블명 MODIFY'을 통해 칼럼을 변경할 수 있다.
ALTER TABLE NEW_EMP MODIFY(ENAME VARCHAR2(40) NOT NULL);

-- 'ALTER TABLE 테이블명 DROP COLUMN 컬럼명'을 통해 칼럼을 삭제할 수 있다.
-- 'DROP'이 아니라 'DROP COLUMN'인 것을 명심해야 한다.
-- 'DROP COLUMN' 뒤에 괄호 없이 바로 삭제할 칼럼이 적힌다는 것을 명심해야 한다.
ALTER TABLE NEW_EMP DROP COLUMN AGE;

-- 'ALTER TABLE 테이블명 REANME COLUMN 기존의 칼럼명 TO 변경될 이름의 칼럼명'을 통해 칼럼명을 변경할 수 있다.
ALTER TABLE NEW_EMP RENAME COLUMN ENAME TO NEW_ENAME;

-- 'DROP TABLE' DDL을 통해서 테이블을 삭제할 수 있다.
CREATE TABLE TEST
(
    TEST_1 VARCHAR2(9),
    TEST_2 NUMBER(4)
);

INSERT INTO TEST VALUES('TEST', 4);

SELECT * FROM TEST;

DROP TABLE TEST;

-- 'DROP TABLE ~ CASCADE CONSTRAINT'를 통해서 해당 테이블을 참조한 슬레이브 테이블과 관련된 제약사항도 삭제할 수 있다.

-- 'CREATE VIEW 뷰명 (AS SELECT ~)' DDL을 통해서 View를 생성할 수 있다.
-- **View**란 **테이블로부터 유도된 가상의 테이블**이다.
-- **실제 데이터를 가지고 있지 않기 때문에 물리적인 공간이 필요하지 않고**, 특정 테이블을 참조해서 원하는 컬럼만 조회할 수 있게 한다.
-- DBA가 View를 통해 특정 칼럼만 가진 가상의 테이블을 생성시키면, 사용자는 실제 테이블에 대해서 자세히 알 수 없다.
-- 결국 실제 존재하는 테이블에 대한 보안성이 향상되고, 기존 테이블의 데이터 관리가 간단해진다.
-- 하나의 테이블에 대한 여러 개의 뷰를 생성할 수 있다.
-- 참조한 테이블이 변경되면 뷰도 변경된다.
-- 한번 생성된 뷰는 변경할 수 없고, 변경을 원하면 삭제 후 재생성해야 한다. 즉, ALTER문을 사용해서 뷰를 변경할 수 없다.

-- 장점 : 보안 기능, 데이터 관리 용이, 사용자의 SELECT문이 간단해짐
-- 단점 : 독자적인 인덱스를 만들 수 없음, 삽입 수정 삭제 연산이 제약됨, 데이터 구조를 변경할 수 없음

```
CREATE VIEW T_EMP AS SELECT EMPNO, NEW_ENAME FROM NEW_EMP;

SELECT * FROM T_EMP
```

-- 'INSERT INTO' DML을 통해 특정 테이블에 인스턴스를 추가할 수 있다.
-- 'VALUES'의 'S'붙는 것 명심
-- 데이터를 입력할 때 문자열을 입력하는 경우에는 작은따옴표를 사용해야 한다.
-- INSERT문을 실행했다고 데이터 파일에 저장되는 것은 아니다. 최종적으로 데이터를 저장하려면 TCL문인 'Commit'을 실행해야 한다.
-- 만약 'Auto Commi't으로 설정된 경우에는 Commit을 실행하지 않아도 바로 데이터 파일에 저장된다.

```
INSERT INTO EMP VALUES(100, '임베스트', 1000, '1000', sysdate);
```

-- 대량의 데이터를 INSERT하면, 오랜 시간이 소요될 수 있다.
-- 데이터베이스에 데이터를 입력할 때마다 로그파일에 그 정보를 기록하는데, 이것이 INSERT 성능을 저하시킨다.
-- 'ALTER TABLE 테이블명 NOLOGGING'을 통해, 로그파일에 대한 기록을 최소화시킬 수 있다.

```
ALTER TABLE NEW_EMP NOLOGGING;
```

-- 'UPDATE 테이블명 SET ~ WHERE ~'DML을 통해, 기존의 테이블에 이미 입력된 데이터 값을 수정할 수 있다.
-- 데이터를 수정할 때 조건절에서 검색되는 행 수만큼 수정이 된다.
-- 만약, WHERE 조건문을 입력하지 않으면 모든 데이터가 수정되므로 유의해야 한다.

```
INSERT INTO NEW_EMP VALUES(102, '가나다라', 3000, '1001', sysdate);
UPDATE NEW_EMP SET SAL = 4000 WHERE SAL = 3000;
SELECT * FROM NEW_EMP;
```

-- 'DELETE FROM 테이블명 WHERE ~'DML을 통해, 조건에 맞는 인스턴스를 삭제할 수 있다.
-- 만약, WHERE절을 입력하지 않으면, 테이블 내 모든 인스턴스가 삭제된다.
-- DELETE를 통해 인스턴스를 삭제한다고 해서, 테이블의 용량이 초기화되지 않는다.

```
INSERT INTO new_emp VALUES(102, '가나다라', 3000, '1001', sysdate);
UPDATE new_emp SET SAL = 4000 WHERE SAL = 3000;
SELECT * FROM NEW_EMP;

DELETE FROM new_emp WHERE new_ename = '가나다라';
SELECT * FROM NEW_EMP;
```

-- 'SELECT 컬럼명 || 문자열 FROM 테이블명'DML을 통해, 특정 컬럼에 들어있는 모든 데이터 값에다 지정한 문자열(무조건 문자열일 필요는 없고 숫자형도 가능)을 결합한 결과물을 조회할 수 있다.

```
SELECT EMPNO, NEW_ENAME || '님' FROM NEW_EMP;
```

-- 'ORDER BY 컬럼명'DML을 통해, 데이터를 오름차순 또는 내림차순으로 출력할 수 있다.
-- 'ORDER BY 컬럼명 DESC'를 통해, 지정한 컬럼에 대한 내림차순으로 출력할 수 있다.
-- 해당 DML은 데이터베이스 메모리를 많이 사용하게 된다. 그러므로 대량의 데이터를 정렬하게 되면, 정렬로 인한 성능 저하가 발생한다.

```
CREATE TABLE EMP
(
EMPNO NUMBER(10),
ENAME VARCHAR2(20),
SAL NUMBER(10),

CONSTRAINT EMPNO_PK PRIMARY KEY(EMPNO)
);

INSERT INTO EMP VALUES(1000, '임베스트', 20000);
INSERT INTO EMP VALUES(1001, '조조', 20000);
INSERT INTO EMP VALUES(1002, '관우', 20000);
INSERT INTO EMP VALUES(999, '가나다라', 10000);

SELECT * FROM EMP ORDER BY EMPNO DESC;
```

-- 'ORDER BY' 정렬 성능 저하는 'Oracle hint'를 사용하여 해결할 수 있다.
-- 'Oracle hint'란 SQL 튜닝의 핵심 부분으로 일종의 지시 구문이다.
-- 즉, 오라클 옵티마이저(Optimizer)에게 SQL문 실행을 위한 데이터를 스캐닝하는 경로, 조인하는 방법 등을 알려주기 위해, SQL사용자가 SQL 구문에 작성하는 것을 뜻한다.
-- 오라클이 항상 최적의 실행 경로를 만들어 내기는 불가능하기 때문에, 직접 최적의 실행 경로를 작성해 주는 것이다.
-- 기본적으로 쿼리의 서두에 힌트를 명시해야 한다.
-- 'SELECT /*+ INDEX_DESC(테이블명 인덱스명) */ 컬럼명'을 입력하면, INDEX를 이용한 정렬을 실시한다.

```
SELECT /*+ INDEX_DESC(EMP EMPNO_PK) */ * FROM EMP;
```

```
-- 실행계획 출력 쿼리
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'ALLSTATS LAST'));




-- DISTINCT 사용법
-- SELECT 내 'DISTINCT'를 컬럼명 앞에 적어주면, 중복되지 않은 데이터를 보여준다.
SELECT DISTINCT SAL FROM EMP;




-- ALIAS 사용법
-- 칼럼에 대한 별명 설정은 'AS ~'로 하면되고, 테이블에 대한 별명 설정은 원래 테이블명 옆에 띄어쓰고 바로 기재해주면 된다.
SELECT EMPNO, ENAME, SAL AS 연봉 FROM EMP E WHERE E.SAL = 20000;




-- WHERE문이 사용하는 연산자

-- '='은 등호를 의미하는 연산자이다. SQL에선 '=='라는 연산자는 없다.
-- '!=', '^=', '<>'는 전부 부등호를 의미하는 연산자이다.
-- 연산식앞에 NOT을 붙일 수 있다. 그래서 'WHERE SAL != 10000'과 'WHERE NOT SAL = 10000'은 같은 의미이다.
SELECT * FROM EMP E WHERE e.sal != 10000;
SELECT * FROM EMP E WHERE NOT E.SAL = 10000;

-- '컬럼명 BETWEEN A AND B' 해당 컬럼 내 A와 B 사이에 존재하는 데이터를 조회한다.
SELECT * FROM EMP E WHERE e.sal BETWEEN 20000 AND 30000;

-- 'IN(list)'는 'OR'을 의미하며, list 값 중 하나만 일치해도 조회된다.
INSERT INTO EMP VALUES(1003, 'HIHI', 15000);
```

```
SELECT * FROM EMP E WHERE E.SAL IN (10000, 15000);

-- 와일드카드(%, _)를 사용하는 'LIKE'문
-- 'WHERE'절에서 와일드카드를 사용하기 위해선 '컬럼명 LIKE ~'으로 시작해야한다.
-- '%'는 어떤 문자를 포함한 도느 것을 조회한다.
SELECT * FROM EMP E WHERE E.ENAME LIKE '임%';

-- '_'는 한 개인 단일 문자를 의미한다.
SELECT * FROM EMP E WHERE E.ENAME LIKE '임베스_';
```

```
-- 'NULL'은 모르는 값을 의미한다.
-- 'NULL'은 값의 부재를 의미한다.
-- 'NULL'과 어떤 값을 비교할 때, '알 수 없음'이 반환된다. 즉, 'SELECT * FROM EMP WHERE SAL > NULL;'를 실행하면, 컬럼들만 있고 인스턴스는 없는 결과가 반환된다.
-- 결론적으로, NULL값에 대한 비교연산은 항상 UNKNOWN을 반환한다.
-- 'NULL'과 숫자 혹은 날짜를 더하면 NULL이 된다.
SELECT 1+NULL FROM DUAL;

-- NULL값을 조회할 경우는 'IS NULL'을 사용하고 NULL값이 아닌 것을 조회할 경우는 'IS NOT NULL'을 사용한다.
INSERT INTO EMP(EMPNO, ename) VALUES(1004, 'BYE');
SELECT * FROM EMP E WHERE SAL IS NULL;

-- NVL(컬럼명, 지정값) : 컬럼 내 NULL값을 '지정값'으로 변형하는 함수
SELECT EMPNO, ENAME, NVL(SAL, 0) AS FIEXED FROM EMP;

-- NVL2(컬럼명, 지정값1, 지정값2) : 컬럼 내 NULL값이 아닌 것을 '지정값1'로 변형하고 NULL값인 것을 '지정값2'로 변형하는 함수
SELECT EMPNO, ENAME, NVL2(SAL, 1, 0) AS FIEXED FROM EMP;

-- NULLIF(컬럼명1, 컬럼명2) : 두 컬럼 내 데이터 값이 같으면 NULL값을, 같지 않으면 '컬럼명1'에 해당하는 값을 반환하는 함수

-- COALESCE(컬럼명1, 컬럼명2, 컬럼명3, ...)['카우얼레스'라고 발음함] : NULL이 아닌 최초의 인자를 반환하는 함수
-- 즉, '컬럼명1'에 해당하는 값이 NULL이 아니면 '컬럼명1'에 해당하는 값을, 그렇지 않으면 그 뒤의 값의 NULL 여부를 판단하여 값을 반환한다.
```

-- 'GROUP BY'구문은 특정 기준에 맞춰 데이터들을 논리적으로 파티셔닝 하는 역할을 한다.
-- 나누고자 하는 그룹 컬럼명을 SELECT절과 GROUP BY구문 뒤에 추가하면 된다.
-- 즉, GROUP BY구문으로 지정된 컬럼은 SELECT절에도 반드시 등장해야 한다.
SELECT DEPTNO, SUM(SAL) AS SALLARY_SUM FROM EMP GROUP BY DEPTNO ;
-- GROUP BY구문을 사용하면서 SELECT절에 'SUM(CASE WHEN ~)'이 기입되어있다면, 해당 'SUM(CASE WHEN ~)'은 GROUP PARTITION 내에서 WHEN 조건에 맞는 데이터들만 합한다.

-- 'HAVING'구를 통해, 'GROUP BY'절에 의한 생성된 파티션 중 원하는 조건에 부합하는 파티션으로만 간추릴 수 있다.
-- 즉, 'HAVING'구가 반환되는 최종 결과물에 대한 마지막 조건문 역할을 수행한다.
SELECT DEPTNO, SUM(SAL) AS SALLARY_SUM FROM EMP GROUP BY DEPTNO HAVING SUM(SAL) < 30000;

-- 집계 함수(aggregate function) : 여러 행 또는 테이블 전체 행으로부터 하나의 결과값을 반환하는 함수
-- 집계 함수(aggregate function) 종류 : COUNT(), SUM(), AVG(), MAX(), MIN(), STDDEV(), VARIAN()
SELECT DEPTNO, AVG(SAL) FROM EMP GROUP BY DEPTNO;
SELECT DEPTNO, MAX(SAL) FROM EMP GROUP BY DEPTNO;
SELECT DEPTNO, STDDEV(SAL) FROM EMP GROUP BY DEPTNO;
SELECT DEPTNO, VARIANCE(SAL) FROM EMP GROUP BY DEPTNO;

-- 'COUNT(*)'는 각 파티션별 NULL값을 포함한 모든 인스턴스의 개수를 계산한다. 하지만 'COUNT(칼럼명)'은 NULL값을 제외한 인스턴스의 개수를 계산한다.

-- SELECT 문의 실행 순서 : FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY

-- 형변환 : 데이터 타입을 변환하는 것이다.
-- 예를 들어 숫자와 문자열의 비교 또는 문자열과 날짜형의 비교를 실시할 때, 두 데이터 타입이 불이치해서 해당 비교가 제대로 실시되지 않는다. 이때 형변환을 실시하여 해당 비교가 실행되도록 한다.
-- 형변환을 실시했다고 해서, 지정된 컬럼의 유형이 완전히 변환되는 것이 아니다. 형변환 함수를 실행했을 때만 임시로 변형되는 것이다.
-- **인덱스가 걸린 컬럼에 형변환을 수행하면, 해당 인덱스를 사용하지 못한다.**

-- 형변환은 '명시적 형변환'과 '암시적 형변환'이 있다.
-- **명시적 형변환** : 형변환 함수를 직접 사용해서 데이터 타입을 변형시키는 것
-- **암시적 형변환** : 형변환 함수를 직접 사용하지 않고, DBMS가 자동으로 형변환을 실시하는 것
-- TO_NUMBER(문자열 컬럼) : 문자열 컬럼을 숫자형 컬럼으로 변형
-- TO_CHAR(숫자형 컬럼(or 날짜형 컬럼), FORMAT) : 숫자 컬럼 또는 날짜형 컬럼을 'FORMAT'매개변수에 따라 문자형 컬럼으로 변형
-- TO_DATE(문자열 컬럼, FORMAT) : 문자열 컬럼을 지정된 FORMAT의 날짜형 컬럼으로 변환한다.

```
SELECT to_number(DEPTNO) FROM DEPT;
SELECT to_char(EMPNO) FROM EMP;
```

-- DUAL 테이블 : Oracle 데이터베이스에 의해 자동으로 생성되는 Dummy 테이블이다.
-- Oracle 데이터베이스 사용자가 임시로 사용할 수 있는 테이블로, 내장형 함수를 실행할 때도 사용할 수 있다.
-- 즉, 해당 테이블 내에서 여러 내장형 함수를 시험해볼 수 있다.
-- Oracle 데이터베이스의 모든 사용자가 사용할 수 있다.

-- 문자열과 관련한 내장 함수 : ASCII(문자(or 숫자)), CHR(ASCII 코드값), SUBSTR(문자열, m, n), CONCAT(문자열1, 문자열2), LOWER(문자열), UPPER(문자열), LEN(문자열), LTRIM(문자열, 지정문자), RTRIM(문자열, 지정문자), TRIM(문자열, 지정문자)

-- SUBSTR(문자열, m, n) : 문자열에서 m번째 위치부터 n개를 자른다.
SELECT SUBSTR('ABCDEFG',3,2) FROM DUAL;

-- CONCAT(문자열1, 문자열2) : '문자열1'과 '문자열2'를 결합한다. Oracle DB에선 'II'를 CONCAT함수 대신 사용할 수 있다.**(concatenate : 사슬같이 잇다; 연쇄시키다)**
SELECT 'AB'||'CD' FROM DUAL;

-- TRIM(문자열, 지정된 문자) : 왼쪽 및 오른쪽에서 지정된 문자를 삭제한다. 만약 지정된 문자를 생략하면, 왼쪽 및 오른쪽의 공백을 제거한다.
SELECT TRIM(' ABCDEF ') FROM DUAL;

-- 날짜형과 관련한 내장 함수 : SYSDATE, EXTRACT(YEAR[or MONTH or DAY.....] FROM 테이블명),
-- SYSDATE : 오늘의 날짜를 날짜형의 값으로 반환한다. **함수이지만 '()'를 이용하여 호출하지 않는다.**
SELECT SYSDATE FROM DUAL;

-- EXTRACT() : 날짜형의 데이터로부터 년, 월, 일을 추출하여 반환한다.
SELECT EXTRACT(YEAR FROM SYSDATE), EXTRACT(MONTH FROM SYSDATE), EXTRACT(DAY FROM SYSDATE) FROM DUAL;

-- 숫자형과 관련한 내장함수 : ABS(숫자), SIGN(숫자), MOD(숫자1, 숫자2), CEIL(숫자), FLOOR(숫자), ROUND(숫자, m), TRUNC(숫자, m)

-- SIGN(숫자) : 양수, 음수, 0을 구별하는 계단함수이다.
SELECT SIGN(10) FROM DUAL;

-- MOD(숫자1, 숫자2) : '숫자1'을 '숫자2'로 나눈 나머지를 반환하는 함수이다. Oracle DB에선 '%'를 MOD함수 대신 사용할 수 있다.
SELECT MOD(10, 3) FROM DUAL;

-- CEIL(숫자) : 숫자보다 크거나 같은 최소의 정수를 반환하는 함수이다.
SELECT CEIL(10.3) FROM DUAL;

-- FLOOR(숫자) : 숫자보다 작거나 같은 최대의 정수를 반환하는 함수이다.
SELECT FLOOR(10.3) FROM DUAL;

-- ROUND(숫자, m) : 소수점 m자리에서 반올림한다. **m이 0일 때 소수점 첫째 자리를 의미한다는 것을 주의해야한다.**
SELECT ROUND(10.34, 1) FROM DUAL; -> 10.3

-- TRUNC(숫자, m) : 소수점 m자리에서 절삭한다. **m이 0일 때 소수점 첫째 자리를 의미한다는 것을 주의해야한다.**
SELECT TRUNC(10.34, 0) FROM DUAL; -> 10

-- DECODE문 : 프로그래밍 언어에서의 IF문을 구현할 수 있다. 즉, 특정조건이 참이면 A, 거짓이면 B로 응답한다.
-- 보통 SELECT문에 DECODE문을 기재한다.
SELECT DECODE(EMPNO, 1000, 'TRUE', 'FALSE') FROM EMP;

-- CASE문 : 프로그래밍 언어에서의 'IF ~ THEN ~ ELSE'문을 구현할 수 있다. 조건을 WHEN구에 사용하고 THEN은 해당 조건이 참이면 실행되고 거짓이면 ELSE구가 실행된다.

```
SELECT
    CASE
        WHEN EMPNO = 1000 THEN 'A'
        WHEN EMPNO = 1001 THEN 'B'
        ELSE 'C'
    END
    FROM EMP;
```

-- 'ROWNUM 컬럼'은 ORACLE 데이터베이스의 SELECT문 결과에 대해서 논리적인 일련번호를 부여한 컬럼이다.
-- ROWNUM 컬럼은 화면에 데이터를 출력할 때 부여되는 논리적 순번이다.
-- 즉, 모든 테이블에 해당 컬럼이 내재되어 있다고 생각하면 되고, SELECT문으로 확인할 수 있다.
SELECT ROWNUM, ENAME FROM EMP;

-- ROWNUM 컬럼은 조회되는 행 수를 제한할 때 많이 사용된다.
-- 즉, ROWNUM을 사용해서 페이지 단위 출력을 할 수 있다. 페이지 단위 출력을 하기 위해서는 'inline view(from절의 서브쿼리)'를 사용해야 한다.
SELECT * FROM(SELECT ROWNUM AS LIST, ENAME FROM EMP) WHERE LIST <= 3;
SELECT * FROM(SELECT ROWNUM AS LIST, ENAME FROM EMP) WHERE LIST BETWEEN 2 AND 4;

-- 'ROWID 컬럼'은 Oracle DB 내에서 데이터를 구분할 수 있는 유일한 값을 가진 컬럼이다.
-- ROWNUM 컬럼처럼 테이블에 내재되어 있고, SELECT문으로 확인할 수 있다.
-- 해당 커럼 값을 통해 특정 인스턴스가 어떤 데이터 파일, 어느 블록에 저장되어 있는지 알 수 있다.
-- **ROWID 해석방법**은 아래와 같다.
-- 1번부터 6번째 자리 : 오브젝트 번호 - 오브젝트 별로 유일한 값을 가지고 있으며, 해당 오브젝트가 속해 있는 값이다.
-- 7번부터 9번째 자리 : 상대 파일 번호 - 테이블스페이스에 속해 있는 데이터 파일에 대한 상대 파일 번호이다.
-- 10번부터 15번째 자리 : 블록 번호 - 데이터 파일 내부에서 어느 블록에 데이터가 있는지 알려준다.
-- 16번부터 18번째 자리 : 데이터 번호 - 데이터 블록에 데이터가 저장되어 있는 순서를 의미한다.

-- WITH구문 : 서브쿼리를 사용해서 가상 테이블이나 뷰처럼 사용할 수 있는 구문
-- 임시테이블을 만든다는 관점에서 본다면, VIEW와 쓰임새가 비슷하다.
-- 둘 간의 차이점은 VIEW는 한 번 만들어놓으면 DROP할 때까지 없어지지 않지만, **WITH은 한 번 실행할 쿼리문 내에 저장되어 있을 경우 그 쿼리문 안에서만 실행된다.** 즉, **WITH을 사용해서 만들어지는 임시테이블은 해당 쿼리문이 끝나면 사라지는 것이다.**
-- **복잡한 처리를 실시할 때 Subquery의 중첩 대신 WITH구문을 사용함으로써, SQL문의 가독성이 높아진다.**
-- **WITH구문을 사용해서 여러 테이블을 정의할 때는 쉼표를 사용해 테이블을 나열한다.(즉, WITH 구문에서 여러 테이블을 정의할 수 있다.)**
-- **처음 기재된 WITH구문을 두 번째 기재된 WITH구문에서 사용할 수 있다.**
-- 옵티마이저는 **WITH구문을 임시테이블로 판단**한다.
-- **WITH구문 형식 : WITH 임시테이블명 AS (SELECT ~)**
-- **WITH구문 안에 들어가는 SELECT절 끝에는 절대 세미콜론이 들어가면 안된다.**

```
WITH VIEWDATA AS (SELECT * FROM EMP) SELECT * FROM VIEWDATA;
```

-- DCL(Data Control Language)는 'GRANT'문과 'REVOKE'문으로 구성되어 있다.

-- GRANT문 : 데이터베이스 사용자에게 권한을 부여한다.
-- 데이터베이스 사용을 위해서는 권한이 필요하며, 연결, 입력, 수정, 삭제, 조회를 할 수 있다.
-- DML 권한을 부여하는 GRANT문 형식 : GRANT 권한명 ON 테이블명 TO 사용자명;
-- 권한 종류 : SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, INDEX, ALL
-- SELECT : 지정된 테이블에 대해서 SELECT권한을 부여
-- INSERT : 지정된 테이블에 대해서 인스턴스 INSERT권한을 부여
-- UPDATE : 지정된 테이블에 대해서 데이터 값 UPDATE권한을 부여
-- REFERENCES : 지정된 테이블을 참조하는 제약조건을 생성하는 권한을 부여
-- ALTER : 지정된 테이블에 대해서 수정할 수 있는 권한을 부여
-- INDEX : 지정된 테이블에 대해서 인덱스를 생성할 수 있는 권한을 부여
-- ALL : 지정된 테이블에 대한 모든 권한을 부여
<SYSTEM 계정에서 해당 코드 실행>
GRANT CREATE SESSION TO us1152;
GRANT CREATE TABLE, CREATE VIEW, CREATE SEQUENCE TO us1152;
GRANT SELECT ON EMP TO us1152;

<us1152 계정에서 해당 코드 실행>
SELECT * FROM SYSTEM.EMP;**(반드시 테이블 앞에 해당 테이블을 생성한 사용자명을 기재해야 한다.)**

-- 'TO' 구문 뒤에 'WITH GRANT OPTION'구문 또는 'WITH ADMIN OPTION'을 추가하면, 현재 부여받은 권한을 다른 사용자에게도 부여할 수 있게 된다.
-- **WITH GRANT OPTION : REVOKE시 REVOKE가 연쇄적으로 발생한다.** 즉, 자신이 부여받은 특정 권한이 철회되었을 때, 해당 권한에 대한 다른 사용자의 권한도 자동적으로 철회된다.
-- **WITH ADMIN OPTION : REVOKE시 REVOKE가 연쇄적으로 발생하지 않는다.**

-- REVOKE문 : 데이터베이스 사용자에게 부여된 권한을 철회한다.
-- DML 권한에 대한 REVOKE문 형식 : REVOKE 권한명 ON 테이블명 FROM 사용자명;
<SYSTEM 계정에서 해당 코드 실행>
REVOKE SELECT ON EMP FROM US1152;

-- TCL(Transaction Control Language)은 'COMMIT', 'ROLLBACK', 'SAVEPOINT'와 같은 트랜잭션을 관리하는 구문으로 구성되어 있다.
-- 트랜잭션 : '데이터 처리 단위'이다. COMMIT을 기준으로 설명하자면, COMMIT이 일어난 시점부터 다음의 COMMIT전까지의 작업이 하나의 트랜잭션이라 생각하면 된다.
-- 'COMMIT'문을 활용하여 해당 트랜잭션을 실행한 후 종료하거나, 'ROLLBACK'문을 활용하여 트랜잭션 실행을 취소할 수 있다.
-- **INSERT, UPDATE, DELETE문으로 변경한 데이터를 데이터베이스에 반영하기 위해선, 반드시 COMMIT문을 실행해야한다.** 즉, 특정 데이터를 INSERT 한 후 해당 테이블을 조회하면, 해당 데이터가 INSERT된 것으로 보이나, COMMIT을 하지 않았기 때문에 ROLLBACK하면 INSERT하기 전의 상태로 돌아간다.
-- COMMIT문을 실행하면, 변경 전 이전 데이터는 사라진다. 즉, A값을 B로 변경하고 COMMIT문을 실행하면, A값은 사라지고 B값을 반영한다.
-- COMMIT이 완료되면, 다른 모든 데이터베이스 사용자는 변경된 데이터를 볼 수 있고 조작할 수 있다.
-- Auto commit : SQL*PLUS 프로그램을 정상적으로 종료하는 경우 자동 COMMIT된다. DDL 및 DCL을 사용하는 경우 자동 COMMIT된다. 'set autocommit on;'을 SQL*PLUS에서 실행하면 자동 COMMIT된다.

```
COMMIT;
```

-- 'ROLLBACK'문을 실행하면, 데이터에 대한 변경 사용을 모두 취소하고 드랜잭션을 종료한다.
-- 즉, INSERT, UPDATE, DELETE문의 작업을 모두 취소한다. **단, 마지막 COMMIT한 지점으로 돌아간다.**

```
ROLLBACK;
```

-- 'SAVEPOINT'문은 트랜잭션을 작게 분할하여 관리하는 것으로, SAVEPOINT를 사용하면 ROLLBACK 했을 때 이전 COMMIT으로 돌아가는 것이 아닌 해당 SAVEPOINT 지점으로 돌아가게 된다.
-- **SAVEPOINT문 형식 : SAVEPOINT 세이브포인트명**
-- 특정 SAVEPOINT로 돌아가고 싶은 경우는 '**ROLLBACK TO 세이브포인트명**'을 실행하면 된다.
-- 만약 그냥 'ROLLBACK'만 실행했을 경우, 마지막 SAVEPOINT로 돌아가는 것이 아닌 이전 COMMIT한 곳으로 돌아간다.

-- 'JOIN'은 여러 개의 테이블을 사용해서 새로운 릴레이션을 만드는 과정이다.
-- **JOIN의 기본은 테이블 간의 교집합을 만드는 것**이다. 즉, 교집합을 중심으로 두 릴레이션을 결합하는 것이다.
-- 'EQUI JOIN'은 두 테이블 간의 컬럼 값들이 서로 정확하게 일치하는 경우에 사용하는 JOIN이다.
-- 즉, 컬럼 값이 정확하게 일치하지 않는 인스턴스들은 EQUI JOIN의 결과물에서 제외된다.
-- EQUI JOIN 형식 : 'SELECT * FROM 테이블1, 테이블2 WHERE 테이블1.컬럼명 = 테이블2.컬럼명'

```
SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO;
```

-- **'INNER JOIN'은 EQUI JOIN과 동일하다.**
-- INNER JOIN 형식 : 'SELECT * FROM 테이블1 INNER JOIN 테이블2 ON 테이블1.컬럼명 = 테이블2.컬럼명'

```
SELECT * FROM EMP A INNER JOIN DEPT B ON A.DEPTNO = B.DEPTNO AND A.ENAME LIKE '임%';
```

-- **EQUI JOIN은 'HASH JOIN'을 발생시킨다.**

-- 'Non-EQUI JOIN'은 EQUI JOIN에서 '='대신 '!='를 사용하는 것이다.
-- 즉, Non-EQUI JOIN은 정확하게 일치하지 않는 것을 조인하는 것이다.
-- 두 테이블의 CROSS JOIN 결과에서 EQUI JOIN의 결과를 뺀 나머지와 같다.

```
SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO != DEPT.DEPTNO;
```

-- 'OUTER JOIN'은 두 개의 테이블 간에 교집합을 조회하고, 한쪽 테이블에만 있는 데이터도 포함시켜서 조회한다.
-- **LEFT JOIN과 RIGHT JOIN의 결과에선 기준 테이블(다른 테이블과 결합 되어지는 테이블)의 행 개수가 유지된다.**
-- **이로써 JOIN 결과물을 해석할 때 발생할 수 있는 오해를 방지할 수 있다.**
-- 예를 들어, EQUI JOIN의 결과물을 본 사람은 해당 EQUI JOIN 결과물에 포함되어 있지 않은 데이터들을 고려하지 않고 잘못된 의사결정을 할 수 있다. 이를 방지하기 위해, OUTER JOIN을 사용하는 것이다.
-- OUTER JOIN은 'LEFT OUTER JOIN', 'RIGHT OUTER JOIN', 'FULL OUTER JOIN'으로 구성되어 있다.
-- FULL OUTER JOIN은 LEFT OUTER JOIN과 RIGHT OUTER JOIN를 모두 하는 것이다.

-- LEFT OUTER JOIN 형식 : 'SELECT * FROM 테이블1 LEFT OUTER JOIN 테이블2 ON 테이블1.컬럼명 = 테이블2.컬럼명'
SELECT * FROM EMP LEFT OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;

-- RIGHT OUTER JOIN 형식 : 'SELECT * FROM 테이블1 RIGHT OUTER JOIN 테이블2 ON 테이블1.컬럼명 = 테이블2.컬럼명'
SELECT * FROM EMP RIGHT OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;

-- FULL OUTER JOIN 형식 : 'SELECT * FROM 테이블1 FULL OUTER JOIN 테이블2 ON 테이블1.컬럼명 = 테이블2.컬럼명'
SELECT * FROM EMP FULL OUTER JOIN DEPT ON EMP.DEPTNO = DEPT.DEPTNO;

-- 'CROSS JOIN'은 두 테이블에 대한 카테시안 곱 연산을 수행하는 것이다. 그러므로 ON구가 존재하지 않는다.
-- CROSS JOIN을 적지 않고 FROM 절에 두 테이블을 나열해도 카테시안 곱 연산을 발생시킨다.
SELECT * FROM EMP CROSS JOIN DEPT;

-- 'UNION' 연산은 두 개의 테이블을 하나로 만드는 연산이다.
-- 즉, 2개의 테이블을 하나로 합치는 것이다. 주의사항은 두 개의 테이블의 컬럼 수와 컬럼의 데이터 형식이 모두 일치해야 한다. 만약 두 개의 테이블에 UNION 연산이 사용될 때 컬럼 수 혹은 데이터 형식이 다르면 오류가 발생한다.
-- UNION 연산은 두 개의 테이블을 하나로 합치면서 **중복된 데이터를 제거**한다. 또한, **SORT과정도 발생**시킨다.

```
SELECT 2 FROM DUAL
UNION
SELECT 1 FROM DUAL;
```

-- **'UNION ALL' 연산은** UNION처럼 **중복을 제거하거나 정렬을 유발하지 않고**, 두 테이블을 합친다.

```
SELECT * FROM EMP
UNION ALL
SELECT * FROM EMP;
```

-- 'INTERSECT'연산은 두 개의 테이블에서 교집합을 조회한다. 즉, 두 개 테이블에서 공통된 값을 조회한다.

```
SELECT DEPTNO FROM EMP
INTERSECT
SELECT DEPTNO FROM DEPT;
```

-- 'MINUS' 연산은 두 개의 테이블에서 차집합을 조회한다. 즉, 먼저 쓴 SELECT문에만 있는 집합을 반환한다.
-- MINUS 연산은 MY-SQL에서 'EXCEPT' 연산과 같다.

```
SELECT DEPTNO FROM DEPT
MINUS
SELECT DEPTNO FROM EMP;
```

-- 'CONNECT BY' 조회는 Oracle DB에서 지원하는 것으로, 트리형으로 데이터를 조회할 수 있다.
-- 즉, 데이터 조회 방식이 '전위순회출력'과 동일하다. '부모노드 내 인스턴스 출력 -> 왼쪽 노드 내 인스턴스 출력 -> 오른쪽노드 내 인스턴스 출력'으로 진행된다.
-- 예를 들어 부장에서 차장, 차장에서 과장, 과장에서 대리, 대리에서 사원 순으로 트리 형태의 구조를 위에서 아래로 탐색하면서 조회하는 것이다. 물론 역방향 조회도 가능하다.
-- 해당 트리 계층의 Level은 1부터 시작한다.
-- 'START WITH'구는 시작 조건을 의미하고, 'CONNECT BY PRIOR'은 조인 조건이다.

```
SELECT * FROM NEW_EMP START WITH MGR IS NULL CONNECT BY PRIOR EMPNO = MGR;
```

-- CONNECT BY 조회를 명확히 보기 위해선, SELECT절에 'LPAD()' 함수를 사용하는 것이 좋다.
-- 'LPAD()' 함수는 특정 대상값이 지정한 문자 길이만큼 되도록, 대상값 왼쪽에 어떤 문자를 채워넣는 함수이다.
-- LPAD 함수 형식 : LPAD(대상값, 총 문자길이, 채움문자)
[채움문자 개수 : 총 문자길이 - LENGTH(대상값)]

```
SELECT LEVEL, LPAD(' ', 4 * (LEVEL-1), ' ') || EMPNO, MGR FROM NEW_EMP START WITH MGR IS NULL CONNECT BY PRIOR EMPNO = MGR;
```

-- 'CONNECT BY' 조회와 관련한 키워드 : LEVEL, CONNECT_BY_ROOT 컬럼명, CONNECT_BY_ISLEAF, SYS_CONNECT_BY_PATH(컬럼명, 문자), NOCYCLE, CONNECT_BY_ISCYCLE

-- CONNECT_BY_ISLEAF : 해당 값을 가진 노드가 말단 노드인지 표시('1'이면 말단노드)

```
SELECT LEVEL, LPAD(' ', 4 * (LEVEL-1), ' ') || EMPNO, MGR, CONNECT_BY_ISLEAF FROM NEW_EMP START WITH MGR IS NULL CONNECT BY PRIOR EMPNO = MGR;
```

-- CONNECT_BY_ROOT 컬럼명 : 루트 노드 내 값을 표기

```
SELECT LEVEL, LPAD(' ', 4 * (LEVEL-1), ' ') || EMPNO, MGR, CONNECT_BY_ROOT EMPNO FROM NEW_EMP START WITH MGR IS NULL CONNECT BY PRIOR EMPNO = MGR;
```

-- SYS_CONNECT_BY_PATH(컬럼명, 문자) : 트리 구조의 전개 경로를 표시

```
SELECT LEVEL, LPAD(' ', 4 * (LEVEL-1), ' ') || EMPNO, MGR, SYS_CONNECT_BY_PATH(EMPNO, '|') FROM NEW_EMP START WITH MGR IS NULL CONNECT BY PRIOR EMPNO = MGR;
```

-- 'Subquery'는 SELECT문 내 특정 구문에 다시 SELECT문을 사용하는 SQL문이다.
-- 즉, 전체 query는 'Main query'와 'Subquery'로 나눠져 있다.
-- **Correlated Subquery를 제외한 나머지 Subquery는 Main Query가 실행되기 이전에 한번 실행된다.**(Correlated Subquery는 해당 Subquery가 여러번 실행된다.)
-- Subquery의 형태 : **Inline View**(FROM구에 SELECT문을 사용하는 경우), **Scala Subquery**(SELECT문에 Subquery를 사용하는 경우), **Nested Subquery**(WHERE구에 SELECT문을 사용하는 경우)
-- Subquery는 괄호로 묶는다.

-- Nested Subquery 형태 :
SELECT ENAME, DEPTNO, JOB FROM new_emp **WHERE DEPTNO = (SELECT DEPTNO FROM new_emp WHERE ENAME = 'TEST1');**
SELECT EMPNO, ENAME, SAL FROM NEW_EMP **WHERE SAL IN (SELECT SAL FROM NEW_EMP WHERE SAL > 2000);**

-- **Inline View 효과** : Inline View는 **가상의 테이블을 만드는 효과**를 얻을 수 있다.
SELECT EMPNO, ENAME FROM **(SELECT EMPNO, ENAME, DEPTNO FROM NEW_EMP WHERE SAL > 2000)**;

-- **Scala Subquery** : SELECT문에 사용된 subquery이고, Scala subquery는 반드시 한 개의 행과 한 개의 컬럼을 반환해야 한다.

-- Subquery는 Single row subquery, Multi row subquery, Multi column subquery로 나뉜다.
-- **Single row subquery(단일 행 서브 쿼리)** : **하나의 컬럼에서 하나의 인스턴스만 반환하는 subquery**로, **비교 연산자는 '=, <=, >=, <>'를 사용**한다.

```
SELECT ENAME, DEPTNO, JOB
FROM new_emp
WHERE DEPTNO = (SELECT DEPTNO FROM new_emp WHERE ENAME = 'TEST1');
```

-- **Multi row subquery** : **하나의 컬럼에서 여러 개의 인스턴스를 반환하는 subquery**로, **'IN, ANY, ALL, EXISTS'를 접목시킨 비교 연산자를 사용**한다.
-- IN : Main query의 비교조건이 Subquery의 결과 중 하나만 동일하면 참이 된다(OR조건)

```
SELECT EMPNO, ENAME, SAL FROM NEW_EMP WHERE SAL IN (SELECT SAL FROM NEW_EMP WHERE SAL > 2000);
```

-- ANY : Main query의 비교조건이 Subquery의 결과 중 하나 이상 동일하면 참이 된다.
-- **> ANY** : Main query의 비교조건이 Subquery 결과의 인스턴스 중 한 개 이상 크다면 참이 된다.

```
SELECT EMPNO, ENAME, SAL FROM NEW_EMP WHERE SAL > ANY (SELECT SAL FROM NEW_EMP WHERE deptno = 20);
```

-- **< ANY** : Main query의 비교조건이 Subquery 결과의 인스턴스 중 한 개 이상 작다면 참이 된다.

```
SELECT EMPNO, ENAME, SAL FROM NEW_EMP WHERE SAL < ANY (SELECT SAL FROM NEW_EMP WHERE deptno = 20);
```

-- ALL : Main query와 Subquery의 결과가 모두 동일하면 참이 된다.
-- **> ALL** : Main query의 비교조건이 Subquery 결과의 모든 인스턴스보다 크다면 참이 된다.

```
SELECT EMPNO, ENAME, SAL FROM NEW_EMP WHERE SAL > ALL (SELECT SAL FROM NEW_EMP WHERE deptno = 20);
```

-- **< ALL** : Main query의 비교조건이 Subquery 결과의 모든 인스턴스보다 작다면 참이 된다.

```
SELECT EMPNO, ENAME, SAL FROM NEW_EMP WHERE SAL < ALL (SELECT SAL FROM NEW_EMP WHERE deptno = 30);
```

-- EXISTS : Subquery 데이터가 존재하는가를 체크해 존재 여부(TRUE,FALSE)를 결과로 반환한다.
-- EXIST가 아니라 **EXISTS**이다.
-- **EXISTS 절의 Subquery는 주로 Main query와 연결이 되는 조인 조건을 가진다.**
-- 즉, EXISTS 절의 Subquery는 주로 '**Correlated Subquery**(Subquery 내에서 Main Query 내의 컬럼을 사용하는 것)'형태를 가진다.
-- **IN, ANY, ALL 구문에서는 Mainquery보다 Subquery를 먼저 실행하였고, Subquery를 단**

한번만 실행하였다. 하지만 Correlated Subquery인 EXISTS 구문은 앞선 동작방식과 다르다.
-- 먼저 Mainquery의 테이블에 접근하여 하나의 레코드를 가져오고, 그 레코드에 대해서 EXISTS 이하의 서브쿼리를 실행한다. 이후 서브쿼리에 대한 결과물이 존재하는지를 확인한다.
SELECT EMPNO, ENAME, MGR FROM NEW_EMP a WHERE EXISTS(SELECT * FROM NEW_EMP b WHERE b.MGR = a.MGR);

-- NOT EXISTS는 EXISTS 동작방식의 반대이다.

-- Multi column subquery : 여러 개의 컬럼을 검색하는 subquery로, 'IN'을 주로 사용한다.
SELECT EMPNO, ENAME, SAL FROM NEW_EMP WHERE (EMPNO, SAL) IN (SELECT EMPNO, SAL FROM NEW_EMP WHERE EMPNO < 1005 AND SAL > 2000);

-- '그룹 함수'는 GROUP BY절에 사용되는 함수이고, 집계 함수와는 다르다.(집계 함수는 SELECT문에서 사용된다.)
-- 특정 상품 매출의 총계와 소계를 구할 때, 해당 함수를 사용하는 것이 매우 용이하다.

-- 'ROLLUP(컬럼명1, 컬럼명2, 컬럼명3, ……)'은 GROUP BY 구문에서 사용되는 함수이고, 해당 함수의 인자로 들어온 컬럼을 오른쪽부터 하나씩 빼면서 GROUP을 만든다.
-- 인자로 들어온 컬럼을 오른쪽부터 하나씩 빼면서 GROUP을 만들기 때문에, 해당 함수의 인자 순서는 매우 중요하다.
SELECT DEPTNO, AVG(SAL) FROM NEW_EMP GROUP BY ROLLUP(DEPTNO);
SELECT DEPTNO, JOB, SUM(SAL) AS 합계 FROM NEW_EMP GROUP BY ROLLUP(DEPTNO, JOB);

-- 'GROUPING SETS(컬럼명1, 컬럼명2, 컬럼명3, ……)'은 인자로 들어온 컬럼의 순서와 관계없이 개별적으로 GROUP 처리한다.(해당 특징이 'ROLLUP()'과 매우 다른 점이다.)
-- 'GROUPING SET'이 아니라 'GROUPING SETES'이다.
SELECT DEPTNO, JOB, SUM(SAL) FROM NEW_EMP GROUP BY GROUPING SETS(DEPTNO, JOB);

-- 'CUBE(컬럼명1, 컬럼명2, 컬럼명3, ……)'은 인자로 들어온 컬럼들로 만들 수 있는 모든 GROUP을 도출한다.

```
SELECT DEPTNO, JOB, SUM(SAL) FROM NEW_EMP GROUP BY CUBE(DEPTNO, JOB);
```

-- 'WINDOW 함수'는 분석함수 중에서 'WINDOWING 절'을 사용하는 함수를 뜻한다.
-- **WINDOW 함수는 특정 인스턴스들의 집합을 가지고 이루어지는 함수이다.**
-- **WINDOW 함수의 종류**(기존에 존재하는 함수도 있고, 새롭게 WINDOW 함수용으로 추가된 함수도 있음) : **집계 함수**, **순위 함수**, **행 순서 관련 함수**
-- WINDOW 함수에선 '**OVER()**' **문구가 키워드로 무조건 포함되어야 한다.**
-- **WINDOW 함수는 GROUP BY 구문과 병행하여 사용할 수 없다.** WINDOW 함수의 PARTITION BY 구문과 GROUP BY 구문은 둘 다 파티션을 분할한다는 의미에서는 유사하다.
-- 그러나 **WINDOW 함수로 인해 반환되는 인스턴스의 수는 줄어들지 않는다. 이 점이 일반 집계 함수와 다르다.**

-- WINDOW 함수의 형식 :
SELECT WINDOW_FUNCTION (ARGUMENTS)
OVER ([PARTITION BY 컬럼] [ORDER BY 절] [WINDOWING 절])
FROM 테이블 명

-- **PARTITION BY 절** : 전체 집합을 기준에 의해 소그룹(PARTITION)으로 나눈다. 해당 절을 기재하지 않으면, 전체 집합이 하나의 PARTITION으로 선정된다.
-- **ORDER BY 절** : PARTITION 내 데이터들을 지정한 열에 대해서 정렬한다. 이로써 SELECT 절의 'ORDER BY'구문을 굳이 사용할 필요가 없어진다.
-- **WINDOWING 절** : **함수의 대상이 되는 인트턴스 범위를 강력하게 지정할 수 있다.** '**ROWS**'는 WINDOW 크기를 물리적인 단위로 행의 집합을 지정하고, '**RANGE**'는 논리적인 주소에 의해 행 집합을 지정하는데, 해당 절에서 둘 중의 하나를 선택해서 사용할 수 있다. 그리고 default로 'RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW'를 실행한다. 또한, **해당 절은 ORDER BY절에 종속한다. 그래서 ORDER BY절 없이 WINDOWING 절을 기재하면, 오류가 발생한다.**
-- PARTITION BY구문과 ORDER BY구문을 각각 사용할 수 있고, 해당 두 개를 조합해서 사용할 수도 있다.

-- 'ROWS'와 'RANGE'와 함께 사용되는 구문 또는 키워드 : BETWEEN ~ AND 절, UNBOUNDED PRECEDING, UNBOUNDED FOLLOWING, CURRENT ROW
-- BETWEEN ~ AND 절 : WINDOW의 시작과 끝 위치를 지정한다.

-- UNBOUNDED PRECEDING : WINDOW의 시작 위치가 첫 번째 인스턴스임을 의미한다.
-- UNBOUNDED FOLLOWING : WINDOW의 마지막 위치가 마지막 인스턴스임을 의미한다.
SELECT DEPTNO, ENAME, SAL, SUM(SAL) OVER (PARTITION BY DEPTNO ORDER BY SAL ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) FROM NEW_EMP;

-- CURRENT ROW : WINDOW의 시작 위치가 현재 인스턴스임을 의미한다.
SELECT DEPTNO, ENAME, SAL, SUM(SAL) OVER (PARTITION BY DEPTNO ORDER BY SAL ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM NEW_EMP

-- WINDOW 함수는 특정 컬럼과 파티션에 대해서 순위를 계산할 수 있는 함수를 제공한다.
-- 순위와 관련된 WINDOW함수 종류 : RANK(), DENSE_RANK(), ROW_NUMBER()
-- **순위와 관련된 WINDOW함수를 호출할 때는, 다른 WINDOW함수와는 다르게 매개변수 없이 호출한다.**
-- 또한, **ORDER BY절을 반드시 기재해야 하고 WINDOWING절은 기재할 필요가 없다.**

-- RANK() : 특정 컬럼 및 파티션에 대해서 순위를 계산한다. 동일한 순위는 동일한 값이 부여된다. 만약 2등이 두 명 나왔다면 3등은 없다.
SELECT DEPTNO, ENAME, SAL, RANK() OVER(ORDER BY SAL)FROM NEW_EMP;

-- DENSE_RANK() : 동일한 순위를 하나의 건수로 계산한다. 그래서 만약 2등이 두 명 나왔더라도 3등이 존재한다.
SELECT DEPTNO, ENAME, SAL, DENSE_RANK() OVER(ORDER BY SAL)FROM NEW_EMP;

-- ROW_NUMBER() : 동일한 순위에 대해서 고유의 순위를 부여한다. 즉, 특정 등수가 중복으로 나오지 않는다.
SELECT DEPTNO, ENAME, SAL, ROW_NUMBER() OVER(ORDER BY SAL)FROM NEW_EMP;

-- 행 순서 관련 WINDOW 함수는 상위 행의 값을 하위에 출력하거나 하위 행의 값을 상위 행에 출력할 수 있다.
-- 특정 위치의 행을 출력할 수 있다.
-- 행 순서 관련 WINDOW 함수 종류 : FIRST_VALUE(), LAST_VALUE(), LAG(), LEAD()

-- FIRST_VALUE(컬럼명) : 파티션 내 특정 컬럼에서 가장 처음에 나온 값을 반환한다.
SELECT DEPTNO, ENAME, SAL, FIRST_VALUE(SAL) OVER(PARTITION BY DEPTNO ORDER BY SAL) AS FIRST_VALUE FROM NEW_EMP;

-- LAST_VALUE(컬럼명) : 파티션 내 특정 컬럼에서 가장 나중에 나오는 값을 반환한다.
SELECT DEPTNO, ENAME, SAL, LAST_VALUE(SAL) OVER(PARTITION BY DEPTNO ORDER BY SAL) AS FIRST_VALUE FROM NEW_EMP;

-- LAG(컬럼명, 현재 인스턴스보다 앞에 존재하는 인스턴스의 위치[default로 1이 설정되어 있음]) : 파티션 내 이전 인스턴스를 반환한다.
SELECT DEPTNO, ENAME, SAL, LAG(SAL) OVER(ORDER BY SAL) AS PREQUEL_SAL FROM NEW_EMP;

-- LEAD(컬럼명, 현재 인스턴스보다 뒤에 존재하는 인스턴스의 위치[default로 1이 설정되어 있음]) : 파티션 내 이후 인스턴스를 반환한다.
SELECT DEPTNO, ENAME, SAL, LEAD(SAL, 2) OVER(ORDER BY SAL) AS PREQUEL_SAL FROM NEW_EMP;

-- 집계와 관련된 WINDOW 함수 종류 : SUM, AVG, COUNT, MAX, MIN
SELECT DEPTNO, ENAME, SAL, SUM(SAL) OVER (PARTITION BY DEPTNO) FROM NEW_EMP;

-- 비율 관련 WINDOW함수는 누적 백분율, 순서별 백분율, 파티션을 N분으로 분할한 결과 등을 조회할 수 있다.
-- 비율 관련 WINDOW함수 종류 : CUME_DIST(), PERCENT_RANK(), NTILE(), RATIO_TO_REPORT()
-- 순위 관련 WINDOW함수와 마찬가지로, 비율 관련 WINDOW함수를 호출할 때는 매개변수 없이 호출한다.

-- CUME_DIST() : 파티션에서 행의 순서별 누적백분율을 반환한다. **값이 아닌 행의 순서에 따라 누적백분율을 반환하는 것이 핵심이다.**

```
SELECT DEPTNO, ENAME, SAL, CUME_DIST() OVER(ORDER BY SAL) AS 누적백분율 FROM NEW_EMP;
```

-- PERCENT_RANK() : **파티션에서 제일 먼저 나온 것을 0으로** 제일 늦게 나온 것을 1로 하여, 값이 아닌 행의 순서별 백분율을 조회한다.

```
SELECT DEPTNO, ENAME, SAL, PERCENT_RANK() OVER(ORDER BY SAL DESC) AS 퍼센트순위 FROM NEW_EMP;
```

-- NTILE() : 파티션별로 전체 건수를 ARGUMENT 값으로 N 등분한 결과를 반환한다.

```
SELECT DEPTNO, ENAME, SAL, NTILE(4) OVER(ORDER BY SAL DESC) FROM NEW_EMP;
```

-- RATIO_TO_PERCENT : 파티션 내 전체 SUM(컬럼)에 대한 행 별 컬럼값의 백분율을 소수점까지 조회한다.

-- 옵티마이저 : SQL 개발자가 SQL을 작성하여 실행할 때, 옵티마이저는 SQL을 어떻게 실행할 것인지를 계획하게 된다. 즉, **옵티마이저는 SQL 실행 계힉을 수립하고 SQL을 실제로 실행하는 역할을 수행한다.**
-- 동일한 결과가 나오는 SQL도 어떻게 실행하느냐에 따라서 성능이 달라진다.

-- 옵티마이저의 실행 방법
-- 1. 개발자가 SQL을 실행하면, '파싱(Parsing)'이 수행된다. 이때, SQL의 문법 검사 및 구문 분석을 수행한다.
-- 2. 파싱이 완료되면, 옵티마이저가 **규칙 기반** 혹은 **비용 기반**으로 실행 계획을 수립한다.
-- 3. 옵티마이저는 기본적으로 **비용 기반 옵티마이저**를 활용해서 최적의 실행 계획을 수립한다.
-- 4. 실행 계획 수립이 완료되면, 최종적으로 SQL을 실행하고 실행이 완료되면 데이터를 인출(Fetch)한다.

-- **규칙 기반 옵티마이저** : 실행 계획을 수립할 때, 15개의 우선순위를 기준으로 실행 계획을 수립한다.
-- **비용 기반 옵티마이저** : 오브젝트 통계 및 시스템 통계를 사용해서 총비용을 계산한다(총 비용 : SQL문을 실행하기 위해서 예상되는 소요 시간 혹은 자원의 사용량). 이후 총 비용이 적은 쪽으로 실행 계획을 수립한다.

-- 옵티마이저가 비효율적으로 실행 계획을 세웠을 때, SQL개발자가 해당 실행 계획을 수정해야한다. 이때 'HINT'를 기재하여 옵티마이저에게 실행 계획을 변경하도록 요청할 수 있다. 'HINT'는 SELECT 절의 맨 앞에 기재하는 것이 원칙이다.
-- 앞서 옵티마이저는 비용 기반 옵티마이저를 활용한다고 설명했다. 옵티마이저가 규칙 기반 옵티마이저를 활용하도록 요청하려면, SELECT 절에 '/*+ RULE */'이라고 기재하면 된다.

-- 데이터베이스 구문, 함수명, 컬럼명, 테이블명 등은 대소문자 구분을 하지 않지만, 데이터 값은 대소문자 구분을 한다.

-- 여러 테이블을 JOIN한 결과에서 특정 컬럼이 어느 테이블로부터 나온 것인지를 파악하는 게 매우 중요하다.

-- 'LISTAGG()'함수를 사용하면 특정 레코드들을 한 개의 그룹(리스트)으로 묶어서, 하나의 데이터 값으로 만들 수 있다.
-- **해당 함수는 집계함수에 속함으로, GROUP BY구문에 의존적이다.**
-- LISTAGG() 함수의 형식 : LISTAGG([합칠 컬럼명], [구분자]) WITHIN GROUP(ORDER BY [정렬 컬럼명])
-- WITHIN GROUP 구문은 그룹 내 데이터들을 정렬할 기준을 설정하는 것이다.

-- Oracle DB에는 'text' type은 없다. 'varchar2' type의 byte 수를 최대로 늘려 사용하던지, 'long' type 또는 다른 type으로 사용하면 된다.

-- GROUP BY절에 기재된 컬럼에 NULL이 있을 때, 해당 NULL에 대한 그룹도 형성된다.

```
SELECT
    team_no,
    COUNT(team_no)
FROM
    GROUP_BY_TEST
GROUP BY
    team_no;

SELECT
    team_no,
    COUNT(*)
FROM
    GROUP_BY_TEST
GROUP BY
    team_no;
```

-- SQL은 일반 프로그래밍언어처럼 순서를 '0'부터 세지 않고, '1'부터 센다.