[2주차] 제2장 SQL 문장의 구성요소 - 박민영

제출일 : 2022년 11월 11일 금요일

작성자 : 박민영

---

1. char vs varchar

1.1 설명

char

- 문자열을 저장하는 데이터 타입으로, 항상 고정된 문자열 길이를 갖는다.
- 문자열은 최대 2,000byte나 2,000자까지 선언할 수 있다.

varchar

- 문자열을 저장하는 데이터 타입이다. 단, CHAR 타입과 다른 점은 문자열 길이가 일정하지 않은 **가변 길이를 갖는다**는 것이다.
- 문자열은 최대 65,532byte나 65,532자까지 선언할 수 있다.

1.2 시나리오 수행

시나리오 내용
1. char, varchar 속성의 컬럼을 가진 TABLE 생성
2. 같은 문자열을 입력
3. 각 컬럼에 저장된 해당 문자열의 길이를 출력
4. 각 칼럼에 저장된 해당 문자열의 공백을 다른 문자로 치환하여 출력

1.3 시나리오 수행 내역

1. char, varchar 속성의 컬럼을 가진 TABLE 생성

```
create table testDB ( c1 char(10), c2 varchar(10) );
```

```
SQL> create table testDB ( c1 char(10), c2 varchar(10) );
Table 'TESTDB' created.
```

2. 같은 문자열을 입력

```
insert into testDB values ('hello', 'hello');
```

```
SQL> insert into testDB values ('hello', 'hello');

1 row inserted.
```

3. 각 컬럼에 저장된 해당 문자열의 길이를 출력

```
select length(c1), length(c2) from testDB where c1 = 'hello';
```

```
SQL> select length(c1), length(c2) from testDB where c1 = 'hello';

LENGTH(C1) LENGTH(C2)
---------- ----------
        10          5

1 row selected.
```

- char로 지정한 컬럼 c1은 길이가 10으로 지정되었다.
- varchar로 지정한 컬럼 c2는 길이가 5로 지정되었다.

4. 각 칼럼에 저장된 해당 문자열의 공백을 다른 문자로 치환하여 출력

```
select replace(c1, ' ', '_'), replace(c2, ' ', '*')
       from testDB where c1 = 'hello';
```

```
SQL> select replace(c1, ' ', '.'), replace(c2, ' ', '.') from testDB whe
re c1 = 'hello';

REPLACE(C1,'','.') REPLACE(C2,'','.')
------------------ ------------------
hello.....         hello

1 row selected.
```

- char로 지정한 컬럼 c1은 'hello ' 로 저장된 것을 확인할 수 있다.
- varchar로 지정한 컬럼 c2는 'hello' 로 저장된 것을 확인할 수 있다.

1.sh

```
-- char vs varchar
SET ECHO ON

create table testDB ( c1 char(10), c2 varchar(10) );
insert into testDB values ('hello', 'hello');
select length(c1), length(c2) from testDB where c1 = 'hello';
select replace(c1, ' ', '.'), replace(c2, ' ', '.') from testDB where c1 = 'hello';


EXIT
```

---

2. NUMBER의 정밀도를 초과하는 자리수의 데이터 저장

2.1 설명

NUMBER

- 정수 또는 실수를 저장하는 데이터타입이다.
- NUMBER 타입이 표현할 수 있는 수의 범위는 음양으로 절댓값이 1.0×10-130 보다 크거나 같고, 1.0×10126 보다 작은 38자리의 수를 표현할 수 있으며 0과 ±무한대를 포함한다.
- NUMBER 타입은 선언할 때 다음과 같이 자릿수를 의미하는 정밀도와 스케일을 함께 정의할 수 있다.

  `NUMBER[(precision[,scale])]`

  - `precision`
    - 정밀도, 유효 숫자의 최대 자릿수
    - 가장 왼쪽의 0이 아닌 숫자부터 가장 오른쪽의 신뢰할 수 있는 숫자까지의 자리수
    - **정밀도를 초과하는 자릿수의 데이터는 저장할 수 없다.**
  - `scale`
    - 스케일, 소수점 아래 가장 오른쪽 유효숫자까지의 자릿수
    - 스케일은 생략할 수 있으며, 생략한 경우 0으로 선언한 것과 동일하다. 이 경우 정수를 표현한다.
    - 스케일을 초과하는 데이터는 반올림을 수행한다.

2.2 시나리오 수행

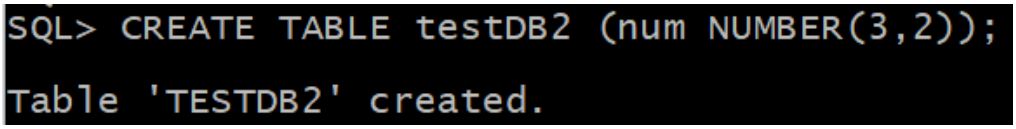
시나리오 내용
1. NUMBER 속성의 컬럼을 가진 TABLE 생성 (총 3자리 수, 소수점 이하 2자리 수)
2. precision 값, scale 값보다 모두 큰 데이터를 입력하는 경우 (총 8자리 수, 소수점 이하 4

자리 수)
3. precision 값보다 더 크고 scale 값과 일치하는 데이터를 입력하는 경우 (총 5자리 수, 소수점 이하 2자리 수)
4. precision 값과 일치하고 scale 값과 보다 작은 데이터를 입력하는 경우 (총 3자리 수, 소수점 이하 0자리 수)
5. precision 값과 일치하고 scale 값과 일치하는 데이터를 입력하는 경우 (총 3자리 수, 소수점 이하 2자리 수)

2.3 시나리오 수행 결과

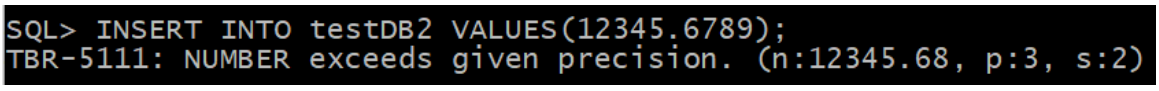
1. NUMBER 속성의 컬럼을 가진 TABLE 생성 (총 3자리 수, 소수점 이하 2자리 수)

```
CREATE TABLE testDB2 (num NUMBER(3,2));
```



2. precision 값, scale 값보다 모두 큰 데이터를 입력하는 경우 (총 9자리 수, 소수점 이하 4자리 수)

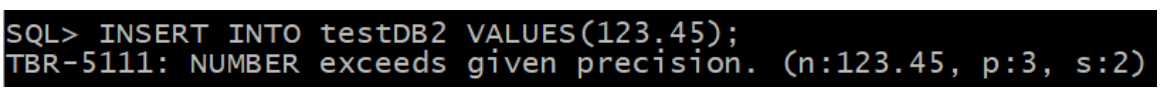
```
INSERT INTO testDB2 VALUES(12345.6789);
```



입력받은 데이터의 총 자리수가 9자리여서 오류가 발생한다. scale의 경우에는 더 큰 자리수가 들어왔지만 오류가 발생하지 않고 반올림해서 정해진 scale로 맞게 입력되려고 하였다.

3. precision 값보다 더 크고 scale 값과 일치하는 데이터를 입력하는 경우 (총 5자리 수, 소수점 이하 2자리 수)

```
INSERT INTO testDB2 VALUES(123.45);
```



입력받은 데이터의 총 자리수가 5자리여서 오류가 발생한다.

4. precision 값과 일치하고 scale 값과 보다 작은 데이터를 입력하는 경우 (총 3자리 수, 소수점 이하 0 자리 수)

```
INSERT INTO testDB2 VALUES(123);
```

```
SQL> INSERT INTO testDB2 VALUES(123);
TBR-5111: NUMBER exceeds given precision. (n:123, p:3, s:2)
```

입력받은 데이터의 총 자리수가 3자리여서 precision에서는 문제가 발생하지 않았지만, scale의 경우에는 정해진 scale만큼 충분히 들어오지 않았기 때문에 오류가 발생했다.

5. precision 값과 일치하고 scale 값과 일치하는 데이터를 입력하는 경우 (총 3자리 수, 소수점 이하 2~3자리 수)

```
INSERT INTO testDB2 VALUES(1.23);
INSERT INTO testDB2 VALUES(0.123);
SELECT num FROM testDB2;
```

```
SQL> INSERT INTO testDB2 VALUES(1.23);

1 row inserted.

SQL> INSERT INTO testDB2 VALUES(0.123);

1 row inserted.

SQL> SELECT num FROM testDB2;

       NUM
----------
      1.23
       .12

2 rows selected.
```

1.23과 0.123는 총 자리수가 3이어서 precision 값과 일치하기 때문에 문제가 발생하지 않는다. scale의 경우에는 정해진 값보다 더 크게 들어와도 반올림하여 처리하기 때문에 정해진 값 이상만 들어오면 된다. 따라서 1.23과 .12 가 정상적으로 삽입될 수 있다.

2.sh

```
CREATE TABLE testDB2 (num NUMBER(3,2));
INSERT INTO testDB2 VALUES(12345.6789);
```

```
INSERT INTO testDB2 VALUES(123.45);
INSERT INTO testDB2 VALUES(123);
INSERT INTO testDB2 VALUES(1.23);
INSERT INTO testDB2 VALUES(0.123);
SELECT num FROM testDB2;
```

---

3. 간격형 데이터 입력 조건

3.1 설명

- 시간이나 날짜 사이의 간격을 저장하는 데이터 타입이다.
- **INTERVAL YEAR TO MONTH**
  - 연도와 월을 이용하여 시간 간격을 표현
  - `INTERVAL YEAR [(year_precision)] TO MONTH`
    - `(year_precision)` : 연도 단위의 자릿수 (기본값: 2)
- **INTERVAL DAY TO SECOND**
  - 일, 시, 분, 초를 이용하여 시간 간격을 표현
  - `INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]`
    - `day_precision` : 일 단위 자릿수 (기본값 : 2)
    - `(fractional_seconds_precision)` : 초 단위의 소수점 자릿수 (0~9 사이의 값 사용, 기본값 : 6)

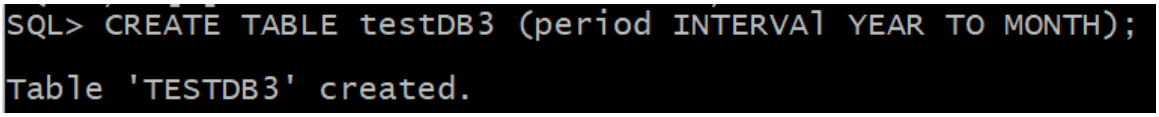
3.2 시나리오 수행

시나리오 내용
1. INTERVAl YEAR TO MONTH 속성의 컬럼을 가진 TABLE 생성
2. INTERVAl YEAR TO MONTH의 조건을 만족해서 테이블에 값이 정상적으로 들어가는 경우
3. INTERVAl YEAR TO MONTH의 조건을 만족하지 않아서 테이블에 값이 들어가지 않고 오류가 발생하는 경우
4. INTERVAl DAY TO SECOND 속성의 컬럼을 가진 TABLE 생성
5. INTERVAl DAY TO SECOND의 조건을 만족해서 테이블에 값이 정상적으로 들어가는 경우
6. INTERVAl DAY TO SECOND의 조건을 만족하지 않아서 테이블에 값이 들어가지 않고 오류가 발생하는 경우

3.3 시나리오 수행 결과

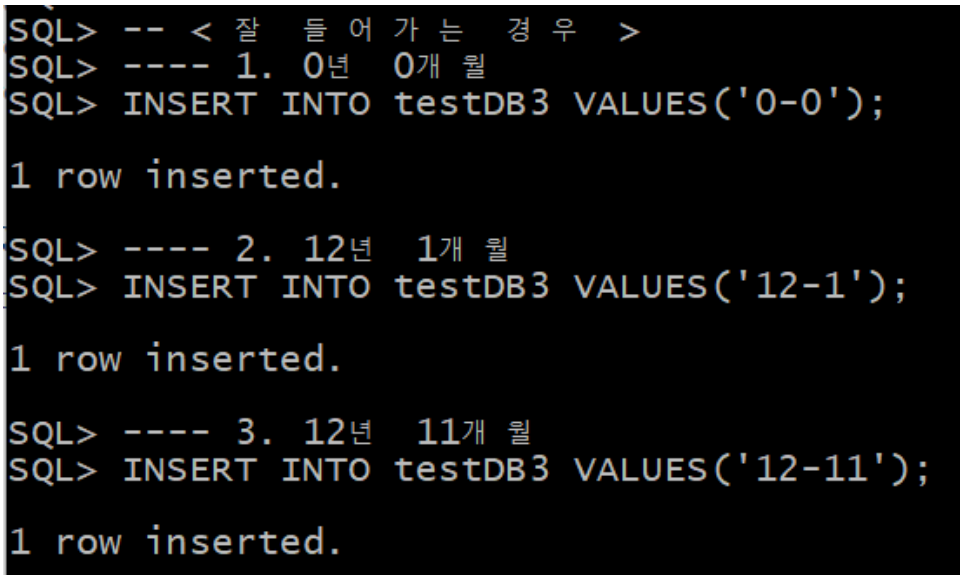
1. INTERVAl YEAR TO MONTH 속성의 컬럼을 가진 TABLE 생성

```
CREATE TABLE testDB3 (period INTERVAl YEAR TO MONTH);
```



2. INTERVAl YEAR TO MONTH의 조건을 만족해서 테이블에 값이 정상적으로 들어가는 경우

```
-- < 잘 들어가는 경우 >
---- 1. 0년 0개월
INSERT INTO testDB3 VALUES('0-0');
---- 2. 12년 1개월
INSERT INTO testDB3 VALUES('12-1');
---- 3. 12년 11개월
INSERT INTO testDB3 VALUES('12-11');
```



- year_precision의 기본값이 2이기 때문에 2자리까지의 연도 데이터들은 정상적으로 삽입된다.
- month는 연도가 같이 들어가면 0 ~ 11개월까지의 값이 들어가야 하기 때문에 0 ~ 11 값까지의 데이터들은 정상적으로 삽입된다.

3. INTERVAl YEAR TO MONTH의 조건을 만족하지 않아서 테이블에 값이 들어가지 않고 오류가 발생하는 경우

```
-- < 오류가 뜨는 경우 >
---- 4. 123년 1개월
INSERT INTO testDB3 VALUES('123-1');
```

```
---- 5. 12년 12개월
INSERT INTO testDB3 VALUES('12-12');
---- 6. 12년 123개월
INSERT INTO testDB3 VALUES('12-123');
```

```
SQL> -- < 오 류 가  뜨 는  경 우  >
SQL> ---- 4. 123년  1개 월
SQL> INSERT INTO testDB3 VALUES('123-1');
TBR-5064: INTERVAL exceeds given precision value.

SQL> ---- 5. 12년  12개 월
SQL> INSERT INTO testDB3 VALUES('12-12');
TBR-5060: Month value is out of range.

SQL> ---- 6. 12년  123개 월
SQL> INSERT INTO testDB3 VALUES('12-123');
TBR-5060: Month value is out of range.
```

연도가 2자리수가 넘어가거나, 개월 수가 12를 초과하는 경우에는 오류가 발생한다.

```
SELECT * FROM testDB3;
```

```
SQL> SELECT * FROM testDB3;

PERIOD
---------------
+00-00
+12-01
+12-11

3 rows selected.
```

2.에서 입력한 데이터만 들어간 것을 확인할 수 있다.

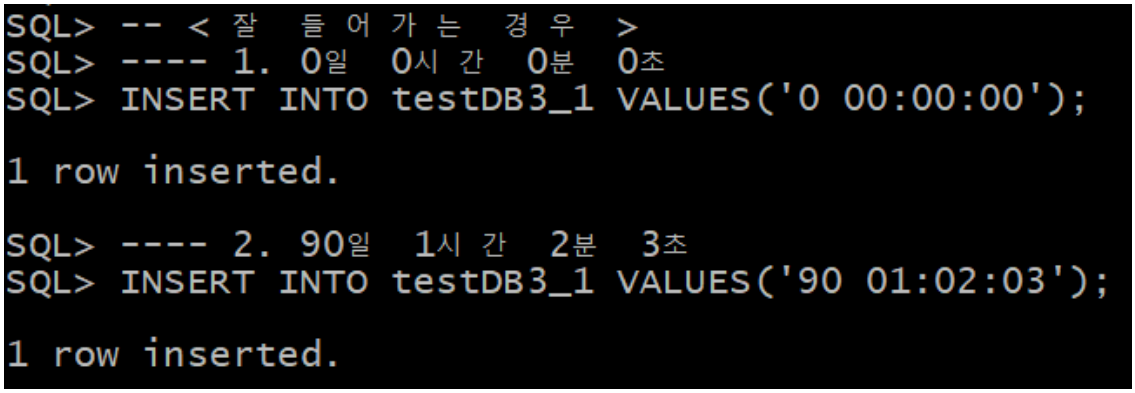
4. INTERVAl DAY TO SECOND 속성의 컬럼을 가진 TABLE 생성

```
CREATE TABLE testDB3_1 (period INTERVAl DAY TO SECOND);
```

```
SQL> CREATE TABLE testDB3_1 (period INTERVAl DAY TO SECOND);

Table 'TESTDB3_1' created.
```

5. INTERVAl DAY TO SECOND의 조건을 만족해서 테이블에 값이 정상적으로 들어가는 경우

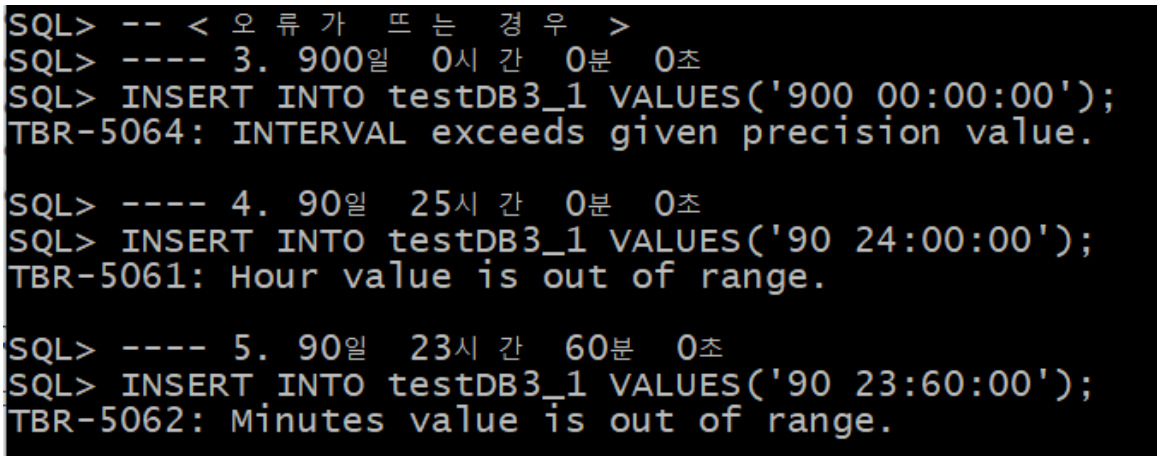
```
-- < 잘 들어가는 경우 >
---- 1. 0일 0시간 0분 0초
INSERT INTO testDB3_1 VALUES('0 00:00:00');
---- 2. 90일 1시간 2분 3초
INSERT INTO testDB3_1 VALUES('90 01:02:03');
```



일 수 기본값이 2자리이기 때문에, 0일 이상의 1~2자리 수까지 들어가면 정상적으로 삽입된다. 시간의 경우에는 시는 0~23, 분은 0~59, 초는 0~59까지의 값이 들어오면 정상적으로 삽입된다.

6. INTERVAl DAY TO SECOND의 조건을 만족하지 않아서 테이블에 값이 들어가지 않고 오류가 발생하는 경우

```
-- < 오류가 뜨는 경우 >
---- 3. 900일 0시간 0분 0초
INSERT INTO testDB3_1 VALUES('900 00:00:00');
---- 4. 90일 25시간 0분 0초
INSERT INTO testDB3_1 VALUES('90 24:00:00');
---- 5. 90일 23시간 60분 0초
INSERT INTO testDB3_1 VALUES('90 23:60:00');
```



일 수가 2자리수가 넘어가거나, 시간의 경우 시는 24, 분과 초는 60을 초과하는 데이터가 들어오면 오류가 발생하면서 삽입되지 않는다.

```
SELECT * FROM testDB3_1;
```

```
SQL> SELECT * FROM testDB3_1;

PERIOD
------------------------------
+00 00:00:00.000000
+90 01:02:03.000000

2 rows selected.
```

5.에서 입력한 데이터만 들어간 것을 확인할 수 있다.

3.sh

```
SET ECHO ON

/* [1] INTERVAL YEAR TO MONTH */
CREATE TABLE testDB3 (period INTERVAl YEAR TO MONTH);

-- < 잘 들어가는 경우 >
---- 1. 0년 0개월
INSERT INTO testDB3 VALUES('0-0');
---- 2. 12년 1개월
INSERT INTO testDB3 VALUES('12-1');
---- 3. 12년 11개월
INSERT INTO testDB3 VALUES('12-11');

-- < 오류가 뜨는 경우 >
---- 4. 123년 1개월
INSERT INTO testDB3 VALUES('123-1');
---- 5. 12년 12개월
INSERT INTO testDB3 VALUES('12-12');
---- 6. 12년 123개월
INSERT INTO testDB3 VALUES('12-123');

SELECT * FROM testDB3;


/* [2] INTERVAL DAY TO SECOND */
CREATE TABLE testDB3_1 (period INTERVAl DAY TO SECOND);

-- < 잘 들어가는 경우 >
---- 1. 0일 0시간 0분 0초
INSERT INTO testDB3_1 VALUES('0 00:00:00');
---- 2. 90일 1시간 2분 3초
INSERT INTO testDB3_1 VALUES('90 01:02:03');

-- < 오류가 뜨는 경우 >
---- 3. 900일 0시간 0분 0초
INSERT INTO testDB3_1 VALUES('900 00:00:00');
---- 4. 90일 25시간 0분 0초
INSERT INTO testDB3_1 VALUES('90 24:00:00');
---- 5. 90일 23시간 60분 0초
INSERT INTO testDB3_1 VALUES('90 23:60:00');
```

```
SELECT * FROM testDB3_1;

EXIT
```

---

4. DATE 리터럴 trunc하기

4.1 설명

DATE 리터럴

- 날짜와 시간 정보를 표현하는 날짜형 리터럴
- 특별한 속성 - 세기, 년, 월, 일, 시, 분, 초의 특별한 속성을 가지고 있다.
- DATE 리터럴을 비교할 때는 리터럴의 시간정보가 포함된 에러인지 확인이 필요하다. 한쪽에만 시간 정보가 있고 다른 쪽에는 시간 정보가 없을 경우 두 날짜가 같다고 비교하기 위해서는 시간 정보를 제거하고 비교해야 하는데 이때 TRUNC 함수를 사용하면 된다.

TRUNC(date)

- date를 format에 명시된 단위로 버림한 결과를 반환하는 함수
- 구성요소
  - date : 날짜를 반환하는 임의의 연산식
  - format : 버림 단위를 명시하는 형식 문자열, format이 명시되지 않는다면 'DD' 형식 문자열을 이용하여 가장 가까운 날로 버림이 된다. format으로 'YEAR', 'MONTH', 'DAY' 등을 사용할 수 있다.

4.2 시나리오 수행

시나리오 내용
1. 날짜만 출력하는 경우
2. 연도만 남기고 날짜는 모두 1월 1일로 초기화하는 경우
3. 해당 날짜에 대한 세기 맨 처음 일자를 출력하는 경우

4.3 시나리오 수행 결과

1. 날짜만 출력하는 경우

```
SELECT TRUNC(TO_DATE('2022/11/10 12:30:14', 'YYYY/MM/DD HH24:MI:SS')) FROM DUAL;
```

```
SQL> -- 날 짜 만  출 력 함
SQL> SELECT TRUNC(TO_DATE('2022/11/10 12:30:14', 'YYYY/MM/DD HH24:MI:SS'
)) FROM DUAL;

TRUNC(TO_DATE('2022/11/1012:30:14','YYYY/MM/DDHH24:MI:SS'))
-----------------------------------------------------------
2022/11/10

1 row selected.
```

2. 연도만 남기고 날짜는 모두 1월 1일로 초기화하는 경우

```
SELECT TRUNC(TO_DATE('2022/11/10', 'YYYY/MM/DD'), 'YEAR') FROM DUAL;
```

```
SQL> -- 연 도 만  남 기 고  날 짜 는  절 삭 함
SQL> SELECT TRUNC(TO_DATE('2022/11/10', 'YYYY/MM/DD'), 'YEAR') FROM DUAL
;

TRUNC(TO_DATE('2022/11/10','YYYY/MM/DD'),'YEAR')
------------------------------------------------
2022/01/01

1 row selected.
```

3. 해당 날짜에 대한 세기 맨 처음 일자를 출력하는 경우

```
SELECT TO_CHAR(TRUNC(TO_DATE('2022/11/10', 'YYYY/MM/DD'), 'CC'), 'YYYY/MM/DD') FROM DUAL;
SELECT TO_CHAR(TRUNC(TO_DATE('-3741/11/10', 'SYYYY/MM/DD'), 'CC'), 'SYYYY/MM/DD') FROM DUAL;
```

```
SQL> -- 세 기  표 시
SQL> SELECT TO_CHAR(TRUNC(TO_DATE('2022/11/10', 'YYYY/MM/DD'), 'CC'), 'Y
YYY/MM/DD') FROM DUAL;

TO_CHAR(TRUNC(TO_DATE('2022/11/10','YYYY/MM/DD'),'CC'),'YYYY/MM/DD')
---------------------------------------------------------------------
2001/01/01

1 row selected.

SQL> SELECT TO_CHAR(TRUNC(TO_DATE('-3741/11/10', 'SYYYY/MM/DD'), 'CC'),
'SYYYY/MM/DD') FROM DUAL;

TO_CHAR(TRUNC(TO_DATE('-3741/11/10','SYYYY/MM/DD'),'CC'),'SYYYY/MM/DD')
-----------------------------------------------------------------------
-3800/01/01

1 row selected.
```

4.sh

```
SET ECHO ON

-- 날짜만 출력함
SELECT TRUNC(TO_DATE('2022/11/10 12:30:14', 'YYYY/MM/DD HH24:MI:SS')) FROM DUAL;

-- 연도만 남기고 날짜는 절삭함
SELECT TRUNC(TO_DATE('2022/11/10', 'YYYY/MM/DD'), 'YEAR') FROM DUAL;

-- 세기 표시
SELECT TO_CHAR(TRUNC(TO_DATE('2022/11/10', 'YYYY/MM/DD'), 'CC'), 'YYYY/MM/DD') FROM DUAL;
SELECT TO_CHAR(TRUNC(TO_DATE('-3741/11/10', 'SYYYY/MM/DD'), 'CC'), 'SYYYY/MM/DD') FROM DUAL;

EXIT
```

---

5. 날짜형 타입의 형식 문자열 RR vs YY

5.1 설명

날짜형 타입의 형식 문자열

- TO_CHAR 함수와 TO_DATE 함수, TO_TIMESTAMP 함수, TO_TIMESTAMP_TZ 함수에서 파라미터로 사용할 수 있다.
- 날짜형 타입에 포함된 '연, 월, 일, 시, 분, 초' 등의 값을 각각 어떤 형식으로 출력할 것인지 지정한다. 예를 들어 연도를 나타내는 형식 요소 문자열인 'YYYY'와 'YY'의 경우 연도의 마지막 네 자리 또는 두 자리만 출력하도록 한다. 즉, 2022년의 경우 각각 '2022'와 '22'를 출력한다.
- 하이픈(-) 또는 슬래시(/)를 삽입할 수 있다. 만약 형식 문자열 내에 형식 요소 이외의 문자열을 삽입하고 싶다면 큰따옴표(" ")를 이용하여 나타낸다.
- 대소문자를 구분하는 형식 요소가 있다. 예를 들어 요일을 출력하기 위한 형식 요소인 'DAY'는 요일 문자열 전체를 대문자로, 'Day'는 맨 앞 글자만 대문자로, 'day'는 전체를 소문자로 출력한다. 월요일의 경우, 각각 'MONDAY', 'Monday', 'monday'로 출력한다.

RR vs YY

1. YY
   - 2자릿수 연도를 출력한다.
2. RR
   - 두 자릿수의 연도의 입력 값에 따라 몇 세기인지 자동으로 조절한다.
   - 다른 세기의 연도 값을 더 간편하게 명시하고 저장할 수 있다.
   - **현재 시스템의 연도의 마지막 두 자리가 00 ~ 49 사이**일 경우
     - 명시한 두 자릿수 연도가 00 ~ 49 사이 : 반환되는 연도는 **현재 연도**와 앞의 두 자리가 같다.

- 명시한 두 자릿수 연도가 50 ~ 99 사이 : 반환되는 연도의 앞의 두 자리는 **현재 연도의 앞의 두 자리에 1을 뺀 값**과 같다.

- **현재 시스템의 연도의 마지막 두 자리가 50 ~ 99 사이**일 경우
  - 명시한 두 자릿수 연도가 00 ~ 49 사이 : 반환되는 연도의 앞의 두 자리는 **현재 연도의 앞의 두 자리에 1을 더한 값**과 같다.
  - 명시한 두 자릿수 연도가 50 ~ 99 사이 : 반환되는 연도는 **현재 연도**와 앞의 두 자리가 같다.

5.2 시나리오 수행

시나리오 내용
1. RR 1.1 RR - 명시한 두 자릿수 연도가 00 ~ 49 사이 1.2 RR - 명시한 두 자릿수 연도가 50 ~ 99 사이
2. YY 2.1 YY - 명시한 두 자릿수 연도가 00 ~ 49 사이 2.2 YY - 명시한 두 자릿수 연도가 50 ~ 99 사이

5.3 시나리오 수행 결과

1. RR

- 1.1 RR - 명시한 두 자릿수 연도가 00 ~ 49 사이

```
SELECT TO_CHAR(TO_DATE('20/08/13', 'RR/MM/DD'), 'YYYY') YEAR FROM DUAL;
```

```
SQL> ---- 1.1 RR - 명시한 두 자릿수 연도가 00 ~ 49 사이
SQL> SELECT TO_CHAR(TO_DATE('20/08/13', 'RR/MM/DD'), 'YYYY') YEAR FROM D
UAL;

YEAR
----
2020

1 row selected.
```

- 1.2 RR - 명시한 두 자릿수 연도가 50 ~ 99 사이

```
SELECT TO_CHAR(TO_DATE('92/02/08', 'RR/MM/DD'), 'YYYY') YEAR FROM DUAL;
```

```
SQL> ---- 1.2 RR - 명시한 두 자릿수 연도가 50 ~ 99 사이
SQL> SELECT TO_CHAR(TO_DATE('92/02/08', 'RR/MM/DD'), 'YYYY') YEAR FROM D
UAL;

YEAR
----
1992

1 row selected.
```

2. YY

- 2.1 YY - 명시한 두 자릿수 연도가 00 ~ 49 사이

```
SELECT TO_CHAR(TO_DATE('20/08/13', 'YY/MM/DD'), 'YYYY') YEAR FROM DUAL;
```

```
SQL> ---- 2.1 YY - 명시한 두 자릿수 연도가 00 ~ 49 사이
SQL> SELECT TO_CHAR(TO_DATE('20/08/13', 'YY/MM/DD'), 'YYYY') YEAR FROM D
UAL;

YEAR
----
2020

1 row selected.
```

- 2.2 YY - 명시한 두 자릿수 연도가 50 ~ 99 사이

```
SELECT TO_CHAR(TO_DATE('98/12/25', 'YY/MM/DD'), 'YYYY') YEAR FROM DUAL;
```

```
SQL> ---- 2.2 YY - 명시한 두 자릿수 연도가 50 ~ 99 사이
SQL> SELECT TO_CHAR(TO_DATE('98/12/25', 'YY/MM/DD'), 'YYYY') YEAR FROM D
UAL;

YEAR
----
2098

1 row selected.
```

5.sh

```
SET ECHO ON

-- 1. RR
---- 1.1 RR - 명시한 두 자릿수 연도가 00 ~ 49 사이
SELECT TO_CHAR(TO_DATE('20/08/13', 'RR/MM/DD'), 'YYYY') YEAR FROM DUAL;
---- 1.2 RR - 명시한 두 자릿수 연도가 50 ~ 99 사이
SELECT TO_CHAR(TO_DATE('92/02/08', 'RR/MM/DD'), 'YYYY') YEAR FROM DUAL;
```

```
-- 2. YY
---- 2.1 YY - 명시한 두 자릿수 연도가 00 ~ 49 사이
SELECT TO_CHAR(TO_DATE('20/08/13', 'YY/MM/DD'), 'YYYY') YEAR FROM DUAL;
---- 2.2 YY - 명시한 두 자릿수 연도가 50 ~ 99 사이
SELECT TO_CHAR(TO_DATE('98/12/25', 'YY/MM/DD'), 'YYYY') YEAR FROM DUAL;

EXIT
```

---

6. ROWNUM와 ORDER BY 절

6.1 설명

ROWNUM

- SELECT 문장의 실행 결과로 나타나는 row에 대하여 순서대로 번호를 부여한다.
- 질의 결과로 반환되는 첫 번째 row는 ROWNUM = 1이며 두 번째 row는 ROWNUM = 2, 세 번째 row는 ROWNUM = 3, …, 등등의 값을 갖는다.
- ROWNUM은 질의를 처리하는 거의 마지막 단계에서 할당되기 때문에 같은 SELECT 문장이라 하더라도 내부적으로 어떤 단계로 질의를 처리하였는가에 따라 다른 결과를 가져올 수 있다.
- ROWNUM을 포함하는 질의가 항상 같은 결과를 반환하도록 하기 위하여 ORDER BY 절을 사용할 수 있다. 하지만, Tibero에서는 WHRER 절을 포함하는 모든 부질의를 처리한 다음에 ORDER BY 절을 처리한다. 따라서 ORDER BY 절을 이용해서 항상 같은 결과를 얻을 수는 없다.

6.2 시나리오 수행

시나리오 내용
1. 테이블 생성 및 데이터 입력
2. 매번 다른 결과가 나올 수 있는 질의
3. 항상 같은 결과를 얻는 질의
4. 하나의 row도 반환되지 않는 질의

6.3 시나리오 수행 결과

1. 테이블 생성 및 데이터 입력

```
-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE LOONA ( name VARCHAR(10), birth_year NUMBER(4) );

INSERT INTO LOONA VALUES ('Heejin', 2000);
INSERT INTO LOONA VALUES ('Hyunjin', 2000);
```

```
INSERT INTO LOONA VALUES ('Haseul', 1997);
INSERT INTO LOONA VALUES ('Yeojin', 2002);
INSERT INTO LOONA VALUES ('Bibi', 1996);
INSERT INTO LOONA VALUES ('Kimlip', 1997);
INSERT INTO LOONA VALUES ('Jinsol', 1997);
INSERT INTO LOONA VALUES ('Choiri', 2001);
INSERT INTO LOONA VALUES ('Eve', 1997);
INSERT INTO LOONA VALUES ('Chuu', 1999);
INSERT INTO LOONA VALUES ('Gowon', 2000);
INSERT INTO LOONA VALUES ('Olivia Hye', 2001);
```

```
SQL> -- 1. 테이블 생성 및 데이터 입력
SQL> CREATE TABLE LOONA ( name VARCHAR(10), birth_year NUMBER(4) );

Table 'LOONA' created.

SQL>
SQL> INSERT INTO LOONA VALUES ('Heejin', 2000);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Hyunjin', 2000);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Haseul', 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Yeojin', 2002);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Bibi', 1996);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Kimlip', 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Jinsol', 1997);

1 row inserted.
```

```
SQL> INSERT INTO LOONA VALUES ('Jinsol', 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Choiri', 2001);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Eve', 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Chuu', 1999);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Gowon', 2000);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Olivia Hye', 2001);

1 row inserted.
```

2. 매번 다른 결과가 나올 수 있는 질의

```
-- 2. 매번 다른 결과가 나올 수 있는 질의
SELECT * FROM LOONA WHERE ROWNUM <= 5 ORDER BY birth_year;
```

```
SQL> -- 2. 매 번  다 른  결 과 가  나 올  수  있 는  질 의
SQL> SELECT * FROM LOONA WHERE ROWNUM <= 5 ORDER BY birth_year;

NAME       BIRTH_YEAR
---------- ----------
Bibi             1996
Haseul           1997
Hyunjin          2000
Heejin           2000
Yeojin           2002

5 rows selected.
```

3. 항상 같은 결과를 얻는 질의

```
-- 3. 항상 같은 결과를 얻는 질의
SELECT * FROM (SELECT * FROM LOONA ORDER BY birth_year) WHERE ROWNUM <= 5;
```

```
SQL> -- 3. 항상 같은 결과를 얻는 질의
SQL> SELECT * FROM (SELECT * FROM LOONA ORDER BY birth_year) WHERE ROWNU
M <= 5;

NAME       BIRTH_YEAR
---------- ----------
Bibi             1996
Haseul           1997
Kimlip           1997
Jinsol           1997
Eve              1997

5 rows selected.
```

4. 하나의 row도 반환되지 않는 질의

```
-- 4. 하나의 row도 반환되지 않는 질의
SELECT * FROM LOONA WHERE ROWNUM > 1;
```

```
SQL> -- 4. 하나의 row도 반환되지 않는 질의
SQL> SELECT * FROM LOONA WHERE ROWNUM > 1;

0 row selected.
```

ROWNUM 값이 정해지기 전에 ROWNUM에 대한 조건식이 수행되기 때문에 위의 SELECT 문의 결과는 첫 번째 row가 ROWNUM = 1이기 때문에 조건식을 만족하지 않는다. 조건식을 만족하지 않으면 ROWNUM 카운터의 값은 변하지 않는다. 따라서 두 번째 결과 row도 ROWNUM = 1이므로 반환되지 않는다. 그 결과 하나의 row도 반환되지 않는다.

6.sh

```
SET ECHO ON

-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE LOONA ( name VARCHAR(10), birth_year NUMBER(4) );

INSERT INTO LOONA VALUES ('Heejin', 2000);
INSERT INTO LOONA VALUES ('Hyunjin', 2000);
INSERT INTO LOONA VALUES ('Haseul', 1997);
INSERT INTO LOONA VALUES ('Yeojin', 2002);
INSERT INTO LOONA VALUES ('Bibi', 1996);
INSERT INTO LOONA VALUES ('Kimlip', 1997);
INSERT INTO LOONA VALUES ('Jinsol', 1997);
INSERT INTO LOONA VALUES ('Choiri', 2001);
INSERT INTO LOONA VALUES ('Eve', 1997);
INSERT INTO LOONA VALUES ('Chuu', 1999);
INSERT INTO LOONA VALUES ('Gowon', 2000);
INSERT INTO LOONA VALUES ('Olivia Hye', 2001);

-- 2. 매번 다른 결과가 나올 수 있는 질의
SELECT * FROM LOONA WHERE ROWNUM <= 5 ORDER BY birth_year;

-- 3. 항상 같은 결과를 얻는 질의
SELECT * FROM (SELECT * FROM LOONA ORDER BY birth_year) WHERE ROWNUM <= 5;
```

```
-- 4. 하나의 row도 반환되지 않는 질의
SELECT * FROM LOONA WHERE ROWNUM > 1;

EXIT
```

---

7. LEVEL 계층 질의

7.1 설명

LEVEL

- 계층 질의를 실행한 결과에 각 로우의 트리 내 계층을 출력하기 위한 컬럼 타입
- 최상위 로우의 LEVEL 값은 1이며, 하위 로우로 갈수록 1씩 증가

CONNECT_BY_ISLEAF

- 현재 로우가 CONNECT BY 조건에 의해 정의된 트리(Tree)의 리프(Leaf)이면 1을 반환하고 그렇지 않을 경우에는 0을 반환한다.
- 이 정보는 해당 로우가 계층 구조(Hierarchy)를 보여주기 위해 확장될 수 있는지 없는지를 나타낸다.

7.2 시나리오 수행

시나리오 내용
1. 테이블 생성 및 데이터 입력
2. 순방향 전개 예제
3. 역방향 전개 예제

7.3 시나리오 수행 결과

1. 테이블 생성 및 데이터 입력

```
-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE testDB7 ( emp VARCHAR(2), mgr VARCHAR(2) );

INSERT INTO testDB7 VALUES ('A', NULL);
INSERT INTO testDB7 VALUES ('B', 'A');
INSERT INTO testDB7 VALUES ('C', 'A');
INSERT INTO testDB7 VALUES ('D', 'C');
INSERT INTO testDB7 VALUES ('E', 'C');
```

```
SQL> -- 1. 테 이 블  생 성  및  데 이 터  입 력
SQL> CREATE TABLE testDB7 ( emp VARCHAR(2), mgr VARCHAR(2) );

Table 'TESTDB7' created.

SQL>
SQL> INSERT INTO testDB7 VALUES ('A', NULL);

1 row inserted.

SQL> INSERT INTO testDB7 VALUES ('B', 'A');

1 row inserted.

SQL> INSERT INTO testDB7 VALUES ('C', 'A');

1 row inserted.

SQL> INSERT INTO testDB7 VALUES ('D', 'C');

1 row inserted.

SQL> INSERT INTO testDB7 VALUES ('E', 'C');

1 row inserted.
```

2. 순방향 전개 예제

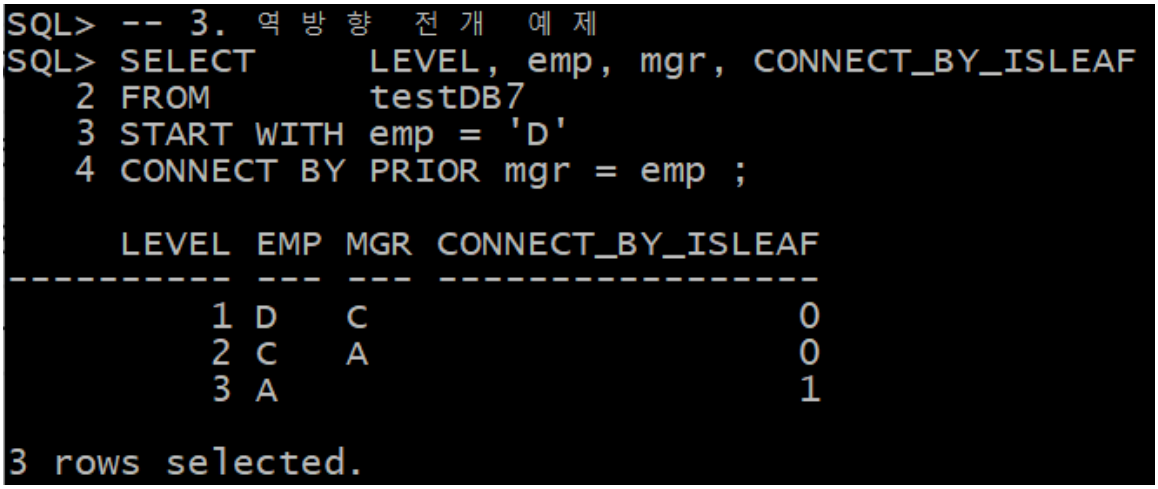
```
-- 2. 순방향 전개 예제
SELECT     LEVEL, emp, mgr, CONNECT_BY_ISLEAF
FROM       testDB7
START WITH mgr IS NULL
CONNECT BY PRIOR emp = mgr ;
```

```
SQL> -- 2. 순 방 향  전 개  예 제
SQL> SELECT     LEVEL, emp, mgr, CONNECT_BY_ISLEAF
  2 FROM       testDB7
  3 START WITH mgr IS NULL
  4 CONNECT BY PRIOR emp = mgr ;

     LEVEL EMP MGR CONNECT_BY_ISLEAF
---------- --- --- -----------------
         1 A                       0
         2 C   A                   0
         3 E   C                   1
         3 D   C                   1
         2 B   A                   1

5 rows selected.
```

관리자 > 사원 방향의 전개이기 때문에 순방향 전개의 예제 : A -> B, C -> D, E

3. 역방향 전개 예제

```
-- 3. 역방향 전개 예제
SELECT     LEVEL, emp, mgr, CONNECT_BY_ISLEAF
FROM       testDB7
START WITH emp = 'D'
CONNECT BY PRIOR mgr = emp ;
```



```
SQL> -- 3. 역 방 향  전 개  예 제
SQL> SELECT     LEVEL, emp, mgr, CONNECT_BY_ISLEAF
  2 FROM       testDB7
  3 START WITH emp = 'D'
  4 CONNECT BY PRIOR mgr = emp ;

     LEVEL EMP MGR CONNECT_BY_ISLEAF
---------- --- --- -----------------
         1 D   C                   0
         2 C   A                   0
         3 A                       1

3 rows selected.
```

사원 D부터 상위 관리자를 찾는 역방향 전개의 예제 : D -> C -> A

7.sh

```
SET ECHO ON

-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE testDB7 ( emp VARCHAR(2), mgr VARCHAR(2) );

INSERT INTO testDB7 VALUES ('A', NULL);
INSERT INTO testDB7 VALUES ('B', 'A');
INSERT INTO testDB7 VALUES ('C', 'A');
INSERT INTO testDB7 VALUES ('D', 'C');
INSERT INTO testDB7 VALUES ('E', 'C');

-- 2. 순방향 전개 예제
SELECT     LEVEL, emp, mgr, CONNECT_BY_ISLEAF
FROM       testDB7
START WITH mgr IS NULL
CONNECT BY PRIOR emp = mgr ;

-- 3. 역방향 전개 예제
SELECT     LEVEL, emp, mgr, CONNECT_BY_ISLEAF
FROM       testDB7
START WITH emp = 'D'
CONNECT BY PRIOR mgr = emp ;
```

8. NULL 에 대한 비교 조건

8.1 설명

NULL을 검사할 수 있는 비교조건

- IS NULL과 IS NOT NULL만 가능
- NULL은 데이터가 없다는 것을 의미하기 때문에 NULL과 NULL, NULL과 NULL이 아닌 다른 값을 서로 비교할 수 없다.
- 다만 DECODE 함수에서는 두 개의 NULL을 비교할 수 있다.

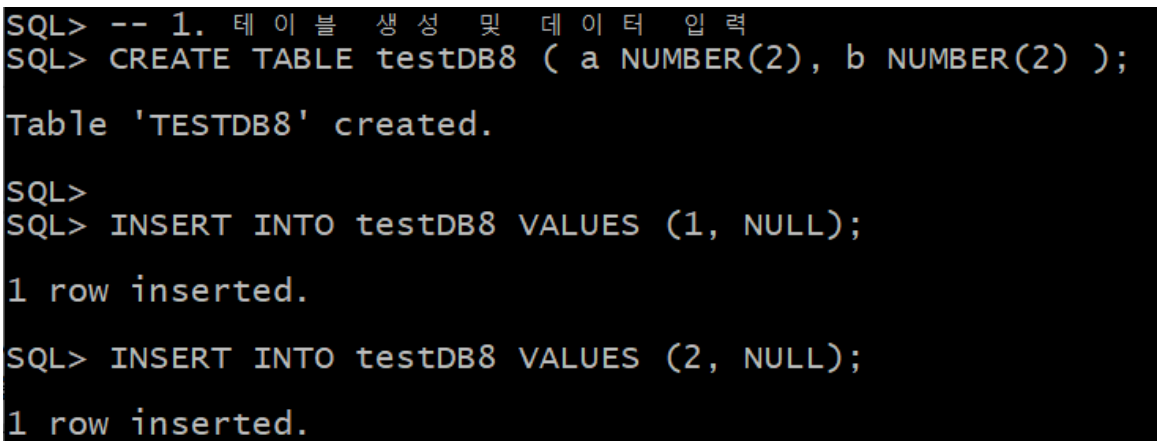
8.2 시나리오 수행

시나리오 내용
1. 테이블 생성 및 데이터 입력
2. NULL을 비교하는 방식 2.1 틀린 방식 2.2 올바른 방식
3. DECODE 함수를 이용해 두 개의 NULL을 비교하는 경우

8.3 시나리오 수행 결과

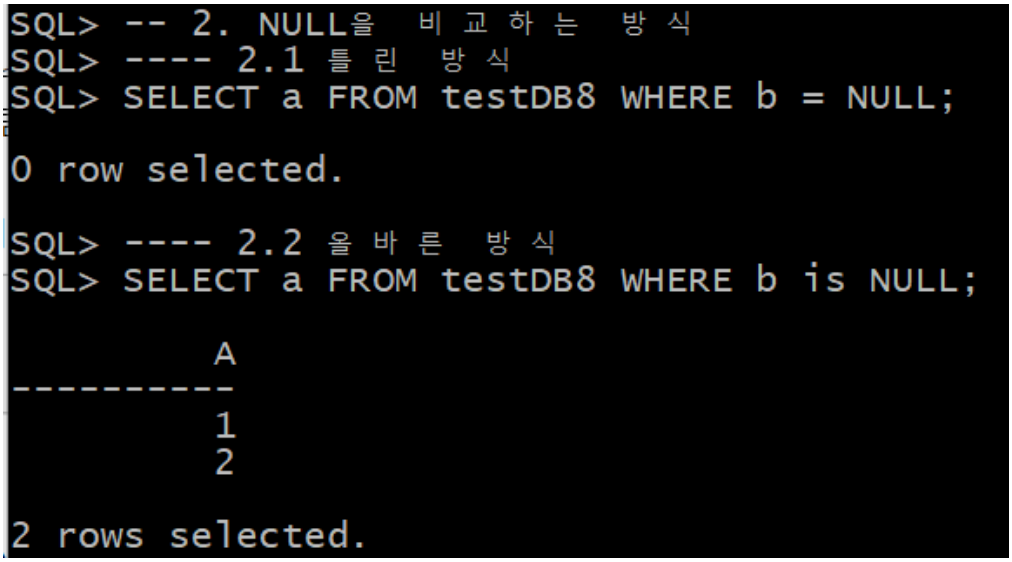
1. 테이블 생성 및 데이터 입력

```
-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE testDB8 ( a NUMBER(2), b NUMBER(2) );

INSERT INTO testDB8 VALUES (1, NULL);
INSERT INTO testDB8 VALUES (2, NULL);
```



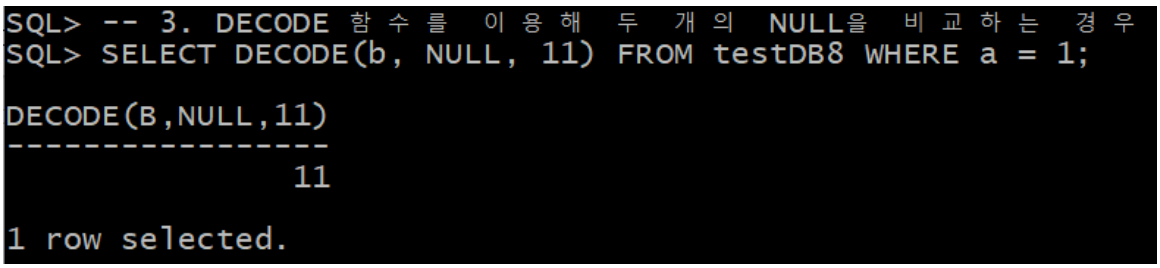
2. NULL을 비교하는 방식
    - 2.1 틀린 방식 & 2.2 올바른 방식

```
-- 2. NULL을 비교하는 방식
---- 2.1 틀린 방식
SELECT a FROM testDB8 WHERE b = NULL;
---- 2.2 올바른 방식
SELECT a FROM testDB8 WHERE b is NULL;
```



3. DECODE 함수를 이용해 두 개의 NULL을 비교하는 경우

```
-- 3. DECODE 함수를 이용해 두 개의 NULL을 비교하는 경우
SELECT DECODE(b, NULL, 11) FROM testDB8 WHERE a = 1;
```



8.sh

```
SET ECHO ON

-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE testDB8 ( a NUMBER(2), b NUMBER(2) );
```

```
INSERT INTO testDB8 VALUES (1, NULL);
INSERT INTO testDB8 VALUES (2, NULL);

-- 2. NULL을 비교하는 방식
---- 2.1 틀린 방식
SELECT a FROM testDB8 WHERE b = NULL;
---- 2.2 올바른 방식
SELECT a FROM testDB8 WHERE b is NULL;

-- 3. DECODE 함수를 이용해 두 개의 NULL을 비교하는 경우
SELECT DECODE(b, NULL, 11) FROM testDB8 WHERE a = 1;

EXIT
```

---

9. 인덱스

9.1 설명

인덱스

- 테이블과 별도의 저장공간을 이용하여 그 테이블의 특정 컬럼을 빠르게 검색 할 수 있도록 해주는 데이터 구조
- 테이블의 소유자는 어떤 컬럼에 대해 하나 이상의 인덱스를 생성 가능
- Tibero에서는 모든 테이블의 기본 키(Primary Key) 컬럼에 대해 자동으로 인덱스를 생성

9.2 시나리오 수행

시나리오 내용
1. 테이블 생성 및 데이터 입력
2. 인덱스 생성하기
3. 인덱스 조회하기
4. 일반 인덱스 삭제

9.3 시나리오 수행 결과

1. 테이블 생성 및 데이터 입력

```
-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE LOONA ( name VARCHAR(10) PRIMARY KEY , debut_order NUMBER(2), birth_year NUMBER(4) );

INSERT INTO LOONA VALUES ('Heejin', 1, 2000);
INSERT INTO LOONA VALUES ('Hyunjin', 2, 2000);
INSERT INTO LOONA VALUES ('Haseul', 3, 1997);
INSERT INTO LOONA VALUES ('Yeojin', 4, 2002);
INSERT INTO LOONA VALUES ('Bibi', 5, 1996);
INSERT INTO LOONA VALUES ('Kimlip', 6, 1997);
INSERT INTO LOONA VALUES ('Jinsol', 7, 1997);
```

```
INSERT INTO LOONA VALUES ('Choiri', 8, 2001);
INSERT INTO LOONA VALUES ('Eve', 9, 1997);
INSERT INTO LOONA VALUES ('Chuu', 10, 1999);
INSERT INTO LOONA VALUES ('Gowon', 11, 2000);
INSERT INTO LOONA VALUES ('Olivia Hye', 12, 2001);
```

```
SQL> -- 1. 테이블 생성 및 데이터 입력
SQL> CREATE TABLE LOONA ( name VARCHAR(10) PRIMARY KEY , debut_order NUMBER(2), birth_year NUMBER(4)
);

Table 'LOONA' created.

SQL>
SQL> INSERT INTO LOONA VALUES ('Heejin', 1, 2000);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Hyunjin', 2, 2000);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Haseul', 3, 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Yeojin', 4, 2002);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Bibi', 5, 1996);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Kimlip', 6, 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Jinsol', 7, 1997);

1 row inserted.
```

```
SQL> INSERT INTO LOONA VALUES ('Choiri', 8, 2001);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Eve', 9, 1997);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Chuu', 10, 1999);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Gowon', 11, 2000);

1 row inserted.

SQL> INSERT INTO LOONA VALUES ('Olivia Hye', 12, 2001);

1 row inserted.
```

2. 인덱스 생성하기

```
-- 2. 인덱스 생성하기
CREATE INDEX order_index ON LOONA(debut_order);
```

```
SQL> -- 2. 인 덱 스  생 성 하 기
SQL> CREATE INDEX order_index ON LOONA(debut_order);

Index 'ORDER_INDEX' created.
```

3. 인덱스 조회하기

```
-- 3. 인덱스 조회하기
SELECT INDEX_NAME, TABLE_OWNER, TABLE_NAME FROM USER_INDEXES WHERE TABLE_NAME = 'LOONA';
```

```
SQL> -- 3. 인 덱 스  조 회 하 기
SQL> SELECT INDEX_NAME, TABLE_OWNER, TABLE_NAME FROM USER_INDEXES WHERE TABLE_NAME = 'LOONA';

INDEX_NAME
--------------------------------------------------------------------------------
TABLE_OWNER
--------------------------------------------------------------------------------
TABLE_NAME
--------------------------------------------------------------------------------
_SYS_CON33000933
SYS
LOONA

ORDER_INDEX
SYS
LOONA


2 rows selected.
```

4. 일반 인덱스 삭제

```
-- 4. 일반 인덱스 삭제
DROP INDEX order_index;
```

```
SQL> -- 4. 인 덱 스  삭 제
SQL> ---- 4.1 일 반  인 덱 스 는  삭 제  가 능
SQL> DROP INDEX order_index;

Index 'ORDER_INDEX' dropped.
```

9.sh

```
SET ECHO ON

-- 1. 테이블 생성 및 데이터 입력
CREATE TABLE LOONA ( name VARCHAR(10) PRIMARY KEY , debut_order NUMBER(2), birth_year NUMBER(4) );

INSERT INTO LOONA VALUES ('Heejin', 1, 2000);
INSERT INTO LOONA VALUES ('Hyunjin', 2, 2000);
INSERT INTO LOONA VALUES ('Haseul', 3, 1997);
INSERT INTO LOONA VALUES ('Yeojin', 4, 2002);
INSERT INTO LOONA VALUES ('Bibi', 5, 1996);
INSERT INTO LOONA VALUES ('Kimlip', 6, 1997);
INSERT INTO LOONA VALUES ('Jinsol', 7, 1997);
INSERT INTO LOONA VALUES ('Choiri', 8, 2001);
INSERT INTO LOONA VALUES ('Eve', 9, 1997);
INSERT INTO LOONA VALUES ('Chuu', 10, 1999);
```

```
INSERT INTO LOONA VALUES ('Gowon', 11, 2000);
INSERT INTO LOONA VALUES ('Olivia Hye', 12, 2001);

-- 2. 인덱스 생성하기
CREATE INDEX order_index ON LOONA(debut_order);

-- 3. 인덱스 조회하기
SELECT INDEX_NAME, TABLE_OWNER, TABLE_NAME FROM USER_INDEXES WHERE TABLE_NAME = 'LOONA';

-- 4. 일반 인덱스 삭제
DROP INDEX order_index;
```

---

10. 시퀀스

10.1 설명

시퀀스

- 유일한 연속적인 값을 생성해 낼 수 있는 스키마 객체
- 이 값은 보통 기본 키 또는 유일 키에 값을 채워 넣을 때 사용된다.
- 항상 시퀀스의 이름에는 의사 컬럼을 붙여서 사용한다.

컬럼	설명
CURRVAL	현재 세션에서 마지막으로 조회한 NEXTVAL 값을 반환한다.
NEXTVAL	시퀀스의 현재 값을 증가시키고 증가된 그 값을 반환한다.

10.2 시나리오 수행

시나리오 내용
1. 시퀀스 생성
2. 테이블 생성 및 데이터 입력
3. 시퀀스 값 조회
4. 시퀀스 삭제

10.3 시나리오 수행 결과

1. 시퀀스 생성

```
-- 1. 시퀀스 생성
CREATE SEQUENCE EX_SEQ --시퀀스이름
INCREMENT BY 1 --증감숫자 1
START WITH 1 --시작숫자 1
MINVALUE 1 --최소값 1
```

```
MAXVALUE 1000; --최대값 1000
;
```

```
SQL> -- 1. 시퀀스 생성
SQL> CREATE SEQUENCE EX_SEQ --시퀀스이름 EX_SEQ
  2 INCREMENT BY 1 --증감숫자 1
  3 START WITH 1 --시작숫자 1
  4 MINVALUE 1 --최소값 1
  5 MAXVALUE 1000 --최대값 1000
  6 NOCYCLE; --순환하지않음
  7 ;

Sequence 'EX_SEQ' created.
```

2. 테이블 생성 및 데이터 입력

```
-- 2. 테이블 생성 및 데이터 입력
CREATE TABLE testDB10 (BOARD_NUM NUMBER(19,6) NOT NULL);

INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);
INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);
INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);
```

```
SQL> -- 2. 테이블 생성 및 데이터 입력
SQL> CREATE TABLE testDB10 (BOARD_NUM NUMBER(19,6) NOT NULL);

Table 'TESTDB10' created.

SQL>
SQL> INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);

1 row inserted.

SQL> INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);

1 row inserted.

SQL> INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);

1 row inserted.
```

3. 시퀀스 값 조회

```
-- 3. 시퀀스 값 조회
SELECT * FROM testDB10;
SELECT EX_SEQ.CURRVAL FROM DUAL; --해당 시퀀스 값 조회
```

```
SQL> -- 3. 시 퀀 스  값  조 회
SQL> SELECT * FROM testDB10;

 BOARD_NUM
----------
         1
         2
         3

3 rows selected.

SQL> SELECT EX_SEQ.CURRVAL FROM DUAL; --해 당  시 퀀 스  값  조 회
   2 ;

   CURRVAL
----------
         3

1 row selected.
```

4. 시퀀스 삭제

```
-- 4. 시퀀스 값 삭제
DROP SEQUENCE EX_SEQ;
```

```
SQL> -- 4. 시 퀀 스  값  삭 제
SQL> DROP SEQUENCE EX_SEQ;

Sequence 'EX_SEQ' dropped.
```

10.sh

```
SET ECHO ON

-- 1. 시퀀스 생성
CREATE SEQUENCE EX_SEQ --시퀀스이름
INCREMENT BY 1 --증감숫자 1
START WITH 1 --시작숫자 1
MINVALUE 1 --최소값 1
MAXVALUE 1000; --최대값 1000
;

-- 2. 테이블 생성 및 데이터 입력
CREATE TABLE testDB10 (BOARD_NUM NUMBER(19,6) NOT NULL);

INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);
INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);
INSERT INTO testDB10(BOARD_NUM) VALUES(EX_SEQ.NEXTVAL);


-- 3. 시퀀스 값 조회
SELECT * FROM testDB10;
```

```
SELECT EX_SEQ.CURRVAL FROM DUAL; --해당 시퀀스 값 조회

-- 4. 시퀀스 값 삭제
DROP SEQUENCE EX_SEQ;
```