SQL 반복문

- SQL 에는 반복문이 존재하지 않는다.
- 반복문을 사용하지 않는 것이 성능에 좋다고 판단했기 때문이다.
- 하지만 반복문에 익숙해진 개발자는 호스트 언어(Java, C, C++, ...)에서 반복 처리를 구현한다.
- **반복문을 사용하면 레코드의 집합에 접근하지 못 하고, 레코드마다 작은 SQL을 사용해서 접근하게 되므로 성능이 좋지 않으므로 반복문을 사용하지 않는 방법을 알아보자.**

1. 반복계의 공포

- **Sales TABLE**

	COMPANY	YEAR	SALE
1	A	2002	50
2	A	2003	52
3	A	2004	55
4	A	2007	55
5	B	2001	27
6	B	2005	28
7	B	2006	28
8	B	2009	30
9	C	2001	40
10	C	2005	39
11	C	2006	38
12	C	2010	35

- **Sales2 TABLE**

	COMPA...	YEAR	SALE	VAR

- **문제1] Sales테이블은 각 기업의 회계연도별 매출을 기록한다. 연도가 연속되지 않는다. 이 데이터를 사용해 특정기업의 매출 변화를 조사할 것 이다. 그리고 결과는 var 필드를 추가한 Sales2 테이블에 등록한다. (Cursor, Procedure, 반복문 등 사용.)**
- Var 필드는 다음과 같은 규칙에 따라 결정된다.

- 이전 데이터가 없는 경우: NULL
- 이전 데이터보다 매출이 오른 경우: +
- 이전 데이터보다 매출이 내린 경우: -
- 이전 데이터와 매출이 같은 경우: =

- **Sale2 테이블은 다음과 같은 데이터가 들어간다.**

	COMPANY	YEAR	SALE	VAR
1	A	2002	50	(null)
2	A	2003	52	+
3	A	2004	55	+
4	A	2007	55	=
5	B	2001	27	(null)
6	B	2005	28	+
7	B	2006	28	=
8	B	2009	30	+
9	C	2001	40	(null)
10	C	2005	39	-
11	C	2006	38	-
12	C	2010	35	-

- **정답 1**

```
-- 정답 1] 반복계 코드
-- 프로시저 생성
CREATE OR REPLACE PROCEDURE PROC_INSERT_VAR
IS

-- 커서(여러 레코드로 구성된 작업영역에서 SQL 문 실행 및 저장에 사용)선언
CURSOR c_sales IS
    SELECT company, year, sale
    FROM Sales
    ORDER BY company, year;
-- 레코드 타입 선언(%ROWTYPE :  컬럼 데이터형, 크기, 속석등을 그대로 사용 할 수
있다.)
rec_sales c_sales%ROWTYPE;
-- 사용할 변수 초기화
i_pre_sale INTEGER := 0;
c_company CHAR(1) := '*';
c_var CHAR(1) := '*';

BEGIN
    -- 커서 오픈: 커서에서 선언된  SELECT 문의 실행을 의미.
    OPEN c_sales;
```

```
        LOOP -- 아래줄의 패치 결과 리턴되는 결과가 여러개인 경우 LOOP 사용.
            -- 레코드를 패치해서 변수에 대입.
            fetch c_sales into rec_sales;
            -- 레코드가 없다면 반복을 종료
            exit when c_sales%notfound;

            IF (c_company = rec_sales.company) THEN
            -- 직전 레코드가 같은 회사의 레코드 일 때
            -- 직전 레코드와 매출을 비교
                IF (i_pre_sale < rec_sales.sale) THEN c_var := '+';
                ELSIF (i_pre_sale > rec_sales.sale) THEN c_var := '-';
                ELSE c_var := '=';
                END IF;
          ELSE c_var := NULL;
          END IF;

          -- 등록 대상인 테이블에 데이터를 등록
          INSERT INTO Sales2 (company, year, sale, var) VALUES
(rec_sales.company, rec_sales.year, rec_sales.sale, c_var);
          c_company := rec_sales.company;
          i_pre_sale := rec_sales.sale;
        END LOOP;
    CLOSE c_sales;
    commit;
END;
-- 프로시저 실행
EXECUTE PROC_INSERT_VAR;
```

- 특정 연도의 레코드와 직전 연도의 레코드를 비교하는 로직을 반복하는 것은 전형적인 'record at a time' (한 번에 한 레코드)적 사고 방식이다.

1) 반복계의 단점 – 성능 : 처리 시간이 처리 대상의 레코드 수에 대해 선형으로 증가한다. 처리시간 = (처리횟수) * (한 회에 걸리는 시간)

- 반복계와 반대되는 포장계의 처리 시간은 대부분 **(log n) 곡선을 그리게 된다.**

- 반복계 성능 저하의 원인

  가) SQL 실행의 오버헤드

  - 오버헤드란 SQL처리 전에 실행하는 부수적인 전처리, 후처리 작업에 드는 시간과 메모리를 말한다.

  - 오버헤드 종류: 1-SQL 구문을 네트워크로 전송, 2-데이터베이스 연결, 3-SQL 구문 파

스, 4-SQL 구문의 실행 계획 생성 또는 평가, 5-결과 집합을 네트워크로 전송

- 1번과 5번은 SQL을 실행하는 애플리케이션과 데이터베이스가 물리적으로 다른 본체에 있을 때 발생하지만 오버헤드가 딱히 일어나지 않는다. 2번은 DB Connection Pool 기술을 사용해서 해결 가능하다.
- 3번과 4번이 오버헤드에 영향을 크게 준다. 파스는 데이터베이스가 SQL을 받을 때마다 실행되므로 작은 SQL을 여러 번 반복하는 반복계에서는 오버헤드가 높아질 수밖에 없다.

나) 병렬 분산이 힘들다.

- 반복 1회마다의 처리가 굉장히 단순하므로 리소스를 분산해서 병렬 처리하는 최적화가 안 된다.

다) 데이터베이스의 진화로 인한 혜택을 받을 수 없다.

2) 반복계의 장점

가) 안정성

- 실행 계획이 단순하기 때문에 실행 계획에 변동 위험이 거의 없다.

나) 예상 처리 시간의 정밀도

- 실행계획이 단순하고 안정적이기 때문에 처리시간 예상이 포장계보다 용이하다.

다) 트랜잭션 제어의 편의

- 예를 들어서 갱신 처리를 반복계에서, 특정 반복 횟수마다 커밋한다고 하자. 만약 중간에 오류가 발생했다고 해도, 중간에 커밋을 했으므로 해당 지점 근처에서 다시 처리를 실행하면 된다.

2. SQL에서는 반복문을 어떻게 표현할까?

1) 포인트는 CASE 식과 윈도우 함수

- **문제2] [문제1]을 반복문을 사용하지 않고, SQL을 사용해서 풀어라. (CASE식과 윈도우 함**

수사용)

- 다음과 같은 테이블이 나오면 된다.

	COMPANY	YEAR	SALE	VAR
1	A	2002	50	(null)
2	A	2003	52	+
3	A	2004	55	+
4	A	2007	55	=
5	B	2001	27	(null)
6	B	2005	28	+
7	B	2006	28	=
8	B	2009	30	+
9	C	2001	40	(null)
10	C	2005	39	-
11	C	2006	38	-
12	C	2010	35	-

- **정답 2**

```
insert into Sales2
select company, year, sale,
    case sign(
            sale - max(sale) over (partition by company order by year
                    rows between 1 preceding and 1 preceding)
        )
    when 0 then '='
    when 1 then '+'
    when -1 then '-'
    else null end as var
from sales;
```

가) SIGN 함수

- **SIGN 함수는 숫자 자료형을 매개변수로 받아 음수라면 -1, 양수라면 1, 0이라면 0을 리턴하는 함수다.**

- 여기서는 직전 연도와의 판매 변화를 알고자 사용했다.

- CASE 식의 조건 부분에 윈도우 함수를 몇 번씩 사용하지 않도록 해주는 기술.

나) ROWS BETWEEN

- 윈도우함수 MAX()에 ROWS BETWEEN 옵션을 사용함.

- 이는 현재 레코드 기준, 대상 범위의 레코드를 제한한다.

- **ROWS BETWEEN 숫자 PRECEDING/FOLLWING AND 숫자 PRECEDING/FOLLWING**

- ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING : 현재 레코드에서 1개 이전부터 1개 이전까지의 레코드 범위.

- 참고

```
select company, year, sale,
    max(sale) over (partition by company order by year
    rows between 1 preceding and 1 preceding) as preceding1,
    max(sale) over (partition by company order by year
    rows between 2 preceding and 2 preceding) as preceding2
 from sales;
```

	COMPANY	YEAR	SALE	PRECEDING1	PRECEDING2
1	A	2002	50	(null)	(null)
2	A	2003	52	50	(null)
3	A	2004	55	52	50
4	A	2007	55	55	52
5	B	2001	27	(null)	(null)
6	B	2005	28	27	(null)
7	B	2006	28	28	27
8	B	2009	30	28	28
9	C	2001	40	(null)	(null)
10	C	2005	39	40	(null)
11	C	2006	38	39	40
12	C	2010	35	38	39

- **문제3] 윈도우 함수로 '직전 회사명'과 '직전 매출' 검색, 데이터 테이블은 문제2번의 데이터를 사용한다.**

- 다음과 같은 테이블이 나와야 한다.

	COMPANY	YEAR	SALE	PRECOMPANY	PRESALE
1	A	2002	50	(null)	(null)
2	A	2003	52	A	50
3	A	2004	55	A	52
4	A	2007	55	A	55
5	B	2001	27	(null)	(null)
6	B	2005	28	B	27
7	B	2006	28	B	28
8	B	2009	30	B	28
9	C	2001	40	(null)	(null)
10	C	2005	39	C	40
11	C	2006	38	C	39
12	C	2010	35	C	38

- **정답 3**

```
select * from sales;
select company, year, sale,
    max(company) over(partition by company order by year
        rows between 1 preceding and 1 preceding) as preCompany,
    max(sale) over(partition by company order by year
        rows between 1 preceding and 1 preceding) as preSale
from sales;
```

2) 최대 반복 횟수가 정해진 경우

- **문제 4] 아래 테이블에 저장된 우편번호의 집합에서, 입력 받은 우편 번호('4130033')와 가장 가까운 지역의 우편번호를 검색해보자. 왼쪽부터 자릿수가 많이 일치할수록 가까운 우편번호다.**

	PCODE	DISTRICT_NAME
1	4130001	서울시 마포구 경리단길 1
2	4130002	서울시 마포구 경리단길 2
3	4130103	서울시 마포구 경리단길 3
4	4130041	서울시 마포구 경리단길 4
5	4103213	서울시 마포구 경리단길 5
6	4380824	서울시 마포구 경리단길 6

- 다음과 같은 결과가 나오면 된다.

	PCODE	DISTRICT_NAME
1	4130001	서울시 마포구 경리단길 1
2	4130002	서울시 마포구 경리단길 2
3	4130041	서울시 마포구 경리단길 4

- **정답 4**

```
select pcode, district_name
from postalCode
where case when pcode like '4130033' then 0
    when pcode like '413003%' then 1
    when pcode like '41300%%' then 2
    when pcode like '4130%%%' then 3
    when pcode like '413%%%%' then 4
    when pcode like '41%%%%%' then 5
    when pcode like '4%%%%%%' then 6
    else null end =
        ( select min(case when pcode like '4130033' then 0
                when pcode like '413003%' then 1
                when pcode like '41300%%' then 2
                when pcode like '4130%%%' then 3
                when pcode like '413%%%%' then 4
                when pcode like '41%%%%%' then 5
                when pcode like '4%%%%%%' then 6
                else null end ) from postalCode
        );
```

- 기본적인 방법은 일단 우편번호 '4130033'이 테이블에 있는지를 찾는다.
- 그리고 없으면 이어서 '413003*', 없으면 '41300**' ... 이와 같은 처리를 진행한다.
- 절차 지향형 사고방식이라면 당연히 반복을 사용해 해결할 것 이다.
- 예제와 같은 데이터라면 단순히 파일 전체에 반복을 최대 7번 돌려 답을 찾을 수 있다. 하지만 이런 방법은 레코드 수가 많아 질수록 성능 측면에서 점점 악화되는 결과를 가져온다.
- **포인트는 순위 이다.** 가장 가까운 우편번호 순위를 0, 가장 먼 우편번호 순위를 6으로 나타내면 된다. 순위를 나타내면 순위가 가장 높은 우편번호를 선택하면 된다. **7회의 반복을 7회의 CASE 식 분기로 변환했다.**

```
select pcode, district_name,
    case when pcode like '4130033' then 0
        when pcode like '413003%' then 1
        when pcode like '41300%%' then 2
        when pcode like '4130%%%' then 3
        when pcode like '413%%%%' then 4
        when pcode like '41%%%%%' then 5
        when pcode like '4%%%%%%' then 6
       else null end as rank
from postalCode;
```

	PCODE	DISTRICT_NAME	RANK
1	4130001	서울시 마포구 경리단길 1	2
2	4130002	서울시 마포구 경리단길 2	2
3	4130103	서울시 마포구 경리단길 3	3
4	4130041	서울시 마포구 경리단길 4	2
5	4103213	서울시 마포구 경리단길 5	5
6	4380824	서울시 마포구 경리단길 6	6

- CASE 식의 WHEN 구는 차례대로 조건을 검사하다가, 조건이 맞으면 이후의 WHEN 구를 평가 하지 않는다. 따라서 순위를 계산할 수 있다.

- **문제 5] [문제4]를 1회의 테이블 스캔, 즉 1회의 SELECT문으로 해결하지 못 했다면 1회의 테이블 스캔으로 문제를 해라.**

- 다음과 같은 테이블이 나오면 된다.

	PCODE	DISTRICT_NAME
1	4130001	서울시 마포구 경리단길 1
2	4130002	서울시 마포구 경리단길 2
3	4130041	서울시 마포구 경리단길 4

- **정답 5**

```
select pcode, district_name
from ( select pcode, district_name,
    case when pcode like '4130033' then 0
        when pcode like '413003%' then 1
        when pcode like '41300%%' then 2
        when pcode like '4130%%%' then 3
        when pcode like '413%%%%' then 4
        when pcode like '41%%%%%' then 5
        when pcode like '4%%%%%%' then 6
        else null end as rank,
    min( case when pcode like '4130033' then 0
        when pcode like '413003%' then 1
        when pcode like '41300%%' then 2
        when pcode like '4130%%%' then 3
        when pcode like '413%%%%' then 4
        when pcode like '41%%%%%' then 5
        when pcode like '4%%%%%%' then 6
        else null end
        ) over ( order by case when pcode like '4130033' then 0
                when pcode like '413003%' then 1
                when pcode like '41300%%' then 2
```

```
                when pcode like '4130%%%' then 3
                when pcode like '413%%%%' then 4
                when pcode like '41%%%%%' then 5
                when pcode like '4%%%%%%' then 6
                else null end) as min_rank
     from postalCode) Foo
where rank = min_rank;
```

- 순위의 최솟값을 서브쿼리에서 찾았기 때문에 테이블 스캔이 2회가 발생했었다. 위와 같이 윈도우 함수를 사용하면 스캔 횟수를 줄일 수 있다.

3) 최대 반복 횟수가 정해지지 않은 경우

- 지금까지 살펴본 예제에서는 최대 반복 횟수가 처음부터 정해졌다.
- 이런 경우에는 SQL 구문에 코드 하나하나 입력해서 반복을 CASE 식을 사용한 분기로 변경할 수 있었다.
- 그런데 반복 횟수가 정해지지 않은 경우에는 어떻게 해야 할까?
- DB설계 시점에 성능을 고려해서 인접 리스트 모델, 중첩 집합 모델 등 에 따라 DB를 설계한다.

가) 인접리스트 모델과 재귀 쿼리

- **문제6] 아래 테이블에서 '지용운'씨가 가장 오래 전에 살았던 주소를 찾아라.**
- 이 테이블에서 현재 주소를 등록할 때는 현재 주소의 우편번호만 등록하고, 이사하는 곳의 우편번호는 NULL 로 한다.
- 이사를 하게 되면 NULL값 이던 필드에 이사하는 곳의 우편번호를 넣어서 레코드를 수정한다. 그리고
- 새로운 우편번호와 'NULL 값을 갖는 새로운 우편번호 필드'를 갖는 레코드를 생성한다.
- 우편번호를 키로 삼아 데이터를 줄줄이 연결하고 있다(포인터 체인).

	NAME	PCODE	NEW_PCODE
1	지용운	4130001	4130002
2	지용운	4130002	4130003
3	지용운	4130103	(null)
4	김수정	4130041	(null)
5	최현웅	4103213	4380824
6	최현웅	4180824	(null)

- 만약 '지용운'씨가 가장 오래 전에 살았던 주소를 검색한다면 답은 '4130001'이다.
- 이것을 찾으려면 현재 주소에서 출발해서 차근차근 이전 주소를 찾아야 할 것 이다.
- 문제는 몇 번을 따라 올라가야만 가장 오래된 주소를 찾을 수 있을 것인지 사전에는 알 수 없다는 점이다.
- **SQL에서 계층 구조를 찾는 방법 중 하나는 재귀 공통 테이블 식을 사용하는 방법이다.**

- **정답 6**

```
-- Oralcle
with explosion (name, pcode, new_pcode, depth)
as
    (select name, pcode, new_pcode, 1
    from postalHistory
    where name = '지용운'
        and new_pcode is null
    union all
    select child.name, child.pcode, child.new_pcode, depth+1
    from explosion parent, postalHistory child
    where parent.pcode = child.new_pcode
        and parent.name = child.name)
select name, pcode, new_pcode
from explosion
where depth = (select max(depth) from explosion);
```

- 재귀 공통 테이블 식 Exposion은 '지용운'씨에 대해서 현재 주소(new_pcode 필드가 NULL)부터 출발해서 포인터 체인을 타고 올라가 과거의 주소를 모두 찾는다.

- 이 때 가장 오래된 주소는 재귀 수준이 가장 깊은 레코드 이므로, 이를 depth 필드로 찾는다.

- depth 필드는 한 번 반복할 때 마다 1씩 증가하므로, depth 필드가 가장 큰 것이 가장 재귀 수준이 깊다.

```sql
-- PostgreSQL
with recursive explosion (name, pcode, new_pcode, depth)
as
(select name, pcode, new_pcode, 1
from postalHistory
where name ='지용운'
    and new_pcode is null
union
select child.name, child.pcode, child.new_pcode, depth+1
from explosion as parent, postalHistory as child
where parent.code = child.new_pcode
    and parent.name = child.name)
select name, pcode, new_pcode
from explosion
where depth = (select max(depth) from explosion);
```

나) 중첩 집합 모델

	NAME	PCODE	LFT	RGT
1	김중수	4130001	0	27
2	김중수	4130002	9	18
3	김중수	4130003	12	15
4	한장희	4130041	0	27
5	김정수	4103213	0	27
6	김정수	4380824	9	18

- 이 방법의 포인트는 **각 레코드의 데이터를 집합(원)으로 보고**, 계층 구조를 집합의 중첩 관계로 나타낸다는 것 이다.
- 우편번호의 데이터를 수치선 상에 존재하는 원으로 생각한다.
- **Lft와 Rft는 원의 왼쪽 끝과 오른쪽 끝에 위치하는 좌표를 나타낸다.**
- 그리고 이사할 때 마다 새로운 우편번호가 이전의 우편번호 '안에' 포함되는 형태로 추가한다.
- '김중수'씨의 가장 오래된 주소는 가장 바깥쪽에 있는 원을 찾기만 하면 되므로 굉장히 간단한 SQL 구문으로 찾을 수 있다.
- NOT EXISTS를 사용하면 쉽게 구할 수 있다.

- **문제 7] '김중수'씨의 가장 오래된 주소를 찾아라.**

- **결과**

	NAME	PCODE
1	김중수	4130001

- **정답 7**

```
select name, pcode
from postalhistory2 ph1
where name = '김중수'
    and not exists (
        select * from postalhistory2 ph2
        where ph2.name = '김중수'
            and ph1.lft > ph2.lft);
```