Oracle 12c New Features focused SQL

Author	이경오
Creation Date	**2018-02**
Last Updated	
Version	**1.0**

Version	변경일자	변경자(작성자)	주요내용
1			

Goodus
Breathing Life into Technology

Contents

1. PATTERN MATCHING(MATCH_RECOGNIZE) 4

1.1. 기능소개 4
1.2. 기본문법 4
1.2.1. PARTITION BY & ORDER BY 4
1.2.2. MEASURES 4
1.2.3. PER MATCH 4
1.2.4. AFTER MATCH SKIP 5
1.2.5. PATTERN 5
1.2.6. DEFINE 5
1.3. 실습 5
1.3.1. 테이블 생성 5
1.3.2. 데이터 입력 6
1.3.3. 데이터 출력 6
1.3.4. Pattern Matching SQL 7
1.3.5. Pattern Matching SQL 결과 7

2. Oracle Native Left Outer Join 기능 개선 8

2.1. 기능소개 8
2.2. 실습 8
2.2.1. Oracle 12c 이전 버전(11gR2) 8
2.2.2. Oracle 12cR1 이후 버전(12cR2) 9

3. Top-n Query 의 새로운 기능 9

3.1. 기능소개 10
3.1.1. offset 10
3.1.2. fetch ~ row 10
3.1.3. fetch ~ percent 10
3.1.4. offest ~ fetch ~ percent 결합 10
3.1.5. with ties 결합 10
3.2. 실습 10
3.2.1. offset 을 이용한 데이터 건너뛰기 11
3.2.2. fetch ~ rows 를 이용한 부분 범위 처리 11
3.2.3. fetch ~ percent 를 이용한 부분 범위 처리 12
3.2.4. offset ~ fetch ~ percent 를 이용한 부분 범위 처리 12
3.2.5. with ties 를 이용한 동순위 데이터 가져오기 12

4. ListAgg Function Enhancements 12

4.1. 기능소개 13
4.2. 실습 13
4.2.1. Oracle 12cR2 이전 버전(11gR2) 13
4.2.2. Oracle 12cR2 버전 13

5. Multi-Column FK - join elimination(SQL Tuning 관점) 14

5.1. 기능 소개 14
5.2. 실습 15
5.2.1. 테이블 생성 15
5.2.2. FK 생성 15
5.2.3. 11gR2 실행 15
5.2.4. 12cR2 실행 15

6. Scalar Subquery Unnesting(SQL Tuning 관점) 16

6.1. 기능 소개 16
6.2. 실습 16
6.2.1. 테이블 생성 16
6.2.2. 스칼라 서브쿼리 Unnesting 이 되지 않은 경우(12cR1 이전) 16
6.2.3. 스칼라 서브쿼리 Unnesting 이 작동한 경우(12cR1 이후) 17

7. 참고문헌 18

1. PATTERN MATCHING(MATCH_RECOGNIZE)

Oracle 12c 버전에서 MATCH_RECOGNIZE 절이 추가되었습니다. 해당 절은 Analytic Function 의 Syntax 와 유사합니다. 이번 장은 **MATCH_RECOGNIZE 의 기능 소개, 기본 문법을 소개하고 간단한 실습을 진행**해 보도록 하겠습니다.

1.1. 기능소개

Oracle 8i 버전에서 처음 소개된 Analytic Function 으로 인해 절차지향적(프로그래밍적)으로 처리되어야할 많은 부분들을 SQL 을 이용하여 처리할수 있게 되었습니다. 이와 비슷한 맥락으로 Oracle 12c 버전에서는 **MATCH_RECOGNIZE** 절이 추가되었으며 해당 절의 추가로 인해 개발자들이 **Pattern Matching 분석 작업을 SQL 을 이용하여 간단하게** 할수 있게 되었습니다. 향후 해당 기능을 활용하여 금융예측, 금융거래통계, 비트코인 등에 사용될것으로 예측됩니다.

1.2. 기본문법

1.2.1. PARTITION BY & ORDER BY

해당 절을 선언하여 Grouping 할 대상을 선정하고 정렬 기준을 정의하겠습니다.

- 문법

```
PARTITION BY 컬럼명
ORDER BY 컬럼명
```

- 예시

```
PARTITION BY product --product 컬럼을 기준으로 grouping 하겠습니다.
ORDER BY tstamp --tstamp 기준으로 order by 하겠습니다.
```

1.2.2. MEASURES

해당 절은 패턴 매칭으로 가져올 컬럼을 지정(정의) 하겠습니다.

- 문법

```
MEASURES
변수명.컬럼명 AS 새로운 컬럼명
```

- 예시

```
MEASURES
STRT.tstamp AS start_tstamp,
LAST(UP.tstamp) AS peak_tstamp, --최고치중 마지막값
LAST(DOWN.tstamp) AS end_tstamp --최저치중 마지막값
```

1.2.3. PER MATCH

패턴을 만족하는 row 를 하나만 보여줄지, 패턴을 만족하는 모든 rows 를 보여줄지 지정하겠습니다.

- 문법

```
[ONE ROW | ALL ROWS] PER MATCH
```

- 예시

```
ONE ROW PER MATCH   --패턴을 만족하는 단 하나의 ROW 를 보여준다.
ALL ROWS PER MATCH  --패턴을 만족하는 모든 ROWS 를 보여준다.
```

1.2.4. AFTER MATCH SKIP

패턴을 만족한 후에 다시 패턴을 찾는 시작점을 지정하겠습니다.

- 문법

```
AFTER MATCH SKIP [TO|PAST] [NEXT|LAST|FIRST] [ROW|pattern_variable]
```

- 예시

```
AFTER MATCH SKIP TO NEXT ROW  --패턴을 만족한 다음 row 부터 다시 패턴을 찾는다.
AFTER MATCH SKIP PAST LAST ROW  --패턴을 만족시키는 마지막 row 부터 패턴매칭을
다시 시작하겠습니다.
AFTER MATCH SKIP TO FIRST pattern_variable  --첫번째 pattern_variable 부터
다시 패턴매칭을 시작하겠습니다.
AFTER MATCH SKIP TO LAST pattern_variable  --마지막 pattern_variable 부터
다시 패턴 매칭을 시작하겠습니다.
```

1.2.5. PATTERN

패턴의 변수명을 정의하겠습니다.

- 문법

```
PATTERN (변수명[+|*] 변수명[+|*] …)
```

- 예시

```
PATTERN (STRT UP+ FLAT* DOWN+) -- UP1 이 한 개 이상, FLAT 는 0 개 이상, DOWN 은 1
개 이상인 경우
```

1.2.6. DEFINE

패턴의 변수를 정의하겠습니다.

- 문법

```
DEFINE
변수명 AS 성립조건
```

- 예시

```
DEFINE --패턴의 변수를 정의
UP AS UP.units_sold > PREV(UP.units_sold),  --UP 은 이전 units_sold 보다 큰
경우
FLAT AS FLAT.units_sold = PREV(FLAT.units_sold), --FLAT 은 units_sold 와 같은
경우
DOWN AS DOWN.units_sold < PREV(DOWN.units_sold)  --DOWN 은 units_sold 보다
작은 경우
```

1.3. 실습

1.3.1. 테이블 생성

실습용 테이블을 생성하겠습니다. sales_history 테이블을 생성하겠습니다.

```
CREATE TABLE sales_history (
  id           NUMBER,
  product      VARCHAR2(20),
  tstamp       TIMESTAMP,
  units_sold   NUMBER,
  CONSTRAINT sales_history_pk PRIMARY KEY (id)
```

```
);
```

1.3.2. 데이터 입력

```
ALTER SESSION SET nls_timestamp_format = 'DD-MON-YYYY';

INSERT INTO sales_history VALUES ( 1, 'TWINKIES', '01-OCT-2014', 17);
INSERT INTO sales_history VALUES ( 2, 'TWINKIES', '02-OCT-2014', 19);
INSERT INTO sales_history VALUES ( 3, 'TWINKIES', '03-OCT-2014', 23);
INSERT INTO sales_history VALUES ( 4, 'TWINKIES', '04-OCT-2014', 23);
INSERT INTO sales_history VALUES ( 5, 'TWINKIES', '05-OCT-2014', 16);
INSERT INTO sales_history VALUES ( 6, 'TWINKIES', '06-OCT-2014', 10);
INSERT INTO sales_history VALUES ( 7, 'TWINKIES', '07-OCT-2014', 14);
INSERT INTO sales_history VALUES ( 8, 'TWINKIES', '08-OCT-2014', 16);
INSERT INTO sales_history VALUES ( 9, 'TWINKIES', '09-OCT-2014', 15);
INSERT INTO sales_history VALUES (10, 'TWINKIES', '10-OCT-2014', 17);
INSERT INTO sales_history VALUES (11, 'TWINKIES', '11-OCT-2014', 23);
INSERT INTO sales_history VALUES (12, 'TWINKIES', '12-OCT-2014', 30);
INSERT INTO sales_history VALUES (13, 'TWINKIES', '13-OCT-2014', 31);
INSERT INTO sales_history VALUES (14, 'TWINKIES', '14-OCT-2014', 29);
INSERT INTO sales_history VALUES (15, 'TWINKIES', '15-OCT-2014', 25);
INSERT INTO sales_history VALUES (16, 'TWINKIES', '16-OCT-2014', 21);
INSERT INTO sales_history VALUES (17, 'TWINKIES', '17-OCT-2014', 35);
INSERT INTO sales_history VALUES (18, 'TWINKIES', '18-OCT-2014', 46);
INSERT INTO sales_history VALUES (19, 'TWINKIES', '19-OCT-2014', 45);
INSERT INTO sales_history VALUES (20, 'TWINKIES', '20-OCT-2014', 30);
COMMIT;

-- 한국인 입장으로 다시 표기
ALTER SESSION SET nls_timestamp_format = 'YYYY-MM-DD';
```

1.3.3. 데이터 출력

```
SELECT id,
       product,
       tstamp,
       units_sold,
       RPAD('#', units_sold, '#') AS graph
FROM   sales_history
ORDER BY id;
```

ID	PRODUCT	TSTAMP	UNITS_SOLD	GRAPH
1	TWINKIES	2014/10/01 00:00:00	17	#################
2	TWINKIES	2014/10/02 00:00:00	19	###################
3	TWINKIES	2014/10/03 00:00:00	23	#######################
4	TWINKIES	2014/10/04 00:00:00	23	#######################
5	TWINKIES	2014/10/05 00:00:00	16	################
6	TWINKIES	2014/10/06 00:00:00	10	##########

7	TWINKIES	2014/10/07 00:00:00	14	##############
8	TWINKIES	2014/10/08 00:00:00	16	################
9	TWINKIES	2014/10/09 00:00:00	15	###############
10	TWINKIES	2014/10/10 00:00:00	17	#################
11	TWINKIES	2014/10/11 00:00:00	23	#######################
12	TWINKIES	2014/10/12 00:00:00	30	##############################
13	TWINKIES	2014/10/13 00:00:00	31	###############################
14	TWINKIES	2014/10/14 00:00:00	29	############################
15	TWINKIES	2014/10/15 00:00:00	25	#########################
16	TWINKIES	2014/10/16 00:00:00	21	#####################
17	TWINKIES	2014/10/17 00:00:00	35	###################################
18	TWINKIES	2014/10/18 00:00:00	46	###
19	TWINKIES	2014/10/19 00:00:00	45	##
20	TWINKIES	2014/10/20 00:00:00	30	##############################

1.3.4. Pattern Matching SQL

```
SELECT *
FROM   sales_history MATCH_RECOGNIZE (
         PARTITION BY product -- Grouping
         ORDER BY tstamp --Ordering
         MEASURES STRT.tstamp AS start_tstamp, --select 될 컬럼지정
                  LAST(UP.tstamp) AS peak_tstamp,
                  LAST(DOWN.tstamp) AS end_tstamp,
                  MATCH_NUMBER() AS mno -- 패턴을 만족한 개수
         ONE ROW PER MATCH -- 패턴을 만족하는 ROW 1 개만
         AFTER MATCH SKIP TO LAST DOWN -- 현재 패턴을 만족 하는 마지막 DOWN 시작점부터
다시 패턴을 찾기 시작함
         PATTERN (STRT UP+ FLAT* DOWN+) -- 패턴을 정의
         DEFINE -- 패턴의 변수를 정의
           UP AS UP.units_sold > PREV(UP.units_sold), -- UP
           FLAT AS FLAT.units_sold = PREV(FLAT.units_sold), -- FLAT
           DOWN AS DOWN.units_sold < PREV(DOWN.units_sold) -- DOWN
       ) MR
ORDER BY MR.product, MR.start_tstamp
;
```

1.3.5. Pattern Matching SQL 결과

PRODUCT	START_TSTAMP	PEAK_TSTAMP	END_TSTAMP	MNO
TWINKIES	2014/10/01 00:00:00	2014/10/03 00:00:00	2014/10/06 00:00:00	1

TWINKIES	2014/10/06 00:00:00	2014/10/08 00:00:00	2014/10/09 00:00:00	2
TWINKIES	2014/10/09 00:00:00	2014/10/13 00:00:00	2014/10/16 00:00:00	3
TWINKIES	2014/10/16 00:00:00	2014/10/18 00:00:00	2014/10/20 00:00:00	4

➢ **SQL 결과 해석**

- TSTAMP 기준으로 ASC 정렬되어 시작하므로 2014 년 10 월 10 이 시작입니다.
- 10 월 01 일부터 판매량이 늘어나다가 10 월 3 일날 최고치를 찍었으며 10 월 5 일이 되서야 처음으로 판매량이 감소하면서 패턴 매칭이 일어나게 되었습니다.
- 10 월 06 일까지 감소하고 (DOWN) 더 이상 감소하지 않으므로 10 월 06 일까지가 **첫번째 구간**이고 해당일부터 다시 패턴 매칭 검색이 시작됩니다.
- 10 월 08 일까지 판매량이 증가하다가 다시 10 월 09 일부터 판매량이 감소하면서 패턴매칭이 일어나게 되었습니다.
- 10 월 09 일까지만 감소하고 더 이상 감소하지 않으므로 10 월 09 일까지가 **두번째 구간**이고 10 월 09 일부터 다시 패턴 매칭 검색이 시작됩니다.
- 10 월 13 일까지 꾸준히 판매량이 증가하다가 10 월 14 일부터 판매량이 감소하기 시작하면서 패턴 매칭이 일어났으며 10 월 16 일까지 계속 하락하고 더 이상 하락하지 않았으므로 10 월 16 일까지가 **세번째 구간**이 되었고. 10 월 16 일부터 다시 검색이 시작되었습니다.
- 10 월 18 일까지 판매량이 증가하다가 10 월 19 일 처음으로 감소하면서 패턴 매칭이 일어나게 되었습니다.
- 10 월 20 일까지가 **네번째 구간**이 되었으며, 더 이상 데이터가 없으므로 검색이 종료되었습니다.

2. Oracle Native Left Outer Join 기능 개선

Oracle Native Left Outer Join 은 Oracle DBMS 만의 조인 방식으로 outer 가 되는 조건에 **(+)부호**를 붙여서 **Outer 조인으로 동작**하게 됩니다. 이번 장에서는 Oracle DBMS 조인 방식의 기능개선에 대해서 알아보고 실습을 진행해 보도록 하겠습니다.

2.1. 기능소개

12c 이전버전에서 Oracle Native Left Outer Join 은 단 하나의 테이블만 Outer 테이블로 지정해야 했었습니다. 하지만 **12c 버전부터는 2 개 이상의 복수 테이블로도 Outer 조인이 가능**합니다.

2.2. 실습

2.2.1. Oracle 12c 이전 버전(11gR2)

- **SQL 문**

```
SELECT A.EMPNO, B.NAME, C.ENAME
  FROM EMPLOYEE A
     , DEPARTMENT B
     , BONUS C
 WHERE A.DEPTNO = B.DEPTNO(+)
   AND A.NAME   = C.ENAME(+)
   AND B.NAME = C.ENAME(+);
```

- 결과

```
ORA-01417: a table may be outer joined to at most one other table
```

➢ **ORA-01417 이 발생하면서 SQL 이 실행되지 않음**

2.2.2. Oracle 12cR1 이후 버전(12cR2)

- SQL 문

```
SELECT A.EMPNO, B.NAME, C.ENAME
  FROM EMPLOYEE A
     , DEPARTMENT B
     , BONUS C
 WHERE A.DEPTNO = B.DEPTNO(+)
   AND A.NAME   = C.ENAME(+)
   AND B.NAME = C.ENAME(+);
```

- 결과

EMPNO	NAME	ENAME
7934	ACCOUNTING	(NULL)
7876	RESEARCH	(NULL)
7369	RESEARCH	(NULL)
7499	SALES	(NULL)
7839	ACCOUNTING	(NULL)
7844	SALES	(NULL)
7521	SALES	(NULL)
7902	RESEARCH	(NULL)
7566	RESEARCH	(NULL)
7654	SALES	(NULL)
7900	SALES	(NULL)
7782	ACCOUNTING	(NULL)
7698	SALES	(NULL)
7788	RESEARCH	(NULL)

➢ **2 개의 테이블에 Outer Join 이 가능**해짐

3. Top-n Query의 새로운 기능

12c 이전버전까지 **Top-n Query** 는 rownum 을 이용하여 SQL 개발이 이루어져왔습니다. 하지만 12c 이후 버전부터 **Top-n Query** 를 처리하기 위한 새로운 방법과 기능이 추가되었습니다. 하지만 성능 적인 관점에서는 기존의 rownum 방식보다 좋다라는 보장이 없으므로 무분별한 적용은 지양하고 검토후 적용할것을 권장하겠습니다. 이번 장에서는 12c 에서 추가된 **Top-n Query** 기능을 알아보고 간단한 실습을 해보기로 하겠습니다.

3.1. 기능소개

3.1.1. offset

offset 기능은 **특정 Row 를 건너뛰고 조회**하는 기능입니다. 해당 기능을 이용함으로써 특정 조건의 순위에서 항상 N 등 이후 데이터를 조회하는 등의 기능 구현이 가능합니다.

- **문법**

```
offset 숫자 rows;
```

3.1.2. fetch ~ row

fetch ~ row 기능은 **특정 집합에서 특정 건수의 rows 를 가져오는 기능**입니다. rownum 의 대체기능으로 사용이 가능합니다.

- **문법**

```
fetch first 숫자 rows only
```

3.1.3. fetch ~ percent

fetch ~ percent 기능은 **특정 집합에서 특정 비율 만큼만** 데이터를 가져오는 기능입니다.

- **문법**

```
fetch first 숫자 percent rows only
```

3.1.4. offest ~ fetch ~ percent 결합

offset 과 fetch 절을 결합하여 사용이 가능합니다. 즉 **몇번째 rows 까지를 건너뛰고 특정번째 rows 부터 특정 percent 비율만큼을** 가져올수 있다.

- **문법**

```
offset 숫자 rows fetch next 숫자 percent rows only
```

3.1.5. with ties 결합

with ties 기능은 **순위상 동률인 경우 동률인 데이터까지 모두 출력**하게 하는 기능입니다.

- **문법**

```
fetch first 숫자 percent rows with ties
```

3.2. 실습

➢ 아래의 SQL 에서 사용되는 **employee** 테이블의 데이터는 아래와 같습니다.

EMPNO	NAME	JOB	BOSS	HIREDATE	SALARY	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	800		20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600	300	30
7521	WARD	SALESMAN	7698	1981-02-22	1250	500	30
7566	JONES	MANAGER	7839	1981-04-02	2975		20
7698	BLAKE	MANAGER	7839	1981-05-01	2850		30
7782	CLARK	MANAGER	7839	1981-06-09	2450		10
7844	TURNER	SALESMAN	7698	1981-09-08	1500	0	30

7654	MARTIN	SALESMAN	7698	1981-09-28	1250	1400	30
7839	KING	PRESIDENT		1981-11-17	5000		10
7902	FORD	ANALYST	7566	1981-12-03	3000		20
7900	JAMES	CLERK	7698	1981-12-03	950		30
7934	MILLER	CLERK	7782	1982-01-23	1300		10
7788	SCOTT	ANALYST	7566	1982-12-09	3000		20
7876	ADAMS	CLERK	7788	1983-01-12	1100		20

3.2.1. offset을 이용한 데이터 건너뛰기

- **SQL 문**

```
SELECT EMPNO,
       HIREDATE
  FROM EMPLOYEE
ORDER BY HIREDATE OFFSET 8 ROWS; -- 8 개의 rows 를 건너뛰고 9 번째 row 부터 출력함
```

- **결과**

EMPNO	HIREDATE
7839	1981-11-17
7902	1981-12-03
7900	1981-12-03
7934	1982-01-23
7788	1982-12-09
7876	1983-01-12

3.2.2. fetch ~ rows를 이용한 부분 범위 처리

- **SQL 문**

```
SELECT EMPNO,
       HIREDATE
  FROM EMPLOYEE
 ORDER BY HIREDATE FETCH FIRST 10 ROWS ONLY; --처음부터 10 개의 rows 를  출력함
```

- **결과**

EMPNO	HIREDATE
7369	1980-12-17
7499	1981-02-20
7521	1981-02-22
7566	1981-04-02
7698	1981-05-01
7782	1981-06-09
7844	1981-09-08
7654	1981-09-28
7839	1981-11-17
7902	1981-12-03

3.2.3. fetch ~ percent를 이용한 부분 범위 처리

- SQL 문

```
SELECT EMPNO,
       HIREDATE
  FROM EMPLOYEE
 ORDER BY HIREDATE FETCH FIRST 25 PERCENT ROWS ONLY; --처음부터 25%의 rows 를 출력함
```

- 결과

EMPNO	HIREDATE
7369	1980-12-17
7499	1981-02-20
7521	1981-02-22
7566	1981-04-02

3.2.4. offset ~ fetch ~ percent를 이용한 부분 범위 처리

- SQL 문

```
SELECT EMPNO,
       HIREDATE
  FROM EMPLOYEE
 ORDER BY HIREDATE OFFSET 4 ROWS FETCH NEXT 25 PERCENT ROWS ONLY; --4 개의 rows 를
건너뛰고 25% 출력
```

- 결과

EMPNO	HIREDATE
7698	1981-05-01
7782	1981-06-09
7844	1981-09-08
7654	1981-09-28

3.2.5. with ties를 이용한 동순위 데이터 가져오기

- SQL 문

```
SELECT EMPNO ,
       NAME  ,
       SALARY
  FROM EMPLOYEE
 ORDER BY SALARY  FETCH FIRST 25 PERCENT
  ROWS WITH TIES; --처음부터 25%를 가져오면서 동일순위까지 가져오기
```

- 결과

EMPNO	NAME	SALARY
7369	SMITH	800
7900	JAMES	950
7876	ADAMS	1100
7654	MARTIN	1250
7521	WARD	1250

4. ListAgg Function Enhancements

Oracle 12c r2 버전부터 ListAgg 함수의 치명적인 약점을 보완하기 위해 새로 나왔습니다. 이번장에서는 기능 소개 및 실습을 진행하기로 하겠습니다.

4.1. 기능소개

Oracle 12c r1 까지의 버전은 ListAgg 함수의 Return 값이 4000Bytes 이상이면 **ora-01489** 가 발생되어 프로그램 자체가 Error 혹은 Exception 처리되는 문제가 있었습니다. 하지만 **Oracle 12c R2 버전부터는 ON OVERFLOW TRUNCATE 기능을 사용함으로써 4000Bytes 가 초과된 경우 오라클 에러를 방지**할수 있게 되었습니다. 또한 아래의 옵셥을 활용하면 다양한 처리가 가능합니다.

ON OVERFLOW ERROR	Default 옵션, 4000Bytes 초과시 에러
ON OVERFLOW TRUNCATE	4000 Bytes 초과시 생략된 문자열 길이를 보여주면 출력
ON OVERFLOW TRUNCATE '^'	4000 Bytes 초과시 '^'를 출력한 후 생략된 문자열 길이를 보여주면 출력
ON OVERFLOW TRUNCATE WITHOUT COUNT	4000 Bytes 초과시 생략된 문자열 길이를 미출력
ON OVERFLOW TRUNCATE '^' WITHOUT COUNT	4000 Bytes 초과시 '^'를 출력한 후 생략된 문자열 길이를 미출력

4.2. 실습

4.2.1. Oracle 12cR2 이전 버전(11gR2)

- **SQL**

```
SELECT LISTAGG(NAME , ',') WITHIN GROUP (ORDER BY NAME ) AS C1
  FROM EMPLOYEE,
       (SELECT *
          FROM DUAL CONNECT BY LEVEL <= 667)
 WHERE EMPNO = '7369' ;
```

- **결과**

```
ORA-01489: result of string concatenation is too long
```

4.2.2. Oracle 12cR2 버전

- **SQL**

```
--4000Bytes 가 초과시 count 가 출력되면서 에러가 발생하지 않음
SELECT LISTAGG(NAME , ',' ON OVERFLOW TRUNCATE) WITHIN GROUP (ORDER BY NAME ) AS
C1
  FROM EMPLOYEE,
       (SELECT *
          FROM DUAL CONNECT BY LEVEL <= 1000)
 WHERE EMPNO  = '7369';
```

- **결과**

C1
SMITH,SMITH

```
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,
SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SM
ITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH
,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,S
MITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMITH,SMI
TH,SMITH,SMITH,SMITH,SMITH,...(339)
```

5. Multi-Column FK - join elimination(SQL Tuning관점)

Oracle 11g R1 버전부터 FK 에 대한 JE(Join Elimination)이 작동하였다. 그후로 Oracle 12c R1 버전까지 Single-Column FK 에 대한 JE 만 작동하였고, Multi-Column FK 인 경우는 작동하지 않았다.

5.1. 기능 소개

Oracle 12c r2 버전부터 **Multi-Column 으로 구성된 FK 에서도 정상적인 JE(Join Elimination)가 작동**하게 되었으며 **이로인해 불필요한 Table Scan 을 방지**할수 있게 되었습니다.

5.2. 실습

5.2.1. 테이블 생성

```
CREATE TABLE t1 (a NUMBER, b NUMBER, c number);
CREATE TABLE t2 (b NUMBER , c NUMBER);
ALTER TABLE t2 ADD CONSTRAINT pk_t2 PRIMARY KEY (b, c);
```

5.2.2. FK 생성

```
ALTER TABLE t1 ADD CONSTRAINT t1_t2_fk FOREIGN KEY (b,c) REFERENCES t2 (b,c);
```

5.2.3. 11gR2 실행

- **select 문**

```
select t1.a, t1.b, t1.c
from t1
where exists (select 1 from t2
              where t2.b = t1.b
                and t2.c = t1.c
                );
```

- **실행계획**

```
*************************[Explain Plan Time: 2018/02/20
10:44:51]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1 Bytes=65)
   1    0   NESTED LOOPS (SEMI) (Cost=2 Card=1 Bytes=65)
   2    1     TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=2 Card=1 Bytes=39)
   3    1     INDEX (UNIQUE SCAN) OF 'PK_T2' (INDEX (UNIQUE)) (Cost=0 Card=1
Bytes=26)
----------------------------------------------------------
Predicate information (identified by operation id):
----------------------------------------------------------
   3 - access("T2"."B"="T1"."B" AND "T2"."C"="T1"."C")
----------------------------------------------------------
```

- T1 테이블을 Table Full Scan 한 후 PK_T2 를 Index Unique Scan 하면서 Nested Loop Semi 조인으로 처리되었음
- T1 테이블의 b,c 컬럼은 T2 의 b, c 컬럼이 반드시 존재해야만 하므로, **PK_T2 인덱스 스캔은 불필요한 Scan 비용**이었음
- 즉 정리하자면 T1 의 Parent 테이블인 T2 에 존재하지 않으면 T1 에도 존재자체가 불가능하기때문에 T1 의 b, c 컬럼을 을 기준으로 Exists 처리할때 T2 테이블을 Scan 할 필요가 전혀 없었으나 Scan 함

5.2.4. 12cR2 실행

- **select 문**

```
select t1.a, t1.b, t1.c
from t1
where exists (select 1 from t2
              where t2.b = t1.b
                and t2.c = t1.c
                );
```

- **실행계획**

```
************************[Explain Plan Time: 2018/02/20
10:51:55]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1 Bytes=39)
   1    0   TABLE ACCESS (FULL) OF 'T1' (TABLE) (Cost=2 Card=1 Bytes=39)
----------------------------------------------------------
Predicate information (identified by operation id):
----------------------------------------------------------
   1 - filter("T1"."B" IS NOT NULL AND "T1"."C" IS NOT NULL)
----------------------------------------------------------
```

- T1 테이블만을 Table Full Scan 하면서 결과를 출력함
- T2 테이블을 Scan 하여 Exists 처리하여도 결과집합은 똑같기 때문에 T2 는 Scan 자체가 논리적으로 불필요했음
- **JE 가 제대로 작동하면서 성능이 개선**됨

6. Scalar Subquery Unnesting(SQL Tuning관점)

12c R1 부터 새로나온 기능으로써, Scalar Subquery 가 Unnesting 되면서 성능이 개선되는 현상입니다.

6.1. 기능 소개

Scalar Subquery 는 결과집합의 건수만큼 Select 절에서 처리됩니다. Scalar Subquery 의 처리는 **메인쿼리의 rows 가 많고 Scalar Subquery 테이블에 적절한 Index 가 미존재시 성능부하의 주 원인**이 되기도 하겠습니다. 이런 경우 Scalar Subquery 가 Unnesting 되어 **Main Query 와 같은 Level 에서 실행되면서 hash join 으로 풀린다면 극적인 성능 개선**이 가능하며 Oracle 12c R1 부터 해당 기능을 제공하기 시작했다.

6.2. 실습

6.2.1. 테이블 생성

```
CREATE TABLE TEST_OBJECTS
AS SELECT * FROM DBA_OBJECTS, (select level from dual connect by level <= 10);

CREATE TABLE TEST_USERS
AS SELECT * FROM DBA_USERS, (select level from dual connect by level <= 10);
```

6.2.2. 스칼라 서브쿼리 Unnesting이 되지 않은 경우(12cR1이전)

- **select 문**

```
SELECT U.USERNAME ,
       (SELECT MAX(CREATED)
          FROM TEST_OBJECTS O
         WHERE O.OWNER = U.USERNAME) as max_created
  FROM TEST_USERS U;
```

- **실행내역**

```
SQL_ID  fjx66fmn3hpqs, child number 0
-------------------------------------
SELECT U.USERNAME ,       (SELECT MAX(CREATED)          FROM
TEST_OBJECTS O         WHERE O.OWNER = U.USERNAME) as max_created
```

```
FROM TEST_USERS U

Plan hash value: 3284448023

-------------------------------------------------------------------------------------------
-------------------
| Id  | Operation          | Name         | Starts | E-Rows | A-Rows |   A-Time   |
Buffers | Reads  |
-------------------------------------------------------------------------------------------
-------------------
|   0 | SELECT STATEMENT   |              |      1 |        |    380 |00:00:00.01 |
14 |      0 |
|   1 |  SORT AGGREGATE    |              |     38 |      1 |     38 |00:00:02.81 |
411K|    411K|
|*  2 |   TABLE ACCESS FULL| TEST_OBJECTS |     38 |   8055 |    453K|00:00:02.78 |
411K|    411K|
|   3 |  TABLE ACCESS FULL | TEST_USERS   |      1 |    380 |    380 |00:00:00.01 |
14 |      0 |
-------------------------------------------------------------------------------------------
-------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("O"."OWNER"=:B1)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

- test_users 의 건수는 380 건이 나왔으며 test_objects 테이블을 380 번 Table Full Scan 을 하게되었음
- 그러면서 **실행시간 2.81 초가 소요되고, Buffer 는 411K Read** 함

6.2.3. 스칼라 서브쿼리 Unnesting이 작동한 경우(12cR1이후)

- **select 문**

```
SELECT U.USERNAME ,
       (SELECT MAX(CREATED)
          FROM TEST_OBJECTS O
         WHERE O.OWNER = U.USERNAME) as max_created
  FROM TEST_USERS U;
```

- **실행내역**

```
SQL_ID  fjx66fmn3hpqs, child number 0
-------------------------------------
SELECT U.USERNAME ,       (SELECT MAX(CREATED)              FROM
TEST_OBJECTS O            WHERE O.OWNER = U.USERNAME) as max_created
FROM TEST_USERS U

Plan hash value: 2019476358

-------------------------------------------------------------------------------------------------------------
| Id  | Operation              | Name           | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
Reads  |  OMem |  1Mem | Used-Mem |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                |      1 |        |    370 |00:00:00.20 |
15048 |  15011 |       |          |
|*  1 |  HASH JOIN OUTER       |                |      1 |    370 |    370 |00:00:00.20 |   15048
|  15011 |  1995K|  1995K| 1543K (0)|
|   2 |   TABLE ACCESS FULL    | TEST_USERS     |      1 |    370 |    370 |00:00:00.01 |
```

```
21 |      0 |          |          |              |
|   3 |    VIEW                  | VW_SSQ_1     |      1 |      25 |      25 |00:00:00.20 |    15027
|  15011 |          |          |              |
|   4 |     HASH GROUP BY        |              |      1 |      25 |      25 |00:00:00.20 |
15027 |  15011 |   1137K|   1137K| 1221K (0)|
|   5 |      TABLE ACCESS FULL| TEST_OBJECTS |      1 |     732K|     732K|00:00:00.09 |
15027 |  15011 |          |          |              |
-------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("ITEM_1"="U"."USERNAME")
```

- test_users 테이블을 build input 으로 하고, test_objects 테이블을 probe input 으로 하여 hash outer join 을 함
- 그러면서 전체 **실행시간 0.20 초가 소요되고, Buffer 는 15048 Read** 함
- 극적인 성능 향상이 이루어짐
- 해당 기능이 작동하지 않는 경우 _OPTIMIZER_UNNEST_SCALAR_SQ 파라미터를 true 로 설정해야함

7. 참고문헌

해당 문서는 아래와 같은 Site 를 참고하여 작성함

https://docs.oracle.com/database/121/DWHSG/pattern.htm#DWHSG8956

http://dbaora.com/enhanced-left-outer-join-syntax-oracle-database-12c-release-1-12-1/

https://oracle-base.com/articles/12c/row-limiting-clause-for-top-n-queries-12cr1

https://oracle-base.com/articles/12c/listagg-function-enhancements-12cr2

https://danischnider.wordpress.com/2015/06/29/join-elimination-difference-in-oracle-11g-and-12c/

https://blog.tanelpoder.com/2013/08/13/oracle-12c-scalar-subquery-unnesting-transformation/

< 끝 >