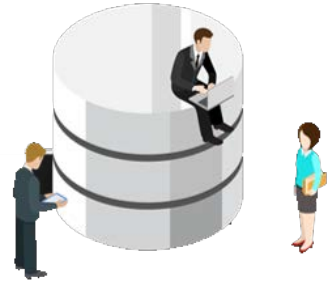


# EXEM SEMINAR

exem-academy.com/#seminar



512시간, 2,000명 이상 강의 세미나 진행, 국내 4대 교육기관 에 콘텐츠 제휴

## 엑셈 아카데미 교육 특장점

- 1 실무 사례 기반의 실전 노하우 전수
- 2 전문 기술 도서 저자의 직접 강의
- 3 기술내재화를 위한 사후 온오프 멘토링
- 4 외부소싱이 아닌 전원 엑셈의 스타강사

▼ 2017년 6월, 고객을 찾아가는 엑셈아카데미\_원주편

▼ 2017년 9월, 고객을 찾아가는 엑셈아카데미\_부산대학교편

공공 대학 기업 일반

▼ 2018년 2월 한국산업단지공단 기업사 임원 대상 빅데이터 분석 사례 세미나

▼ 2018년 3월 LS오토모티브 4차산업혁명 및 빅데이터 분석 교육

▼ 엑셈아카데미, 빅데이터/AI/DB/클라우드 세미나

▼ 2018년 1월 환경관리공단 IT 실무자 빅데이터 분석 및 블록체인 강의 세미나

▼ 2018년 3월 에너지밸리기업개발원 주최 광주·전남 지역 대상 클라우드 기반 빅데이터 분석과 활용 사례 발표

▼ 2018년 4월 LS오토모티브 제조 빅데이터 분석 입문·기초·심화 교육

▼ 2018년 1월, 라이나생명 디지털 전략 과정 - 빅데이터 분석 과정

엑셈 온사이트 세미나

4차 산업혁명의 핵심 기술들을 다루는 엑셈 아카데미의 지식 전문 세미나가, 여러분이 있는 곳으로 직접 찾아갑니다. 현재 공공기관, 기업 및 일반, 대학교 등 다양한 사이트에서 엑셈 온사이트 세미나는, 고객 여러분들이 원하는 맞춤형 4차 산업혁명 세미나를 제공합니다. 자세한 문의는 하단의 교육문의 메일을 참조하십시오.



## SQL 튜닝의 시작은?

- SQL 튜닝의 시작
  - \* SQL의 작성 의도를 제대로 파악하라



CONTENTS

# 1. SQL 튜닝의 시작

## SQL 튜닝의 시작?



**SQL의 의미를  
제대로  
파악하는 것**

# 1. SQL 튜닝의 시작

## SQL의 작성의도를 제대로 파악하라

### 성능 문제 SQL

```
SELECT *  
FROM (  
  SELECT /*+ INDEX_DESC(A IDX_MBOX_SENDDATE) */  
    a.*,  
    rownum as rnum  
  FROM tbs mbox a  
  WHERE userid = :b1  
    AND status = :b2  
    AND ROWNUM <= :b3  
)  
where rnum >= :b4 ;  
  
select statement - choose- cost estimate:3  
view  
count stopkey  
table access by index rowid :imsi.tbs_mbox  
index range scan descending :imsi.idx_mbox_senddate
```

### INDEX 정보

INDEX_NAME	COLUMN LIST
idx_mbox_status	userid, status
idx_mbox_senddate	userid, senddate

userid status senddate

idx\_mbox\_status  
인덱스를 사용?



SQL에 숨은  
Order By가  
존재한다



Index에 Status  
컬럼을 추가하여  
재 생성



# 1. SQL 튜닝의 시작

## SQL의 작성의도를 제대로 파악하라

IDX\_MBOX\_SENDDATE INDEX가 INVALID 상태가 된다면?

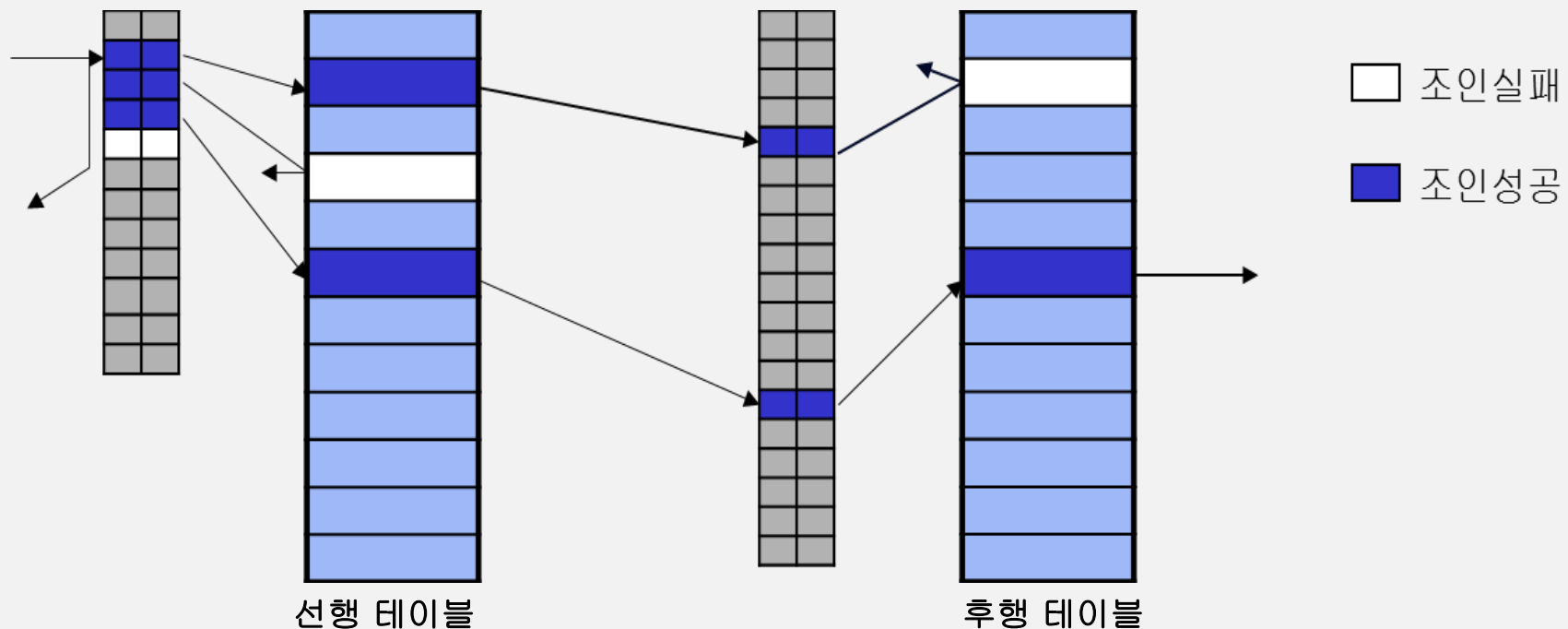
```
SELECT *
FROM (
  SELECT *
  FROM (
    SELECT /*+ INDEX_DESC(A IDX_MBOX_SENDDATE) */
          a.* ,
          ROWNUM AS rnum
    FROM   tbs_mbox a
    WHERE  userid = :b1
    AND    status = :b2
    ORDER  BY senddate DESC
  )
  WHERE  ROWNUM <= :b3
)
WHERE  rnum >= :b4 ;
```



# 1. SQL 튜닝의 시작

## 조인 방법 – Nested Loops Join

선행 테이블의 추출건수가 많으면 비효율적이다.  
후행 테이블에서 Join Key에 대한 조건을 활용할 수 있다.

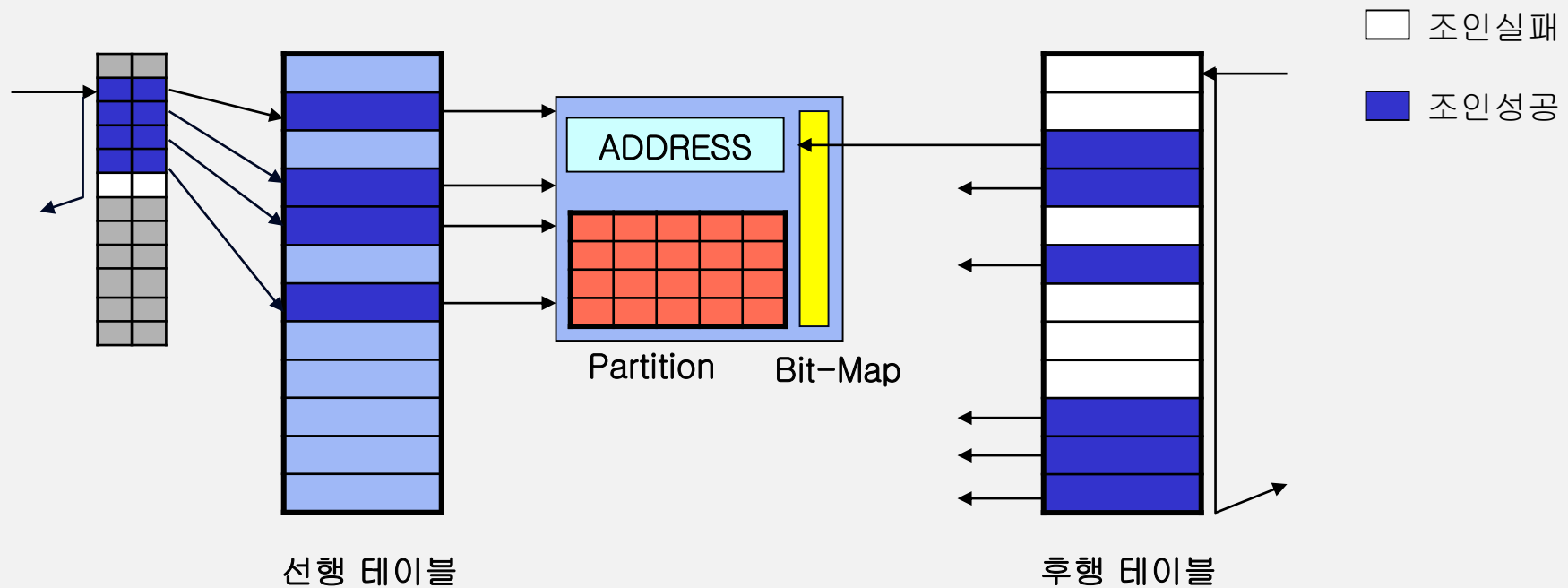


# 1. SQL 튜닝의 시작

## 조인 방법 - Hash Join

선행 테이블의 추출건수가 많아 후행 테이블을 반복적으로 탐색하면서 발생하는 비효율을 제거 할 때 유리하다.

후행 테이블에서 Join Key에 대한 조건을 활용할 수 없다.



# 1. SQL 튜닝의 시작

## PLAN을 읽는 방법

안에서 밖으로, 위에서 아래로, JOIN은 PAIR로, 각 Operation에 따라

```
SELECT c1, c2, c3
  FROM SUBQUERY_T2 t2
 WHERE c1 >= :b1 AND c1 <= :b2
    AND EXISTS ( SELECT /*+ UNNEST HASH_SJ */
                  'x'
                FROM SUBQUERY_T1 t1
                WHERE t1.c6 = t2.c3
                  AND t1.c6 >= :b1 )
```

INDEX_NAME	COLUMN_NAME
PK_SUQUERY_2	C1
INDEX_NAME	COLUMN_NAME
SUBQUERY_T1_IDX_01	C4, C5
SUBQUERY_T1_IDX_02	C5

Rows	Row Source OPERATION
11	FILTER (cr=37422 pr=0 pw=0 time=1910470 us) ①
11	HASH JOIN SEMI (cr=37422 pr=0 pw=0 time=1910466 us) ②
221	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=5 pr=0 pw=0 time=42 us) ③
221	INDEX RANGE SCAN PK_SUQUERY_2 (cr=3 pr=0 pw=0 time=31 us) ④
6400640	TABLE ACCESS FULL SUBQUERY_T1 (cr=37417 pr=0 pw=0 time=6261 us) ⑤



# 1. SQL 튜닝의 시작

## PLAN을 읽는 방법

```
SELECT ROWNUM rnum,
       X.*
FROM (
    SELECT /*+ USE_NL(T1 T2 T3) */
           t1.c1,
           t1.c2,
           t1.c3,
           t2.c3 AS t2_c3,
           t3.c3 AS t3_c3
    FROM SCALAR_T1 T1,
         SCALAR_T2 T2,
         SCALAR_T3 T3
    WHERE t1.c1 = t2.c1(+)
          AND t1.c1 = t3.c1(+)
    ORDER BY t1.c1, t1.c2 ) X
WHERE ROWNUM <= 10 ;
```

Rows	Row Source Operation	
-----	-----	
10	COUNT STOPKEY (cr=3003406 pr=0 pw=0 time=4448209 us)	①
10	VIEW (cr=3003406)	②
10	SORT ORDER BY STOPKEY (cr=3003406)	③
500000	NESTED LOOPS OUTER (cr=3003406)	④
500000	NESTED LOOPS OUTER (cr=1502292)	⑤
500000	TABLE ACCESS FULL SCALAR_T1 (cr=1178)	⑥
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T2 (cr=1501114)	⑦
500000	INDEX RANGE SCAN SCALAR_T2_IDX_01 (cr=1001114)	⑧
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T3 (cr=1501114)	⑨
500000	INDEX RANGE SCAN SCALAR_T3_IDX_01 (cr=1001114)	⑩

수행 순서는?



- ⑥ ⑧ ⑦ ⑤ ⑩ ⑨ ④ ③ ② ①



# 1. SQL 튜닝의 시작

## Trace 분석 방법

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c1 >= :b1 AND c1 <= :b2
AND EXISTS ( SELECT /*+ UNNEST HASH_SJ */
              'x'
              FROM SUBQUERY_T1 t1
              WHERE t1.c6 = t2.c3
              AND t1.c6 >= :b1 )
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.90	1.91	0	37422	0	11
total	4	1.90	1.92	0	37422	0	11

Rows

Row Source OPERATION

-----

11 FILTER (cr=37422 pr=0 pw=0 time=1910470 us)

11 HASH JOIN SEMI (cr=37422 pr=0 pw=0 time=1910466 us)

221 TABLE ACCESS BY INDEX ROWID SUBQUERY\_T2 (cr=5 pr=0 pw=0 time=42 us)

221 INDEX RANGE SCAN PK\_SUQUERY\_2 (cr=3 pr=0 pw=0 time=31 us)

6400640 TABLE ACCESS FULL SUBQUERY\_T1 (cr=37417 pr=0 pw=0 time=6261 us)

## 서브쿼리와 성능 문제 이해하기



- 서브쿼리에 대한 기본 내용 이해하기
  - \* 서브쿼리란
  - \* 서브쿼리의 동작 방식 이해하기
  - \* 서브쿼리 실행계획 제어하기
- 서브쿼리를 활용한 SQL 성능개선
  - \* 비효율 MINUS 대신 EXISTS를 사용하자
  - \* WHERE절의 서브쿼리를 조인으로 변경하자

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리란?

- ◆ WHERE절에 비교 조건으로 사용되는 SELECT 쿼리를 의미한다.
- 집합적인 사고를 필요로 하는 조인 보다는 절차적이므로 사용하기 쉽다.
- 무분별하게 남용하여 사용할 경우 성능 문제가 발생할 확률이 높다.

### 사용 패턴 1

```
SELECT *
FROM emp
WHERE sal > ( SELECT AVG( sal )
              FROM emp )
```

서브쿼리 추출 결과는 반드시 1건  
서브쿼리 ➡ Main SQL

### 사용 패턴 2

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c2 = 'A'
AND EXISTS (
    SELECT /*+ NO_UNNEST */
        'x'
    FROM SUBQUERY_T1 t1
    WHERE t1.c5 = t2.c2
)
```

서브쿼리 추출 결과가 여러 건 추출  
서브쿼리 ↔ Main SQL  
성능문제가 주로 발생하는 패턴

# 1. 서브쿼리에 대한 기본 내용 이해하기

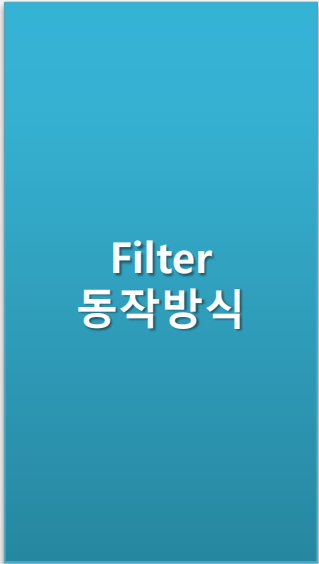
## 서브쿼리의 동작방식 이해하기

Filter 동작방식

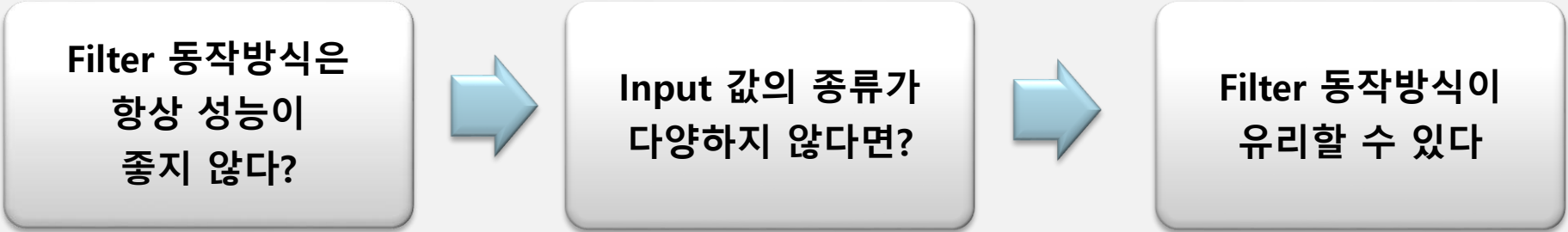
조인 동작방식

# 1. 서브쿼리에 대한 기본 내용 이해하기

## Filter 동작방식



- ◆ 최대 Main SQL에서 추출된 데이터 건수 만큼 서브쿼리가 반복적으로 수행되며 처리 되는 방식
  - Main SQL의 추출건수가 100만 건이면 서브쿼리는 최대 100만 번 수행된다.
  - Input 값에 해당하는 값의 종류가 적은 경우에는 Filter Optimization 작업을 통해 오히려 조인 방식보다 효율적일 수 있다.
  - 연결 컬럼에 대한 INDEX는 반드시 존재해야 한다.
  - 한가지 처리 방법만을 고수한다.





# 1. 서버쿼리에 대한 기본 내용 이해하기

## Filter 동작방식에 대한 TEST(1)

◆ Main SQL의 추출건수가 많고, Input 값이 Unique 한 경우

Main SQL추출 건수 : 380,001건  
t2.c1은 Unique한 컬럼임

```
SELECT c1, c2, c3
  FROM SUBQUERY_T2 t2
 WHERE c1 >= :b1 AND c1 <= :b2
       AND EXISTS ( SELECT /*+ NO_UNNEST */
                    'x'
                    FROM SUBQUERY_T1 t1
                    WHERE t1.c4 = t2.c1 )
```

BIND VALUE

-----

B1 = 20000

B2 = 400000

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.01	0.00	0	0	0	0
Fetch	15335	2.34	2.41	0	1156653	0	230001
total	15337	2.35	2.42	0	1156653	0	230001

Rows	Row	Source OPERATION
230001	<b>FILTER</b>	(cr=1156653 pr=0 pw=0 time=1846157 us)
380001	FILTER	(cr=16650 pr=0 pw=0 time=3336 us)
380001	TABLE ACCESS FULL	SUBQUERY_T2 (cr=16650 pr=0 pw=0 time=35462 us)
230001	INDEX RANGE SCAN	SUBQUERY_T1_IDX_01 (cr=1140003 pr=0 ...)

추출건수 만큼  
반복 수행되어  
비효율 발생

# 1. 서버쿼리에 대한 기본 내용 이해하기

## Filter 동작방식에 대한 TEST(2)

◆ Main SQL의 추출건수가 적고, Input 값이 Unique 한 경우

Main SQL추출 건수 : 5건  
t2.c1은 Unique한 컬럼임

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c1 >= :b1 AND c1 <= :b2
AND EXISTS ( SELECT /*+ NO_UNNEST */
              'x'
              FROM SUBQUERY_T1 t1
              WHERE t1.c4 = t2.c1 )
```

BIND VALUE

-----

B1 = 20000

B2 = 20004

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.01	0.01	0	1332	0	5
total	4	0.01	0.01	0	1332	0	5

Rows	Row	Source OPERATION
5	<b>FILTER</b>	(cr=1332 pr=0 pw=0 time=12743 us)
5	FILTER	(cr=1317 pr=0 pw=0 time=12657 us)
5	TABLE ACCESS FULL	SUBQUERY_T2 (cr=1317 pr=0 pw=0 time=12652 us)
5	INDEX RANGE SCAN	SUBQUERY_T1_IDX_01 (cr=15 pr=0 pw=0 time=58 us)

추출건수가  
5건에 불과해  
Filter 처리도  
효율적임

# 1. 서버쿼리에 대한 기본 내용 이해하기

## Filter 동작방식에 대한 TEST(3)

◆ Main SQL의 추출건수는 많지만, Input 값의 종류가 26가지인 경우

Main SQL추출 건수 : 380,001건  
t2.c2는 값의 종류가 26 가지

BIND VALUE  
-----  
B1 = 20000  
B2 = 400000

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c1 >= :b1 AND c1 <= :b2
AND EXISTS ( SELECT /*+ NO_UNNEST */
              'x'
              FROM SUBQUERY_T1 t1
              WHERE t1.c5 = t2.c2 )
```

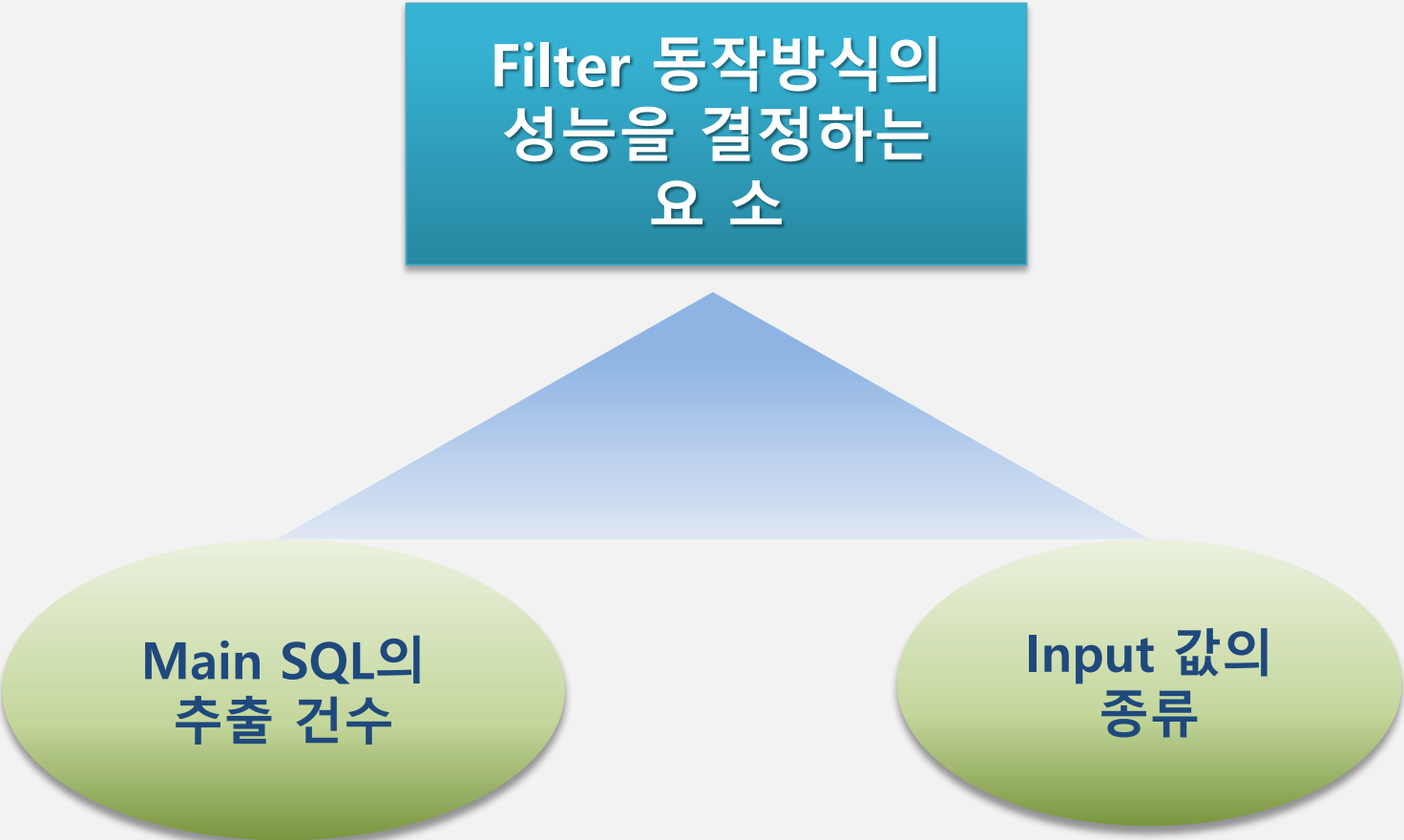
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	25335	1.01	1.25	0	26728	0	380001
total	25337	1.01	1.25	0	26728	0	380001

Rows	Row	Source OPERATION
380001	<b>FILTER</b>	(cr=26728 pr=0 pw=0 time=385929 us)
380001	FILTER	(cr=26650 pr=0 pw=0 time=3480 us)
380001	TABLE ACCESS FULL	SUBQUERY_T2 (cr=26650 pr=0 pw=0 time=40002 us)
26	INDEX RANGE SCAN	SUBQUERY_T1_IDX_02 (cr=78 pr=0 pw=0 time=223 us)

값의 종류는26 가지  
Filter Optimization에  
의해 서버쿼리가  
26번만 수행됨

# 1. 서브쿼리에 대한 기본 내용 이해하기

## Filter 동작방식과 SQL의 성능



# 1. 서브쿼리에 대한 기본 내용 이해하기

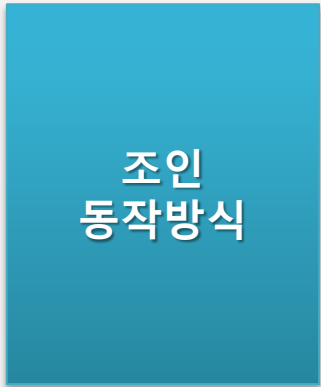
## Filter 동작방식에 대한 특징

구분	내용
수행 순서	✓ Main SQL이 먼저 수행된다.
Main SQL 추출 건수	✓ 최대 Main SQL을 추출 건수 만큼 서브 쿼리가 수행된다. ✓ Main SQL의 추출 건수가 적을 경우에는 Filter 동작방식은 불리하지 않다.
Input 값의 종류	✓ Unique 할 경우 Main SQL을 추출 건수만큼 서브쿼리가 수행된다. ✓ 값의 종류가 적을 경우, 최소 값의 종류만큼만 서브쿼리가 수행된다.
유 연 성	✓ Main SQL을 먼저 수행해야만 하므로 다양한 상황에서 유연하게 대처하기는 어려운 동작 방식.

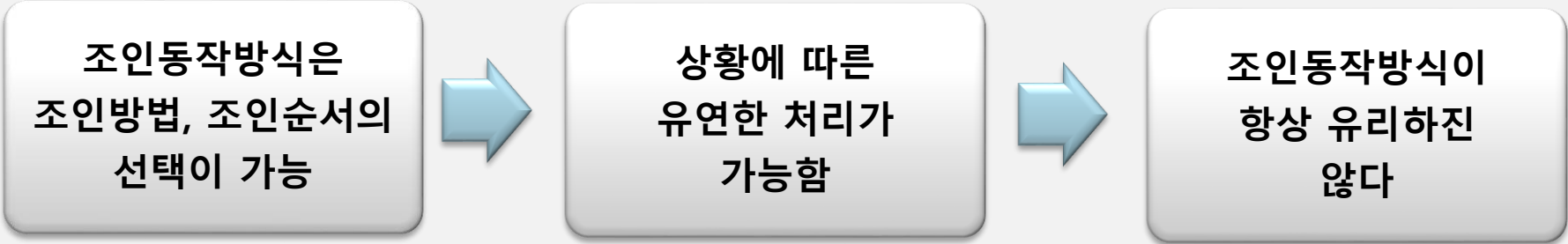


# 1. 서버쿼리에 대한 기본 내용 이해하기

## 조인 동작방식



- ◆ 조인 동작방식은 Filter 동작 방식에 비해 유연한 대처가 가능하다.
- 조인방법, 조인순서 유리한 것을 선택 할 수 있다.
- Filter Optimization 효과를 이용할 수 없다. Input이 동일한 값이 많다면 Filter 동작방식이 유리할 수 있다.



# 1. 서버쿼리에 대한 기본 내용 이해하기

## Filter 동작방식으로 수행되어 성능 문제가 발생하는 SQL 개선

◆ Sub Query에 적절한 Index가 없어 비효율이 발생한 경우

```
SELECT c1, c2, c3
  FROM SUBQUERY_T2 t2
 WHERE c1 >= :b1 AND c1 <= :b2
    AND EXISTS ( SELECT /*+ NO_UNNEST */
                  'x'
                FROM SUBQUERY_T1 t1
                WHERE t1.c6 = t2.c3 AND t1.c6 >= :b1 )
```

BIND VALUE

-----

B1 = 249990

B2 = 250210

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	156.11	157.03	0	7863857	0	11
total	4	156.12	157.03	0	7863857	0	11

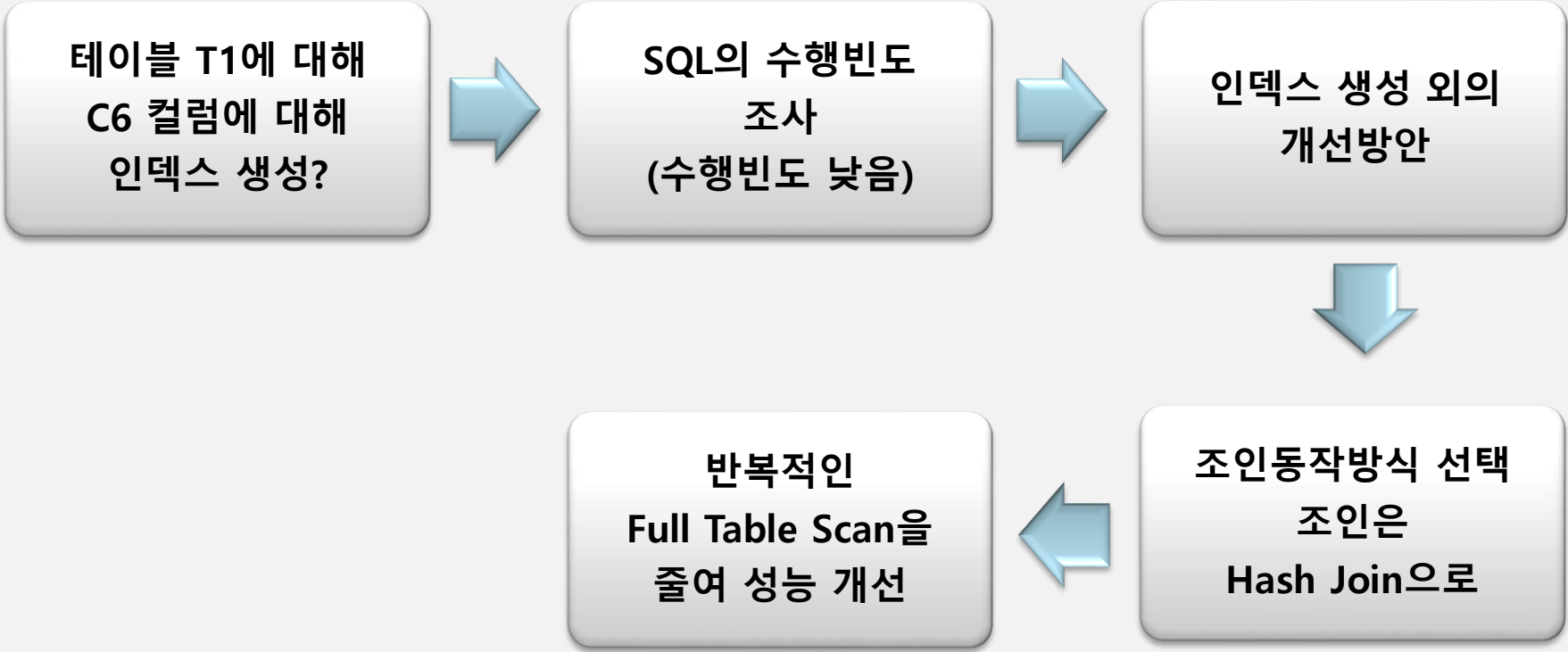
Rows	Row Source OPERATION
11	FILTER (cr=7863857 pr=0 pw=0 time=157033184 us)
221	FILTER (cr=7 pr=0 pw=0 time=2467 us)
221	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=7 pr=0 pw=0 time=2125 us)
221	INDEX RANGE SCAN PK_SUQUERY_2 (cr=4 pr=0 pw=0 time=690 us)
11	FILTER (cr=7863850 pr=0 pw=0 time=157028384 us)
11	TABLE ACCESS FULL SUBQUERY_T1 (cr=7863850 pr=0 pw=0 time=157028031 us)

221번 Full Table Scan

# 1. 서버쿼리에 대한 기본 내용 이해하기

Filter 동작방식으로 수행되어 성능 문제가 발생하는 SQL 개선

◆ 개선 방안 도출



# 1. 서버쿼리에 대한 기본 내용 이해하기

## Filter 동작방식으로 수행되어 성능 문제가 발생하는 SQL 개선

◆ Index를 생성하지 않고 조인동작방식(Hash)으로 처리하여 개선한 사례

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c1 >= :b1 AND c1 <= :b2
AND EXISTS ( SELECT /*+ UNNEST HASH_SJ */
              'x'
              FROM SUBQUERY_T1 t1
              WHERE t1.c6 = t2.c3
              AND t1.c6 >= :b1 )
```

BIND VALUE

-----

B1 = 249990

B2 = 250210

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.90	1.91	0	37422	0	11
total	4	1.90	1.92	0	37422	0	11

Rows	Row Source OPERATION
11	FILTER (cr=37422 pr=0 pw=0 time=1910470 us)
11	HASH JOIN SEMI (cr=37422 pr=0 pw=0 time=1910466 us)
221	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=5 pr=0 pw=0 time=42 us)
221	INDEX RANGE SCAN PK_SUQUERY_2 (cr=3 pr=0 pw=0 time=31 us)
6400640	TABLE ACCESS FULL SUBQUERY_T1 (cr=37417 pr=0 pw=0 time=6261 us)

1번

Full Table Scan

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 동작방식을 제어하는 힌트

HINT 명	설명
NO_UNNEST	✓ 서브쿼리를 Filter 방식으로 유도하는 HINT
UNNEST	✓ 서브쿼리를 조인동작방식으로 유도하는 HINT
NL_SJ	✓ 조인동작방식 중 Nested Loops Join Semi로 유도하는 HINT
HASH_SJ	✓ 조인동작방식 중 Hash Join Semi로 유도하는 HINT
NL_AJ	✓ 조인동작방식 중 Nested Loops Join Anti로 유도하는 HINT
HASH_AJ	✓ 조인동작방식 중 Hash Join Anti로 유도하는 HINT
ORDERED	✓ FROM절에 나열된 순서대로 조인 순서를 정하는 HINT (SUB QUERY가 존재하면 서브쿼리부터 수행하도록 유도함)
QB_NAME	✓ QUERY BLOCK의 이름을 지정하는 HINT
SWAP_JOIN_INPUTS	✓ HASH OUTER JOIN과 같이 순서가 고정된 상황에서 조인 순서를 바꾸도록 유도하는 HINT
NO_SWAP_JOIN_INPUTS	✓ HASH OUTER JOIN과 같이 순서가 고정된 상황에서 조인 순서를 바꾸지 못하도록 유도하는 HINT
PUSH_SUBQ	✓ 서브쿼리가 먼저 수행하도록 유도하는 HINT

## 1. 서브쿼리에 대한 기본 내용 이해하기



서브쿼리 실행계획 조절하기

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 실행계획 조절하기

### ◆ 예제 SQL

```
SELECT c4, c5, c6
FROM   SUBQUERY_T1 T1
WHERE  c6 >= :b1
AND    c6 <= :b2
AND    EXISTS ( SELECT 'x'
                  FROM   SUBQUERY_T2 T2
                  WHERE  t2.c1 = t1.c4
                  AND    t2.c3 >= :b3
                  AND    t2.c3 <= :b4 ) ;
```

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 실행계획 조절하기 -1-

1) 예제 SQL을 Filter 동작 방식으로 수행하도록 제어하여 보자.

**HINT : NO\_UNNEST**

2) 예제 SQL을 조인동작 방식 Nested Loops Semi Join으로 수행하도록 제어하여 보자.

**HINT : UNNEST, NL\_SJ**

3) 예제 SQL을 Hash Semi Join으로 수행하되, 서브쿼리가 Main SQL 보다 먼저 수행하도록 제어하여 보자.

**HINT : UNNEST, HASH\_SJ, SWAP\_JOIN\_INPUTS**

4) 예제 SQL을 Hash Semi Join으로 수행하되, Main SQL이 먼저 수행하도록 제어하여 보자.

**HINT : UNNEST, HASH\_SJ, NO\_SWAP\_JOIN\_INPUTS**



# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 실행계획 조절하기 -2-

5) 예제 SQL을 Nested Loops Join으로 수행하되, 서브쿼리부터 수행하도록 제어하여 보자.

**HINT : UNNEST, QB\_NAME, LEADING, USE\_NL**

6) 예제 SQL을 Hash Join으로 수행하되, 서브쿼리부터 수행하도록 제어하여 보자.

**HINT : UNNEST, QB\_NAME, LEADING, USE\_HASH**

7) NOT EXISTS를 사용한 예제 SQL을 Nested Loops Anti Join으로 수행하도록 제어하여 보자.

**HINT : UNNEST, NL\_AJ**

8) NOT EXISTS를 사용한 예제 SQL을 Hash Join Anti로 수행하도록 제어하여 보자.

**HINT : UNNEST, HASH\_AJ**

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 실행계획 조절하기 심층학습

◆ 아래의 SQL에서 서브쿼리를 먼저 읽은 후, Nested Loops Join으로 수행하도록 할 수 있겠는가?

```
SELECT *  
FROM emp a  
WHERE empno IN ( SELECT max(empno)  
                  FROM emp x  
                  GROUP BY deptno );
```

# 1. 서버쿼리에 대한 기본 내용 이해하기

## 서버쿼리 실행계획 조절하기 심층학습

조인순서(LEADING), 조인방법( USE\_NL), QUERY BLOCK명 지정 (QB\_NAME)힌트들을 사용하여 SQL의 실행계획을 제어하여 보자.

```
SELECT /*+ LEADING(X@SUB) QB_NAME(MAIN) USE_NL(A@MAIN) */
      *
FROM   emp a
WHERE  empno IN (  SELECT /*+ UNNEST QB_NAME(SUB) */
                  max(empno)
                  FROM   emp x
                  GROUP BY deptno );
```

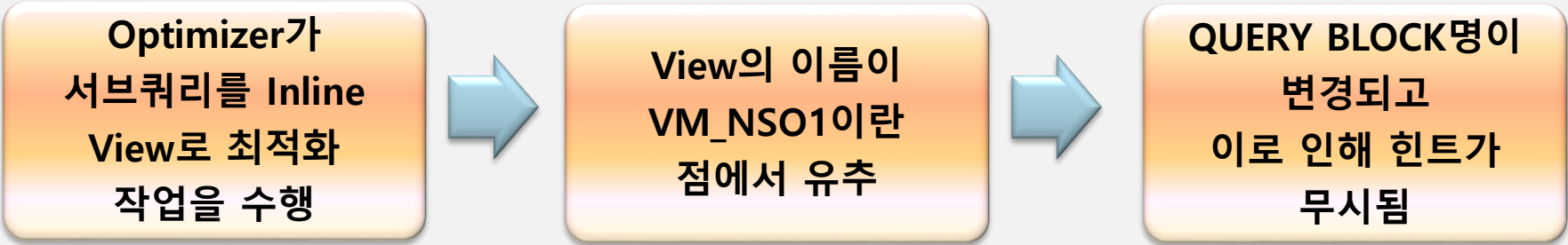
Id	OPERATION	Name	Rows	Bytes
0	SELECT STATEMENT		3	153
1	MERGE JOIN		3	153
2	TABLE ACCESS BY INDEX ROWID	EMP	13	494
3	INDEX FULL SCAN	PK_EMP	13	
* 4	SORT JOIN		3	39
5	VIEW	VW_NSO_1	3	39
6	HASH GROUP BY		3	21
7	TABLE ACCESS FULL	EMP	13	91

실행계획이  
의도한 대로  
적용되지 않음

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 실행계획 조절하기 심층학습

무엇 때문에 실행계획이 의도한 대로 제어 되지 않았을까?



Id	OPERATION	Name	Rows	Bytes
0	SELECT STATEMENT		3	153
1	MERGE JOIN		3	153
2	TABLE ACCESS BY INDEX ROWID	EMP	13	494
3	INDEX FULL SCAN	PK_EMP	13	
* 4	SORT JOIN		3	39
5	VIEW	VW_NSO_1	3	39
6	HASH GROUP BY		3	21
7	TABLE ACCESS FULL	EMP	13	91

# 1. 서브쿼리에 대한 기본 내용 이해하기

## 서브쿼리 실행계획 조절하기 심층학습

그렇다면 힌트로 실행 계획을 유도할 방법은 없을까?

- Optimizer는 서브쿼리를 Inline View로 변경할 경우 **From 절의 앞에** 위치 시킨다.
- ORDERED 힌트는 Query Block Name과 상관 없이 **From절에 나열된 순서대로** 수행한다는 특징이 있음.



**ORDERED HINT를  
활용하자!**

# 1. 서버쿼리에 대한 기본 내용 이해하기

## 서버쿼리 실행계획 조절하기 심층학습

ORDERED HINT와 USE\_NL을 이용하여 원하는 대로 실행 계획을 제어하여 보자.

```
SELECT /*+ ORDERED USE_NL(A) */
      *
FROM   emp a
WHERE  empno IN ( SELECT /*+ UNNEST */
                  max(empno)
                  FROM   emp x
                  GROUP BY deptno );
```

Id	OPERATION	Name	Rows	Bytes
0	SELECT STATEMENT		3	153
1	NESTED LOOPS		3	153
2	VIEW	VW_NSO_1	3	39
3	HASH GROUP BY		3	21
4	TABLE ACCESS FULL	EMP	13	91
5	TABLE ACCESS BY INDEX ROWID	EMP	1	38
* 6	INDEX UNIQUE SCAN	PK_EMP	1	

의도한 대로  
실행계획이  
제어됨

## 2. 서브쿼리를 활용한 SQL 성능개선

### 비효율적인 MINUS 대신 NOT EXISTS를 사용하자

◆ MINUS와 NOT EXISTS 비교

비교 대상	MINUS	NOT EXISTS
수행 SQL	SELECT ... FROM A MINUS SELECT ... FROM B	SELECT ... FROM A WHERE NOT EXISTS ( SELECT ... FROM B WHERE B.XX = A.XX )
수행 방식	1) 테이블 A에서 데이터 추출 2) 추출된 데이터 SORT 연산 3) 테이블 B에서 데이터 추출 4) 추출된 데이터 SORT 연산 5) 2번과 4번 데이터 비교 후 최종 데이터 추출	1) 테이블 A에서 데이터 추출 2) 1번에서 추출한 데이터와 서브쿼리 테이블 B 데이터와 비교 후 최종 데이터 추출
수행 순서	고정 (A → B)	변경 가능(A → B or B → A)
테이블 수행방식	테이블 A와 상관없이 별도로 데이터 추출 후 SORT 연산 수행	테이블 A의 추출 데이터를 이용한 인덱스 스캔 가능 (조인 연결 키) 및 별도 수행도 가능
SQL 성능	<b>불리:</b> 1) 테이블 A, B에서 추출한 데이터를 SORT 연산 시 성능 저하 2) 테이블 A에서 추출한 데이터가 적고, 테이블 B에 아무런 조건이 없는 경우 Full Table Scan 으로 처리하여야 하고, 추출된 데이터를 SORT 연산을 수행해야 함.	<b>유리:</b> 적절한 인덱스 스캔을 수행하거나 Full Table Scan을 수행 하는 등 SQL 성능에 가장 효율적인 방법을 선택하여 적용이 가능함.

## 2. 서버쿼리를 활용한 SQL 성능개선

### 비효율적인 MINUS 대신 NOT EXISTS를 사용하자

◆ MINUS를 사용한 성능문제 발생 SQL

```
SELECT c1, c2, c3
  FROM SUBQUERY_T2
 WHERE c2 = :b1
AND c1 >= :b2
AND c1 <= :b3
MINUS
SELECT c4, c5, c6
  FROM SUBQUERY_T1
```

BIND VALUE

-----

B1 = A

B2 = 200000

B3 = 300000

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	130	13.01	16.09	15273	56207	0	1923
total	132	13.01	16.09	15273	56207	0	1923

Rows	Row Source OPERATION
1923	MINUS (cr=56207 pr=15273 pw=0 time=16095391 us)
3846	SORT UNIQUE (cr=247 pr=0 pw=0 time=7290 us)
3846	FILTER (cr=247 pr=0 pw=0 time=3906 us)
3846	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=247 pr=0 ...)
3846	INDEX RANGE SCAN SUBQUERY_T2_IDX_01 (cr=12 pr=0 ...)
250000	SORT UNIQUE (cr=55960 pr=15273 pw=0 time=16054420 us)
16000000	TABLE ACCESS FULL SUBQUERY_T1 (cr=55960 pr=15273 pw=0 time=15833 us)

사이즈가 큰  
SUBQUERY\_T1  
테이블에  
조회조건이 없어  
FULL TABLE SCAN  
수행



## 2. 서버쿼리를 활용한 SQL 성능개선

### 비효율적인 MINUS 대신 NOT EXISTS를 사용하자

◆ MINUS를 NOT EXISTS로 변경 한 후

```
SELECT c1, c2, c3
  FROM SUBQUERY_T2 T2
 WHERE c2 = :b1
 AND c1 >= :b2
 AND c1 <= :b3
 AND NOT EXISTS ( SELECT /*+ UNNEST NL_AJ */
                   'x'
                   FROM SUBQUERY_T1 T1
                  WHERE t1.c4 = t2.c1
                    AND t1.c5 = t2.c2
                    AND t1.c6 = t2.c3 )
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	130	0.03	0.03	0	10241	0	1923
total	132	0.03	0.03	0	10241	0	1923

Rows	Row	Source	OPERATION
1923		FILTER	(cr=10241 pr=0 pw=0 time=30102 us)
1923		NESTED LOOPS ANTI	(cr=10241 pr=0 pw=0 time=28175 us)
3846		TABLE ACCESS BY INDEX ROWID SUBQUERY_T2	(cr=495 pr=0 pw=0 time=3888 us)
3846		INDEX RANGE SCAN SUBQUERY_T2_IDX_01	(cr=140 pr=0 ...)
1923		TABLE ACCESS BY INDEX ROWID SUBQUERY_T1	(cr=9746 pr=0 ...)
1923		INDEX RANGE SCAN SUBQUERY_T1_IDX_01	(cr=7823 pr=0 ...)

NOT EXISTS로  
변경한 후  
INDEX를 사용

## 2. 서버쿼리를 활용한 SQL 성능개선

### 비효율적인 MINUS 대신 NOT EXISTS를 사용하자

◆ MINUS와 NOT EXISTS 사용시 주의 할 점

#### MINUS 사용 SQL

Rows	Row Source OPERATION
-----	-----
1923	MINUS (cr=56207 pr=15273 pw=0 time=16095391 us)
3846	SORT UNIQUE (cr=247 pr=0 pw=0 time=7290 us)
3846	FILTER (cr=247 pr=0 pw=0 time=3906 us)
3846	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=247 pr=0 ...)
3846	INDEX RANGE SCAN SUBQUERY_T2_IDX_01 (cr=12 pr=0 ...)
250000	SORT UNIQUE (cr=55960 pr=15273 pw=0 time=16054420 us)
16000000	TABLE ACCESS FULL SUBQUERY_T1 (cr=55960 pr=15273 pw=0 time=15833 us)

SORT UNIQUE  
OPERATION 수행  
즉 중복을 제거함

#### NOT EXISTS 사용 SQL

Rows	Row Source OPERATION
-----	-----
1923	FILTER (cr=10241 pr=0 pw=0 time=30102 us)
1923	NESTED LOOPS ANTI (cr=10241 pr=0 pw=0 time=28175 us)
3846	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=495 pr=0 pw=0 time=3888 us)
3846	INDEX RANGE SCAN SUBQUERY_T2_IDX_01 (cr=140 pr=0 ...)
1923	TABLE ACCESS BY INDEX ROWID SUBQUERY_T1 (cr=9746 pr=0 ...)
1923	INDEX RANGE SCAN SUBQUERY_T1_IDX_01 (cr=7823 pr=0 ...)

SORT UNIQUE  
OPERATION이  
존재하지 않음

## 2. 서브쿼리를 활용한 SQL 성능개선

### 비효율적인 MINUS 대신 NOT EXISTS를 사용하자

◆ 데이터 정합성을 위해 항상 DISTINCT 처리를 해야 할까?

```
SELECT c1, c2, c3
  FROM SUBQUERY_T2
 WHERE c2 = :b1
AND c1 >= :b2
AND c1 <= :b3
```

```
MINUS
SELECT c4, c5, c6
  FROM SUBQUERY_T1
```

Rows	Row Source OPERATION
-----	-----
1923	MINUS (cr=56207 pr=15273 pw=0 time=16095391 us)
3846	SORT UNIQUE (cr=247 pr=0 pw=0 time=7290 us)
3846	FILTER (cr=247 pr=0 pw=0 time=3906 us)
3846	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=247 pr=0 ...)
3846	INDEX RANGE SCAN SUBQUERY_T2_IDX_01 (cr=12 pr=0 ...)
250000	SORT UNIQUE (cr=55960 pr=15273 pw=0 time=16054420 us)
16000000	TABLE ACCESS FULL SUBQUERY_T1 (cr=55960 pr=15273 pw=0 time=15833 us)

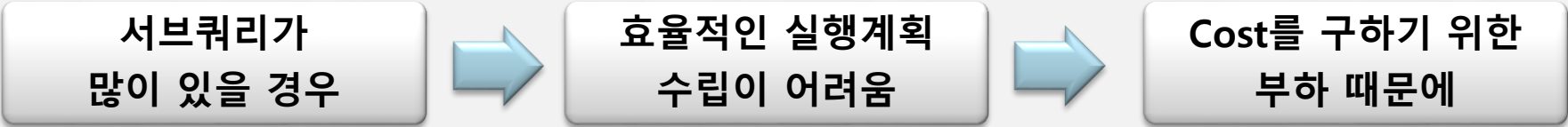
SUBQUERY\_T1의  
C1컬럼은 Unique함.

SELECT LIST절  
추출 컬럼 조합이  
Unique 하면  
Distinct 처리  
불필요

## 2. 서브쿼리를 활용한 SQL 성능개선

### WHERE절의 서브쿼리를 조인으로 변경하자

◆ 서브쿼리가 여러 개 존재하는 SQL의 성능문제



```
SELECT t1.*
FROM SUBQUERY_T1 t1
WHERE EXISTS ( SELECT 'x'
               FROM SUBQUERY_T2 t2
               WHERE t2.c2 like :b1
                 AND t2.c3 >= :b2
                 AND t2.c3 <= :b3
                 AND t2.c1 = t1.c4
                 AND t2.c2 = t1.c5 )
AND EXISTS ( SELECT 'x'
             FROM SUBQUERY_T3 t3
             WHERE t3.c2 LIKE :b4
               AND t3.c3 >= :b5
               AND t3.c3 <= :b6
               AND t3.c1 = t1.c4 )
AND EXISTS ( SELECT 'x'
             FROM SUBQUERY_T1 t11
             WHERE t11.c5 LIKE :b7
               AND t11.c4 >= :b8
               AND t11.c4 <= :b9
```

먼저 수행하면  
유리함을  
우리는 알고 있다  
Optimizer의  
선택은?

BIND VALUE	
-----	
B1 =	A
B2 =	200000
B3 =	200100
B4 =	%
B5 =	100000
B6 =	300000
B7 =	%
B8 =	100000
B9 =	300000

## 2. 서브쿼리를 활용한 SQL 성능개선

### WHERE절의 서브쿼리를 조인으로 변경하자

◆ Optimizer의 선택은?



## 2. 서버쿼리를 활용한 SQL 성능개선

### WHERE절의 서버쿼리를 조인으로 변경하자

◆ Optimizer의 선택은?

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	19	5.75	90.32	22044	478496	0	256
total	21	5.75	90.32	22044	478496	0	256

Rows	Row Source OPERATION
256	FILTER (cr=478496 pr=22044 pw=0 time=90748742 us)
256	NESTED LOOPS SEMI (cr=478496 pr=22044 pw=0 time=90748735 us)
246144	NESTED LOOPS SEMI (cr=466943 pr=22031 pw=0 time=997164558 us)
246144	NESTED LOOPS (cr=459246 pr=22031 pw=0 time=993718531 us)
100002	SORT UNIQUE (cr=974 pr=728 pw=0 time=510612 us)
100002	TABLE ACCESS BY INDEX ROWID SUBQUERY_T3 (cr=974 pr=728 ...)
200001	INDEX RANGE SCAN SUBQUERY_T3_IDX_01 (cr=504 pr=376 ...)
246144	TABLE ACCESS BY INDEX ROWID SUBQUERY_T1 (cr=458272 pr=21303 ...)
246144	INDEX RANGE SCAN SUBQUERY_T1_IDX_01 (cr=212128 pr=13827 ...)
3846	INDEX RANGE SCAN SUBQUERY_T1_IDX_01 (cr=7697 pr=0 ...)
4	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=11553 pr=13 ...)
3846	INDEX RANGE SCAN SUBQUERY_T2_IDX_01 (cr=7707 pr=13 ...)

## 2. 서브쿼리를 활용한 SQL 성능개선

### WHERE절의 서브쿼리를 조인으로 변경하자

#### ◆ SUBQUERY\_T2를 가장 먼저 수행하도록 SQL을 변경하자

```

SELECT /*+ LEADING(T2 T1) USE_NL(T2 T1) */
      t1.*
FROM SUBQUERY_T1 t1,
      ( SELECT DISTINCT t2.c1, t2.c2
        FROM SUBQUERY_T2 t2
        WHERE t2.c2 like :b1
              AND t2.c3 >= :b2 AND t2.c3 <= :b3 ) t2
WHERE t2.c1 = t1.c4
      AND t2.c2 = t1.c5
      AND EXISTS ( SELECT 'x'
                  FROM SUBQUERY_T3 t3
                  WHERE t3.c2 LIKE :b4
                        AND t3.c3 >= :b5 AND t3.c3 <= :b6 AND t3.c1 = t1.c4 )
      AND EXISTS ( SELECT 'x'
                  FROM SUBQUERY_T1 t11
                  WHERE t11.c5 LIKE :b7
                        AND t11.c4 >= :b8 AND t11.c4 <= :b9
                        AND t11.c4 = t1.c4 )

```

서브쿼리에서  
조인으로 변경  
(DISTINCT 포함)

SUBQUERY를 Inline View로 바꾸어 SQL을 작성 한 후  
Leading Hint를 사용하여 실행계획을 제어한다.

## 2. 서버쿼리를 활용한 SQL 성능개선

### WHERE절의 서버쿼리를 조인으로 변경하자

◆ SUBQUERY\_T2를 가장 먼저 수행하도록 SQL을 변경한 후 Trace 결과

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.03	0	0	0	0
Execute	1	0.01	0.02	0	0	0	0
Fetch	19	0.00	0.02	29	827	0	256
total	21	0.01	0.08	29	827	0	256

Rows	Row Source OPERATION
256	NESTED LOOPS SEMI (cr=827 pr=29 pw=0 time=46064 us)
256	NESTED LOOPS SEMI (cr=811 pr=29 pw=0 time=37370 us)
256	NESTED LOOPS (cr=799 pr=29 pw=0 time=35047 us)
4	VIEW (cr=512 pr=29 pw=0 time=25797 us)
4	HASH UNIQUE (cr=512 pr=29 pw=0 time=25787 us)
4	FILTER (cr=512 pr=29 pw=0 time=60 us)
4	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=512 pr=29 ...)
7692	INDEX RANGE SCAN SUBQUERY_T2_IDX_01 (cr=42 pr=29 ...)
256	TABLE ACCESS BY INDEX ROWID SUBQUERY_T1 (cr=287 pr=0 ...)
256	INDEX RANGE SCAN SUBQUERY_T1_IDX_01 (cr=31 pr=0 ...)
4	INDEX RANGE SCAN SUBQUERY_T1_IDX_01 (cr=12 pr=0 ...)
4	TABLE ACCESS BY INDEX ROWID SUBQUERY_T3 (cr=16 pr=0 ...)
4	INDEX RANGE SCAN SUBQUERY_T3_IDX_01 (cr=12 pr=0 ...)



## 2. 서브쿼리를 활용한 SQL 성능개선

### Summary

SUBQUERY 란?

SUBQUERY 동작방식

SUBQUERY 제어하기

## 스칼라 서브쿼리의 이해와 특성 이해하기



### - 스칼라 서브쿼리의 특성 이해하기

- \* 스칼라 서브쿼리란

### - 스칼라 서브쿼리와 조인의 이해 및 활용하기

- \* 스칼라 서브쿼리에서 발생할 수 있는 성능문제
- \* 스칼라 서브쿼리는 최종 결과 만큼 수행하자
- \* 스칼라 서브쿼리와 조인관계로 보는 SQL 성능문제
- \* 스칼라 서브쿼리는 최종 결과 만큼 수행하자

# 1. 스칼라 서브쿼리의 특성 이해하기

## 스칼라 서브쿼리란?

### ◆ Select Column List 절에 사용된 서브쿼리

- 최대 결과 건수만큼 반복적으로 수행된다.
- 추출되는 데이터는 항상 1건만 유효하다.
- 데이터가 추출 되지 않아도 된다.

## 사용 예

```
SELECT c1, c2, c3,  
       (SELECT t2.c1  
        FROM SCALAR_T2 T2  
        WHERE t2.c2 = t1.c2  
          AND ROWNUM <= 1 ) AS t2_c1  
FROM SCALAR_T1 T1  
WHERE c2 = 'A'  
      AND ROWNUM <= 1
```

스칼라 서브쿼리 추출 결과는 1건을 초과하면 안 된다.

# 1. 스칼라 서브쿼리의 특성 이해하기

## 최대 결과 건수만큼 반복적으로 수행된다

◆ 스칼라 서브쿼리는 Deterministic 속성을 가지고 있다

Deterministic 속성이란?

Multi Buffer란?  
\_query\_execution\_cache\_max\_size

예

SQL 추출건수가  
100건



Input 값으로  
사용되는 값은  
모두 동일



1회 수행

# 1. 스칼라 서브쿼리의 특성 이해하기

## 추출되는 데이터는 항상 1건만 유효하다

◆ 스칼라 서브쿼리의 결과가 2건 이상을 추출할 경우 SQL은 수행 에러가 발생한다

SCALAR\_T2의 C2컬럼은 Unique하지 않음

### 에러가 발생하는 경우

```
SELECT c1,
       c2,
       c3,
       (SELECT t2.C1
        FROM SCALAR_T2 T2
        WHERE t2.c2 = t1.c2
       ) AS t2_c1
FROM SCALAR_T1 T1
WHERE c2 = 'A'
      AND ROWNUM <= 1;
```

ERROR at line 4:  
ORA-01427: single-row subquery returns more than one row

### 해결 방안

```
SELECT c1,
       c2,
       c3,
       (SELECT t2.C1
        FROM SCALAR_T2 T2
        WHERE t2.c2 = t1.c2
              AND ROWNUM <= 1
       ) AS t2_c1
FROM SCALAR_T1 T1
WHERE c2 = 'A'
      AND ROWNUM <= 1 ;
```

C1	C2	C3	T2_C1
26	A	100025	26

# 1. 스칼라 서브쿼리의 특성 이해하기

## 데이터가 추출되지 않아도 된다

◆ 스칼라 서브쿼리의 결과가 NULL 데이터를 추출해도 SQL 수행에 영향을 미치지 않는다

### 사용 예

```
SELECT c1,
       c2,
       c3,
       NVL((SELECT NULL
            FROM SCALAR_T2 T2
            WHERE t2.c2 = t1.c2
            AND ROWNUM <= 1
            ), 'ISNULL') AS t2_c1
FROM SCALAR_T1 T1
WHERE c2 = 'A'
AND ROWNUM <= 1 ;
```

C1	C2	C3	T2_C1
26	A	100025	ISNULL



스칼라 서브쿼리는  
SQL의 추출건수에는  
영향을 주지 않음  
(Outer Join)

## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

스칼라 서브쿼리에서 발생할 수 있는 성능문제 유형



수행 위치



조인 관계

## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리는 최종 결과 만큼 수행하자

◆ SCALAR\_T1 테이블 전체 데이터 대상으로, C1, C2 컬럼으로 오름 차순 정렬 후 10건만 가져오는 SQL을 작성하고자 한다. 이때 추출한 데이터는 SCALAR\_T1의 C1, C2, C3와 SCALAR\_T2의 C3 컬럼 값이다

#### 작성 SQL [1]

```
SELECT ROWNUM rnum,
       x.*
FROM (
    SELECT c1,
           c2,
           c3,
           (SELECT t2.c3
            FROM SCALAR_T2 T2
            WHERE t2.c1 = t1.c1) AS t2_c3
    FROM SCALAR_T1 T1
    ORDER BY c1, c2
) x
WHERE ROWNUM <= 10 ;
```

#### 작성 SQL [2]

```
SELECT ROWNUM rnum,
       x.*,
       (SELECT t2.c3
        FROM SCALAR_T2 T2
        WHERE t2.c1 = x.c1) AS t2_c3
FROM (
    SELECT c1,
           c2,
           c3
    FROM SCALAR_T1 T1
    ORDER BY c1, c2
) x
WHERE ROWNUM <= 10 ;
```

SQL[1]과 SQL[2]는 작성방법은 다르지만  
동일한 데이터를 추출하는 SQL



## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리는 최종 결과 만큼 수행하자

◆ SQL[1], SQL[2] 트레이스 결과

SQL [1]

```
SELECT ROWNUM rnum,
       x.*
FROM (
  SELECT c1,
         c2,
         c3,
         (SELECT t2.c3
          FROM SCALAR_T2 T2
          WHERE t2.c1 = t1.c1) AS t2_c3
  FROM SCALAR_T1 T1
  ORDER BY c1, c2
) x
WHERE ROWNUM <= 10 ;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	2.02	2.05	0	1502292	0	10
total	4	2.02	2.05	0	1502292	0	10

Rows	Row Source Operation
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T2 (cr=1501114 pr=0 ...)
500000	INDEX RANGE SCAN SCALAR_T2_IDX_01 (cr=1001114 pr=0 pw=0 time=1084517 us)
10	COUNT STOPKEY (cr=1502292 pr=0 pw=0 time=2051304 us)
10	VIEW (cr=1502292 pr=0 pw=0 time=2051288 us)
10	SORT ORDER BY STOPKEY (cr=1502292 pr=0 pw=0 time=2051285 us)
500000	TABLE ACCESS FULL SCALAR_T1 (cr=1178 pr=0 pw=0 time=52 us)

SQL [2]

```
SELECT ROWNUM rnum,
       x.*,
       (SELECT t2.c3
        FROM SCALAR_T2 T2
        WHERE t2.c1 = x.c1) AS t2_c3
FROM (
  SELECT c1,
         c2,
         c3
  FROM SCALAR_T1 T1
  ORDER BY c1, c2
) x
WHERE ROWNUM <= 10 ;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.14	0.13	0	1210	0	10
total	4	0.14	0.13	0	1210	0	10

Rows	Row Source Operation
10	TABLE ACCESS BY INDEX ROWID SCALAR_T2 (cr=32 pr=0 pw=0 time=120 us)
10	INDEX RANGE SCAN SCALAR_T2_IDX_01 (cr=22 pr=0 pw=0 time=72 us)
10	COUNT STOPKEY (cr=1178 pr=0 pw=0 time=132413 us)
10	VIEW (cr=1178 pr=0 pw=0 time=132397 us)
10	SORT ORDER BY STOPKEY (cr=1178 pr=0 pw=0 time=132385 us)
500000	TABLE ACCESS FULL SCALAR_T1 (cr=1178 pr=0 pw=0 time=47 us)

## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제

- ◆ SQL[1] SCALAR\_T1 테이블만 FROM절에 위치 시킨 후 나머지는 스칼라 서브쿼리로 작성
- ◆ SQL[2] SCALAR\_T1, SCALAR\_T2, SCALAR\_T3를 모두 조인으로 작성

#### 작성 SQL [1]

```
SELECT t1.c1,
       t1.c2,
       t1.c3,
       (SELECT t2.C3
        FROM SCALAR_T2 T2
        WHERE t2.c1 = t1.c1) AS t2_c3,
       (SELECT t3.C3
        FROM SCALAR_T3 T3
        WHERE t3.c1 = t1.c1) AS t3_c3
FROM SCALAR_T1 T1
ORDER BY t1.c1, t1.c2 ;
```

전체 데이터를 추출

#### 작성 SQL [2]

```
SELECT ROWNUM rnum,
       X.*
FROM (
        SELECT /*+ USE_NL(T1 T2 T3) */
              t1.c1,
              t1.c2,
              t1.c3,
              t2.c3 AS t2_c3,
              t3.c3 AS t3_c3
        FROM SCALAR_T1 T1,
              SCALAR_T2 T2,
              SCALAR_T3 T3
        WHERE t1.c1 = t2.c1(+)
              AND t1.c1 = t3.c1(+)
        ORDER BY t1.c1, t1.c2 ) X
WHERE ROWNUM <= 10 ;
```

일부 데이터를 추출



## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

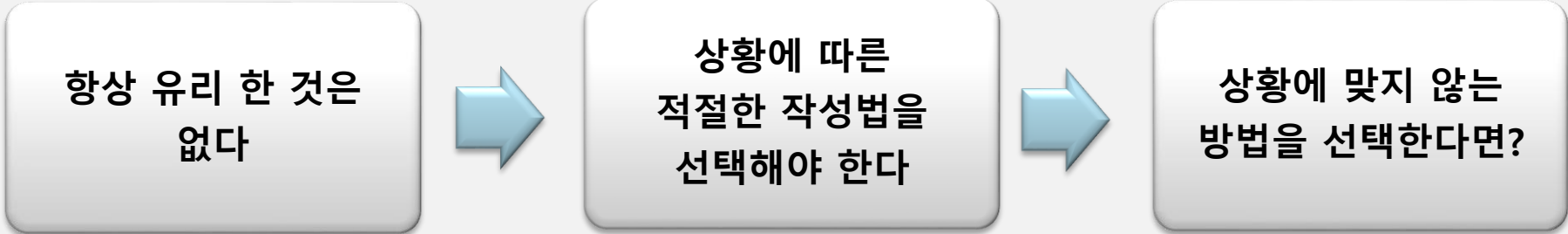
스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제



두가지 중  
어느 작성법이  
더 유리할까?

## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제



## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기



SQL[1] SQL[2]의 성능을 개선해 보자

## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제

◆ SQL[1] 개선 전 Trace결과

#### 작성 SQL [1]

```
SELECT t1.c1,
       t1.c2,
       t1.c3,
       (SELECT t2.C3
        FROM SCALAR_T2 T2
        WHERE t2.c1 = t1.c1) AS t2_c3,
       (SELECT t3.C3
        FROM SCALAR_T3 T3
        WHERE t3.c1 = t1.c1) AS t3_c3
FROM SCALAR_T1 T1
ORDER BY t1.c1, t1.c2 ;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	33335	4.18	4.20	0	3003406	0	500000
total	33337	4.18	4.20	0	3003406	0	500000

Rows	Row Source Operation
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T2 (cr=1501114 pr=0 pw=0 time=1597402 us)
500000	INDEX RANGE SCAN SCALAR_T2_IDX_01 (cr=1001114 pr=0 ...)
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T3 (cr=1501114 pr=0 pw=0 time=1640391 us)
500000	INDEX RANGE SCAN SCALAR_T3_IDX_01 (cr=1001114 pr=0 ...)
500000	SORT ORDER BY (cr=3003406 pr=0 pw=0 time=3775664 us)
500000	TABLE ACCESS FULL SCALAR_T1 (cr=1178 pr=0 pw=0 time=37 us)

대량의 데이터를 추출하는 SQL에서 반복적으로 수행되는 스칼라 서브쿼리로 인해 비효율 발생.



개선 포인트 : 반복으로 수행되는 비효율을 제거하자.

## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제

#### SQL [1] 개선 방향 도출

##### 현상 분석

- 1) SCALAR\_T1 추출건수는 50만건.
- 2) SCALAR\_T2와 SCALAR\_T3 (스칼라서브쿼리) 를 50만 번 반복적으로 수행.
- 3) 최종 추출건수는 50만건 모두 추출.



##### 개선 방향

- 즉 많은 데이터를 추출함에도 불과하고, 이를 스칼라 서브쿼리를 사용하여, 반복 Access에 의한 비효율이 발생.
- 스칼라 서브쿼리를 조인으로 변경한 후 반복적인 INDEX 탐색에 의한 비효율을 제거하기 위해 HASH JOIN으로 유도.





## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제

◆ SQL[2] 개선 전 Trace결과

#### 작성 SQL [2]

```
SELECT ROWNUM rnum,
       X.*
FROM (
    SELECT /*+ USE_NL(T1 T2 T3) */
           t1.c1,
           t1.c2,
           t1.c3,
           t2.c3 AS t2_c3,
           t3.c3 AS t3_c3
    FROM SCALAR_T1 T1,
         SCALAR_T2 T2,
         SCALAR_T3 T3
    WHERE t1.c1 = t2.c1(+)
          AND t1.c1 = t3.c1(+)
    ORDER BY t1.c1, t1.c2 ) X
WHERE ROWNUM <= 10 ;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	4.41	4.44	0	3003406	0	10
total	4	4.41	4.45	0	3003406	0	10

Rows	Row	Source	Operation
10	COUNT	STOPKEY	(cr=3003406 pr=0 pw=0 time=4448209 us)
10	VIEW		(cr=3003406 pr=0 pw=0 time=4448205 us)
10	SORT	ORDER BY	STOPKEY (cr=3003406 pr=0 pw=0 time=4448202 us)
500000	NESTED	LOOPS OUTER	(cr=3003406 pr=0 pw=0 time=4500101 us)
500000	NESTED	LOOPS OUTER	(cr=1502292 pr=0 pw=0 time=2000090 us)
500000	TABLE	ACCESS FULL	SCALAR_T1 (cr=1178 pr=0 pw=0 time=46 us)
500000	TABLE	ACCESS BY INDEX ROWID	SCALAR_T2 (cr=1501114 pr=0 ...)
500000	INDEX	RANGE SCAN	SCALAR_T2_IDX_01 (cr=1001114 pr=0 ...)
500000	TABLE	ACCESS BY INDEX ROWID	SCALAR_T3 (cr=1501114 pr=0 pw=0 time=1929888)
500000	INDEX	RANGE SCAN	SCALAR_T3_IDX_01 (cr=1001114 pr=0 ...)

최종 추출건수는 10건인데  
모두 조인을 시도하는 비효율 발생.



개선 포인트 : 최종 추출건수에 대해서만  
조인을 시도하자.



## 2. 스칼라 서브쿼리와 조인의 이해 및 활용하기

### 스칼라 서브쿼리와 조인의 관계로 보는 SQL 성능문제

#### SQL [2] 개선 방향 도출

##### 현상 분석

- 1) SCALAR\_T1 추출건수는 50만건.
- 2) SCALAR\_T2와 SCALAR\_T3 인덱스 스캔 50만번 반복 수행 (Nested Loops Join).
- 3) 최종 추출건수는 10건만 추출.



##### 개선 방향

- 10건만 추출하면 되나, Join을 모두 수행한 후 10건만 가져오므로 비효율 발생.
- SCALAR\_T2, SCALAR\_T3 데이터 조회 방법을 스칼라 서브쿼리로 변경하여 성능개선.



## 3. Summary

### Summary

스칼라 서브쿼리란?

스칼라 서브쿼리의 특징

스칼라 서브쿼리와 성능



CONTENTS



## WITH절에 대한 기본 내용 이해하기

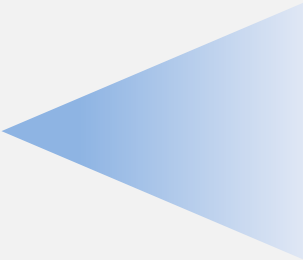
- WITH절에 대한 기본 내용 이해하기
- WITH절 동작 방식 이해하기
  - \* Materialize 동작 방식
  - \* Inline View 동작방식
- SQL 성능 개선을 위한 WITH절 활용하기
  - \* View Predicating 성능 문제 제거하기
  - \* 계층 쿼리의 데이터 최소화 하기
- WITH절 사용시 주의점

# 1. With절에 대한 기본 내용 이해하기

With절이란?

- ◆ SQL내에서 동일한 데이터가 반복적으로 사용되는 경우 With절을 사용한다.
- Materialize 동작 방식 (Global Temporary Table에 저장)
- Inline 동작 방식

Materialize  
VS  
Inline



Executions

Hint

## 2. With절 동작방식 이해하기

### Materialize 동작방식

```
WITH T_T1 AS (  
    SELECT *  
    FROM T1  
    WHERE c2 IN ('A','B','C')  
) , T_T2 AS (  
    SELECT *  
    FROM T2  
    WHERE c2 IN ('A','B','C')  
    AND c3 <= 10  
)  
SELECT t1.*,  
t2.*  
FROM T_T1 T1,  
T_T2 T2  
WHERE t1.c1 = T2.c1  
AND t1.c2 = 'A' ;
```

With절 T\_T1,  
T\_T2 각각 한번씩  
호출

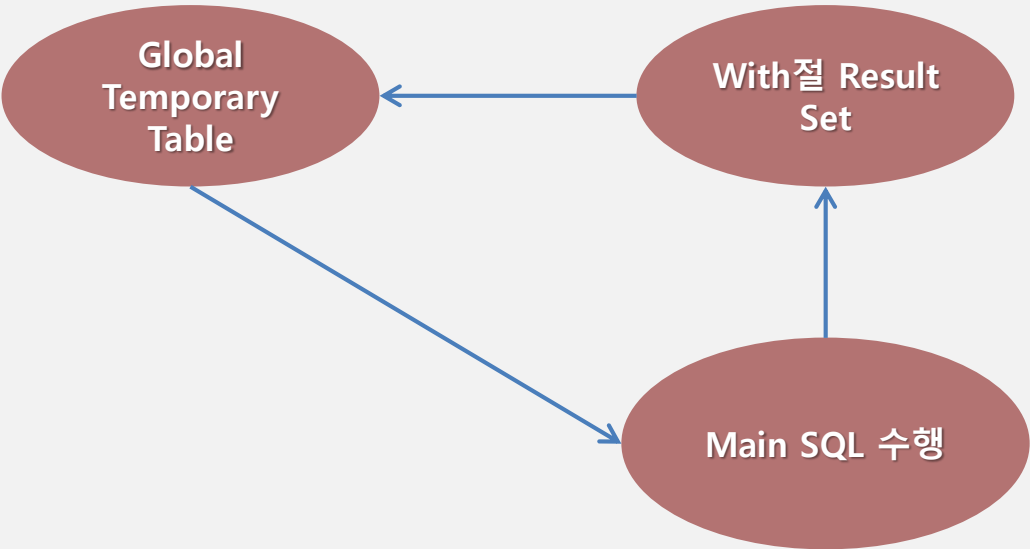


Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1092	17472	335 (2)	00:00:05
* 1	HASH JOIN		1092	17472	335 (2)	00:00:05
* 2	TABLE ACCESS FULL	T2	1092	10920	277 (1)	00:00:04
* 3	TABLE ACCESS FULL	T1	3870	23220	57 (4)	00:00:01

## 2. With절 동작방식 이해하기

### Materialize 동작방식

```
WITH T_T1 AS (  
    SELECT /*+ MATERIALIZE */  
        *  
    FROM T1  
    WHERE c2 IN ('A','B','C')  
) , T_T2 AS (  
    SELECT /*+ MATERIALIZE */  
        *  
    FROM T2  
    WHERE c2 IN ('A','B','C')  
    AND c3 <= 10  
)  
SELECT T1.*,  
    T2.*  
FROM T_T1 T1,  
    T_T2 T2  
WHERE t1.c1 = t2.c1  
    AND t1.c2 = 'A' ;
```



Id	Operation	Name	Rows	Bytes	Time
0	SELECT STATEMENT		122	5490	00:00:05
1	TEMP TABLE TRANSFORMATION				
2	LOAD AS SELECT				
* 3	TABLE ACCESS FULL	T1	11172	67032	00:00:01
4	LOAD AS SELECT				
* 5	TABLE ACCESS FULL	T2	1092	10920	00:00:04
* 6	HASH JOIN		122	5490	00:00:01
7	VIEW		1092	31668	00:00:01
8	TABLE ACCESS FULL	SYS_TEMP_OFD9D6683	1092	10920	00:00:00
* 9	VIEW		11172	174K	00:00:01
10	TABLE ACCESS FULL	SYS_TEMP_OFD9D6682	11172	67032	00:00:00

## 2. With절 동작방식 이해하기

### Inline 동작방식

```

WITH T_T1 AS (
    SELECT *
    FROM T1
    WHERE c2 IN ('A','B','C')
), T_T2 AS (
    SELECT *
    FROM T2
    WHERE c2 IN ('A','B','C')
    AND c3 <= 10
)
SELECT T1.*,
T2.*
FROM T_T1 T1,
T_T2 T2
WHERE t1.c1 = t2.c1
AND t1.c2 = 'A'
UNION ALL
SELECT T1.*,
T2.*
FROM T_T1 T1,
T_T2 T2
WHERE t1.c1 = t2.c1
AND t1.c2 = 'B' ;

```

With절 T\_T1,  
T\_T2 각각 한번  
이상 호출

## 2. With절 동작방식 이해하기

### Inline 동작방식

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		244	10980	13 (54)	00:00:01
1	TEMP TABLE TRANSFORMATION					
2	LOAD AS SELECT					
* 3	TABLE ACCESS FULL	T1	11172	67032	58 (6)	00:00:01
4	LOAD AS SELECT					
* 5	TABLE ACCESS FULL	T2	1092	10920	277 (1)	00:00:04
6	UNION-ALL					
* 7	HASH JOIN		122	5490	7 (15)	00:00:01
8	VIEW		1092	31668	2 (0)	00:00:01
9	TABLE ACCESS FULL	SYS_TEMP_0FD9D	1092	10920	2 (0)	00:00:00
* 10	VIEW		11172	174K	4 (0)	00:00:01
11	TABLE ACCESS FULL	SYS_TEMP_0FD9D	11172	67032	4 (0)	00:00:00
* 12	HASH JOIN		122	5490	7 (15)	00:00:01
13	VIEW		1092	31668	2 (0)	00:00:01
14	TABLE ACCESS FULL	SYS_TEMP_0FD9D	1092	10920	2 (0)	00:00:00
* 15	VIEW		11172	174K	4 (0)	00:00:01
16	TABLE ACCESS FULL	SYS_TEMP_0FD9D	11172	67032	4 (0)	00:00:00

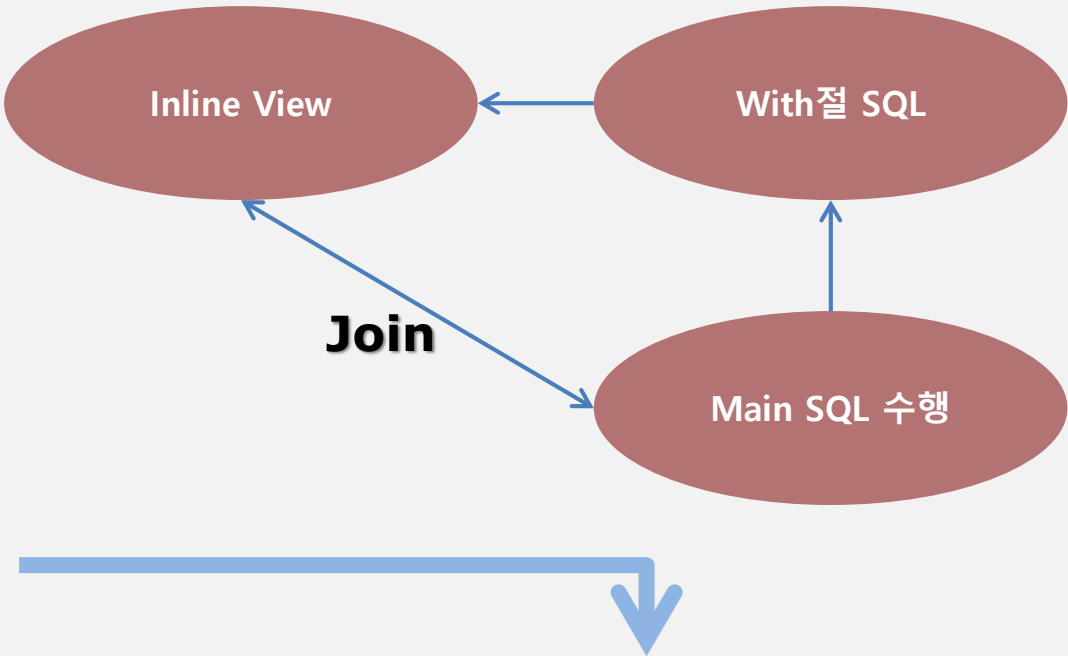




## 2. With절 동작방식 이해하기

### Inline 동작방식

```
WITH T_T1 AS (  
  SELECT /*+ INLINE */  
    *  
  FROM T1  
  WHERE c2 IN ('A','B','C')  
) , T_T2 AS (  
  SELECT /*+ INLINE */  
    *  
  FROM T2  
  WHERE c2 IN ('A','B','C')  
    AND c3 <= 10  
)  
SELECT T1.*,  
T2.*  
FROM T_T1 T1,  
T_T2 T2  
  WHERE t1.c1 = t2.c1  
    AND t1.c2 = 'A'  
UNION ALL  
SELECT T1.*,  
T2.*  
FROM T_T1 T1,  
T_T2 T2  
  WHERE t1.c1 = t2.c1  
    AND t1.c2 = 'B' ;
```



Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		2184	34944	669 (51)	00:00:09
1	UNION-ALL					
* 2	HASH JOIN		1092	17472	335 (2)	00:00:05
* 3	TABLE ACCESS FULL	T2	1092	10920	277 (1)	00:00:04
* 4	TABLE ACCESS FULL	T1	3870	23220	57 (4)	00:00:01
* 5	HASH JOIN		1092	17472	335 (2)	00:00:05
* 6	TABLE ACCESS FULL	T2	1092	10920	277 (1)	00:00:04
* 7	TABLE ACCESS FULL	T1	3687	22122	57 (4)	00:00:01

### 3. SQL 성능 개선을 위한 With절 활용하기

#### 데이터 중복 Access 제거하기

동일 데이터의  
반복 Access  
제거

◆ **Materialize 동작 방식을 이용하자!**

- With절의 Result Set을 Global Temporary Table에 저장한다.
- 여러 번 사용하더라도 저장된 Result Set만 읽어 처리할 수 있다.
- 따라서, I/O 처리량이 대폭 감소되어 양호한 SQL 수행 성능을 기대할 수 있다.
- 단, With절 Result set의 추출건수가 많다면?

Global Temporary Table에 저장하는 비용증가!

## 3. SQL 성능 개선을 위한 With절 활용하기

### View Predicating 성능문제 제거하기

#### View Predicating?

◆ **Inline View 외부의 조건을 Inline View 내부로 침투시키는 것!**

- 간혹, 뷰 외부 조건이 내부로 침투되지 못하는 경우 발생한다.
- 이 경우 뷰의 데이터를 모두 처리한 이후에 조인 연결 조건을 Filter 조건으로 사용하여 심각한 성능 문제를 유발하는 경우가 있다.
- 사례를 통해 자세히 알아보자.

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

##### ◆ View Predicating 성능문제 SQL

```

SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 T1,
     WITH_T2 T2,
(
SELECT /*+ NO_MERGE */
      c1, c2, sum(c3) c3
  FROM WITH_T3
 GROUP BY c1, c2
) T3
WHERE t1.c1 = t2.c1(+)
      AND t1.c1 = t3.c1(+)
      AND t1.c2 = 'A'
      AND t1.c3 <= 11000;

```

##### ◆ 전제조건

- Outer Join이므로 with\_T1테이블을 먼저 수행한다.
- T1과 조인 연결컬럼인 T2와 T3 테이블의 c1 컬럼 값은 각각 unique 하다.

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL

```
SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 T1,
     WITH_T2 T2,
(
  SELECT /*+ NO_MERGE */
        c1, c2, sum(c3) c3
    FROM WITH_T3
   GROUP BY c1, c2
) T3
WHERE t1.c1 = t2.c1(+)
      AND t1.c1 = t3.c1(+)
      AND t1.c2 = 'A'
      AND t1.c3 <= 11000;
```



Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		39	1716
* 1	HASH JOIN OUTER		39	1716
2	NESTED LOOPS OUTER		39	702
3	TABLE ACCESS BY INDEX ROWID	WITH_T1	39	429
* 4	INDEX RANGE SCAN	WITH_T1_IDX_02	39	
5	TABLE ACCESS BY INDEX ROWID	WITH_T2	1	7
* 6	INDEX RANGE SCAN	WITH_T2_IDX_01	1	
7	VIEW		500K	12M
8	HASH GROUP BY		500K	5371K
9	TABLE ACCESS FULL	WITH_T3	500K	5371K

조인 조건이 인라인 뷰 T3에 침투  
되지 못하면서 With\_T3 테이블에  
Table Full Scan 발생!

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL – View Predicating이 이루어지면 효율적인가?

```
SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 T1,
     WITH_T2 T2,
     (
       SELECT /*+ NO_MERGE */
              c1, c2, sum(c3) c3
              FROM WITH_T3
              GROUP BY c1, c2
     ) T3
WHERE t1.c1 = t2.c1(+)
      AND t1.c1 = t3.c1(+)
      AND t1.c2 = 'A'
      AND t1.c3 <= 11000;
```

```
SELECT count(*)
FROM WITH_T1
WHERE c2 = 'A'
AND c3 <= 11000 ;

COUNT(*)
-----
38 ----> 38건 추출
```

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL – 개선안 도출

#### 개선안

- ◆ With\_T1에서 추출한 값을 인라인 뷰로 만든 이후, 인라인 뷰 T3에 강제로 조건을 추가하는 방법.
- ◆ With절을 선언하여 필요한 데이터를 미리 추출한 후, 필요할 때마다 재 사용하도록 SQL을 작성하는 방법.

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL – 첫 번째 개선안

```
SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 T1,
     WITH_T2 T2,
(
SELECT /*+ NO_MERGE */
      c1, c2, sum(c3) c3
  FROM WITH_T3
 GROUP BY c1, c2
) T3
WHERE t1.c1 = t2.c1(+)
      AND t1.c1 = t3.c1(+)
      AND t1.c2 = 'A'
      AND t1.c3 <= 11000;
```



```
SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 t1
   ,WITH_T2 t2
   ,( SELECT /*+ LEADING(T1) USE_NL(T1 T3) */
      t3.c1, t3.c2, SUM(t3.c3) c3
    FROM WITH_3 t3,
      (
        SELECT c1, c2
        FROM WITH_T1
        WHERE c2 = 'A' AND c3 <= 11000
      ) t1
    WHERE t1.c1 = t3.c1
    GROUP BY t3.c1, t3.c2 ) t3
WHERE t1.c1 = t2.c1(+)
      AND t1.c1 = t3.c1(+)
      AND t1.c2 = 'A' AND t1.c3 <= 11000 ;
```



### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL – 첫 번째 개선안

```
SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 t1
,WITH_T2 t2
,( SELECT /*+ LEADING(T1) USE_NL(T1 T3) */
    t3.c1, t3.c2, SUM(t3.c3) c3
  FROM WITH_3 t3,
    (
      SELECT c1, c2
      FROM WITH_T1
      WHERE c2 = 'A' AND c3 <= 11000
    ) t1
  WHERE t1.c1 = t3.c1
  GROUP BY t3.c1, t3.c2 ) t3
WHERE t1.c1 = t2.c1(+)
AND t1.c1 = t3.c1(+)
AND t1.c2 = 'A' AND t1.c3 <= 11000 ;
```

Id	Operation	Name	Rows	Bytes
0	SELECT STATEMENT		39	1716
1	NESTED LOOPS OUTER		39	1716
* 2	HASH JOIN OUTER		39	1443
3	TABLE ACCESS BY INDEX ROWID	WITH_T1	39	429
* 4	INDEX RANGE SCAN	WITH_T1_IDX_02	39	
5	VIEW		38	988
	GROUP BY		38	836
* 6	TABLE ACCESS BY INDEX ROWID	WITH_T3	1	11
	NESTED LOOPS		38	836
7	TABLE ACCESS BY INDEX ROWID	WITH_T1	39	429
* 10	INDEX RANGE SCAN	WITH_T1_IDX_02	39	
* 11	INDEX RANGE SCAN	WITH_T3_IDX_01	1	
12	TABLE ACCESS BY INDEX ROWID	WITH_T2	1	7
* 13	INDEX RANGE SCAN	WITH_T2_IDX_01	1	

Good?

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL – 두 번째 개선안

```
SELECT t1.c1,
       t1.c2,
       t2.c1,
       t2.c2,
       t3.c3
FROM WITH_T1 T1,
     WITH_T2 T2,
(
SELECT /*+ NO_MERGE */
      c1, c2, sum(c3) c3
  FROM WITH_T3
 GROUP BY c1, c2
) T3
WHERE t1.c1 = t2.c1(+)
      AND t1.c1 = t3.c1(+)
      AND t1.c2 = 'A'
      AND t1.c3 <= 11000;
```



```
WITH t1 AS (
  SELECT /*+ materialize */
        c1 ,
        c2
  FROM   with_t1
 WHERE  c2 = 'A'
 AND    c3 <= 11000
)
SELECT t1.c1 ,
       t1.c2 ,
       t2.c1 ,
       t2.c2 ,
       t3.c3
FROM   t1 ,
       with_t2 t2 ,
(
  SELECT /*+ leading(t1) use_nl(t1 t3) */
        t3.c1 ,
        t3.c2 ,
        SUM( t3.c3 ) c3
  FROM   with_t3 t3 ,
        t1
 WHERE  t1.c1 = t3.c1
 AND    t1.c2 = t3.c2
 GROUP  BY t3.c1 ,
          t3.c2
) t3
WHERE  t1.c1 = t2.c1( + )
 AND   t1.c1 = t3.c1( + );
```

### 3. SQL 성능 개선을 위한 With절 활용하기

#### View Predicating 성능문제 제거하기

◆ View Predicating 성능문제 SQL – 두 번째 개선안

Id	Operation	Name	Rows
0	SELECT STATEMENT		39
1	TEMP TABLE TRANSFORMATION		
2	LOAD AS SELECT		
3	TABLE ACCESS BY INDEX ROWID	WITH_T1	39
* 4	INDEX RANGE SCAN	WITH_T1_IDX_02	39
5	NESTED LOOPS OUTER		39
* 6	HASH JOIN OUTER		39
7	VIEW		39
8	TABLE ACCESS FULL	SYS_TEMP_OFD9D6602_16CD8855	39
9	VIEW		39
10	HASH GROUP BY		39
* 11	TABLE ACCESS BY INDEX ROWID	WITH_T3	1
12	NESTED LOOPS		39
13	VIEW		39
14	TABLE ACCESS FULL	SYS_TEMP_OFD9D6602_16CD8855	39
* 15	INDEX RANGE SCAN	WITH_T3_IDX_01	1
* 16	TABLE ACCESS BY INDEX ROWID	WITH_T2	1
* 17	INDEX RANGE SCAN	WITH_T2_IDX_01	1

With절 수행부분

SQL 수행부분

Good!

### 3. SQL 성능 개선을 위한 With절 활용하기

#### 계층 쿼리의 데이터 최소화 하기

◆ 계층 쿼리 성능문제 SQL

```
SELECT apt_id ,
       COUNT( subsno ) apt_cnt_subsno_1 ,
       SUBSTR( MAX( sys_connect_by_path( subsno , ',' ) ) , 2 )
subsno_lst
FROM (
  SELECT a.* ,
         ROW_NUMBER( ) over( PARTITION BY tmp_key
                               ORDER BY subsno ) rnum
  FROM (
    SELECT ...
    FROM   tb_logdata a
    WHERE  ... 생략 ...
    GROUP BY a.apt_id ,
             a.subsno
    HAVING COUNT( a.subsno ) >= 2
  ) a
) b
START WITH rnum = 1
CONNECT BY PRIOR rnum = rnum - 1
AND PRIOR tmp_key = tmp_key
GROUP BY apt_id ;
```

계층구조 분석대상 데이터 추출  
부분

Start With

Connect By

Where

### 3. SQL 성능 개선을 위한 With절 활용하기

#### 계층 쿼리의 데이터 최소화 하기

◆ 계층 쿼리 성능문제 SQL – Optimizer의 실수로 인한 비효율 발생

```
SELECT apt_id ,
       COUNT( subsno ) apt_cnt_subsno_1 ,
       SUBSTR( MAX( sys_connect_by_path( subsno , ',' ) ) , 2 )
subsno_lst
FROM (
    SELECT a.* ,
           ROW_NUMBER( ) over( PARTITION BY tmp_key
                                ORDER BY subsno ) rnum
    FROM (
        SELECT ...
        FROM   tb_logdata a
        WHERE  ... 생략 ...
        GROUP BY a.apt_id ,
                 a.subsno
        HAVING COUNT( a.subsno ) >= 2
    ) a
    ) b
START WITH rnum = 1
CONNECT BY PRIOR rnum = rnum - 1
AND PRIOR tmp_key = tmp_key
GROUP BY apt_id ;
```



계층구조 데이터  
분석



Group By 수행

### 3. SQL 성능 개선을 위한 With절 활용하기

#### 계층 쿼리의 데이터 최소화 하기

◆ 계층 쿼리 성능문제 SQL – 개선 후 SQL

```
WITH temp_t1 AS (  
  SELECT /*+ materialize */  
    a.*,  
    row_number() over(partition by tmp_key order by  
subsno) rnum  
  FROM (  
    SELECT /*+ full(a) */  
      ...  
    FROM tb_logdata a  
    WHERE ... 생략 ...  
    GROUP BY a.apl_id, a.subsno  
    HAVING count(a.subsno) >= 2  
  ) a  
)  
SELECT apl_id ,  
  count(subsno) apl_cnt_subsno_1 ,  
  substr(max(sys_connect_by_path(subsno, ','), 2) subsno_lst  
FROM temp_t1 ---> With절에서 선언한 명칭  
START WITH rnum = 1  
CONNECT BY PRIOR rnum = rnum - 1  
  AND PRIOR tmp_key = tmp_key  
GROUP BY apl_id ;
```

With절을 이용하여 대상  
데이터를 미리 추출

Grouping된 데이  
터를 미리 추출



해당 데이터로 계  
층구조 분석

## 4. With절 사용시 주의점?

### 동시성이 높은 경우

- ◆ 동시성이 높은 SQL을 Materialize 방식으로 수행한다면?
  - Control file sequential reads Wait Event 대기현상으로 인한 성능저하.

### 많은 추출건수

- ◆ With절의 Result Set 건수가 많은 경우 Materialize 방식으로 수행한다면?
  - Global Temporary Table에 많은 데이터를 저장하고 읽어야 하는 부하가 커진다.

### With절의 위치

- ◆ SQL의 가장 앞에 위치시키자!
  - SQL의 데이터 정합성이 훼손될 여지가 있다.

### Hint

- ◆ With절 동작방식을 결정짓는 힌트를 사용하자.
  - With절로 SQL구조를 변경하더라도 힌트를 적용하지 않으면 동작방식에는 변화가 없을 수 있기 때문이다.

## Summary

With절의 기본

With절 동작방식 이해

With절의 활용

With절 사용시 주의사항





## SQL 튜닝 기초 이해하기

- Join 동작 방식
- Plan 읽는 방법
- Trace 분석 방법



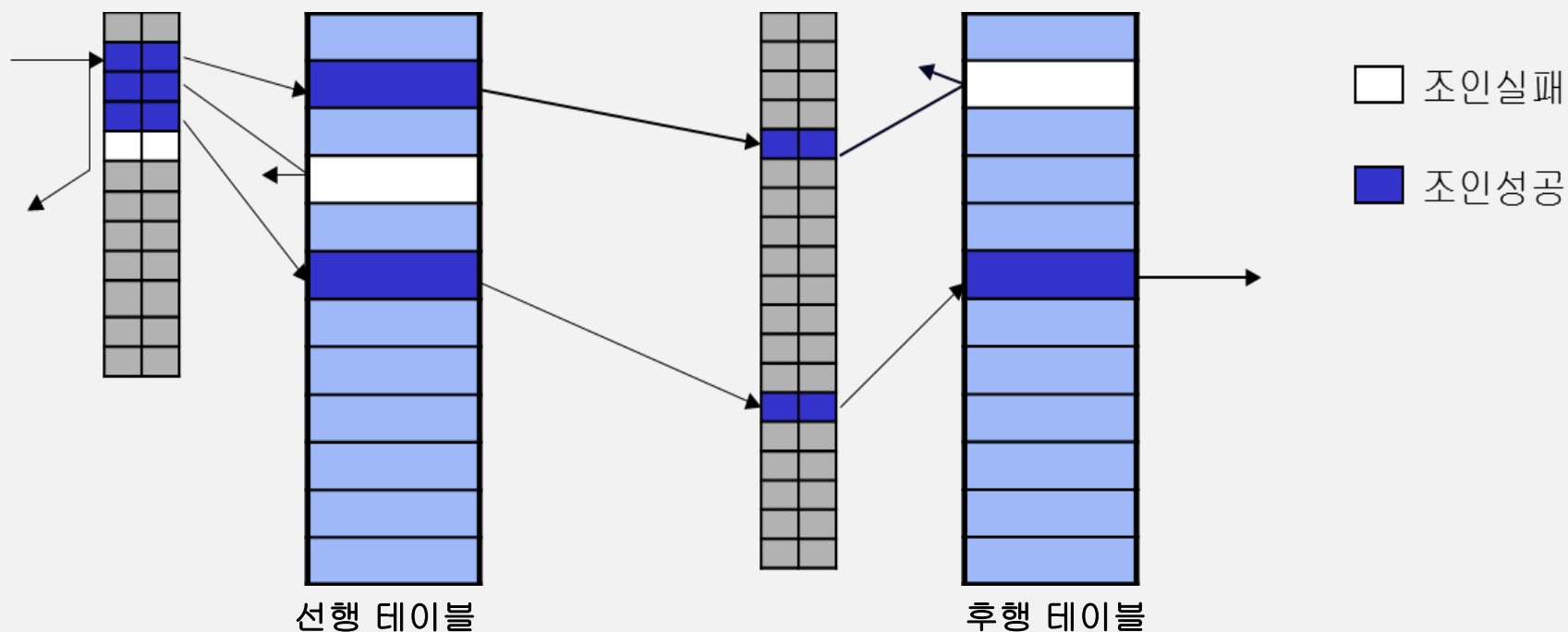
CONTENTS

# 1. SQL 튜닝 기본 지식

## 조인 방법 – Nested Loops Join

선행 테이블의 추출건수가 많으면 비효율적이다.

후행 테이블에서 Join 조건에 대한 INDEX를 활용할 수 있다.

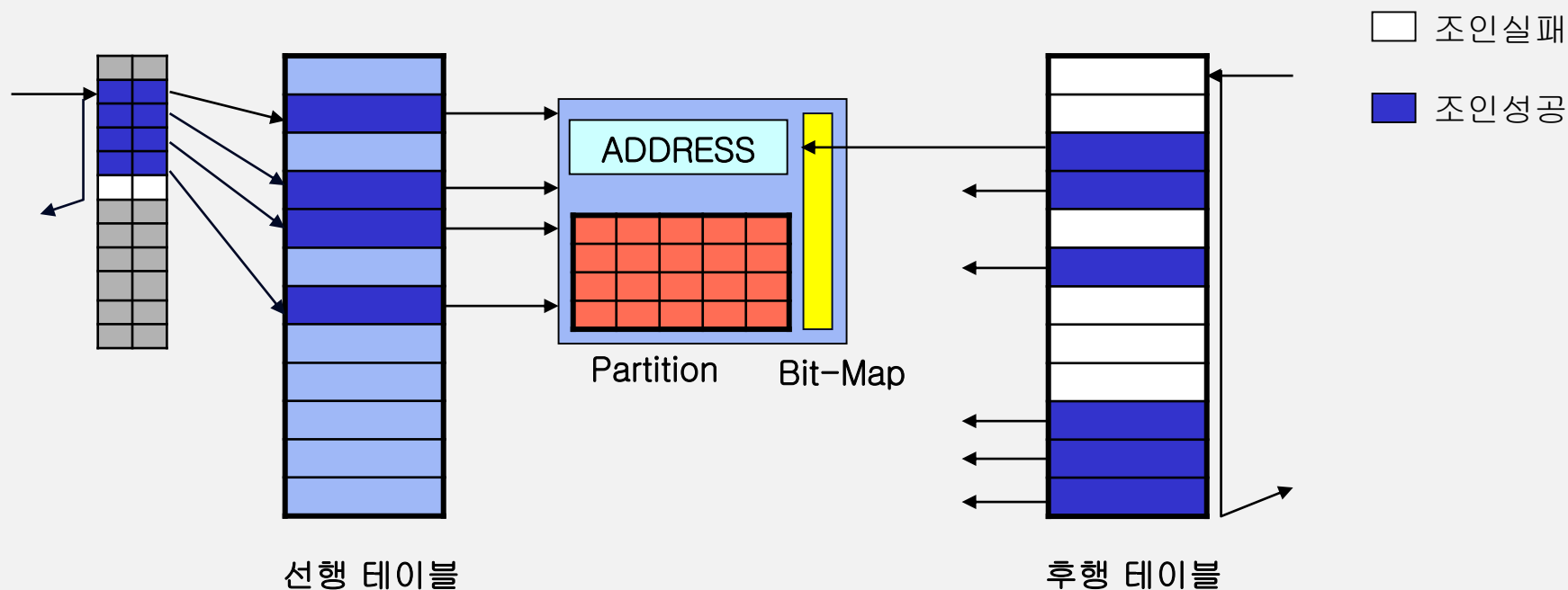


# 1. SQL 튜닝 기본 지식

## 조인 방법 - Hash Join

선행 테이블의 추출건수가 많아 후행 테이블을 반복적으로 탐색하면서 발생하는 비효율을 제거 할 때 유리하다.

후행 테이블에서 Join 조건에 대한 INDEX를 활용할 수 없다.



# 1. SQL 튜닝 기본 지식

## PLAN을 읽는 방법

안에서 밖으로, 위에서 아래로, JOIN은 PAIR로, 각 Operation에 따라

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c1 >= :b1 AND c1 <= :b2
AND EXISTS ( SELECT /*+ UNNEST HASH_SJ */
              'x'
              FROM SUBQUERY_T1 t1
              WHERE t1.c6 = t2.c3
              AND t1.c6 >= :b1 )
```

INDEX_NAME	COLUMN_NAME
PK_SUQUERY_2	C1
SUBQUERY_T1_IDX_01	C4, C5
SUBQUERY_T1_IDX_02	C5

Rows	Row Source OPERATION
11	FILTER (cr=37422 pr=0 pw=0 time=1910470 us) ①
11	HASH JOIN SEMI (cr=37422 pr=0 pw=0 time=1910466 us) ②
221	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=5 pr=0 pw=0 time=42 us) ③
221	INDEX RANGE SCAN PK_SUQUERY_2 (cr=3 pr=0 pw=0 time=31 us) ④
6400640	TABLE ACCESS FULL SUBQUERY_T1 (cr=37417 pr=0 pw=0 time=6261 us) ⑤



# 1. SQL 튜닝 기본 지식

## PLAN을 읽는 방법

```
SELECT ROWNUM rnum,
       X.*
FROM (
  SELECT /*+ USE_NL(T1 T2 T3) */
        t1.c1,
        t1.c2,
        t1.c3,
        t2.c3 AS t2_c3,
        t3.c3 AS t3_c3
  FROM SCALAR_T1 T1,
       SCALAR_T2 T2,
       SCALAR_T3 T3
 WHERE t1.c1 = t2.c1(+)
       AND t1.c1 = t3.c1(+)
       ORDER BY t1.c1, t1.c2 ) X
WHERE ROWNUM <= 10 ;
```

Rows	Row Source Operation	
-----	-----	
10	COUNT STOPKEY (cr=3003406 pr=0 pw=0 time=4448209 us)	①
10	VIEW (cr=3003406)	②
10	SORT ORDER BY STOPKEY (cr=3003406)	③
500000	NESTED LOOPS OUTER (cr=3003406)	④
500000	NESTED LOOPS OUTER (cr=1502292)	⑤
500000	TABLE ACCESS FULL SCALAR_T1 (cr=1178)	⑥
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T2 (cr=1501114)	⑦
500000	INDEX RANGE SCAN SCALAR_T2_IDX_01 (cr=1001114)	⑧
500000	TABLE ACCESS BY INDEX ROWID SCALAR_T3 (cr=1501114)	⑨
500000	INDEX RANGE SCAN SCALAR_T3_IDX_01 (cr=1001114)	⑩

수행 순서는?



- ⑥ ⑧ ⑦ ⑤ ⑩ ⑨ ④ ③ ② ①

# 1. SQL 튜닝 기본 지식

## Trace 분석 방법

```
SELECT c1, c2, c3
FROM SUBQUERY_T2 t2
WHERE c1 >= :b1 AND c1 <= :b2
AND EXISTS ( SELECT /*+ UNNEST HASH_SJ */
              'x'
              FROM SUBQUERY_T1 t1
              WHERE t1.c6 = t2.c3
              AND t1.c6 >= :b1 )
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.90	1.91	0	37422	0	11
total	4	1.90	1.92	0	37422	0	11

Rows      Row Source OPERATION

11	FILTER (cr=37422 pr=0 pw=0 time=1910470 us)
11	HASH JOIN SEMI (cr=37422 pr=0 pw=0 time=1910466 us)
221	TABLE ACCESS BY INDEX ROWID SUBQUERY_T2 (cr=5 pr=0 pw=0 time=42 us)
221	INDEX RANGE SCAN PK_SUQUERY_2 (cr=3 pr=0 pw=0 time=31 us)
6400640	TABLE ACCESS FULL SUBQUERY_T1 (cr=37417 pr=0 pw=0 time=6261 us)

## Merge 구문의 이해와 효율적인 SQL 작성하기



- Merge 구문의 구성요소 알기

- Merge 구문 작성 시 발생할 수 있는 에러와 해결방법

- \* USING절 컬럼 중에서 On절에 사용된 컬럼 값은 반드시 Unique 해야 한다
- \* UPDATE 컬럼은 ON 절에 사용할 수 없다

- Merge 구문은 다양한 형태의 DML을 지원한다

- \* 다양한 CASE 1 / 2 / 3

- Merge 구문을 성능 문제에 활용하자

- \* 대량 UPDATE시 SET절 서브쿼리 성능을 개선해 보자
- \* 어플리케이션 로직에 따른 MERGE 구문 활용

# 1. MERGE 구문의 구성요소 알기

## MERGE

◆ Oracle에서 UPDATE & INSERT의 MULTIPLE OPERATION을 지원하기 위한 DML 구문이다.

➤ UPSERT, Only UPDATE를 처리하는데 많이 사용된다.

구성 요소	예 문	설 명
INTO	MERGE /*+ <b>LEADING(st)</b> */ INTO merge_t1 tt	<ul style="list-style-type: none"><li>➤ Target Table 지정.</li><li>➤ Hint 사용이 가능함.</li></ul>
USING	USING ( SELECT /*+ <b>FULL(t2)</b> */ c1, c2, c3 FROM MERGE_T2 t2 WHERE c1 >= 99990 AND c1 <= 100090 ) st	<ul style="list-style-type: none"><li>➤ Source Table을 지정.</li><li>➤ Hint 사용이 가능함.</li><li>➤ On절에서 Target Table과 조인할 컬럼은 반드시 Unique 해야 함.</li></ul>
ON	ON (tt.c1 = st.c1) <b>WHEN MATCHED THEN</b> UPDATE SET tt.c2 = st.c2, tt.c3 = st.c3 DELETE WHERE (tt.c2 = 'A') <b>WHEN NOT MATCHED THEN</b> INSERT (tt.c1, tt.c2, tt.c3) values (st.c1, st.c2, st.c3) WHERE (st.c2 = 'A');	<ul style="list-style-type: none"><li>➤ Target Table의 데이터 중 Source Table에서 일치하는 데이터 인지 체크 함. ON (tt.c1 = st.c1) JOIN</li><li>➤ 일 치 : UPDATE와 DELETE 수행 불일치 : INSERT 수행</li></ul>



## 2. MERGE 구문 작성 시 발생할 수 있는 에러와 해결방법

USING절 컬럼 중에서 On절에 사용된 컬럼 값은 반드시 Unique 해야 한다

MERGE\_T1 테이블과 MERGE\_T2 테이블의 통계정보

◆ [TABLE (Column) STATISTICS ]  
Table : merge\_t1

COLUMN_NAME	DATA_TYPE	DATA LEN	PREC SCAL	NUMBER N	NUMBER DISTINCT	NUM_NULLS
C1	NUMBER	22		Y	100000	0
C2	VARCHAR2	2		Y	26	0
C3	NUMBER	22		Y	100000	0

◆ [TABLE (Column) STATISTICS ]  
Table : merge\_t2

COLUMN_NAME	DATA_TYPE	DATA LEN	PREC SCAL	NUMBER N	NUMBER DISTINCT	NUM_NULLS
C1	NUMBER	22		Y	500000	0
C2	VARCHAR2	2		Y	26	0
C3	NUMBER	22		Y	9	50000

## 2. MERGE 구문 작성 시 발생할 수 있는 에러와 해결방법

### USING절 컬럼 중에서 On절에 사용된 컬럼 값은 반드시 Unique 해야 한다

ON절에 조인으로 사용된 컬럼 값이 Unique 하지 않다면, MERGE 구문은 에러가 발생한다.

```
MERGE INTO MERGE_T1 tt
USING (
  SELECT c1, c2, c3
  FROM MERGE_T2, (SELECT LEVEL FROM DUAL CONNECT BY LEVEL <= 2)
  WHERE c1 >= 99990
  AND c1 <= 100090
) st
ON ( tt.c1 = st.c1 )
WHEN MATCHED THEN
  UPDATE SET tt.c2 = st.c2, tt.c3 = st.c3
  DELETE WHERE (tt.c2 = 'A')
WHEN NOT MATCHED THEN
  INSERT (tt.c1, tt.c2, tt.c3) VALUES (st.c1, st.c2, st.c3)
  WHERE (st.c2 = 'A') ;
```

임의적으로  
중복 값을 생성

```
MERGE INTO merge_t1 tt
*
ERROR at line 1:
ORA-30926: unable to get a stable set of rows in the source tables
```

컬럼에 중복 값이  
존재하여  
에러 발생

## 2. MERGE 구문 작성 시 발생할 수 있는 에러와 해결방법

**USING절 컬럼 중에서 On절에 사용된 컬럼 값은 반드시 Unique 해야 한다**

DISTINCT나 GROUP BY 처리를 하여 중복을 제거하면, 정상적인 처리가 가능하다.

```
MERGE INTO MERGE_T1 tt
USING (
  SELECT DISTINCT c1, c2, c3
  FROM MERGE_T2, (SELECT LEVEL FROM DUAL CONNECT BY LEVEL <= 2)
  WHERE c1 >= 99990
  AND c1 <= 100090
) st
ON ( tt.c1 = st.c1 )
WHEN MATCHED THEN
  UPDATE SET tt.c2 = st.c2, tt.c3 = st.c3
  DELETE WHERE (tt.c2 = 'A')
WHEN NOT MATCHED THEN
  INSERT (tt.c1, tt.c2, tt.c3) VALUES (st.c1, st.c2, st.c3)
  WHERE (st.c2 = 'A') ;
```

**DISTINCT를 사용하여  
중복을 제거하자  
정상 수행**

14 rows merged.

## 2. MERGE 구문 작성 시 발생할 수 있는 에러와 해결방법

### UPDATE 컬럼은 ON절에 사용할 수 없다

UPDATE 대상 컬럼을 ON절에 다시 사용 할 경우 ERROR가 발생한다.

```
MERGE INTO MERGE_T1 tt
USING (
  SELECT c1, c2, c3
  FROM MERGE_T2
  WHERE c1 >= 99990
  AND c1 <= 100090
) st
ON ( tt.c1 = st.c1 )
WHEN MATCHED THEN
  UPDATE SET tt.c1 = st.c1, tt.c2 = st.c2, tt.c3 = st.c3
  DELETE WHERE (tt.c2 = 'A')
WHEN NOT MATCHED THEN
  INSERT (tt.c1, tt.c2, tt.c3) VALUES (st.c1, st.c2, st.c3)
  WHERE (st.c2 = 'A')
```

ON절에 사용된  
tt.c1을 UPDATE  
수행.

```
tt.c1 = st.c1
*
ERROR at line 9:
ORA-38104: Columns referenced in the ON절 cannot be updated: "TT"."C1"
```

ON절에 사용된  
C1컬럼을 UPDATE  
하여 에러 발생.

## 2. MERGE 구문 작성 시 발생할 수 있는 에러와 해결방법

### UPDATE 컬럼은 ON절에 사용할 수 없다

ROWID를 활용하면, 에러 발생 없이 처리할 수 있다.

```
MERGE INTO MERGE_T1 tt
USING (
    SELECT st.c1, st.c2, st.c3, tt.ROWID as rid
    FROM MERGE_T2 st, MERGE_T1 tt
    WHERE st.c1 >= 99990
    AND st.c1 <= 100090
    AND st.c1 = tt.c1(+)
) st
ON ( tt.ROWID = st.rid )
WHEN MATCHED THEN
    UPDATE SET tt.c1 = st.c1, tt.c2 = st.c2, tt.c3 = st.c3
    DELETE WHERE (tt.c2 = 'A')
WHEN NOT MATCHED THEN
    INSERT (tt.c1, tt.c2, tt.c3) VALUES (st.c1, st.c2, st.c3)
    WHERE (st.c2 = 'A') ;
```

ON절에 tt.c1 대신  
ROWID를 사용하면  
정상 수행됨.

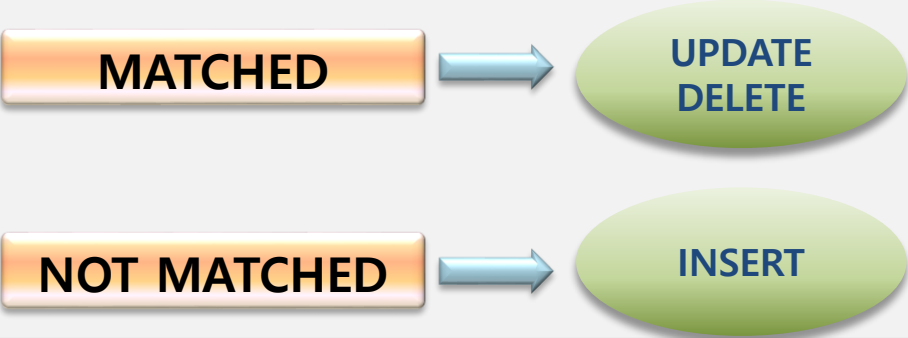
14 rows merged.

### 3. MERGE 구문은 다양한 형태의 DML을 지원한다

#### MEGER 구문으로 처리 가능한 트랜잭션 유형 CASE 1

CASE 1. UPDATE & DELETE & INSERT 또는 UPDATE & INSERT

```
MERGE INTO MERGE_T1 tt
USING ( SELECT st.c1, st.c2, st.c3
        FROM MERGE_T2 st
        WHERE st.c1 >= 99990
              AND st.c1 <= 100090 ) ST
on ( tt.c1 = st.c1 )
WHEN MATCHED THEN
    UPDATE SET tt.c2 = st.c2, tt.c3 = st.c3
    DELETE WHERE (tt.c2 = 'A')
WHEN NOT MATCHED THEN
    INSERT (tt.c1, tt.c2, tt.c3) VALUES (st.c1, st.c2, st.c3)
    WHERE (st.c2 = 'A') ;
```

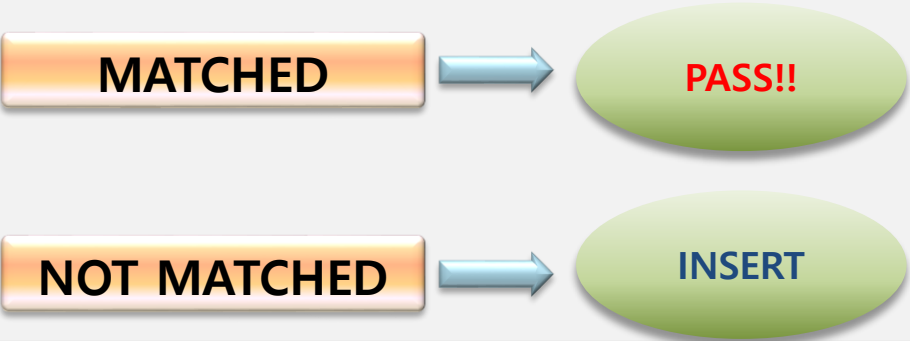


### 3. MERGE 구문은 다양한 형태의 DML을 지원한다

#### MEGER 구문으로 처리 가능한 트랜잭션 유형 CASE 2

CASE 2. Only INSERT

```
MERGE INTO MERGE_T1 tt
USING (      SELECT st.c1, st.c2, st.c3
              FROM MERGE_T2 st
              WHERE st.c1 >= 99990  AND st.c1 <= 100090 ) st
ON ( tt.c1 = st.c1 )
WHEN NOT MATCHED THEN
  INSERT (tt.c1, tt.c2, tt.c3) VALUES(st.c1, st.c2, st.c3)
  WHERE (st.c2 = 'A') ;
```



### 3. MERGE 구문은 다양한 형태의 DML을 지원한다

#### MEGER 구문으로 처리 가능한 트랜잭션 유형 CASE 3

CASE 3. Only UPDATE or UPDATE & DELETE

```
MERGE INTO MERGE_T1 tt
USING ( SELECT st.c1, st.c2, st.c3
        FROM MERGE_T2 st
        WHERE st.c1 >= 99990
              AND st.c1 <= 100090 ) st
ON ( tt.c1 = st.c1 )
WHEN MATCHED THEN
  UPDATE SET tt.c2 = st.c2, tt.c3 = st.c3
  DELETE WHERE (tt.c2 = 'A') ;
```

**MATCHED**



**UPDATE  
DELETE**

**NOT MATCHED**



**PASS!!**



# 4. MERGE 구문을 성능 문제에 활용하자

## 대량의 UPDATE시 SET절 서브쿼리 성능을 개선해 보자

SET절에 서브쿼리가 존재하는 UPDATE문

```
UPDATE emp a
SET  ename = (SELECT  dname
                FROM    dept b
                WHERE   a.deptno = b.deptno)
WHERE a.empno > 0 ;
```

UPDATE 대상이  
100만 건이라면?

최대 100 만번  
수행됨.

SET 절 서브쿼리가 반복 수행됨으로써  
비효율 발생 !!  
  
(단 DEPTNO 값의 종류가 많다고 가정)

## 4. MERGE 구문을 성능 문제에 활용하자

대량의 UPDATE시 SET절 서브쿼리 성능을 개선해 보자

9i -> UPDATABLE JOIN VIEW를 이용해 처리하자.

```
UPDATE /*+ BYPASS_UJVC LEADING(A) USE_HASH(B) */  
( SELECT b.dname, a.ename, a.deptno  
  FROM emp a, dept b  
 WHERE a.deptno = b.deptno (+)  
       AND a.empno > 0 )  
SET ename = dname ;
```



조인으로 한번에  
처리

## 4. MERGE 구문을 성능 문제에 활용하자

대량의 UPDATE시 SET절 서브쿼리 성능을 개선해 보자

MERGE 구문을 이용하여 PARALLEL DML 처리

```
ALTER SESSION ENABLE PARALLEL DML;

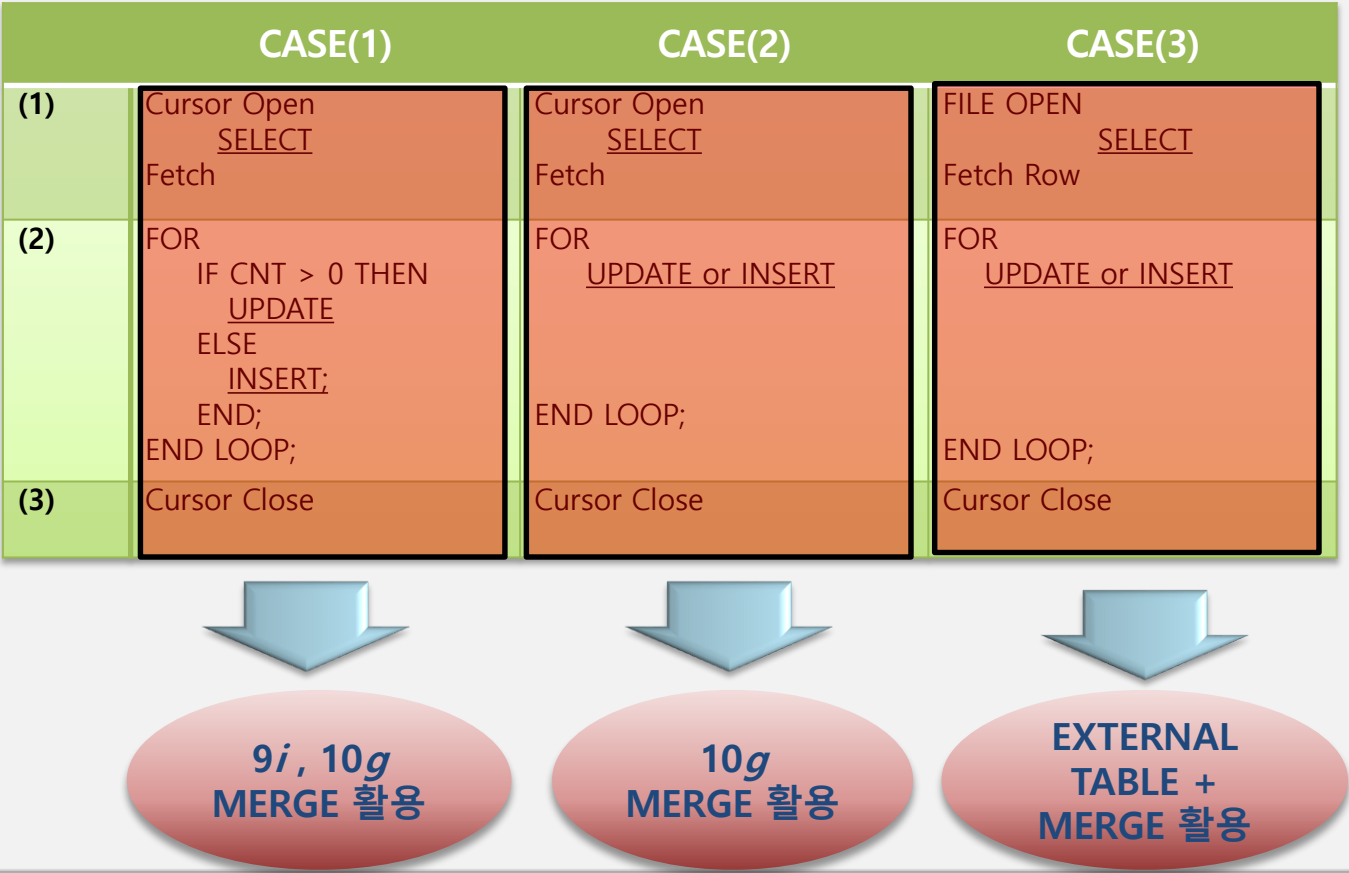
MERGE /*+ PARALLEL(A 2) USE_HASH(B) */ INTO emp a
USING (
    SELECT /*+ FULL(C) PARALLEL(C 2) */
           c.dname, c.deptno
    FROM dept c
) b
ON (
    a.deptno = b.deptno(+) and a.empno > 0
)
WHEN MATCHED THEN
    UPDATE SET a.ename = b.dname ;
```

MERGE 구문으로  
변경 후  
PARALLEL DML  
처리 가능

# 4. MERGE 구문을 성능 문제에 활용하자

## 애플리케이션 로직에 따른 MERGE 구문 활용

아래와 같은 어플리케이션 로직을 MERGE 구문으로 변경하여 보자.



## 5. Summary

### Summary

MERGE 구문 구성요소

MERGE 구문의 오류와 해결책

MEGER 구문이 지원하는 DML 유형

MEGER 구문을 SQL 성능개선에 활용

## FUNCTION 수행과 SQL 성능문제 이해하기



### - FUNCTION 기본 내용들 이해하기

- \* WAS 서버와 네트워크 부하를 감소 시킬 수 있다
- \* 유지보수 측면에서 매우 효율적이다.

### - USER DEFINED FUNCTION의 종류

### - FUNCTION의 동작 방식 이해하기

- \* FUNCTION의 동작방식

### - FUNCTION 수행과 SQL 성능 문제

- \* FUNCTION은 최종 추출 결과만큼만 수행하자
- \* FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자
- \* SELECT절에 사용된 FUNCTION을 조인으로 변경하자
- \* WHERE절의 FUNCTION을 SELECT절로 옮기자

# 1. FUNCTION 기본 내용들 이해하기

## USER DEFINED FUNCTION

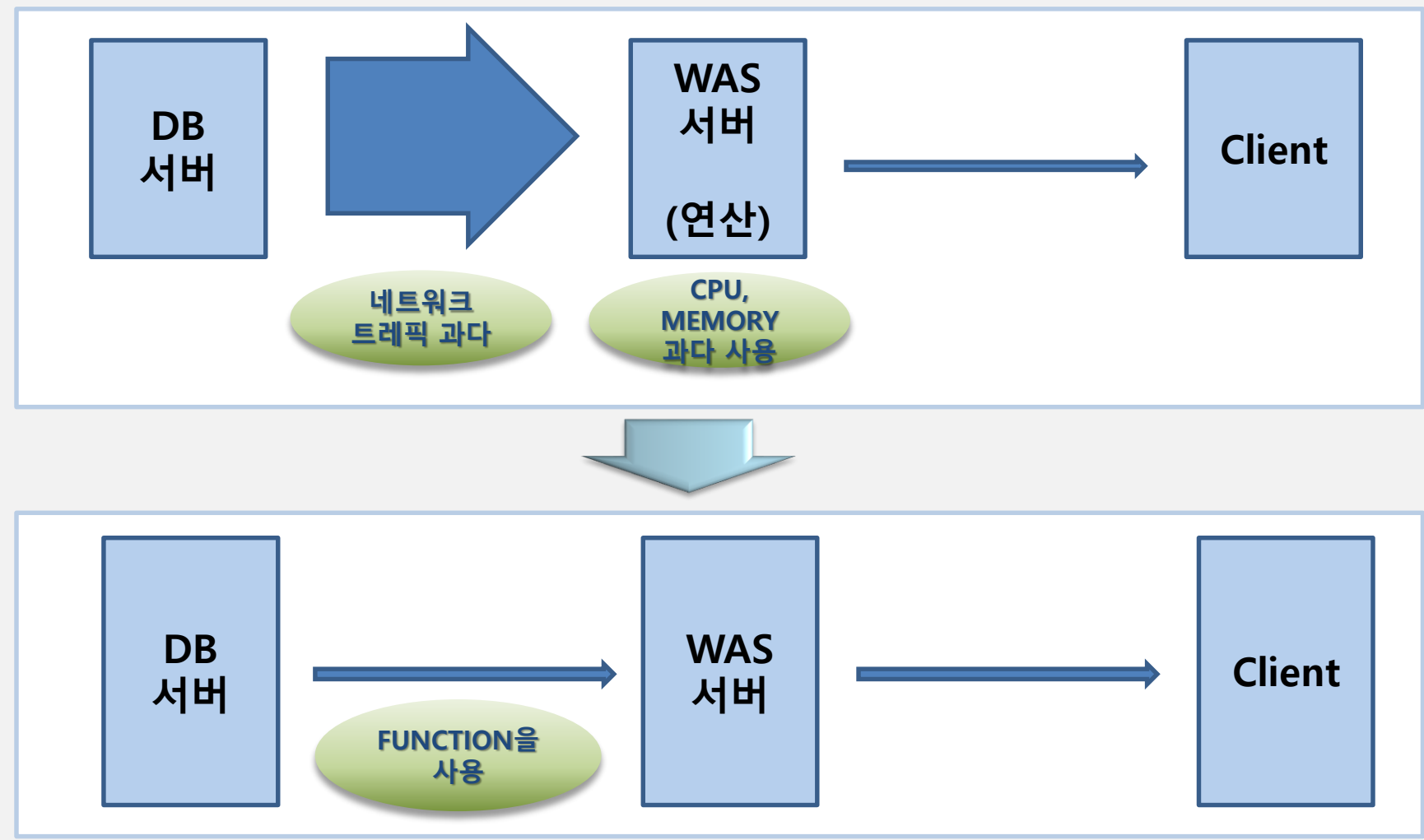
- ◆ 사용자의 필요에 의해 만들어진 FUNCTION으로 DB 객체로 저장되며, 컴파일 된 상태에서 수행된다.
- USER DEFINED FUNCTION의 특징
  - ① 리턴 값이 있다.
  - ② 데이터베이스 객체로 저장되어, 컴파일 된 상태에서 수행된다.
  - ③ 단독적인 사용보다, SQL내에서 많이 사용된다.
  - ④ 예외 처리가 가능하다.

## USER DEFINED FUNCTION 장 점

- ◆ 모듈화된 프로그래밍 가능하다.
- ◆ 변수 및 다양한 제어 문 사용이 가능하여 복잡한 비즈니스 로직 구현이 쉬워진다.
- ◆ WAS 서버와 네트워크 부하를 줄일 수 있다.
- ◆ 유지보수 측면에서 매우 효율적이다.

## 2. USER DEFINED FUNCTION의 종류

WAS 서버와 네트워크 부하를 감소 시킬 수 있다.





## 2. USER DEFINED FUNCTION의 종류

유지 보수 측면에서 매우 효율적이다.



금액 + 부가세  
= 판매가격



직접 SQL에  
기술한 경우



금액 + 부가세  
= 판매가격



FUNCTION  
사용한 경우



## 2. USER DEFINED FUNCTION의 종류

### FUNCTION의 종류

**NOT  
DETERMINISTIC  
FUNCTION**

**DETERMINISTIC  
FUNCTION**

**PIPELINE TABLE  
FUNCTION**

### 3. FUNCTION의 동작 방식 이해하기

#### FUNCTION의 동작방식

◆ FUNCTION이 사용된 위치에 따라 동작 방식을 크게 두가지로 나뉜다면?

**SELECT절에  
사용된  
FUNCTION**

**WHERE 절에  
사용된  
FUNCTION**

### 3. FUNCTION의 동작 방식 이해하기

#### SELECT절에 사용하는 FUNCTION의 동작방식

◆ 데이터가 10만 건인 FUNCTION\_TABLE 아무 조건 없이 조회 한 경우

```
SELECT c1,
       FN_C1_CODENM(c1) c2,
       c3
FROM FUNCTION_TABLE
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.05	0	1	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	6668	4.85	4.60	0	6858	0	100000
total	6670	4.85	4.65	0	6859	0	100000

Rows Row Source Operation

100000	TABLE ACCESS FULL FUNCTION_TABLE (cr=6858 pr=0 pw=0 time=2838 us)
--------	-------------------------------------------------------------------

Main SQL의  
추출 데이터  
10만 건

I/O 6,859 Block  
효율적일까요?

[FUNCTION 수행 내역]

**FUNCTION**  
수행내역을 확인해야 함

```
SELECT c2
FROM C1_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	100000	5.77	5.84	0	0	0	0
Fetch	100000	0.65	3.47	211	300000	0	100000
total	200001	6.42	9.31	211	300000	0	100000

추출 건수와  
동일하게  
10만 번 수행됨.

### 3. FUNCTION의 동작 방식 이해하기

#### SELECT절에 사용하는 FUNCTION의 동작방식

◆ 조건을 추가하여 추출 데이터 건수를 줄여 보자.

```
SELECT c1,
       FN_C1_CODENM(c1) c2,
       c3
FROM FUNCTION_TABLE
WHERE c2 = 0
      AND c3 = 'A'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.03	0	2	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	258	0.23	0.18	0	461	0	3846
total	260	0.23	0.21	0	463	0	3846

Main SQL의  
추출 데이터  
3,846 건

[FUNCTION 수행 내역]

```
SELECT c2
FROM C1_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	3846	0.21	0.21	0	0	0	0
Fetch	3846	0.03	0.03	0	11538	0	3846
total	7693	0.24	0.25	0	11538	0	3846

FUNCTION은 몇 번  
수행 될까?

추출 건수와  
동일하게  
3,846번 수행됨

### 3. FUNCTION의 동작 방식 이해하기

#### SELECT절에 사용하는 FUNCTION의 동작방식

◆ ROWNUM <= 1 조건을 추가하여 1건이 추출되도록 한다면?

```
SELECT c1,
       FN_C1_CODENM(c1) c2,  c3
FROM FUNCTION_TABLE
WHERE c2 = 0
      AND c3 = 'A' AND ROWNUM = 1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.02	0	4	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	3	0	1
total	4	0.01	0.02	0	7	0	1

Rows

Row Source Operation

-----

1 COUNT STOPKEY (cr=3 pr=0 pw=0 time=45 us)

1 TABLE ACCESS BY INDEX ROWID FUNCTION\_TABLE (cr=3 pr=0 pw=0 time=41 us)

1 INDEX RANGE SCAN IDX\_FUNCTION\_TABLE (cr=2 pr=0 pw=0 time=24 us)

Main SQL의  
추출 데이터  
1 건

[FUNCTION 수행내역]

FUNCTION은 몇 번  
수행 될까?

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	0.00	0.00	0	3	0	1
total	3	0.00	0.00	0	3	0	1

추출 건수와  
동일하게  
수행됨.

### 3. FUNCTION의 동작 방식 이해하기

| SELECT절에 사용하는 FUNCTION의 동작방식

SELECT절의  
사용되는  
FUNCTION



데이터 Fetch시에  
수행된다.

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 1]

◆ T1을 먼저 읽고 HASH JOIN으로 수행할 경우

```
SELECT /*+ LEADING(T1) USE_HASH(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	5	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	53	4.11	4.36	0	725	0	768
total	55	4.11	4.37	0	730	0	768

Rows

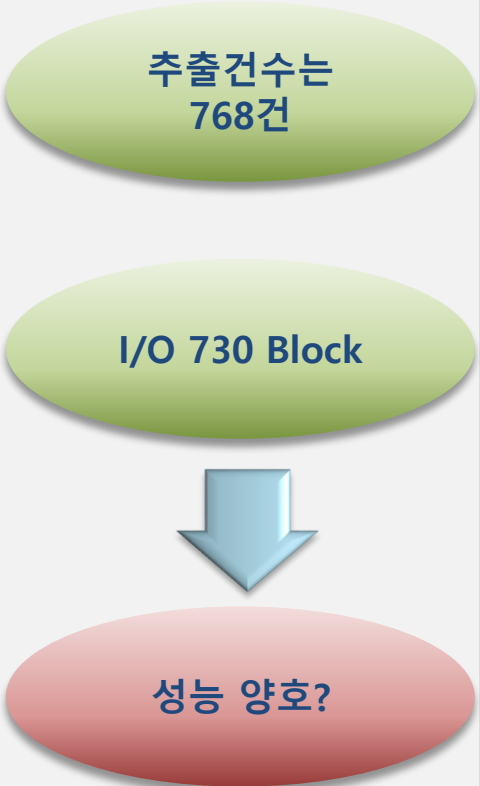
Row Source Operation

-----

768 HASH JOIN (cr=160725 pr=0 pw=0 time=10384443 us)

3846 TABLE ACCESS FULL FUNCTION\_TABLE (cr=204 pr=0 pw=0 time=3894 us)

15000 TABLE ACCESS FULL C1\_CODE\_NM (cr=160521 pr=0 pw=0 time=10352492 us)





### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 1]

◆ T1을 먼저 읽고 HASH JOIN으로 수행할 경우

```
SELECT /*+ LEADING(T1) USE_HASH(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

테이블 T1 조건  
3846 건 추출

FUNCTION 수행내역을  
살펴보자!

T2 테이블에 대한  
상수조건이 없다.

[FUNCTION의 수행내역]

```
SELECT c2
FROM C2_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	100000	5.81	5.57	0	0	0	0
Fetch	100000	0.39	0.43	0	160000	0	60000
total	200001	6.20	6.02	0	160000	0	60000

T2 테이블  
전체 건수만큼  
FUNCTION  
수행

### 3. FUNCTION의 동작 방식 이해하기

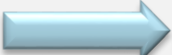
#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 2]

◆ [CASE 1]에서 T2 테이블에 대한 조건을 추가

```
SELECT /*+ LEADING(T1) USE_HASH(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c4 in (2, 4)
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```



CASE1에 대해  
조건을 추가함



동일하게  
10만 번 수행될까?

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.02	0	5	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	27	1.88	1.85	0	699	0	384
total	29	1.88	1.88	0	704	0	384

Rows	Row	Source	Operation
384		HASH JOIN	(cr=60699 pr=0 pw=0 time=4278423 us)
3846		TABLE ACCESS FULL	FUNCTION_TABLE (cr=204 pr=0 pw=0 time=82 us)
5000		TABLE ACCESS FULL	C1_CODE_NM (cr=60495 pr=0 pw=0 time=4265307 us)

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 2]

◆ [CASE 1]에서 T2 테이블에 대한 조건을 추가

```
SELECT /*+ LEADING(T1) USE_HASH(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c4 in (2, 4)
      AND t2.c3 = FN_C2_CODENM(t1.c1);
```

[FUNCTION의 수행내역]

```
SELECT c2
FROM C2_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	40000	2.18	2.25	0	0	0	0
Fetch	40000	0.18	0.16	0	60000	0	20000
total	80001	2.37	2.42	0	60000	0	20000

FUNCTION은 몇 번  
수행 될까?

SELECT COUNT(\*) CNT  
FROM C1\_CODE\_NM T2  
WHERE t1.c4 in (2, 4)

CNT  
-----  
40,000

FUNCTION  
수행 전  
t2.c4 in (2, 4)  
조건을 먼저  
수행

40,000번  
수행

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 3]

◆ [CASE 1]에서 조인 방법이 NESTED LOOPS JOIN로 바뀐 경우

```
SELECT /*+ LEADING(T1) USE_NL(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c3 = FN_C2_CODENM(t2.C4)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.07	0	5	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	53	0.21	1.83	0	8002	0	768
total	55	0.23	1.90	0	8007	0	768

Rows      Row Source Operation

768	NESTED LOOPS (cr=14156 pr=0 pw=0 time=2151094 us)
3846	TABLE ACCESS FULL FUNCTION_TABLE (cr=256 pr=0 pw=0 time=249993 us)
768	TABLE ACCESS BY INDEX ROWID C1_CODE_NM (cr=13900 pr=0 pw=0 time=1895996 us)
3846	INDEX UNIQUE SCAN IDX_C1_CODE_NM (cr=3900 pr=0 pw=0 time=542664 us)

HASH JOIN시  
730 Block

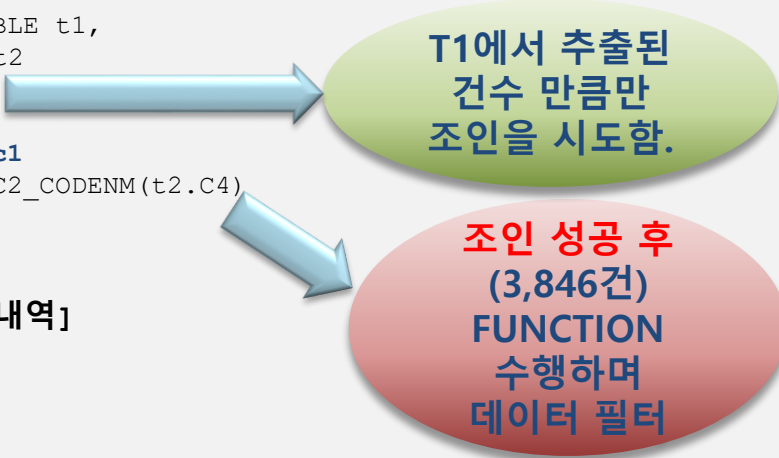
더  
비효율적일까?

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 3]

◆ [CASE 1]에서 조인 방법이 NESTED LOOPS JOIN로 바뀐 경우

```
SELECT /*+ LEADING(T1) USE_NL(T1 T2) */
      t1.*,
      t2.*
FROM  FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c3 = FN_C2_CODENM(t2.C4)
```



**FUNCTION은 몇 번  
수행 될까?**

**[FUNCTION의 수행내역]**

```
SELECT c2
FROM  C2_CODE_NM
WHERE c1 = :B1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	3846	0.26	0.28	0	0	0	0
Fetch	3846	0.00	0.02	0	6154	0	2308
total	7693	0.26	0.32	0	6154	0	2308

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 4]

◆ CASE[3]에 T2테이블에 대해 T2.C4 IN (2,4) 상수 조건을 부여한 경우

```
SELECT /*+ LEADING(T1)  USE_NL(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE T1,
      C1_CODE_NM T2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.C1
      AND t2.c4 IN (2, 4)
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	5	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	27	0.14	0.09	0	7950	0	384
total	29	0.14	0.11	0	7955	0	384

Rows	Row	Source Operation
384		NESTED LOOPS (cr=10257 pr=0 pw=0 time=189078 us)
3846		TABLE ACCESS FULL FUNCTION_TABLE (cr=230 pr=0 pw=0 time=7748 us)
384		TABLE ACCESS BY INDEX ROWID C1_CODE_NM (cr=10027 pr=0 pw=0 time=180366 us)
3846		INDEX UNIQUE SCAN IDX_C1_CODE_NM (cr=3874 pr=0 pw=0 time=9164 us)



### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 4]

◆ CASE[3]에 T2테이블에 대해 T2.C4 IN (2,4) 상수 조건을 부여한 경우

```
SELECT /*+ LEADING(T1) USE_NL(T1 T2) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE T1,
      C1_CODE_NM T2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.C1
      AND t2.c4 IN (2, 4)
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

FUNCTION은 몇 번  
수행 될까?

조인 시도  
3,846번



t2.c4가  
2, 4 인지  
체크



Function  
수행



[FUNCTION의 수행내역]

```
SELECT c2
FROM   C2_CODE_NM
WHERE  c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1538	0.04	0.08	0	0	0	0
Fetch	1538	0.00	0.00	0	2307	0	769
total	3077	0.04	0.09	0	2307	0	769

```
SELECT COUNT(*) CNT
FROM FUNCTION_TABLE T1,
      C1_CODE_NM T2
WHERE T1.C2 = 0
      AND T1.C3 = 'A'
      AND T1.C1 = T2.C1
      AND T2.C4 IN (2, 4) ;

CNT
-----
1538
```

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 5]

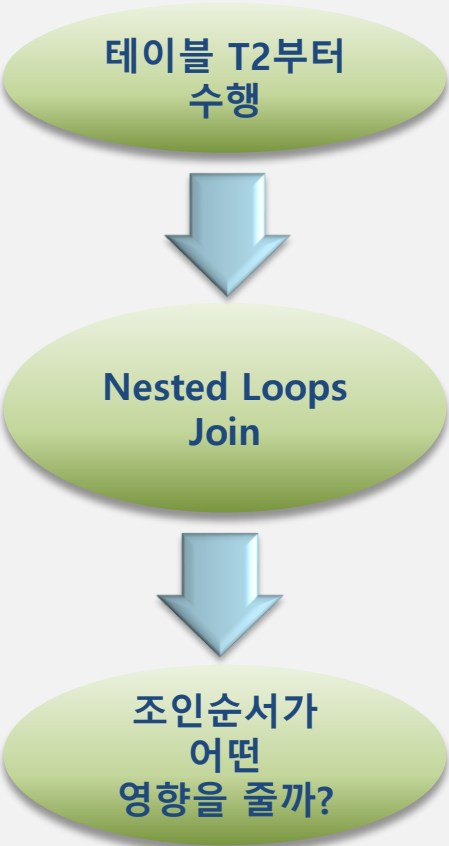
◆ [CASE 3]에서 T2부터 수행하고 NESTED LOOPS JOIN을 선택할 경우

```
SELECT /*+ LEADING (T2) USE_NL(T2 T1) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE T1,
      C1_CODE_NM T2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.05	0	5	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	53	4.33	5.86	211	30575	0	768
total	55	4.33	5.91	211	30580	0	768

Rows	Row Source Operation
768	NESTED LOOPS (cr=190575 pr=211 pw=0 time=12052722 us)
15000	TABLE ACCESS FULL C1_CODE_NM (cr=160521 pr=0 pw=0 time=10710013 us)
768	TABLE ACCESS BY INDEX ROWID FUNCTION_TABLE (cr=30054 pr=211 time=1332356 )
15000	INDEX UNIQUE SCAN IDX_FUNCTION_TABLE_C1 (cr=15054 pr=211 time=1291172 us)





### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 5]

◆ [CASE 3]에서 T2부터 수행하고 NESTED LOOPS JOIN을 선택할 경우

```
SELECT /*+ LEADING(T2) USE_NL(T2 T1) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE T1,
      C1_CODE_NM T2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.c1
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

**FUNCTION은 몇 번  
수행 될까?**

테이블 T2를 먼저  
읽을 경우 범위를  
줄일 상수  
조건이 없다.

T2 테이블  
전체 건수만큼  
FUNCTION  
수행

[FUNCTION의 수행내역]

```
SELECT c2
FROM C2_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	100000	6.05	5.74	0	0	0	0
Fetch	100000	0.42	0.45	0	160000	0	60000
total	200001	6.47	6.19	0	160000	0	60000

### 3. FUNCTION의 동작 방식 이해하기

#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 6]

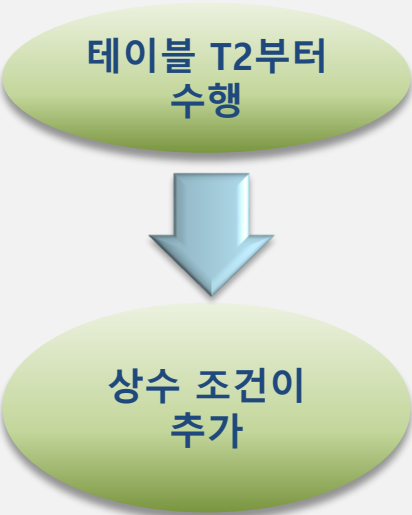
◆ [CASE 5]에 대해 상수 조건이 추가 될 경우

```
SELECT /*+ LEADING(T2) USE_NL(T2 T1) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.C1
      AND t2.c4 IN (2, 4)
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.00	0	5	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	27	1.80	1.87	0	10523	0	384
total	29	1.82	1.88	0	10528	0	384

Rows      Row Source Operation

384	NESTED LOOPS (cr=70523 pr=0 pw=0 time=4301208 us)
5000	TABLE ACCESS FULL C1_CODE_NM (cr=60495 pr=0 pw=0 time=4265206 us)
384	TABLE ACCESS BY INDEX ROWID FUNCTION_TABLE (cr=10028 pr=0 ...)
5000	INDEX UNIQUE SCAN IDX_FUNCTION_TABLE_C1 (cr=5028 pr=0 pw=0 time=20161 us)



### 3. FUNCTION의 동작 방식 이해하기

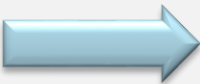
#### WHERE절에 사용하는 FUNCTION의 동작방식 [CASE 6]

◆ [CASE 5]에 대해 상수 조건이 추가 될 경우

```
SELECT /*+ LEADING(T2) USE_NL(T2 T1) */
      t1.*,
      t2.*
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c3 = 'A'
      AND t1.c1 = t2.C1
      AND t2.c4 IN (2, 4)
      AND t2.c3 = FN_C2_CODENM(t2.c4)
```



테이블 T2에  
FUNCTION을  
수행하기 전  
C4가 2, 4인지  
확인



```
SELECT COUNT(*) CNT
FROM C1_CODE_NM T2
WHERE t1.c4 in (2, 4)

CNT
-----
40,000
```



FUNCTION  
40,000 번  
수행

[FUNCTION의 수행내역]							
<pre>SELECT c2 FROM C2_CODE_NM WHERE c1 = :b1</pre>							
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	40000	2.27	2.25	0	0	0	0
Fetch	40000	0.17	0.16	0	60000	0	20000
<hr/>							
total	80001	2.44	2.42	0	60000	0	20000

### 3. FUNCTION의 동작 방식 이해하기

WHERE절에 사용하는 FUNCTION의 동작방식

WHERE절의  
사용되는  
FUNCTION



조인방법  
조인순서  
조회조건

# 4. FUNCTION 수행과 SQL 성능 문제

## FUNCTION은 최종 추출 결과만큼만 수행하자.

### ◆ FUNCTION의 수행위치와 SQL의 성능

```
SELECT Z.*
FROM (
  SELECT a.apply_code ,
         b.branch_code ,
         get_com_branch_name( b.branch_code ) AS branch_name,
         get_saf_lecture_k2_name( d.lecture_kind2 ) AS lecture_kind_name,
         get_commem_name( a.member_code ) AS member_name
  FROM saf_test_apply a,
       saf_brn_test b,
       com_member c,
       saf_test d
  WHERE a.test_code = b.test_code
        AND a.branch_code = b.branch_code
        AND a.member_code = c.member_code
        AND a.test_code = d.test_code
        AND b.state <> 'C'
        AND a.state = 'A'
  ORDER BY a.insert_date DESC
) Z
WHERE ROWNUM <= 3
```

옆에 SQL문은 insert\_date 컬럼을 기준으로 최근 3건만 추출하는 SQL이다.

1,901 Block  
성능 양호?

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.02	0.03	0	2	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	28.74	30.63	0	1899	0	3
total	4	28.76	30.67	0	1901	0	3

Rows	Row Source Operation
3	COUNT STOPKEY (cr=605408 pr=0 pw=0 time=0 us)
3	VIEW (cr=605408 pr=0 pw=0 time=0 us cost=528 size=38511004 card=6356)
3	SORT ORDER BY STOPKEY (cr=605408 pr=0 pw=0 time=0 us ...)
60344	HASH JOIN (cr=1901 pr=0 pw=0 time=311686 us ...)
60345	HASH JOIN (cr=1081 pr=0 pw=0 time=167352 us ...)
2000	TABLE ACCESS FULL SAF_TEST (cr=38 pr=0 pw=0 time=874 us ...)
60345	HASH JOIN (cr=1043 pr=0 pw=0 time=89621 us ...)
2343	TABLE ACCESS FULL SAF_BRN_TEST (cr=38 pr=0 pw=0 time=986 us ...)
60716	TABLE ACCESS FULL SAF_TEST_APPLY (cr=1005 pr=0 pw=0 time=30293 us ...)
389419	INDEX FAST FULL SCAN SYS_C0011071 (cr=820 pr=0 pw=0 time=169690 us ...)

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION은 최종 추출 결과만큼만 수행하자.

◆ FUNCTION의 수행위치와 SQL의 성능

```
SELECT Z.*
FROM (
  SELECT a.apply_code ,
         b.branch_code ,
         get_com_branch_name( b.branch_code ) AS branch_name,
         get_saf_lecture_k2_name( d.lecture_kind2 ) AS lecture_kind_name,
         get_commem_name( a.member_code ) AS member_name
  FROM saf_test_apply a,
       saf_brn_test b,
       com_member c,
       saf_test d
  WHERE a.test_code = b.test_code
        AND a.branch_code = b.branch_code
        AND a.member_code = c.member_code
        AND a.test_code = d.test_code
        AND b.state <> 'C'
        AND a.state = 'A'
  ORDER BY a.insert_date DESC
) Z
WHERE ROWNUM <= 3
```

FUNCTION의  
과도한 수행으로  
비효율 발생

[GET_COM_BRANCH_NAME() 수행부분]							
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	60344	0.81	1.06	0	0	0	0
Fetch	60344	0.55	0.61	0	120688	0	60344
total	120689	1.36	1.67	0	120688	0	60344
[GET_SAF_Lecture_K2_NAME() 수행부분]							
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	60344	0.80	0.94	0	0	0	0
Fetch	60344	1.62	1.75	0	241372	0	60344
total	120689	2.42	2.69	0	241372	0	60344
[GET_COMMEM_NAME() 수행부분]							
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	60344	1.04	1.14	0	0	0	0
Fetch	60344	0.73	0.77	0	241387	0	60344
total	120689	1.77	1.92	0	241387	0	60344

## 4. FUNCTION 수행과 SQL 성능 문제

**FUNCTION은 최종 추출 결과만큼만 수행하자.**

◆ FUNCTION의 수행위치와 SQL의 성능



**왜 이런 일이  
발생했으며  
해결 할 방법은  
없을까?**

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION은 최종 추출 결과만큼만 수행하자.

◆ FUNCTION의 수행위치를 변경하여 성능을 개선하자.

FUNCTION의  
수행 위치를  
변경하자!

```
SELECT x.apply_code,
       x.branch_code,
       ---> FUNCTION 수행 위치를 SQL의 가장 밖으로 옮김
       get_com_branch_name( x.branch_code ) AS branch_name ,
       get_saf_lecture_k2_name( x.lecture_kind2 ) AS lecture_kind_name ,
       get_commem_name( x.member_code ) AS member_name
FROM (
  SELECT Z.*
  FROM (
    SELECT /*+ LEADING(A) USE_NL(A B C D) index(A INX_SAF_TEST_APPLY_05) */
           a.apply_code ,
           b.branch_code,
           ---> 주식 시작 (FUNCTION 수행 위치를 변경하기 위해 주식 처리함)
           --get_com_branch_name( b.branch_code ) AS branch_name ,
           --get_saf_lecture_k2_name( d.lecture_kind2 ) AS lecture_kind_name ,
           --get_commem_name( a.member_code ) AS member_name
           ---> 주식 끝
           d.lecture_kind2, -- 수행 위치를 옮긴 후
                           -- function수행에 필요한 INPUT 값을 추출
           a.member_code   -- 수행 위치를 옮긴 후
                           -- function수행에 필요한 INPUT 값을 추출
    FROM saf_test_apply a ,
         saf_brn_test b ,
         com_member c ,
         saf_test d
    WHERE a.test_code = b.test_code
    AND   a.branch_code = b.branch_code
    AND   a.member_code = c.member_code
    AND   a.test_code = d.test_code
    AND   b.state <> 'C'
    AND   a.state = 'A'
    ORDER BY a.state, a.insert_date DESC
  ) Z
  WHERE ROWNUM <= 3
) X
```



# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION은 최종 추출 결과만큼만 수행하자.

◆ FUNCTION의 수행위치를 변경하여 성능을 개선하자.

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	30	0	3
total	4	0.00	0.00	0	30	0	3

Rows	Row	Source	Operation
3	VIEW	(cr=30 pr=0 pw=0 time=0 us cost=49 size=180 card=3)	
3	COUNT STOPKEY	(cr=30 pr=0 pw=0 time=0 us)	
3	VIEW	(cr=30 pr=0 pw=0 time=0 us cost=49 size=180 card=3)	
3	NESTED LOOPS	(cr=30 pr=0 pw=0 time=0 us cost=49 size=204 card=3)	
3	NESTED LOOPS	(cr=22 pr=0 pw=0 time=112 us cost=46 size=189 card=3)	
3	NESTED LOOPS	(cr=14 pr=0 pw=0 time=84 us cost=43 size=208 card=4)	
3	TABLE ACCESS BY INDEX ROWID	SAF_TEST_APPLY (cr=6 pr=0 pw=0 time=26 us)	
3	INDEX RANGE SCAN	INX_SAF_TEST_APPLY_05 (cr=3 pr=0 pw=0 time=18 us)	
3	TABLE ACCESS BY INDEX ROWID	SAF_BRN_TEST (cr=8 pr=0 pw=0 time=0 us)	
3	INDEX RANGE SCAN	SAF_BRN_TEST_IDX_BT (cr=6 pr=0 pw=0 time=0 us ...)	
3	TABLE ACCESS BY INDEX ROWID	SAF_TEST (cr=8 pr=0 pw=0 time=0 us cost=1)	
3	INDEX UNIQUE SCAN	SYS_C0011754 (cr=5 pr=0 pw=0 time=0 us cost=0 ...)	
3	INDEX UNIQUE SCAN	SYS_C0011071 (cr=8 pr=0 pw=0 time=0 us cost=1 size=5)	

TRACE 결과는  
부분범위 처리

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION은 최종 추출 결과만큼만 수행하자.

◆ FUNCTION의 수행위치를 변경하여 성능을 개선하자.

GET\_COM\_BRANCH\_NAME() 수행부분

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	3	0.00	0.00	0	0	0	0
Fetch	3	0.00	0.00	0	6	0	3
total	7	0.00	0.00	0	6	0	3

GET\_SAF\_LECTURE\_K2\_NAME() 수행부분

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	3	0.00	0.00	0	0	0	0
Fetch	3	0.00	0.00	0	12	0	3
total	7	0.00	0.00	0	12	0	3

GET\_COMMEM\_NAME() 수행부분

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	3	0.00	0.00	0	0	0	0
Fetch	3	0.00	0.00	0	12	0	3
total	7	0.00	0.00	0	12	0	3

Good!

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자.

◆ INPUT 값이 종류가 몇 건 되지 않는 경우 스칼라 서브쿼리를 이용하자.

```
SELECT ---> Function 수행부분 시작
      jisaname( gr.jisacode1 , '00' , SYSDATE ) jisaname1 ,
      jisaname( gr.jisacode1 , gr.jisacode2 , SYSDATE ) jisaname2 ,
      groupname( gr.parent_groupno ) parent_groupname ,
      groupname( gr.groupno ) groupname ,
      ---> Function 수행부분 끝
      gr.groupno,
      gr.parent_groupno,
      gr.jisacode1,
      gr.jisacode2,
      mem.memberno
FROM   t_member mem inner join t_group gr
      ON gr.groupno = mem.groupno
WHERE  ... 이하 생략 .....
```

5,359 Block  
성능이 양호할까?

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	315	4.99	5.19	0	5353	0	4708
total	317	4.99	5.19	0	5353	0	4708
Rows	Row Source Operation						
4708	SORT ORDER BY (cr=52433 pr=0 pw=0 time=12159 us ...)						
4708	HASH JOIN RIGHT ANTI (cr=5353 pr=0 pw=0 time=97956 us ...)						
.....	이하 생략 .....						

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자.

◆ INPUT 값이 종류가 몇 건 되지 않는 경우 스칼라 서브쿼리를 이용하자.

[JISANAME () 수행 내역]

```
SELECT MAX(jisaname)
FROM T_JISACODE
WHERE jisacode1 = :b3
AND jisacode2 = :b2
AND :b1 BETWEEN startday AND endday
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	9416	0.36	0.34	0	0	0	0
Fetch	9416	0.17	0.22	0	18832	0	9416
total	18833	0.53	0.56	0	18832	0	9416

[GROUPNAME () 수행 내역]

```
SELECT MAX(groupname)
FROM T_GROUP WHERE groupno = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	9416	0.17	0.27	0	0	0	0
Fetch	9416	0.18	0.21	0	28248	0	9416
total	18833	0.35	0.49	0	28248	0	9416

KEY POINT

FUNTION의 INPUT  
값으로 사용되는  
컬럼은 Distinct 값이 낮다.

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자.

◆ INPUT 값이 종류가 몇 건 되지 않는 경우 스칼라 서브쿼리를 이용하자.

```
SELECT
    ---> FUNCTION을 스칼라 서브쿼리로 변경 시작
    (SELECT jisaname( gr.jisacode1 , '00' , SYSDATE ) FROM DUAL) jisaname1,
    (SELECT jisaname( gr.jisacode1 , gr.jisacode2 , SYSDATE ) FROM DUAL) jisaname2,
    (SELECT groupname( gr.parent_groupno ) FROM DUAL) parent_groupname,
    (SELECT groupname( gr.groupno ) FROM DUAL) groupname,
    ---> FUNCTION을 스칼라 서브쿼리로 변경 끝
    gr.groupno,
    gr.parent_groupno,
    gr.jisacode1,
    gr.jisacode2,
    mem.memberno
FROM    t_member mem INNER JOIN t_group gr ON gr.groupno = mem.groupno
WHERE   ... 이하 생략 .....
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.03	0.07	0	0	0	0
Fetch	315	1.44	1.58	0	5353	0	4708
total	317	1.47	1.66	0	5353	0	4708

Rows	Row Source Operation								
1	FAST DUAL	(cr=0 pr=0 pw=0 time=0 us cost=2 size=0 card=1)							
1	FAST DUAL	(cr=0 pr=0 pw=0 time=0 us cost=2 size=0 card=1)							
498	FAST DUAL	(cr=0 pr=0 pw=0 time=0 us cost=2 size=0 card=1)							
1715	FAST DUAL	(cr=0 pr=0 pw=0 time=0 us cost=2 size=0 card=1)							

.....이하 생략 .....

스칼라 서브쿼리가 가지고 있는 MULTI BUFFER를 사용할 수 있다

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자.

◆ INPUT 값이 종류가 몇 건 되지 않는 경우 스칼라 서브쿼리를 이용하자.

[JISANAME () 수행 내역]

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	2	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	4	0	2
total	5	0.00	0.00	0	4	0	2

[GROUPNAME () 수행 내역]

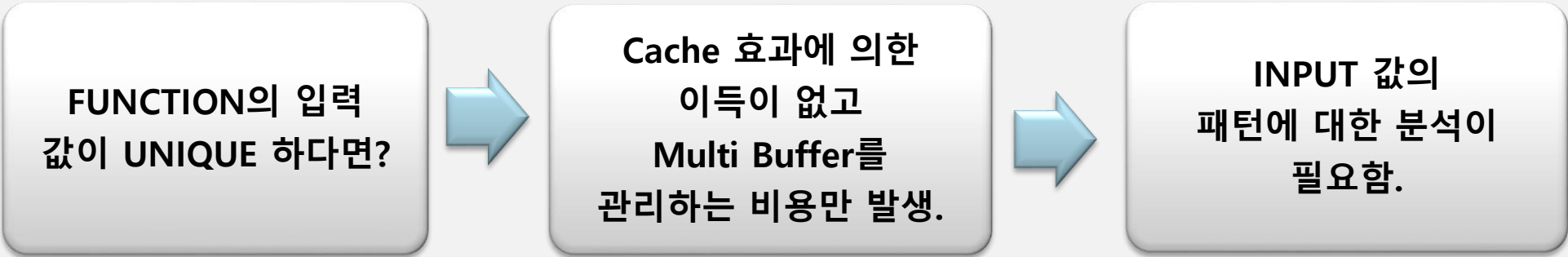
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	2213	0.09	0.08	0	0	0	0
Fetch	2213	0.14	0.06	0	6639	0	2213
total	4427	0.23	0.14	0	6639	0	2213

Good!

## 4. FUNCTION 수행과 SQL 성능 문제

**FUNCTION을 호출하는 값의 패턴을 분석하자.**

◆ FUNCTION을 스칼라 서브쿼리에서 수행하면 항상 성능이 좋아질까?



## 4. FUNCTION 수행과 SQL 성능 문제

**FUNCTION을 호출하는 값의 패턴을 분석하자.**

◆ FUNCTION을 스칼라 서브쿼리에서 수행하면 항상 성능이 좋아질까?

```
CASE1  select fn_c1_codenm(c1), c1 from function_table
```

```
CASE2  select fn_c1_codenm(c4), c4 from function_table
```

**CASE1 VS CASE2**

**INPUT 값의 C1과 C2의  
분포도를 알아야 한다.**



**최신의 통계정보가 존재한다면  
NUM DISTINCT 값을  
이용해 예측 가능.**



# 4. FUNCTION 수행과 SQL 성능 문제

## FUNCTION을 호출하는 값의 패턴을 분석하자.

◆ FUNCTION에 INPUT에 해당하는 데이터 분포도를 분석해 보기

### [테이블 통계정보]

TABLE NAME	NUM_ROWS	AVG ROW LEN	BLOCKS	LAST_ANALYED
TABSPACE NAME			EMP.B	
FUNCTION_TABLE (SCOTT)	100000	12	253	2012-02-17
USERS				

### [컬럼 통계정보]

COLUMN_NAME	DATA_TYPE	DATA_LEN	DENSIT	NULLABLE	NUM_NULLS	NUM_DISTINCT
C1	NUMBER	22	0.000010	Y	0	100000
C2	NUMBER	22	0.500000	Y	0	2
C3	VARCHAR2	4	0.038462	Y	0	26
C4	NUMBER	22	0.333333	Y	0	3



C1은 Unique값, C4는 값의 종류가 3가지

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자.

◆ CASE1 성능

Trace 결과

```
SELECT (SELECT fn_c1_codenm(c1) FROM DUAL), c1
FROM FUNCTION_TABLE;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.03	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	6668	4.52	4.52	0	6898	0	100000
total	6670	4.52	4.55	0	6898	0	100000

Rows	Row Source Operation
100000	FAST DUAL (cr=0 pr=0 pw=0 time=46297 us)
100000	TABLE ACCESS FULL FUNCTION_TABLE (cr=6898 pr=0 pw=0 time=3112 us)

Function 수행 내역

```
SELECT c2
FROM C1_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	100000	5.38	5.68	0	0	0	0
Fetch	100000	0.74	0.72	0	300000	0	100000
total	200001	6.13	6.41	0	300000	0	100000

# 4. FUNCTION 수행과 SQL 성능 문제

FUNCTION이 스칼라 서브쿼리에서 수행하도록 변경하자.

◆ CASE2 성능

```
SELECT (SELECT fn_c1_codenm(c4) FROM DUAL), c4
FROM FUNCTION_TABLE;
```

Trace 결과

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	6668	0.20	0.20	0	6898	0	100000
total	6670	0.20	0.20	0	6898	0	100000
Rows Row Source Operation							
3	FAST DUAL (cr=0 pr=0 pw=0 time=3 us)						
100000	TABLE ACCESS FULL FUNCTION_TABLE (cr=6898 pr=0 pw=0 time=1405 us)						

Function 수행 내역

```
SELECT c2
FROM C1_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	3	0.00	0.00	0	0	0	0
Fetch	3	0.00	0.00	0	9	0	3
total	7	0.00	0.00	0	9	0	3

# 4. FUNCTION 수행과 SQL 성능 문제

SELECT절에 사용된 FUNCTION을 조인으로 변경하자.

◆ NUM DISTINCT가 높고 Function 수행 횟수가 매우 높은 경우



```
SELECT b.c2, a.c1
FROM  function_table a,
      c1_code_nm b
WHERE a.c1 = b.c1(+)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.02	0	0	0	0
Execute	1	0.01	0.00	0	0	0	0
Fetch	6668	0.31	0.28	0	7350	0	100000
total	6670	0.32	0.31	0	7350	0	100000

Rows	Row Source Operation
100000	HASH JOIN OUTER (cr=7350 pr=0 pw=0 time=165798 us)
100000	TABLE ACCESS FULL FUNCTION_TABLE (cr=245 pr=0 pw=0 time=104 us)
100000	TABLE ACCESS FULL C1_CODE_NM (cr=7105 pr=0 pw=0 time=1188 us)

## 4. FUNCTION 수행과 SQL 성능 문제

### WHERE절의 FUNCTION을 SELECT절로 옮기자.

- ◆ NUM DISTINCT가 높고 FUNCTION 수행 횟수가 매우 높은 경우

```
SELECT /*+ first_rows */
      ..... 생략 .....
FROM t_regmember r INNER JOIN t_member m
      ..... 생략 .....
      ON r.memberno = m.memberno
WHERE 1=1
      AND r.joinday BETWEEN '20100219' AND '20100301'
      AND r.regjoincode IN ('01','02','03','04','05','09','10')
      AND rp.sunabclscodel IN ('04','05','06','07','08')
      AND rp.regcyclecode IN ('01','02')
      AND get_birthday_by_juminno(m.jumin1,jumin2_decrypt(m.jumin2enc)) BETWEEN
      to_date('19000101','yyyymmdd') AND to_date('19911231','yyyymmdd')
```

T\_MEMBER 테이블이 WHERE절 조건이 없어  
HASH JOIN으로 수행할 경우 T\_MEMBER 테이블  
전체 건수만큼 FUNCTION 수행됨.

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자.

◆ NUM DISTINCT가 높고 Function 수행 횟수가 매우 높은 경우

```
SELECT *
FROM (
  SELECT /*+ first_rows */
    ..... 생략 .....
    ,get_birthday_by_juminno(m.jumin1,jumin2_decrypt(m.jumin2enc))
    AS function_call --> WHERE절에서 스칼라 서브쿼리로 Function 수행 위치 변경
  FROM t_regmember r INNER JOIN t_member m
    ON r.memberno = m.memberno
    ..... 생략 ..... WHERE 1=1
    AND r.joinday BETWEEN '20100219' AND '20100301'
    AND r.regjoincode IN ('01','02','03','04','05','09','10')
    AND rp.sunabclscodel IN ('04','05','06','07','08')
    AND rp.regcyclecode IN ('01','02')
) A
WHERE A.function_call BETWEEN to_date('19000101', 'yyyymmdd')
AND to_date('19911231', 'yyyymmdd')
```

스칼라 서브쿼리에서 추출한 이후  
FUNCTION 데이터 체크

## 4. FUNCTION 수행과 SQL 성능 문제

WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

◆ 10gR2에서는 오히려 성능이 저하된다.

```
SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
      t1.c3, t2.c4
  FROM FUNCTION_TABLE t1,
       C1_CODE_NM t2
 WHERE t1.c2 = 0
        AND t1.c4 = 2
        AND t1.c1 = t2.c1
        AND t2.c3 IN ( 'A' )
        AND fn_c2_codnm(T2.C4) BETWEEN 'A' AND 'B'
```

Trace 결과는  
어떠할까?

SQL설명 : FUNCTION\_TABLE과 C1\_CODE\_NM 테이블은 조인 후에 데이터가 많이 걸러진다.  
FN\_C2\_CODENM(T2.C4) BETWEEN 'A' AND 'B' 조건은 데이터가 걸러지는 주요 조건은 아니다.

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

SQL의 본문 Trace결과와 Function 수행 내역을 확인.

```
SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
      t1.c3, t2.c4
FROM   FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE  t1.c2 = 0
      AND t1.c4 = 2
      AND t1.c1 = t2.c1
      AND t2.c3 IN ( 'A' )
      AND fn_c2_codenm(T2.C4) BETWEEN 'A' AND 'B'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	113	0.45	0.41	0	838	0	1667
total	115	0.45	0.41	0	838	0	1667

Rows Row Source Operation

1667	HASH JOIN (cr=20826 pr=0 pw=0 time=995069 us)
16667	TABLE ACCESS FULL FUNCTION_TABLE (cr=245 pr=0 pw=0 time=52 us)
5000	TABLE ACCESS FULL C1_CODE_NM (cr=20581 pr=0 pw=0 time=973594 us)

```
SELECT c2
FROM   C2_CODE_NM
WHERE  c1 = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	10000	0.54	0.53	0	0	0	0
Fetch	10000	0.01	0.05	0	20000	0	10000
total	20001	0.56	0.58	0	20000	0	10000

828 Block으로  
성능이 양호해 보인다

FUNCTION은 몇 번  
수행 될까?

5,000번이 아닌  
10,000번 수행?



# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

FUNCTION이 예상보다 많이 수행 된 이유는 무엇일까?

```
SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
      t1.c3, t2.c4
FROM FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE t1.c2 = 0
      AND t1.c4 = 2
      AND t1.c1 = t2.c1
      AND t2.c3 IN ( 'A' )
      AND fn_c2_codenm(T2.C4) BETWEEN 'A' AND 'B'
```

Rows	Row Source Operation
-----	-----
1667	HASH JOIN (cr=20826 pr=0 pw=0 time=995069 us)
16667	TABLE ACCESS FULL FUNCTION_TABLE (cr=245 pr=0 pw=0 time=52 us)
5000	TABLE ACCESS FULL C1_CODE_NM (cr=20581 pr=0 pw=0 time=973594 us)

**SELECT COUNT(\*) FROM C1\_CODE\_NM T2 WHERE T2.C3 IN ('A')**



**BETWEEN이 OPTIMIZER에 의해 변경되었기 때문이다.**

**T2.C3=A AND FN\_C2\_CODENM(T2.C4) => 'A' AND  
FN\_C2\_CODENM(T2.C4) <= 'B'**

## 4. FUNCTION 수행과 SQL 성능 문제

### WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 사용된 FUNCTION에 대한 비효율을 개선하자

#### 현상 분석

1. SQL의 최종 추출 데이터는 1,667건이지만, FUNCTION의 수행횟수는 10,000번이다.
2. WHERE절에 존재하는 FUNCTION을 사용한 조건은 데이터를 거의 걸러내지 못한다.



#### 개선 방향

1. FUNCTION의 수행 결과를 SELECT절에 위치 시킨 후 ALIAS를 지정한다.
2. INLINE VIEW로 만든 후 FUNCTION 수행결과 ALIAS를 이용해 INLINE VIEW 밖에서 조건을 수행하도록 한다.

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자.

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

### 변경 전

```
SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
      t1.c3, t2.c4
FROM   FUNCTION_TABLE t1,
      C1_CODE_NM t2
WHERE  t1.c2 = 0
      AND t1.c4 = 2
      AND t1.c1 = t2.c1
      AND t2.c3 IN ( 'A' )
      AND fn_c2_codenm(T2.C4) BETWEEN 'A' AND 'B'
```



### 변경 후 SQL

```
SELECT /*+ NO_MERGE(A) */ *
FROM   (
        SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
              t1.c3,
              t1.c4,
              fn_c2_codenm(t2.c4) AS ft2c4
        FROM   FUNCTION_TABLE t1,
              C1_CODE_NM t2
        WHERE  t1.c2 = 0
              AND t1.c4 = 2
              AND t1.c1 = t2.c1
              AND t2.c3 IN ( 'A' )
        ) A
WHERE  ft2c4 BETWEEN 'A' AND 'B'
```

- 1. FUNCTION의 수행 결과를 SELECT절에 위치 시킨 후 ALIAS를 지정한다.
- 2. INLINE VIEW로 만든 후 FUNCTION 수행결과 ALIAS를 이용해 INLINE VIEW 밖에서 조건을 수행하도록 한다.

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

### SQL Trace 결과

변경 전  
CPU TIME  
0.45

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	113	0.51	0.59	0	618	0	1667
total	115	0.51	0.59	0	618	0	1667

Rows	Row	Source	Operation
1667	VIEW	(cr=24384 pr=0 pw=0 time=1448413 us)	
1667	HASH JOIN	(cr=20826 pr=0 pw=0 time=1234042 us)	
16667	TABLE ACCESS FULL	FUNCTION_TABLE (cr=245 pr=0 pw=0 time=58 us)	
5000	TABLE ACCESS FULL	C1_CODE_NM (cr=20581 pr=0 pw=0 time=1213581 us)	

### Function 수행내역

OOPS!!  
BAD!

SELECT c2 FROM C2_CODE_NM WHERE c1 = :b1							
call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	11891	0.90	0.79	0	0	0	0
Fetch	11891	0.06	0.07	0	23782	0	11891
total	23783	0.96	0.86	0	23782	0	11891

## 4. FUNCTION 수행과 SQL 성능 문제

**WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]**

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

FUNCTION을 SELECT절에 위치 시키자  
FUNCTION의 수행횟수가 더 늘었다  
어떻게 된 일일까?

11,891 번이  
SELECT에서?  
불가능하다

원인 파악을 위해 INLINE VIEW 외부  
조건을 주석 처리한 후  
SELECT만 수행한 결과를 보도록 하자

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

### VIEW 외부조건 조석처리 후

```
SELECT /*+ NO_MERGE(A) */ *
FROM (
    SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
        t1.c3,
        t1.c4,
        fn_c2_codenm(t2.c4) AS ft2c4
    FROM FUNCTION_TABLE t1,
        C1_CODE_NM t2
    WHERE t1.c2 = 0
        AND t1.c4 = 2
        AND t1.c1 = t2.c1
        AND t2.c3 IN ( 'A' )
    ) A
-- WHERE ft2c4 BETWEEN 'A' AND 'B' 주석처리
```



### FUNCTION 수행 내역

```
SELECT c2
FROM C2_CODE_NM
WHERE c1 = :b1
```

call	count	cpu	elapsed	disk	query	rows
Parse	1	0.00	0.00	0	0	0
Execute	1891	0.14	0.10	0	0	0
Fetch	1891	0.03	0.01	0	3782	1891
total	3783	0.17	0.11	0	3782	1891

뷰 외부 조건을 제거하자 FUNCTION의 수행횟수가  
SELECT절에서 **1,891번** 수행된다.

WHERE절 FUNCTION 수행횟수 + SELECT절 FUNCTION 수행횟수 = 11,891  
( 10,000 ) ( 1,891 )

## 4. FUNCTION 수행과 SQL 성능 문제

**WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]**

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

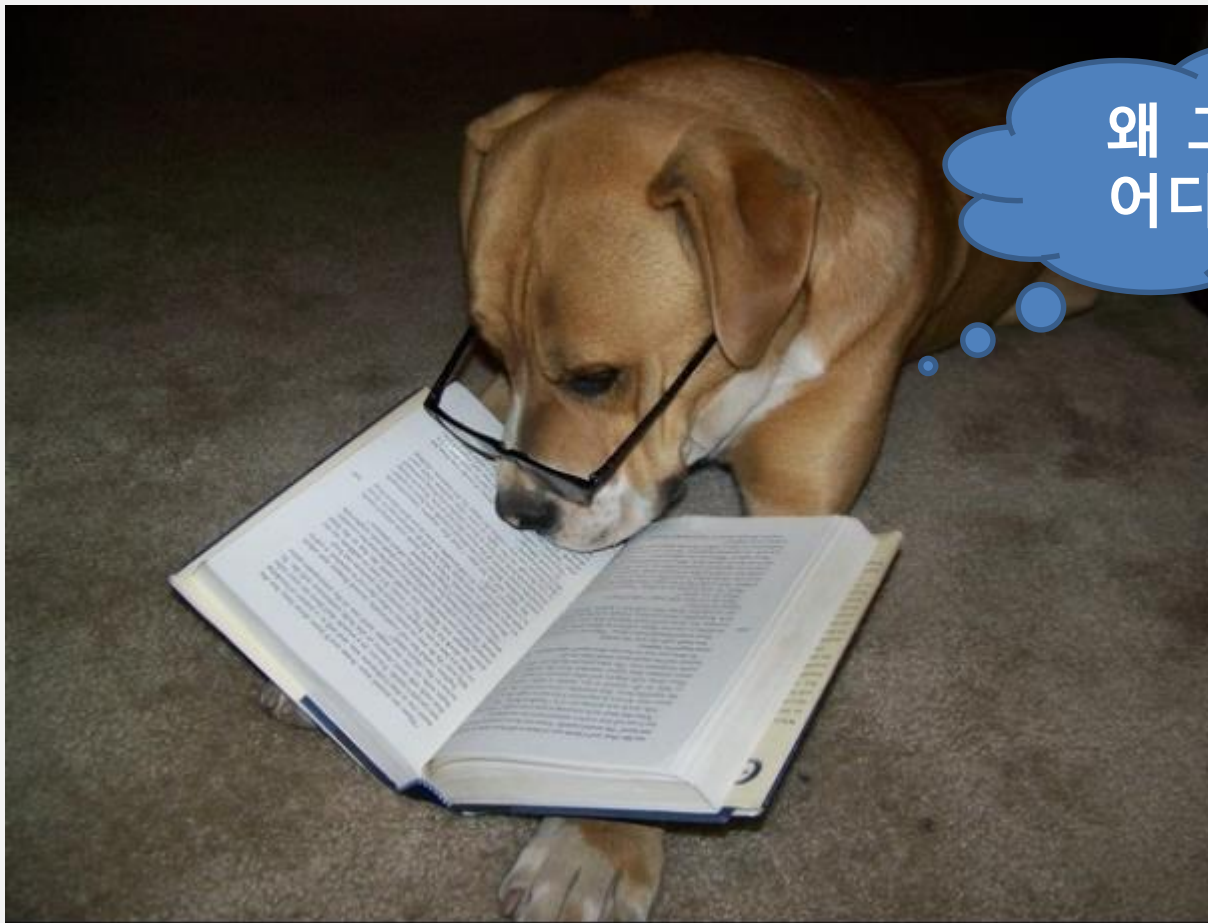


잉? 대체 무슨 짓을  
한 거야?

## 4. FUNCTION 수행과 SQL 성능 문제

**WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]**

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.



왜 그랬을까?  
어디 보자~~



## 4. FUNCTION 수행과 SQL 성능 문제

**WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]**

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.



10053 Trace와  
DBMS\_XPLAN을 보면  
알 수 있어요

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

### 10053 TRACE 결과

```
*****
Predicate Move-Around (PM)
*****
PM: Considering predicate move-around in SEL$1 (#0).
PM:  Checking validity of predicate move-around in SEL$1 (#0).
PM:  Passed validity checks.
FPD: Considering simple filter push in SEL$1 (#0)
FPD:  Current where clause predicates in SEL$1 (#0) :
      "A"."FT2C4">='A' AND "A"."FT2C4"<='B'
kkogcp: try to generate transitive predicate from check constraints for SEL$1 (#0)
predicates with check constraints: "A"."FT2C4">='A' AND "A"."FT2C4"<='B' AND 'B'>='A'
after transitive predicate generation: "A"."FT2C4">='A' AND "A"."FT2C4"<='B' AND 'A'<='B'
finally: "A"."FT2C4">='A' AND "A"."FT2C4"<='B' AND 'A'<='B'
FPD:  Following transitive predicates are generated in SEL$1 (#0) :  'A'<='B'
JPPD:  JPPD bypassed: View not on right-side of outer join
FPD:  Following are pushed to where clause of SEL$2 (#0) : -> simple filter를 view로 침투시켜 본다.
      "SCOTT"."FN_C2_CODENM"("T2"."C4")>='A' AND "SCOTT"."FN_C2_CODENM"("T2"."C4")
      <='B' AND 'A'<='B'
FPD: Considering simple filter push in SEL$2 (#0)
FPD:  Current where clause predicates in SEL$2 (#0) :
      "T1"."C2"=0 AND "T1"."C4"=2 AND "T1"."C1"="T2"."C1" AND "T2"."C3"='A' AND
      "SCOTT"."FN_C2_CODENM"("T2"."C4")>='A' AND "SCOTT"."FN_C2_CODENM"("T2"."C4")
      <='B' AND 'A'<='B'
.....
```



## 4. FUNCTION 수행과 SQL 성능 문제

### WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

#### 현상 분석

1. 10053 TRACE 분석 결과 Optimizer가 FPD(Filter Push Down)에 성공하여 뷰 밖의 조건을 뷰 안으로 침투 시킴
2. QUERY TRANSFORMATION에 FPD는 WHERE절에 FUNCTION을 2회 수행시킬 뿐 아니라 SELECT절에 FUNCTION은 유지함으로써 원 SQL보다 수행횟수가 더 늘어남



수행횟수가 증가한 이유가 FPD와 SELECT 절 사용된 FUNCTION 때문인지  
어떻게 알 수 있을까?

**DBMS\_XPLAN**

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

Predicate Information과 Column Projection Information에 그 답이 있다

### DBMS\_XPLAN 결과

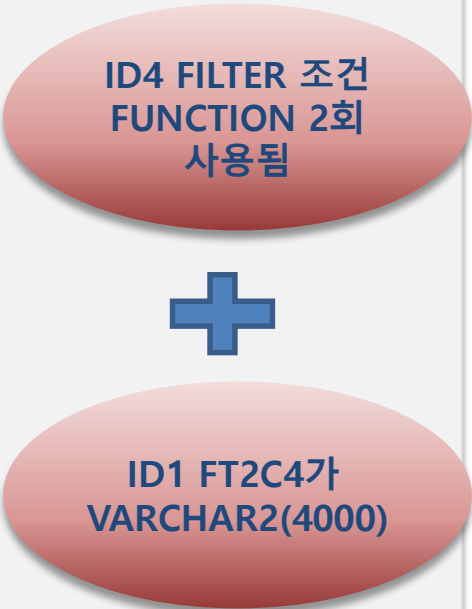
Id	Operation	Name	Rows	Bytes	Cost	Time
0	SELECT STATEMENT				170	
1	VIEW		12	24K	170	00:00:03
2	HASH JOIN		12	264	170	00:00:03
3	TABLE ACCESS FULL	FUNCTION_TABLE	16K	195K	59	00:00:01
4	TABLE ACCESS FULL	C1_CODE_NM	12	120	110	00:00:02

Predicate Information:

2 - access("T1"."C1"="T2"."C1")  
3 - filter(("T1"."C4"=2 AND "T1"."C2"=0))  
4 - filter(("T2"."C3"='A' AND "FN\_C2\_CODENM" ("T2"."C4")>='A' AND "FN\_C2\_CODENM" ("T2"."C4")<='B'))

Column Projection Information (identified by operation id):

1 - "A"."C3"[VARCHAR2,4], "A"."C4"[NUMBER,22], "FT2C4"[VARCHAR2,4000]  
2 - (#keys=1) "T1"."C3"[VARCHAR2,4], "T2"."C4"[NUMBER,22]  
3 - "T1"."C1"[NUMBER,22], "T1"."C3"[VARCHAR2,4]  
4 - "T2"."C1"[NUMBER,22], "T2"."C4"[NUMBER,22]



# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

**OPTIMIZER에 의한 변경된 SQL 모습은 예측해 보면 아래와 같다**

```
SELECT /*+ NO_MERGE(A) */
      *
FROM (
SELECT /*+ LEADING ("T1" "T2") USE_HASH ("T2") USE_HASH ("T1") */
      t1.c3,
      t2.c4,
      FN_2_CODENM(t2.c4) FT2C4 --> 불필요하게 나열된 SELECT절의 FUNCTION
FROM   FUNCTION_TABLE T1 ,
      C1_CODE_NM T2
WHERE  t1.c4 =2
AND    t1.c2 =0
AND    t1.c1 =t2.C1
AND    t2.c3 ='A'
AND    FN_C2_CODENM(t2.c4) >='A' ----> FPD에 의해 조건이 안으로 침투
AND    FN_C2_CODENM(t2.c4) <='B' ----> FPD에 의해 조건이 안으로 침투
) A
```

## 4. FUNCTION 수행과 SQL 성능 문제

**WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]**

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.



해결책은 ?

동작방식을 변경하자!!

## 4. FUNCTION 수행과 SQL 성능 문제

### WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

#### 변경 후

```
SELECT /*+ NO_MERGE(A) */ *
FROM (
    SELECT /*+ LEADING(T1 T2) USE_HASH(T1 T2) */
        t1.c3,
        t1.c4,
        (SELECT fn_c2_codenm(t2.c4) FROM DUAL) AS ft2c4
    FROM FUNCTION_TABLE t1,
        C1_CODE_NM t2
    WHERE t1.c2 = 0
        AND t1.c4 = 2
        AND t1.c1 = t2.c1
        AND t2.c3 IN ( 'A' )
    ) A
WHERE ft2c4 BETWEEN 'A' AND 'B'
```

SELECT절에 존재하는  
FUNCTION의 동작방식을  
변경하자

# 4. FUNCTION 수행과 SQL 성능 문제

## WHERE절의 FUNCTION을 SELECT절로 옮기자. [심화학습]

WHERE절의 FUNCTION을 SELECT절로 옮겨보자.

### DBMS\_XPLAN 결과

-----					
Id	Operation	Name	A-Rows	A-Time	Buffers
-----					
1	FAST DUAL		1	00:00:00.01	0
2	VIEW		1667	00:00:00.03	828
* 3	FILTER		1667	00:00:00.03	828
* 4	HASH JOIN		1667	00:00:00.03	826
* 5	TABLE ACCESS FULL	FUNCTION_TABLE	1667	00:00:00.01	245
* 6	TABLE ACCESS FULL	C1_CODE_NM	5000	00:00:00.01	581
7	FAST DUAL		1	00:00:00.01	0
-----					
Predicate Information (identified by operation id):					
-----					
3 - filter(>='A' AND <='B')					
4 - access("T1"."C1"="T2"."C1")					
5 - filter(("T1"."C4"=2 AND "T1"."C2"=0))					
6 - filter("T2"."C3"='A')					

### Function 수행내역



SELECT C2 FROM C2_CODE_NM WHERE C1 = :B1							
call	count	cpu	elapsed	disk	query	current	rows
-----							
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	0.00	0.00	0	2	0	1
-----							
total	3	0.00	0.00	0	2	0	1



## 5. Summary

### Summary

FUNCTION의 장점

USER DEFINED FUNCTION의 종류

FUNCTION의 동작 방식

FUNCTION 수행과 SQL 성능 문제

## DECODE & CASE WHEN 이해 및 조건 문 처리하기



### - DECODE 기본 내용 이해하기

- \* DECODE
- \* DECODE 중첩

### - DECODE와 성능이슈

### - CASE 기본 내용 이해하기

- \* 구문설명
- \* CASE 활용 방법
- \* 중첩 CASE 실전 예제

# 1. DECODE 기본 내용 이해하기

## DECODE 함수

- ◆ SQL내에서 IF – THEN\_ ELSE IF – END 로직의 처리를 가능하게 해주는 함수이다.
- ◆ TRUE or FALSE 결과에 따라 각각 다른 값을 추출할 때 유용하게 사용된다.
- ◆ SQL내에서 FROM절을 제외한 모든 위치에서 사용 가능하다.

## DECODE 구문

DECODE({column | expression}, search1, result1 [,search2,result2] ... [,default] )

# 1. DECODE

DECODE 구문

```
DECODE({column | expression}, search1, result1 [,search2,result2] ... [,default] )
```




테이블의 컬럼이나 계산식 등을 사용할 수 있음.

# 1. DECODE

## DECODE 구문

DECODE({column | expression}, search1, result1 [,search2,result2] ... [,default] )



**COLUMN | EXPRESSION의 결과와 비교할 값을 지정.**

예) DECODE ( 9+1 (COLUMN|RESSION), 10 (SEARCH1), '정답')

# 1. DECODE

DECODE 구문

```
DECODE({column | expression}, search1, result1 [,search2,result2] ... [,default] )
```



COLUMN | EXPRESSION과 SEARCH1이 값이 같다면 리턴 할 값을 지정.

```
예) DECODE( 9+1 , 10, '정답') ➡ '정답' 추출  
      DECODE( 9+1 , 11, '정답') ➡ NULL 추출
```

# 1. DECODE

DECODE 구문

DECODE({column | expression}, search1, result1 [,search2,result2] ... [,default] )



SEARCH2와 RESULT2는 항상 같이 기술되며, IF절에서 ELSIF와 같은 역할을 함.

예) DECODE (9+1, 9, '정답1', 10, '정답2') ➡ '정답2' 추출  
DECODE (8+1, 9, '정답1', 10, '정답2') ➡ '정답1' 추출

DECODE(9+1(COLUMN|EXPRESSION), 11(SEARCH1), '정답')  
➡ 두 번의 비교가 모두 FALSE이면 NULL 추출

# 1. DECODE

## DECODE 구문

```
DECODE({column | expression}, search1, result1 [,search2,result2] ... [,default] )
```



모든 비교가 FALSE일 경우 NULL값이 아닌 값을 강제로 지정하고자 할 때, DEFAULT를 이용해 값을 지정할 수 있음. IF문의 ELSE와 같은 역할을 함.

```
예) DECODE (9+1, 9, '정답1', 10, '정답2', '정답3')
```

- ➡ 9+1 을 9와 비교 - FALSE이므로 다음 IF로 이동
- 9+1 을 10과 비교 - TRUE이므로 '정답2' 리턴
- 만약 9+2로 비교한다면 - 두 번의 비교 모두 FALSE이므로 DEFAULT로 지정한 '정답3' 리턴



# 1. DECODE

DECODE 구문

DECODE (9+1, 9, '정답1', 10, '정답2', '정답3')



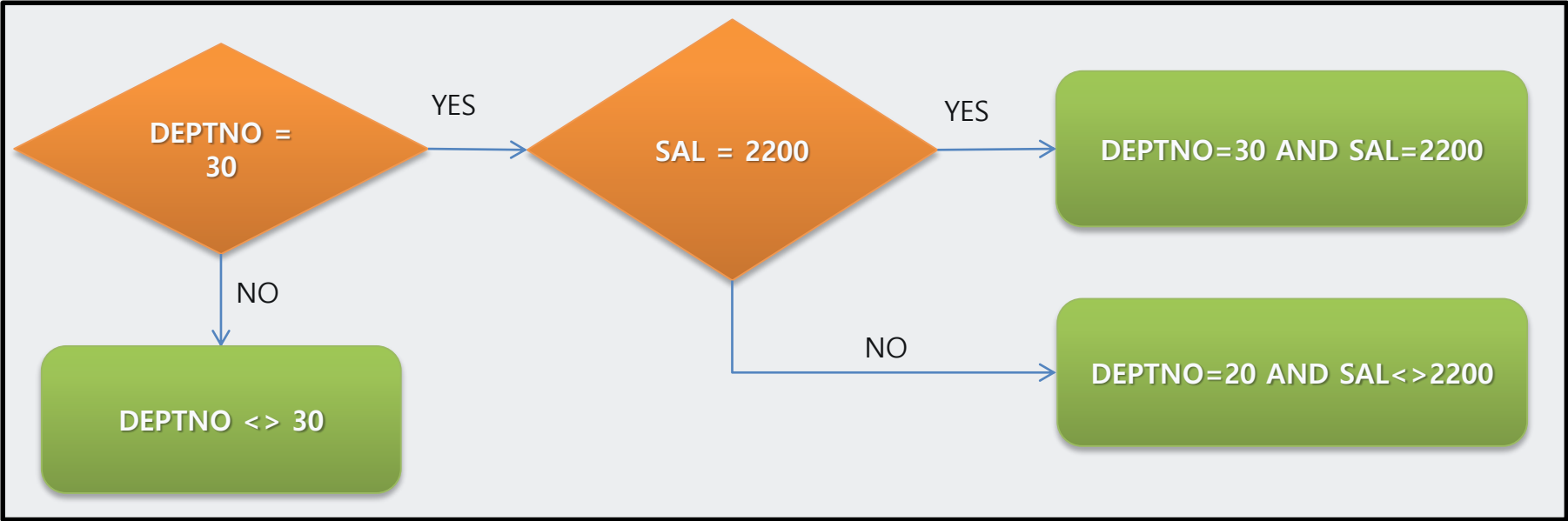
PL/SQL로 표현

DECODE ( <u>9+1</u> , <u>9</u> , <u>'정답1'</u> , <u>10</u> , <u>'정답2'</u> , <u>'정답3'</u> )				
①	②	③	④	⑤
IF 9+1 = 9 THEN			①	
RETURN '정답1'			②	
ELSIF 9+1 = 10 THEN			③	
RETURN '정답2'			④	
ELSE				
RETURN '정답3'			⑤	
END IF;				

# 1. DECODE

## DECODE 중첩

```
SELECT deptno,  
       sal ,  
       DECODE( deptno , 30 , DECODE( sal , 2200 , 'DEPTNO=30 AND SAL=2200' ,  
                                     'DEPTNO=30 AND SAL<>2200' ) , 'DEPTNO <> 30' )  
FROM emp ;
```



# 1. DECODE

## DECODE 중첩

```
SELECT deptno,  
       sal ,  
       DECODE( deptno , 30 , DECODE( sal , 2200 , 'DEPTNO=30 AND SAL=2200' ,  
                                     'DEPTNO=30 AND SAL<>2200' ) , 'DEPTNO <> 30' )  
FROM emp ;
```



PL/SQL로 표현

```
IF DEPTNO = 30 THEN  
  IF SAL = 2200 THEN  
    RETURN 'DEPTNO=30 AND SAL=2200' ;  
  ELSE  
    RETURN 'DEPTNO=30 AND SAL<>2200' ;  
  END IF;  
ELSE  
  RETURN 'DEPTNO <> 30';  
END IF;
```

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar Sub Query를 이용한 작성법

```
SELECT '2011/12/01' saledt_1201,
      (SELECT SUM(target) FROM decode_t1 WHERE sale_dt='20111201') AS target_1201,
      (SELECT SUM(salecnt) FROM decode_t1 WHERE sale_dt ='20111201') AS sale_1201,
      -- 중략
      '2011/12/10' saledt_1210,
      (SELECT SUM(target) FROM decode_t1 WHERE sale_dt='20111210') AS target_1210,
      (SELECT SUM(salecnt) FROM decode_t1 WHERE sale_dt ='20111210') AS sale_1210
FROM DUAL;
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.68	1.69	0	19744	0	1
total	4	1.68	1.69	0	19744	0	1

Rows	Row	Source	Operation
	1		SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85366 us)
100000		TABLE ACCESS BY INDEX ROWID	DECODE_T1 (cr=1040 pr=0 pw=0 time=50 us)
100000		INDEX RANGE SCAN	IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6194 us)
1			SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85824 us)
100000		TABLE ACCESS BY INDEX ROWID	DECODE_T1 (cr=1040 pr=0 pw=0 time=31 us)
100000		INDEX RANGE SCAN	IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6441 us)
		.....	-- 중간 생략. --
1			SORT AGGREGATE (cr=1058 pr=0 pw=0 time=88706 us)
99785		TABLE ACCESS BY INDEX ROWID	DECODE_T1 (cr=1058 pr=0 pw=0 time=30 us)
99785		INDEX RANGE SCAN	IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6535 us)
			FAST DUAL (cr=0 pr=0 pw=0 time=1 us)

DECODE를 사용하여  
작성할 수 있다.

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar Sub Query를 이용한 작성법

```
SELECT '2011/12/01' saledt_1201,  
      (SELECT SUM(target) FROM decode_t1 WHERE sale_dt='20111201') AS target_1201,  
      (SELECT SUM(salecnt) FROM decode_t1 WHERE sale_dt ='20111201') AS sale_1201,  
      -- 중략  
      '2011/12/10' saledt_1210,  
      (SELECT SUM(target) FROM decode_t1 WHERE sale_dt='20111210') AS target_1210,  
      (SELECT SUM(salecnt) FROM decode_t1 WHERE sale_dt ='20111210') AS sale_1210  
FROM DUAL;
```

Scalar Sub Query를 이용하여 2011년 12월 1일 부터 10일까지의 데이터 추출

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.68	1.69	0	19744	0	1
total	4	1.68	1.69	0	19744	0	1

Rows	Row Source Operation
1	SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85366 us)
100000	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1040 pr=0 pw=0 time=50 us)
100000	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6194 us)
1	SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85824 us)
100000	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1040 pr=0 pw=0 time=31 us)
100000	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6441 us)
1	SORT AGGREGATE (cr=1002 pr=0 pw=0 time=101328 us)
	..... -- 중간 생략. --
1	SORT AGGREGATE (cr=1058 pr=0 pw=0 time=88706 us)
99785	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1058 pr=0 pw=0 time=30 us)
99785	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6535 us)
	FAST DUAL (cr=0 pr=0 pw=0 time=1 us)

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar Sub Query를 이용한 작성법

```
SELECT '2011/12/01' saledt_1201,  
      (SELECT SUM(target) FROM decode_t1 WHERE sale_dt='20111201') AS target_1201,  
      (SELECT SUM(salecnt) FROM decode_t1 WHERE sale_dt ='20111201') AS sale_1201,  
      -- 중략  
      '2011/12/10' saledt_1210,  
      (SELECT SUM(target) FROM decode_t1 WHERE sale_dt='20111210') AS target_1210,  
      (SELECT SUM(salecnt) FROM decode_t1 WHERE sale_dt ='20111210') AS sale_1210  
FROM DUAL;
```

Scalar Sub Query를 이용하여 2011년 12월 1일 부터 10일까지의 데이터 추출

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.68	1.69	0	19744	0	1
total	4	1.68	1.69	0	19744	0	1

Rows	Row Source Operation
1	SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85366 us)
100000	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1040 pr=0 pw=0 time=50 us)
100000	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6194 us)
1	SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85824 us)
100000	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1040 pr=0 pw=0 time=31 us)
100000	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6441 us)
1	SORT AGGREGATE (cr=1002 pr=0 pw=0 time=101328 us)
.....	-- 중간 생략. --
1	SORT AGGREGATE (cr=1058 pr=0 pw=0 time=88706 us)
99785	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1058 pr=0 pw=0 time=30 us)
99785	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6535 us)
	FAST DUAL (cr=0 pr=0 pw=0 time=1 us)

동일 테이블을 Scalar Sub Query 수만큼 반복 탐색하는 비효율 발생

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – DECODE를 이용한 작성법

```
SELECT '2011/12/01' saledt_1201,
       SUM(DECODE(sale_dt, '20111201',target, 0)) AS target_1201,
       SUM(DECODE(sale_dt, '20111201',salecnt, 0)) AS sale_1201,
       '2011/12/02' saledt_1202,
       SUM(DECODE(sale_dt, '20111202',target, 0)) AS target_1202,
       SUM(DECODE(sale_dt, '20111202',salecnt, 0)) AS sale_1202,
       '2011/12/03' saledt_1203,
       SUM(DECODE(sale_dt, '20111203',target, 0)) AS target_1203,
       SUM(DECODE(sale_dt, '20111203',salecnt, 0)) AS sale_1203,
       -- 중략
       '2011/12/08' saledt_1208,
       SUM(DECODE(sale_dt, '20111208',target, 0)) AS target_1208,
       SUM(DECODE(sale_dt, '20111208',salecnt, 0)) AS sale_1208,
       '2011/12/09' saledt_1209,
       SUM(DECODE(sale_dt, '20111209',target, 0)) AS target_1209,
       SUM(DECODE(sale_dt, '20111209',salecnt, 0)) AS sale_1209,
       '2011/12/10' saledt_1210,
       SUM(DECODE(sale_dt, '20111210',target, 0)) AS target_1210,
       SUM(DECODE(sale_dt, '20111210',salecnt, 0)) AS sale_1210
FROM DECODE_T1
WHERE SALE_DT BETWEEN '20111201' AND '20111210' ;
```

기존 Scalar Sub Query로  
추출하던 방식을 Decode로  
변경.

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – DECODE를 이용한 작성법

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	1	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.85	1.87	0	7184	0	1
total	4	1.87	1.88	0	7185	0	1
Rows	Row Source Operation						
1	SORT AGGREGATE (cr=7184 pr=0 pw=0 time=1878579 us)						
957364	TABLE ACCESS FULL DECODE_T1 (cr=7184 pr=0 pw=0 time=61 us)						

반복적으로 동일한 테이블을 탐색하던 비효율이 단 한번의 Table Access만으로 완료됨을 알 수 있다.



## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar vs Decode

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.68	1.69	0	19744	0	1
total	4	1.68	1.69	0	19744	0	1

Rows	Row Source Operation
1	SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85366 us)
100000	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1040 pr=0 pw=0 time=50 us)
100000	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6194 us)
1	SORT AGGREGATE (cr=1040 pr=0 pw=0 time=85824 us)
100000	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1040 pr=0 pw=0 time=31 us)
100000	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6441 us)
1	SORT AGGREGATE (cr=1002 pr=0 pw=0 time=101328 us)
	..... -- 중간 생략. --
1	SORT AGGREGATE (cr=1058 pr=0 pw=0 time=88706 us)
99785	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=1058 pr=0 pw=0 time=30 us)
99785	INDEX RANGE SCAN IDX_DECODE_T1 (cr=281 pr=0 pw=0 time=6535 us)
	FAST DUAL (cr=0 pr=0 pw=0 time=1 us)



Scalar Sub Query 방식에서  
의 10046 Trace 결과

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar vs Decode

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	1	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.85	1.87	0	7184	0	1
total	4	1.87	1.88	0	7185	0	1
Rows	Row Source Operation						
1	SORT AGGREGATE (cr=7184 pr=0 pw=0 time=1878579 us)						
957364	TABLE ACCESS FULL DECODE_T1 (cr=7184 pr=0 pw=0 time=61 us)						



DECODE 방식으로 추출한  
SQL의 10046 Trace 결과

Is Winner  
DECODE?

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar vs Decode

**Winner is  
Decode!**

- ◆ Scalar Sub Query를 사용한 경우는 Index Clustering Factor가 양호하여 테이블을 반복해서 읽는 경우, 발생하는 비효율이 많이 감소하였으며, 동시에 데이터 블록이 모두 Caching 된 상태이므로 Disk I/O가 전혀 없었다.
- ◆ DECODE를 사용한 SQL의 경우 Select List절에 추출된 데이터로 인해, DECODE를 반복적으로 수행하는 횟수가 많으므로 처리 시간이 길었다는 점이다.

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar vs Decode

Low  
Clustering  
Factor



```
CREATE TABLE decode_temp AS SELECT * FROM decode_t1 ORDER BY target;  
CREATE INDEX idx_decode_temp ON decode_temp (sale_dt);  
CREATE INDEX idx_decode_temp_02 ON decode_temp (empno);
```

통계정보 수집  
및  
Buffer Cache  
Flush



```
EXEC dbms_stats.gather_table_stats('scott', 'decode_temp');  
ALTER SYSTEM FLUSH BUFFER_CACHE;
```

## 2. DECODE와 성능이슈

### Row를 Column으로 변환 – Scalar vs Decode

Scalar Sub Query

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.03	0.03	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	3.02	37.27	9932	32720	0	1
total	4	3.05	37.31	9932	32720	0	1

VS

DECODE is Win!

DECODE

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	2.09	3.27	7175	7184	0	1
total	4	2.09	3.28	7175	7184	0	1

## 2. DECODE와 성능이슈

### DECODE 사용시 생기는 비효율



## 2. DECODE와 성능이슈

### DECODE 사용시 생기는 비효율 - Default 값 지정

```
SELECT '2011/12/01' saledt_1201,
       SUM(DECODE(sale_dt, '20111201',target, 0)) AS target_1201,
       SUM(DECODE(sale_dt, '20111201',salecnt, 0)) AS sale_1201,
       '2011/12/02' saledt_1202,
       SUM(DECODE(sale_dt, '20111202',target, 0)) AS target_1202,
       SUM(DECODE(sale_dt, '20111202',salecnt, 0)) AS sale_1202,
       '2011/12/03' saledt_1203,
       SUM(DECODE(sale_dt, '20111203',target, 0)) AS target_1203,
       SUM(DECODE(sale_dt, '20111203',salecnt, 0)) AS sale_1203,
       '2011/12/04' saledt_1204,
       SUM(DECODE(sale_dt, '20111204',target, 0)) AS target_1204,
       SUM(DECODE(sale_dt, '20111204',salecnt, 0)) AS sale_1204,
       -- 중략
       '2011/12/08' saledt_1208,
       SUM(DECODE(sale_dt, '20111208',target, 0)) AS target_1208,
       SUM(DECODE(sale_dt, '20111208',salecnt, 0)) AS sale_1208,
       '2011/12/09' saledt_1209,
       SUM(DECODE(sale_dt, '20111209',target, 0)) AS target_1209,
       SUM(DECODE(sale_dt, '20111209',salecnt, 0)) AS sale_1209,
       '2011/12/10' saledt_1210,
       SUM(DECODE(sale_dt, '20111210',target, 0)) AS target_1210,
       SUM(DECODE(sale_dt, '20111210',salecnt, 0)) AS sale_1210
FROM DECODE_T1
```

NULL 값에 따른 틀린 데이터  
의 추출이 우려되어 Default  
값을 0으로 지정

## 2. DECODE와 성능이슈

### DECODE 사용시 생기는 비효율 - Default 값 지정

#### Default값 지정 시 IF문으로 변환

```
IF 조건 식 THEN
  RETURN 값
ELSE
  RETURN 값
END IF
```

#### Default값 지정 시 IF문으로 변환

```
IF 조건 식 THEN
  RETURN 값
END IF
```



추가되는 ELSE 로직으로 인하여  
CPU를 더 오래 점유함.



## 2. DECODE와 성능이슈

### DECODE 사용시 생기는 비효율 - Default값 제거

```
SELECT '2011/12/01' saledt_1201,
       SUM(DECODE(sale_dt, '20111201',target)) AS target_1201,
       SUM(DECODE(sale_dt, '20111201',salecnt)) AS sale_1201,
       '2011/12/02' saledt_1202,
       SUM(DECODE(sale_dt, '20111202',target)) AS target_1202,
       SUM(DECODE(sale_dt, '20111202',salecnt)) AS sale_1202,
       '2011/12/03' saledt_1203,
       SUM(DECODE(sale_dt, '20111203',target)) AS target_1203,
       SUM(DECODE(sale_dt, '20111203',salecnt)) AS sale_1203,
       '2011/12/04' saledt_1204,
       SUM(DECODE(sale_dt, '20111204',target)) AS target_1204,
       SUM(DECODE(sale_dt, '20111204',salecnt)) AS sale_1204,
       -- 중략
       '2011/12/08' saledt_1208,
       SUM(DECODE(sale_dt, '20111208',target)) AS target_1208,
       SUM(DECODE(sale_dt, '20111208',salecnt)) AS sale_1208,
       '2011/12/09' saledt_1209,
       SUM(DECODE(sale_dt, '20111209',target)) AS target_1209,
       SUM(DECODE(sale_dt, '20111209',salecnt)) AS sale_1209,
       '2011/12/10' saledt_1210,
       SUM(DECODE(sale_dt, '20111210',target)) AS target_1210,
       SUM(DECODE(sale_dt, '20111210',salecnt)) AS sale_1210
FROM DECODE_T1
```

Default 값을 지정하지 않음

## 2. DECODE와 성능이슈

### DECODE 사용시 생기는 비효율 - NVL함수 사용

```
SELECT '2011/12/01' saledt_1201,
       NVL(SUM(DECODE(sale_dt, '20111201',target)),0) AS target_1201,
       NVL(SUM(DECODE(sale_dt, '20111201',salecnt)),0) AS sale_1201,
       '2011/12/02' saledt_1202,
       NVL(SUM(DECODE(sale_dt, '20111202',target)),0) AS target_1202,
       NVL(SUM(DECODE(sale_dt, '20111202',salecnt)),0) AS sale_1202,
       '2011/12/03' saledt_1203,
       NVL(SUM(DECODE(sale_dt, '20111203',target)),0) AS target_1203,
       NVL(SUM(DECODE(sale_dt, '20111203',salecnt)),0) AS sale_1203,
       '2011/12/04' saledt_1204,
       NVL(SUM(DECODE(sale_dt, '20111204',target)),0) AS target_1204,
       NVL(SUM(DECODE(sale_dt, '20111204',salecnt)),0) AS sale_1204,
       -- 중략
       '2011/12/08' saledt_1208,
       NVL(SUM(DECODE(sale_dt, '20111208',target)),0) AS target_1208,
       NVL(SUM(DECODE(sale_dt, '20111208',salecnt)),0) AS sale_1208,
       '2011/12/09' saledt_1209,
       NVL(SUM(DECODE(sale_dt, '20111209',target)),0) AS target_1209,
       NVL(SUM(DECODE(sale_dt, '20111209',salecnt)),0) AS sale_1209,
       '2011/12/10' saledt_1210,
       NVL(SUM(DECODE(sale_dt, '20111210',target)),0) AS target_1210,
       NVL(SUM(DECODE(sale_dt, '20111210',salecnt)),0) AS sale_1210
FROM DECODE_T1
```

Default값을 지정하지 않는  
대신 NVL 함수 사용

## 2. DECODE와 성능이슈

### 3가지 경우의 성능 비교분석

Default 값 지정

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.85	1.85	0	7184	0	1
total	4	1.85	1.85	0	7184	0	1

Default 값 지정  
하지 않은 경우

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.15	1.15	0	7184	0	1
total	4	1.15	1.15	0	7184	0	1

NVL함수 사용

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.17	1.17	0	7184	0	1
total	4	1.17	1.17	0	7184	0	1

## 2. DECODE와 성능이슈

### DECODE의 잘못된 사용에 따른 Index 사용 불가 이슈와 해결방안

```
SELECT /*+ NO_EXPAND */  
      *  
FROM DECODE_T1 a  
WHERE empno = DECODE(:b1, NULL, a.empno, :b1);
```

**:B1 is NULL ?**

**OR**

**:B1 is NOT NULL ?**

## 2. DECODE와 성능이슈

### DECODE의 잘못된 사용에 따른 Index 사용 불가 이슈와 해결방안

**:B1 is NOT NULL ?**

```
VAR  b1 NUMBER
EXEC :b1 := 10

SELECT /*+ NO_EXPAND */ *
  FROM DECODE_T1 a
 WHERE empno = DECODE(:b1, NULL, a.empno, :b1)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.02	0	0	0	0
Execute	1	0.01	0.13	0	0	0	0
Fetch	2	0.32	0.41	1	7185	0	10
total	4	0.34	0.57	1	7185	0	10

Rows

Row Source Operation

10 TABLE ACCESS FULL DECODE\_T1 (cr=7185 pr=1 pw=0 time=417594 us)

**BAD**

## 2. DECODE와 성능이슈

### DECODE의 잘못된 사용에 따른 Index 사용 불가 이슈와 해결방안

```
SELECT * FROM decode_t1 WHERE empno = :b1 AND :b1 IS NOT NULL
UNION ALL
SELECT * FROM decode_t1 WHERE empno = empno AND :b1 IS NULL
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.01	0	0	0	0
Fetch	2	0.00	0.00	0	14	0	10
total	4	0.00	0.01	0	14	0	10

Rows	Row Source Operation
10	UNION-ALL (cr=14 pr=0 pw=0 time=171 us)
10	FILTER (cr=14 pr=0 pw=0 time=164 us)
10	TABLE ACCESS BY INDEX ROWID DECODE_T1 (cr=14 pr=0 pw=0 time=149 us)
10	INDEX RANGE SCAN IDX_DECODE_T1_02 (cr=4 pr=0 pw=0 time=60 us)
0	FILTER (cr=0 pr=0 pw=0 time=1 us)
0	TABLE ACCESS FULL DECODE_T1 (cr=0 pr=0 pw=0 time=0 us)

Good!



### 3. CASE

#### 구문 설명

##### CASE 구문

```
CASE [ expression ]  
  WHEN condition_1 THEN result_1  
  WHEN condition_2 THEN result_2  
  ...  
  WHEN condition_n THEN result_n  
  ELSE result  
END  
)
```

SQL에서 CASE 구문을 시작할 때 사용한다. (선언)

### 3. CASE

#### 구문 설명

#### CASE 구문

```
CASE [ expression ]  
  WHEN condition_1 THEN result_1  
  WHEN condition_2 THEN result_2  
  ...  
  WHEN condition_n THEN result_n  
  ELSE result  
END  
)
```

비교 조건을 기입하는 부분으로, 조건 (CONDITION\_1)이 TRUE이면 THEN절의 값을 리턴한다. (RESULT\_1). 그런데 FALSE라면 이후의 WHEN절의 조건 (CONDITION\_2~N)에 대해 TRUE인지 FALSE인지 비교해 가며 만족하는 값을 추출한다.



### 3. CASE

#### 구문 설명

#### CASE 구문

```
CASE [ expression ]  
  WHEN condition_1 THEN result_1  
  WHEN condition_2 THEN result_2  
  ...  
  WHEN condition_n THEN result_n  
  ELSE result  
END  
)
```

WHEN절의 비교가 모두 FALSE인 경우 ELSE절의 값을 리턴한다. 만약, ELSE절이 없다면 NULL을 리턴한다. ELSE절은 IF문의 ELSE와 같은 역할을 수행한다.

### 3. CASE

#### 조건검색 CASE 사용법

```
SELECT EMPNO,  
       ENAME,  
       CASE WHEN SAL < 1800 THEN 'F등급'  
            WHEN SAL >= 1800 AND SAL < 3000 THEN 'E등급'  
            WHEN SAL >= 3000 AND SAL < 4500 THEN 'D등급'  
            WHEN SAL >= 4500 AND SAL < 6000 THEN 'C등급'  
            WHEN SAL >= 6000 AND SAL < 8000 THEN 'B등급'  
            WHEN SAL >= 8000 THEN 'A등급'  
       END  
FROM EMP
```

- DECODE로 구현하기 복잡한 조건에 대한 보다 용이한 작성 가능.
- 정교한 조건식을 쉽게 보다 작성 가능

### 3. CASE

#### 예제를 통한 CASE 사용법

##### 예제1) CASE 9+1 WHEN 10 THEN '정답' END

```
로직 : IF 9+1 = 10 THEN
      RETURN '정답';
      END IF
변환 : DECODE(9+1, 10, '정답')
```

##### 예제2) CASE 9+1 WHEN 9 THEN '정답1' WHEN 10 THEN '정답2' END

```
로직 : IF 9+1 = 9 THEN
      RETURN "정답1"
    ELSIF 9+1 = 10 THEN
      RETURN "정답2"
    END IF;
변환 : DECODE (9+1, 9, "정답1", 10, "정답2")
```

### 3. CASE

#### 예제를 통한 CASE 사용법

예제3) CASE 9+1 WHEN 9 THEN '정답1' WHEN 10 THEN '정답2' ELSE '정답3' END

```
로직 : IF 9+1 = 9 THEN  
      RETURN '정답1'  
      ELSE IF 9+1= 10 THEN  
            RETURN '정답2'  
      ELSE  
            RETURN '정답3'  
      END IF;  
변환 : DECODE (9+1, 9, '정답1', 10, '정답2', '정답3')
```

## 중첩 CASE 실전예제



**CASE문을 이용하여 SQL 작성하기**

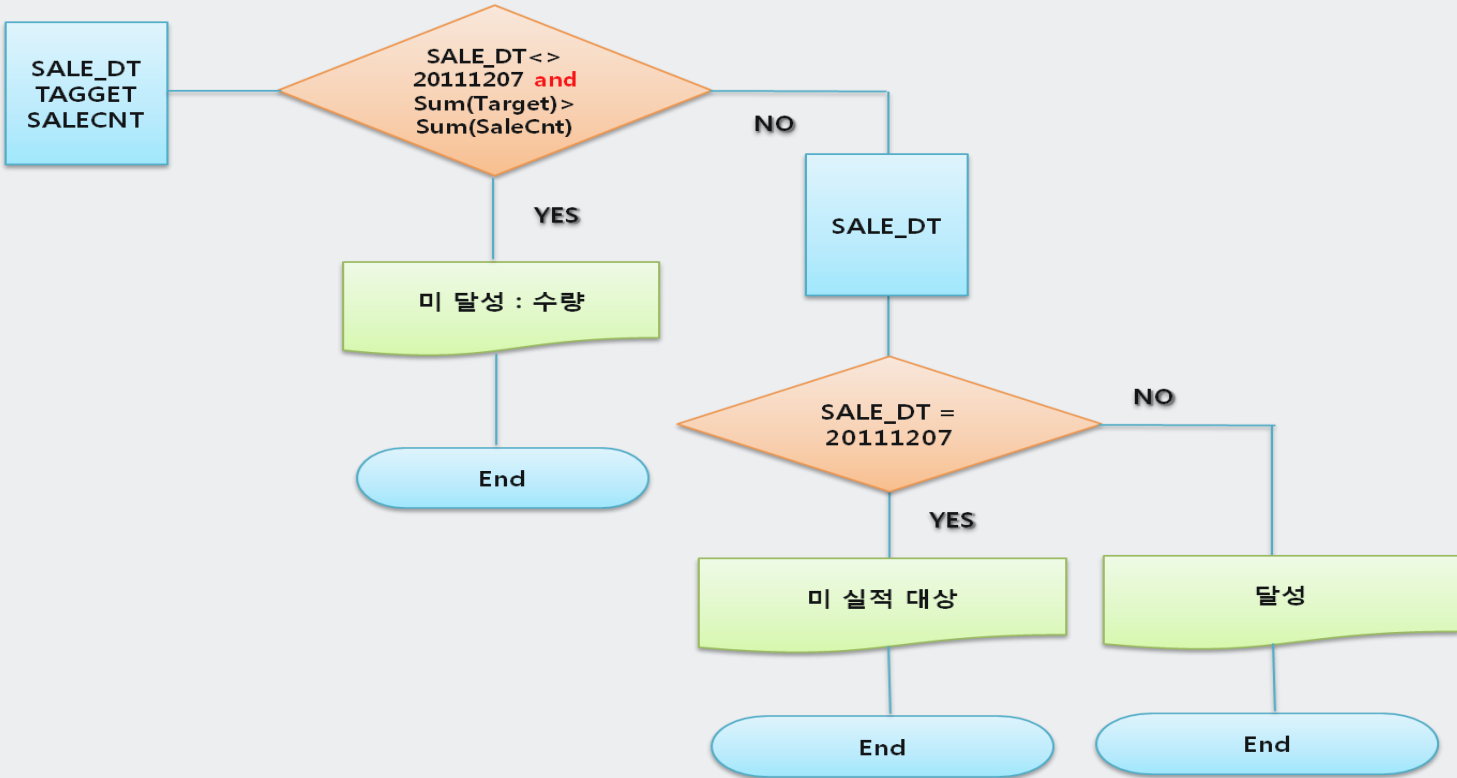
# 중첩 CASE 실전예제

## 개요

◆ 스키용품 점 A사의 2011년 12월의 일자 별 판매 목표량에 따른 판매 실적 달성 정도에 대한 SQL을 작성해보자.

## 조건

◆ 2011/12/07은 실적을 계산하지 않는 날이다. 2011/12/07을 제외한 날짜 중 판매 목표보다 판매량이 낮으면 비교란에 "미 달성" 문구와 함께 목표량과 비교해 부족한 수치를 보여주고, 판매량이 목표량 보다 높으면 "달성"이란 문구를 추출한다. 또한, 2011/12/07인 경우 비교란에 "미 실적대상"으로 표기하기로 한다.



## Summary

DECODE

DECODE와 성능이슈

CASE

# 감사합니다.



서울특별시 강서구 염창동 240-21 우림 비즈니스센터  
Tel. 02) 6203-6300, Fax. 02) 6203-6301  
<http://www.ex-em.com>