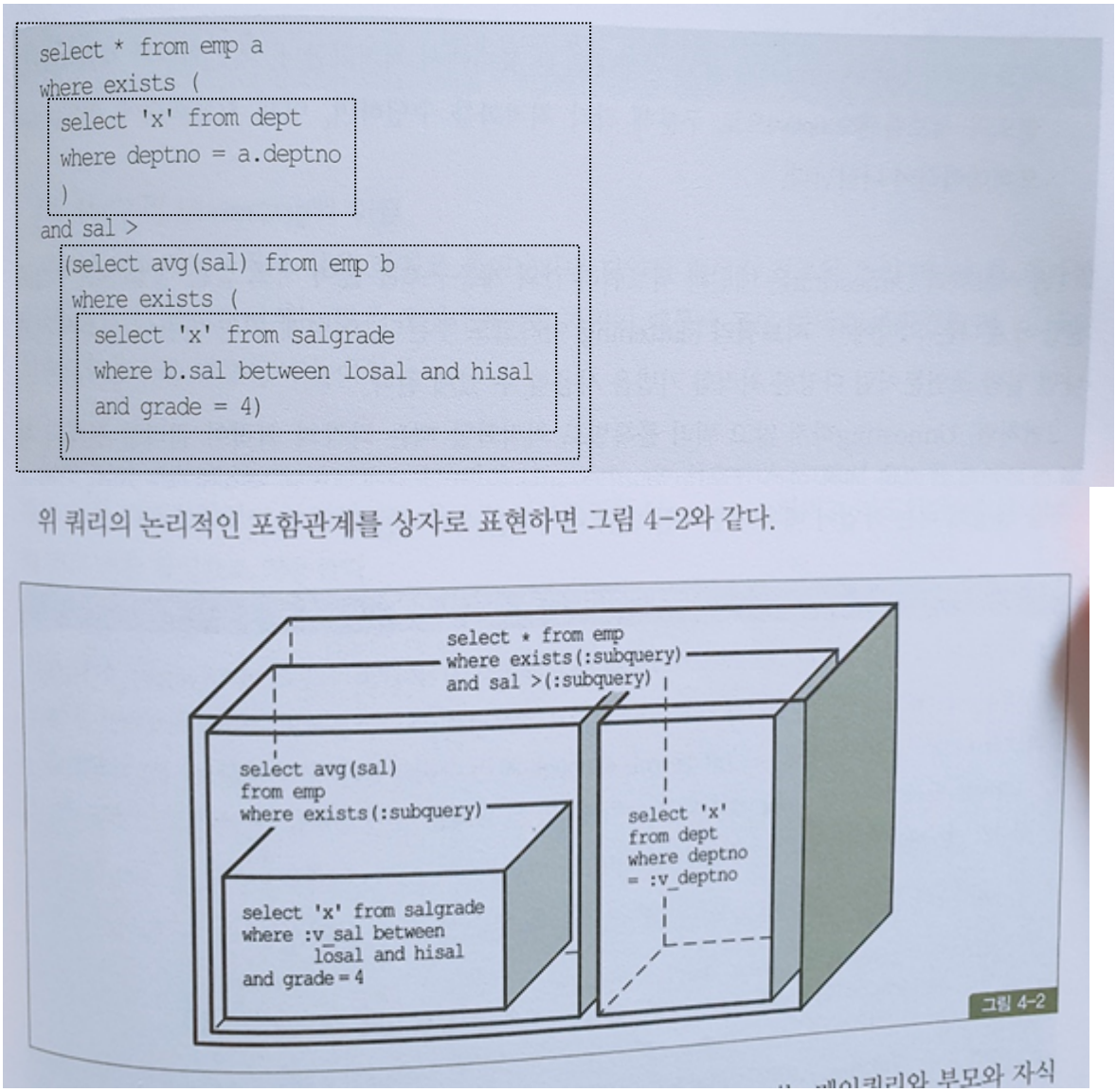
Sub Query

다른 쿼리 블록에 포함된 쿼리 블록을 서브쿼리, 다른 쿼리 블록을 포함한 쿼리 블록을 메인쿼리.



```
select * from emp a
where exists (
  select 'x' from dept
  where deptno = a.deptno
  )
and sal >
  (select avg(sal) from emp b
  where exists (
    select 'x' from salgrade
    where b.sal between losal and hisal
    and grade = 4)
  )
```

위 쿼리의 논리적인 포함관계를 상자로 표현하면 그림 4-2와 같다.

그림 1출처 오라클 성능 고도화 원리와 해법 2

서브쿼리의 사용위치에 따른 구분

Sub Query 는 오는 위치에 따라서 그 이름이 다릅니다.

SELECT (Sub Query) <- 1 행만 반환할 경우 Scalar Sub Query(스칼라 서브쿼리)

FROM (Sub Query) <- Inline View (인라인 뷰) – View 장에서 배웁니다

WHERE (Sub Query) <- Sub Query 라고 부릅니다.

1.1 비상관 서브 쿼리(non_correlated subquery - 연관쿼리)

1.1.1 단일행 비상관 서브 쿼리

1.1.2 다중행 비상관 서브 쿼리

1.2 상관 서브 쿼리(correlated subquery)

1.2.1 단일행 상관 서브 쿼리

1.2.2 다중행 상관 서브 쿼리

사용기준

	비상관 서브 쿼리	상관 서브 쿼리
단일 행	비교 조건	비교 조건
다중 행	IN 조건, NOT IN 조건	EXISTS 조건, NOT EXISTS 조건

비상관 / 상관 쿼리

서브쿼리가 외부(outer)쿼리의 어떤 것도 참조하지 않고 단독적으로 사용된다 -> **비상관 쿼리**
그에 반해 **상관쿼리는** 내부쿼리의 값이 결정되는데 외부쿼리에 의존함

일반적인 예시로는 외부쿼리에 있는 행들에서 관련테이블에 없는 모든 행들 찾는 것

```
SELECT
    A.부서번호
   ,A.부서명
FORM 부서 A
WHERE A.부서번호 IN (SELECT B.부서번호 FROM 사원 B) -
```

```
SELECT
    A.부서번호
   ,A.부서명
FORM 부서 A
WHERE A.부서번호 IN (SELECT B.부서번호 FROM 사원 B WHERE B.USER_ID = A.USER_ID)
```

IN과 EXISTS의 차이

조건	의미
IN 조건	서브 쿼리를 먼저 조회하여, 메인 쿼리에 값을 공급
EXISTS 조건	메인 쿼리를 먼저 조회하여, 서브 쿼리로 존재 여부를 확인

EXISTS는 서브쿼리에서 메인쿼리 컬럼이 참조되므로 자동적으로 상관(연관)쿼리라고 판단.

```
-- EXISTS() 사용
SELECT
    PLAYER_NAME 선수명
    , POSITION 포지션
    , BACK_NO 백넘버
    , HEIGHT 키
    , TEAM_ID 팀아이디
FROM PLAYER P
WHERE EXISTS (
                SELECT
                    *
                FROM TEAM_INFO T
                WHERE  T.TEAM_ID = P.TEAM_ID
                 )
ORDER BY PLAYER_NAME ;

--IN() 사용
SELECT
    PLAYER_NAME
    ,POSITION
    ,BACK_NO
    ,HEIGHT
    ,TEAM_ID
FROM PLAYER P
WHERE TEAM_ID IN (
                    SELECT
                        TEAM_ID
                    FROM TEAM_INFO T
                    )
ORDER BY PLAYER_NAME;
```

주의해야 할 점 :
서브 쿼리 내 NULL이 존재할 경우, NOT EXISTS 조건은 정상적인 결과를 반환함.

고로, NOT IN 보다 NOT EXISTS조건 사용을 권장함 **<<결론 : NOT EXISTS에서 NULL은 조인에 참여하지 않음**

EX)

```
SELECT *
  FROM TEST1 A
WHERE NOT EXISTS (SELECT 1 FROM TEST2 B WHERE A.NO = B.NO)
```

여기서 EXISTS는 서브쿼리가 TRUE인지 FALSE인지 체크하는 것으로 NOT EXISTS는 서브쿼리가 FALSE이면 전체적으로 TRUE. 이는 서브쿼리에서 test1과 test2의 조인 시 null은 빠짐.
해당 서브쿼리의 값의 존재유무로 EXISTS는 참조하기 때문이고 NOT IN은 NULL의 비교연산은 UNKNOWN값을 반환하므로 해당 쿼리의 결과가 없는 것으로 나옴.
If) NOT IN 에서 NULL 값도 비교하게 하고 싶다면 NVL를 이용하여 NULL 처리
*NVL : 값이 NULL인 경우 지정 값을 출력하는 함수
보통,

IN() -> INNER JOIN or EXISTS

NOT IN() -> LEFT OUTER JOIN or NOT EXISTS 로 대체해서 보통 씀

아래는 NOT EXIST 조건을 사용한 쿼리다.

```
1  SELECT a.* FROM t1 a WHERE NOT EXISTS (SELECT 1 FROM t2 x WHERE x.c1 = a.c1);
2
3  C1 C2
4  -- --
5   3  4
6
7  1 행이 선택되었습니다.
```

위 쿼리는 아래의 슈도 코드로 표현할 수 있다. NOT EXISTS 조건은 서브 쿼리와의 조인이 1번이라도 성공하면 행을 반환하지 않는다. 이런 조인 방식을 안티 조인(anti join)이라고 한다.

```
1  <<outer>>
2  FOR a IN (SELECT * FROM t1) LOOP
3      FOR b IN (SELECT * FROM t2 b WHERE b.c1 = a.c1) LOOP
4          CONTINUE outer;
5      END LOOP;
6      결과 반환
7  END LOOP outer;
```

P.294

*Anti JOIN : 테이블의 값을 추출할 때 조인의 대상이 되는 테이블과 일치하지 않는 데이터를 추출하는 조인방식.
**원래 안티조인은 메인쿼리와 서브쿼리에 널이 아니라는 보장이 있어야 가능했지만 오라클 11g 의 Optimizer 는 널을 인지하면서 조인을 실행하는 Null Aware Anti Join(ANTI NA)이라는 새로운 Join Operation 을 추가했다. 즉 서브쿼리, 메인쿼리에 IS NOT NULL 조건을 주지 않아도 안티조인이 된다는 이야기.

***하지만 실제 테스트 해보면 **머지 안티 조인**의 경우 IS NOT NULL 조건을 안주어도 잘되지만 **해시 안티 조인**의 경우에는 서브쿼리, 메인쿼리에 IS NOT NULL 조건을 줘야만 동작한다.

<SAMPLE

비상관 쿼리에 있어서

단일행(> = <) / 다중행 쿼리(IN ANY ALL) *대체적으로 =보단 IN을 권장함

검색되는 행이

1_단일인가? 다중인가

2_ 사용되는 연산자의 종류에 따른 구별

연산자	기 능
IN	검색된 값 중에 하나만 일치하면 참이다.
ANY	검색된 값 중에 조건에 맞는 것이 하나 이상 있으면 참이다.
ALL	모든 검색된 값과 조건에 맞아야한다.

ALL과 ANY 정리

사 용	의 미	같은 표현
컬럼 > ANY	가장 작은 값보다 크다.	컬럼 > MIN
컬럼 < ANY	가장 큰 값보다 작다.	컬럼 < MAX
컬럼 > ALL	가장 큰 값보다 크다.	컬럼 > MAX
컬럼 < ALL	가장 작은 값보다 작다.	컬럼 < MIN

```
SELECT
      PLAYER_NAME 선수명
      , POSITION 포지션
      , BACK_NO 백넘버
FROM PLAYER
WHERE HEIGHT > ANY(
                        SELECT
                              HEIGHT
                        FROM PLAYER
                        )
ORDER BY PLAYER_NAME;
```

최소값 165보다 큰 키를 찾음

스칼라 서브 쿼리

- 스칼라: 단일 값(단일 행, 단일 열)
- 함수처럼 한 레코드 당 하나의 값만을 리턴하는 서브쿼리(=결과를 1행씩 반환)

If) 한 개의 테이블에서 복수개의 컬럼을 가져오고 싶은 경우, 스칼라를 사용하지 말고 join 사용 (== 두 개의 컬럼을 ||로 묶어서 하나의 컬럼을 리턴

*이는 테이블 전체를 (FULL SCAN 해야하는) 다 읽어야 하는 비효율이 존재함, in-line view 안에서 스칼라 서브쿼리를 한 번 사용하고, 바깥쪽에서 substr을 이용해 잘라서 사용 _ex만들어보기)

Ex)

```
//TODO EMP 테이블 전체를 다 읽어야 하는 비효율 발생 왜?
SELECT D.DEPTNO, D.DNAME, AVG_SAL, MIN_SAL, MAX_SAL
    FROM DEPT D,
            (SELECT DEPTNO, AVG(SAL) AVG_SAL, MIN(SAL) MIN_SAL, MAX(SAL) MAX_SAL
FROM EMP GROUP BY DEPTNO) E
```

```
WHERE D.DEPTNO = E.DEPTNO(+)
    AND D.LOC = 'CHICAGO';    <<-- EMP 테이블 전체를 다 읽어야 하는 비효율 발생
```

- 주로 select-list에서 사용(select list scalar subquery), 컬럼이 올 수 있는 대부분 위치에 사용가능.
- 서브쿼리의 결과가 없을 경우 NULL을 돌려준다. 이는 outer join과 동일(outer 조인문과 100% 같은 결과를 출력한다는 의미)
- 스칼라 서브쿼리를 사용할 경우, **NL조인**에서 inner 쪽 인덱스와 테이블에 나타나는 버퍼 pinning 효과는 사라짐.

>>참조link :

https://ukja.tistory.com/94

https://m.blog.naver.com/PostView.nhn?blogId=kmymk&logNo=110082741528&proxyReferer=https%3A%2F%2Fwww.google.com%2F

오라클에서 스칼라 서브 쿼리 **동작 원리**

1 메인 쿼리를 수행한 후 스칼라 서브쿼리에 필요한 값을 제공
2 스칼라 서브쿼리를 수행하기 위해 필요한 데이터가 들어있는 블록을 메모리로 로딩
3 메인 쿼리에서 주어진 조건을 가지고 필요한 값을 찾아 메모리에 입력값과 출력값을 메모리 내의 query exectution cache(내부 캐시)에 저장해 둠
  여기서 입력 값은 메인쿼리에서 주어진 값이고 출력값은 스칼라 서브쿼리를 수행한 후 나온 결과값
  이 값을 저장하는 캐쉬값을 저장하는 파라미터는 _query_execution_cache_max_size
  Oracle 8i, 9i -256개 엔트리를 캐싱, 10g에선 파라미터에 의해 chche size가 결정.
4 다음 조건이 메인 쿼리에서 스칼라 서브쿼리로 들어오면 해쉬 함수를 이용해 해당 값이 캐쉬에 존재하는지 찾고, 있으면 즉시 결과값을 출력하고 없으면 다시 블록을 엑세스 해서 해당 값을 찾은 후 다시 메모리에 캐쉬해 둔다.
  If) 해시 충돌이 발생할 시 오라클은 기존 캐시 엔트리를 그대로 둔 채, 스칼라 서브쿼리만 반복수행. So, 입력 값이 반복적으로 입력되면 스칼라 서브쿼리도 반복수행
5 메인 쿼리가 끝날 때까지 반복

So, 데이터 종류와 개수도 적은 코드성 테이블에서 데이터를 가져올 때 사용하는 것을 권장
So2, 데이터의 분포도나 다량의 건수를 처리하고자 할 때는 nested loop join과 마찬가지로 전체적인 처리량은 랜덤 블럭 엑세스로 인해 성능저하가 있으므로 hash join이나 sort merge join을 이용해서 outer join을 하는 것이 유리함.

"메인 쿼리와 서브 쿼리의 조인 차수가 M:1이기 때문에 스칼라 서브 쿼리를 사용할 필요가 없다. 스칼라 서브 쿼리는 메인 쿼리와 서브 쿼리의 조인 차수가 1:M일 때 사용하는 것이 일반적이다."
-p.307 >서브쿼리 사용을 이용한 access횟수에 있어 효율성을 가져와야 하기 때문.
(스칼라 서브 쿼리의 핵심은 캐싱을 통한 수행 횟수 최소화, 이에 따라 동일한 결과 값을 사용할 수 있는 활용도가 높기 때문에 캐싱을 통한 높은 효율을 거둘 수 있음.)

Q) P.308에서 "필수 관계라도 스칼라- 서브 쿼리에 일반 조건이 존재하면 아우터 조인으로 조인해야 한다"

스칼라 서브쿼리의 권장사용 상황

1. 코드성 테이블의 명칭만 가져오는 경우 *하지만 결과집합이 대량인 경우는 조인사용(반복access 비효율)
2. 단순히 count, sum 등의 간단한 집계 작업인 경우
3. 조인과 group by가 같이 있는 경우

   Ex) 접근하는 페이지의 수는 동일하나, 사람 별로 제안건수가 10만 건 씩이면
   조인, group by의 경우 100만 건의 조인된 집합이 생성된 후 group by를 통해 count 실행
   하지만, 스칼라의 경우 조인연산이 필요 없고 조인된 집합을 가져가는 부하가 없어지기 때문

서브쿼리 권장사용 상황

1. SELECT 절에서 해다 테이블의 컬럼을 조회하지 않는 경우
2. OUTER 조인을 사용하지 않는 테이블
3. 서브쿼리로 변경할 테이블에 의해 최종 추출 결과가 증가하지 않을 경우
   -> 서브쿼리는 추출되는 데이터를 감소시키는 역할을 수행하기 때문임(조인횟수 감소)

일반 조인 -> 스칼라 서브쿼리,

서브쿼리로의 **변경기준 이유**

스칼라 서브쿼리는 추출되는 데이터에 어떠한 변화를 줄 수 없으며, 서브쿼리는 추출되는 데이터를 감소시키는 역할을 수행함. FROM절을 이용한 일반 조인도 데이터에 변화를 발생시키지 않는다면 스칼라 서브쿼리를 고려할 수 있으며, 추출되는 데이터를 감소시킨다면 서브쿼리 고려가능

인라인 뷰

서브 쿼리의 일종으로 보통 FROM 절에 위치해서 테이블처럼(view) 사용하는 것
일반적인 뷰를 정적 뷰라하고, 인라인 뷰를 동적 뷰라고 함
SQL문이 실행될 때만 임시적으로 생성되는 동적 뷰. So, DB에 해당 정보가 저장되지 않음.
보통 Top-N을 구할 때 사용

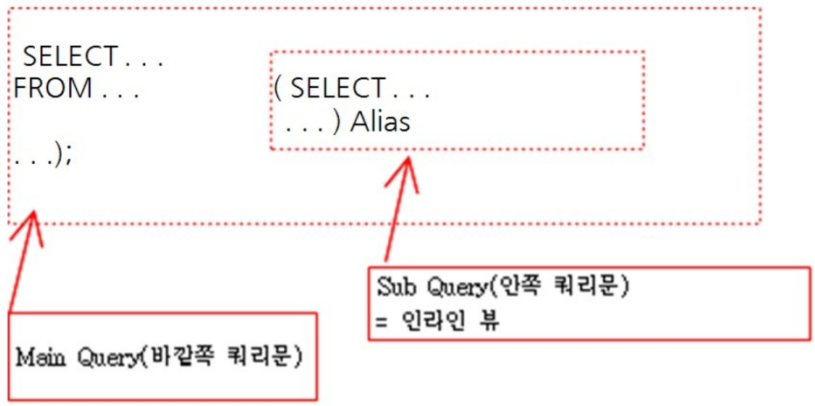


그림 2 인라인 뷰

기본적으로 조인 < 서브쿼리 < EXISTS 순으로 사용하는 경향이 있음

So, 인라인 뷰를 올바른 사용 법은 처리 범위를 감소시키는 형태로 수행되어야 함.

* 튜닝 전

```
SELECT MIN(T2.상품명) 상품명, SUM(T1.판매수량) 판매수량, SUM(T1.판매금액) 판매금액
FROM 일별상품판매 T1, 상품 T2
WHERE T1.상품코드 = T2.상품코드
AND T1.판매일자 BETWEEN '20090101' AND '20091231'
GROUP BY T2.상품코드;
```

* 튜닝 후

포인트 : 인라인 뷰를 활용해서 Group by 결과를 줄이고 조인 한다.

```
SELECT T2.상품명, T1.판매수량, T1.판매금액
FROM 상품 T2, (SELECT 상품코드, SUM(판매수량) 판매수량, SUM(판매금액) 판매금액
                FROM 일별상품판매
                WHERE 판매일자 BETWEEN '20090101' AND '20091231'
                GROUP BY 상품코드) T1
WHERE T2.상품코드 = T1.상품코드;
```

```
CREATE OR REPLACE VIEW empsal_vw AS
      SELECT
            ROWNUM rown,
            employee_id,
            first_name,
            salary
      FROM
            (
                  SELECT
                        ROWNUM,
                        employee_id,
                        first_name,
                        salary
                  FROM
                        employees
                  ORDER BY
                        salary DESC
            )
      WHERE
            ROWNUM <= 5;
```

그림 3 ROWNUM을 이용한 인라인 뷰 이용

*ROWNUM : 테이블에 주어지는 고유번호, 입력한 순서대로 1 부터 부여, ORDER BY 를 해도 지정된 ROWNUM 은 불변

**오라클은 테이블에서 정보를 조회하거나 조인된 row 가 반환되는 순서에 따라 ROWNUM 을 부여

ROWNUM 을 사용한 view 이기 때문에 기본 테이블의 데이터<-> 뷰의 데이터 각 수정 시 연쇄적으로 DATA 수정이 이뤄짐

다음과 같은 질의문은 첫번째 query 를 제외하고는 어떠한 결과도 출력되지 않는다.

- select * FROM emp where ROWNUM = 1; (<= 가능) (o)
- select * FROM emp where ROWNUM > 1; (x)

- select * FROM emp where ROWNUM = 2 (x)
- select * FROM emp where ROWNUM BETWEEN 2 and 10; (x)
-> ROWNUM 은 < 조건이나 <=조건을 사용해야 함

분석함수를 이용한 TOP-N
보통 분석함수를 통하여 TOP 처리를 하는 때는 동 순위의 처리를 위해 사용하는 경우
그 외에는 ROWNUM 을 권장

(*분석함수의 사용법에 따라 분석함수의 AccessPath 가 달라지기 때문. 이는 인덱스나, 데이터 타입 등 신경 써야 할 부분이 많아짐)

```
WINDOW (SORT)
WINDOW (SORT PUSHED RANK)
WINDOW (NOSORT)
WINDOW (NOSORT STOPKEY)   <-
WINDOW (BUFFER)
WINDOW (BUFFER PUSHED RANK)
WINDOW (CHILD PUSHED RANK)
WINDOW (IN SQL MODEL) SORT
```

*TMI 로 분석함수의 내부 operation 은 총 8 가지

WHY?
TOP N 은 '<' or '<=' 사용하게 되는데
'<='조건에서 기존의 인덱스가 있는 상황일 경우 분석함수는 'WINDOW' 'NOSORT' 'STOPKEY PLAN'을 보면 1 건 더 스캔이 이뤄짐

즉 '1 위 그룹'은 109 건이지만 109 건을 scan 하였고 WINDOW FILTER 과정에서 1 건이 제거되었음.
ROWNUM 을 사용 시엔 정확히 108 건만 scan 함
즉, ROWNUM 은 정확히 case 를 잘라내지만 분석함수는 operation 에 따라 scan 률이 천차만별임
+조건을 <로 변경 시 이는 비효율이 더 크게 드러남
(*분석함수를 이용한 TOP 처리 시 V_RANK 에 2 를 대입하면 3 위 그룹의 첫 번째 ROW 까지 스캔 하기 때문
So, WINDOW STOPKEY 사용 시엔 =를 생략하면 안됨)

```
CREATE INDEX SH.SALES_IDX01 ON SH.SALES
(AMOUNT_SOLD);
```

```
SELECT /*+ gather_plan_statistics FIRST_ROWS(1) */
       *
  FROM (SELECT prod_id, amount_sold,
               RANK () OVER (ORDER BY amount_sold DESC) top_sales
          FROM sh.sales a)
 WHERE top_sales <= :V_RANK;   --> 숫자 1 대입
```

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                              | Name        | Starts | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |             |      1 |    108 |00:00:00.01 |      11 |
|*  1 |  VIEW                                  |             |      1 |    108 |00:00:00.01 |      11 |
|*  2 |   WINDOW NOSORT STOPKEY                |             |      1 |    108 |00:00:00.01 |      11 |
|   3 |    TABLE ACCESS BY GLOBAL INDEX ROWID| SALES       |      1 |    109 |00:00:00.01 |      11 |
|   4 |     INDEX FULL SCAN DESCENDING         | SALES_IDX01 |      1 |    109 |00:00:00.01 |       4 |
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("TOP_SALES"<=:V_RANK)
   2 - filter(RANK() OVER ( ORDER BY INTERNAL_FUNCTION("AMOUNT_SOLD") DESC )<=:V_RANK)
```

그림 4분석 함수 방식

```
SELECT /*+ GATHER_PLAN_STATISTICS FIRST_ROWS(1) */ *
  FROM (SELECT prod_id, amount_sold
          FROM sh.sales a
         ORDER BY amount_sold DESC )
WHERE ROWNUM <= V_RANK --> 108 대입;
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name        | Starts | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |             |      1 |    108 |00:00:00.01 |      10 |
|*  1 |  COUNT STOPKEY                      |             |      1 |    108 |00:00:00.01 |      10 |
|   2 |   VIEW                              |             |      1 |    108 |00:00:00.01 |      10 |
|   3 |    TABLE ACCESS BY GLOBAL INDEX ROWID| SALES       |      1 |    108 |00:00:00.01 |      10 |
|   4 |     INDEX FULL SCAN DESCENDING      | SALES_IDX01 |      1 |    108 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(ROWNUM<:V_RANK)
```

그림 5 ROWNUM 방식

```
SELECT /*+ gather_plan_statistics FIRST_ROWS(1) */
       *
  FROM (SELECT prod_id, amount_sold,
               RANK () OVER (ORDER BY amount_sold DESC) top_sales
          FROM sh.sales a)
 WHERE top_sales < V_RANK;  --> 숫자 2 대입
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name        | Starts | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |             |      1 |    108 |00:00:00.01 |      13 |
|*  1 |  VIEW                               |             |      1 |    108 |00:00:00.01 |      13 |
|*  2 |   WINDOW NOSORT STOPKEY             |             |      1 |    115 |00:00:00.01 |      13 |
|   3 |    TABLE ACCESS BY GLOBAL INDEX ROWID| SALES       |      1 |    116 |00:00:00.01 |      13 |
|   4 |     INDEX FULL SCAN DESCENDING      | SALES_IDX01 |      1 |    116 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("TOP_SALES"<:V_RANK)
   2 - filter(RANK() OVER ( ORDER BY INTERNAL_FUNCTION("AMOUNT_SOLD") DESC )<:V_RANK)
```

그림 6 분석 함수 방식<조건으로 변경한 결과

*그림 6에서 유의해야 할 점은 바인딩 변수의 데이터 타입을 문자형으로 인식하게 되어 STOPKEY(부분범위 처리에 특화 됨)가 적용되지 않아서 index scan 부분범위 처리가 아닌 fullscan 전체 범위 처리가 되어 성능의 하락이 도출됨.

```
-------------------------------------------------------------------------------------------------------------
| Id | Operation | Name | Starts | A-Rows | A-Time | Buffers |
-------------------------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 1 | 108 |00:00:00.01 | 20972 |
|* 1 | VIEW | | 1 | 108 |00:00:00.01 | 20972 |
| 2 | WINDOW NOSORT | | 1 |918K|00:00:00.91 | 20972 |
| 3 | TABLE ACCESS BY GLOBAL INDEX ROWID| SALES | 1 | 918K|00:00:00.56 | 20972 |
| 4 | INDEX FULL SCAN DESCENDING | SALES_IDX01 | 1 | 918K|00:00:00.13 | 2455 |
-------------------------------------------------------------------------------------------------------------

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

1 - filter("TOP_SALES"<=TO_NUMBER(:V_RANK))
```

그림 7 바인딩 변수에 데이터 형을 지정할 수 있는 다른 툴에서 숫자형으로 지정하는 경우

*Id 2 에서 window nosort 에서 stopkey operation 이 적용되지 않아 부정확하게 그림 6 처럼 도출 됨. So, NOSORT 시 STOPKEY 를 위해 Date type 확인 필수

일반적으로 P.401 과 같이 WINDOW SORT PUSHED RANK operation 이 발생함 이는 Full Table Scan 에서 분석함수 컬럼을 filter 조건으로 사용했기 때문

참조 링크 : https://scidb.tistory.com/entry/%EB%B6%84%EC%84%9D%ED%95%A8%EC%88%98%EC%9D%98-%EC%8B%A4%ED%96%89%EA%B3%84%ED%9A%8D-1%EB%B6%80

ROW LIMITING 절

12c 부터 추가된 기능이 존재함.

ANSI 기준

offset

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

문법

```
offset 숫자 rows;
```

fetch ~ row

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

문법

```
fetch first 숫자 rows only
```

fetch ~ percent

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

문법

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

offest ~ fetch ~ percent 결합

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

문법

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

with ties 결합

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

문법

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

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

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

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

Sample 테이블에서 각 항목에 대한 case

offset을 이용한 데이터 건너뛰기

SQL 문

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

결과

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

fetch ~ rows를 이용한 부분 범위 처리

SQL 문

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

결과

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

fetch ~ percent를 이용한 부분 범위 처리

SQL 문

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

결과

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

offset ~ fetch ~ percent를 이용한 부분 범위 처리

SQL 문

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

결과

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

with ties를 이용한 동순위 데이터 가져오기

SQL 문

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

결과

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

TOP-N 쿼리와 UNION ALL 연산자

*UNION ALL 연산자 : 여러 개의 SQL문의 결과에 대한 합집합과 공통부분을 더한 합집합
Ex) 1번째 select(x,y,z) + 2번째 select(o,p,q,z)인 경우 union all = x,y,z,z,o,p,q

```
1  SELECT *
2    FROM (SELECT 1 AS tp, deptno AS no, dname AS name FROM dept
3          UNION ALL
4          SELECT 2 AS tp, empno  AS no, ename AS name FROM emp
5          ORDER BY tp, no)
6   WHERE ROWNUM <= 3;
7
```

그림 8 데이터 집합 별로 Top-N 처리를 수행x
Select union all select order by

```
1  SELECT *
2    FROM (SELECT *
3            FROM (SELECT 1 AS tp, deptno AS no, dname AS name FROM dept ORDER BY no)
4           WHERE ROWNUM <= 3
5          UNION ALL
6          SELECT *
7            FROM (SELECT 2 AS tp, empno  AS no, ename AS name FROM emp  ORDER BY no)
8           WHERE ROWNUM <= 3)
9   WHERE ROWNUM <= 3;
```

그림 9 데이터 집합 별로 Top-N 처리를 수행O
Select order by union all select order by
Top n 시 발생하는 소트의 부하를 경감할 수 있음