2주차 과제 풀이

조인은 작은 집합에서 큰 집합으로 진행한다. 단, 조인조건으로 드라이빙/필터링된 이후의 결과값을 기준으로 작은 집합과 큰 집합을 구분한다. 드라이빙 테이블이 되려면 하나 이상의 드라이빙 조건이 있어야 한다.

룩업 테이블에는 조인 연결고리 + 추가조건 컬럼이 인덱스 구성요소 컬럼이 된다.

드라이빙 조건인 경우 between 이나 like 조건의 차이는 없다. LIke를 드라이빙 조건으로 수행하면 수행결과에 filter 가 찍히지만 성능상의 차이는 없다.

룩업 테이블의 인덱스에 조인연결고리가 추가조건 컬럼보다 앞에 오는게 범용성 측면에서 낫다.

드라이빙 테이블의 탐색결과 수 만큼 조인을 진행한다.

룩업테이블의 루트인덱스는 항상 거쳐가므로 룩업테이블의 루트인덱스에 대한 버퍼 피닝이 발생한다.

테이블 액세스 비효율을 낮추는 법은 검색조건에서 사용된 컬럼이 인덱스에 모두 포함되는 것이다. 인덱스 매칭도를 높히는 방법은 최대한 많은 조건이 드라이빙 조건으로 읽히도록 하는 것이다.

룩업 테이블은 조인의 수 만큼 수직적 탐색을 진행하므로 룩업 테이블 인덱스의 DEPTH를 낮춰야 수직적 탐색 효율이 높아진다.

지난주 복습

NL JOIN

드라이빙 테이블과 룩업 테이블이 존재한다. 조인연결컬럼은 룩업 테이블에만 존재하면 된다. 반드시 존재해야 한다. 소량의 데이터를 위해 인덱스를 사용하는 랜덤엑세스 위주의 조인이다. = 조인이 아니라도 가능하다.

HASH JOIN

큰 집합과 작은 집합을 구분한 다음 작은 집합의 조인연결고리컬럼 값을 HASH FUNCTION 에 넣어서 나온 결과값을 PGA의 해시테이블에 적재한다. 해시테이블의 크기가 적다면 명령어를 통해 사이즈를 키울 수 있다. 해쉬조인은 = 연산자 조인만 가능하다.

```
-- 해시테이블의 총 크기를 지정한다.
alter session set Hash_Area_Size = '2GB';
```

큰 집합의 조인연결고리컬럼 값을 HASH FUNCTION에 넣어 나온 결과값을 해시테이블과 비교하여 같은 값이면 조인한다. 해시조인의 성능의 성패를 결정하는 요인은 두 가지다.

- Build input table 의 크기가 해시 테이블에 들어갈만큼 충분히 작아야한다.

- 1:M 관계에서 1쪽 집합이 Build input으로 올라가는게 해시링크가 적게 발생해 성능에 유리하다

조인순서는 작은쪽에서 큰쪽으로 하는게 유리하다. 단, 소트머지조인의 작은쪽 테이블이 sort area 에 담지 못할만큼 큰 경우 큰쪽에서 작은쪽으로 조인하는게 유리하다 그 이유는 Lookup table에서 발생하는 디스크 I/O를 최소화하기 위함이다.

OUTER JOIN

일반적인 조인에서는 Outer table 이 항상 driving table 이 된다. 하지만 Hash join 에서는 Inner table 이 Driving Table 로 올라갈 수 있다.

```
-- 특정 테이블을 DRIVING 테이블로 올린다.
/*+swap_join_inputs*/
```

해시조인의 실행계획에는 계단식, 등호형이 있다.

계단식은 선두 테이블의 조인 결과를 probeinput으로 후행 테이블을 probeinput으로 삼아 조인한다.

일반적인 실행계획 해석 순서와 달라지는 경우가 3가지 존재한다.

1. select 절에 있는 스칼라서브쿼리
2. 해쉬조인 계단식 (상위에 기술된 테이블이 가장 마지막에 종료된다. 재귀적으로)
3. Prefetch

스칼라 서브쿼리

스칼러 서브쿼리의 정의는 하나의 컬럼에 대응되는 값을 반환하는 서브쿼리를 말한다. select, where 에 위치할 수 있다.

select 절에 위치하는 경우 하나의 값만을 반환해야하고 조건절의 경우 In 연산자 뒤에 위치하는 경우 다수의 값을 반환할 수 있다.

스칼라 서브쿼리는 캐싱을 한다. 캐싱되는 값은 서브쿼리 내부에 존재하는 부모 쿼리의 테이블의 연결컬럼값과 서브쿼리로부터 반환되는 값 두 가지이다. 배치프로그램에서는 스칼라 서브쿼리를 사용해선안된다.

```
-- 대량의 데이터가 들어가기 때문에 해쉬조인으로 유도해야 한다.
-- 스칼러 서브쿼리를 풀때는 Outer Join으로 풀어야 한다

insert into t_x
select c1, c2 ...(, sclar subquery)
from  t1,t2
where
```

누적매출 구하기

```
select
  T1.지점, T1.판매월, MIN(T1.매출액) 매출액, sum(t2.매출액) 누적매출액
from 누적매출 t1, 누적매출 t2
where t2.지점 = t1.지점
and t2.판매월 <= t1.판매월
group by t1.지점, t1.판매월
```

윈도우 함수

```
select sum(sal) over ( partition by 부서 order by 직급
  rows between unbounded preceding and current row
) 부서별 연봉 총액
From emp
order by 부서, 직급
```

선분이력 끊기

```
-- 월말 기준으로 선분을 끊는 경우
DROP TABLE HALIN
CREATE TABLE HALIN
       (
        CUSTNO VARCHAR2(10),
        STARTDT DATE,
        ENDDT DATE,
        RATIO NUMBER(1,1)
        );

INSERT INTO HALIN VALUES
('C101', TO_DATE('20190601', 'YYYYMMDD'), TO_DATE('20190708', 'YYYYMMDD'),
0.3);
INSERT INTO HALIN VALUES
('C101', TO_DATE('20190709', 'YYYYMMDD'), TO_DATE('20190807', 'YYYYMMDD'),
0.2);
INSERT INTO HALIN VALUES
('C101', TO_DATE('20190808', 'YYYYMMDD'), TO_DATE('20190828', 'YYYYMMDD'),
0.5);
INSERT INTO HALIN VALUES
('C101', TO_DATE('20190821', 'YYYYMMDD'), TO_DATE('20190910', 'YYYYMMDD'),
0.2);
INSERT INTO HALIN VALUES
('C101', TO_DATE('20190911', 'YYYYMMDD'), TO_DATE('99991231', 'YYYYMMDD'),
0.3);

select CUSTNO,
TO_CHAR(GREATEST(STARTDT, TO_DATE('20190801', 'YYYYMMDD')), 'YYYYMMDD')
STARTDT,
TO_CHAR(LEAST(ENDDT, TO_DATE('20190801', 'YYYYMMDD')),'YYYYMMDD') ENDDT,
RATIO
from halin
where custno = 'C101'
AND STARTDT <= TO_DATE('20190831', 'YYYYMMDD')
AND ENDDT >= TO_DATE('20190801', 'YYYYMMDD')
```

데이터 복제를 통한 소계 구하기

```
SELECT B.NO,
A.DEPTNO,
A.EMPNO,
A.SAL
FROM EMP A,
(
  SELECT ROWNUM NO FROM DUAL
  CONNECT BY LEVEL <= 2
) B
ORDER BY 2;
;

SELECT
A.DEPTNO DEPTNO,
DECODE(B.NO, 1 , TO_CHAR(A.EMPNO), 2, 'BUSEO TOTAL') EMPNO,
SUM(A.SAL) T_SAL,
ROUND(AVG(A.SAL)) A_SAL
FROM EMP A,
(
  SELECT ROWNUM NO FROM DUAL
  CONNECT BY LEVEL <= 2
) B
GROUP BY A.DEPTNO, B.NO, DECODE(B.NO, 1 , TO_CHAR(A.EMPNO), 2, 'BUSEO
TOTAL')
ORDER BY 1, 2;

-- rollup 함수 활용 단순하게 작성

SELECT
DEPTNO, EMPNO,
  CASE WHEN GROUPING(DEPTNO) = 1 AND GROUPING(EMPNO) = 1 THEN 'TOTAL'
  WHEN GROUPING(EMPNO) = 1 THEN 'BUSEO TOTAL'
  ELSE TO_CHAR(EMPNO)
END DIV,
SUM(SAL) T_SAL,
ROUND(AVG(SAL)) A_SAL
FROM EMP
GROUP BY ROLLUP(DEPTNO, EMPNO)
ORDER BY DEPTNO, DIV;
```

GROPING SET

```
--- 일별 SET
SELECT
SUBSTR(ORDER_DT, 1, 8)
CUST_NO,
SUM(ORDER_PRICE)
FROM T_ORDER74
WHERE ORDER_DT >= '20160601'
AND ORDER_DT < '20160701'
```

```
GROUP BY SUBSTR(ORDER_DT, 1, 8), CUST_NO
--- 월별 SET
SELECT
SUBSTR(ORDER_DT, 1, 6)
CUST_NO,
SUM(ORDER_PRICE)
FROM T_ORDER74
WHERE ORDER_DT >= '20160601'
AND ORDER_DT < '20160701'
GROUP BY SUBSTR(ORDER_DT, 1, 6), CUST_NO;


--- GROUPING SET 활용

SELECT 구분, DECODE(ORDER_MM, NULL, ORDER_DT, ORDER_MM) 일자
,CUST_NO, ORDER_PRICE_SUM
FROM
(
    SELECT DECODE(TO_CHAR(ORDER_DT, 'YYYYMM'), NULL, '일별', '월별') 구분
        , TO_CHAR(ORDER_DT, 'YYYYMMDD') ORDER_DT
        , TO_CHAR(ORDER_DT, 'YYYYMM') ORDER_MM
        , CUST_NO
        , SUM(ORDER_PRICE) ORDER_PRICE_SUM
    FROM T_ORDER74
    WHERE ORDER_DT BETWEEEN TO_DATE('20160601', 'YYYYMMDD')
          AND TO_DATE('20160630 235959', 'YYYYMMDD HH24MISS')
    GROUP BY GROUPING SETS ( -- N개의 grouping by를 union all 한다.
      (TO_CHAR(ORDER_DT, 'YYYYMMDD', CUST_NO)),
      (TO_CHAR(ORDER_DT, 'YYYYMM', CUST_NO))
    )
)
ORDER BY DECODE(구분, '일별', 1, 2), ORDER_DT, ORDER_MM, CUST_NO

SELECT 'MONTHLY' DIV
,SUBSTR(TO_CHAR(HIREDATE, 'YYYYMMDD'), 1, 6) D
,COUNT(EMPNO) COUNT
FROM EMP
GROUP BY SUBSTR(TO_CHAR(HIREDATE, 'YYYYMMDD'), 1, 6)
UNION ALL
SELECT 'YEALY' DIV
,SUBSTR(TO_CHAR(HIREDATE, 'YYYYMMDD'), 1, 4) D
,COUNT(EMPNO) COUNT
FROM EMP
GROUP BY SUBSTR(TO_CHAR(HIREDATE, 'YYYYMMDD'), 1, 4)
ORDER BY D
;

SELECT DECODE(TO_CHAR(HIREDATE, 'YYYYMM'), NULL, 'YEARY', 'MONTHRY') DIV
, TO_CHAR(HIREDATE, 'YYYY') YEARLY
, TO_CHAR(HIREDATE, 'YYYYMM') MONTHRY
, COUNT(EMPNO) COUNT
FROM EMP
GROUP BY GROUPING SETS (
  (TO_CHAR(HIREDATE, 'YYYY'))
```

```
  ,(TO_CHAR(HIREDATE, 'YYYYMM'))
);
```

아크일때 물리 모델의 2가지 상황

아크 : 한 컬럼의 값이 있으면 한 컬럼에는 값이 없는 상호배타적인 두 컬럼 (티켓 온라인권 & 티켓 실권)

```
--- 외래키 분리방법

-- 온라인권
-- # 온라인권번호
-- * 발행일시
-- * 유효기간

-- 실권
-- # 실권번호
-- * 발행일시

-- 상품권 결제
-- # 온라인권번호(FK)
-- # 실권번호(FK)
-- * 결제금액
-- * 결제일자

SELECT a.주문번호, a.결제일자, a.결제금액
, nvl(b.온라인권번호, c.실권번호) 상품권번호
, nvl(b.발행일시, c.발행일시) 발행일시
FROM 상품권결제 a, 온라인권 b, 실권 c
WHERE a.결제일자 between :date1 and :date2
AND b.온라인권번호(+) = a.온라인권번호
AND c.실권번호(+) = a.실권번호

-- 외래키 통합 방법

-- 온라인권
-- # 온라인권번호
-- * 발행일시
-- * 유효기간

-- 실권
-- # 실권번호
-- * 발행일시

-- 상품권 결제
-- # 상품권번호(FK)
-- * 상품권구분코드
-- * 결제금액
-- * 결제일자

-- 상품권 결제 인덱스가 [결제일자 + 상품권구분] 일 경우
```

```
-- 수직적탐색 효율화
-- 상품권번호를 보면 온라인/실권 중 어느 테이블과 조인 가능한지 알 수 있다.

SELECT a.주문번호, a.결제일자, a.결제금액
, nvl(b.온라인권번호, c.실권번호) 상품권번호
, nvl(b.발행일시, c.발행일시) 발행일시
FROM 상품권결제 a, 온라인권 b, 실권 c
WHERE a.결제일자 between :date1 and :date2
-- 상품권 구분코드가 1이 아니면  null로 치환해 조인하지 않는다.
AND b.온라인권번호(+) = decode(a.상품권구분, '1', a.상품권번호)
-- 상품권 구분코드가 2이 아니면  null로 치환해 조인하지 않는다.
AND c.실권번호(+) = decode(a.상품권구분, '2', a.상품권번호)
-- 이를 통해 조인 불가능한 테이블과의 조인을 막아 수직적 탐색을 방지한다.


-- 상품권 결제 인덱스가 [상품권구분 + 결제일자] 일 경우
SELECT a.주문번호, a.결제일자, a.결제금액, b.발행일시
FROM 상품권결제 a, 온라인권 b
WHERE a.상품권구분 = '1'
and a.결제일자 Between :date1 and date2
and B.온라인권번호 = a.상품번호
UNION ALL
SELECT a.주문번호, a.결제일자, a.결제금액, b.발행일시
FROM 상품권결제 a, 실권 b
WHERE a.상품권구분 = '2'
and a.결제일자 Between :date1 and date2
and B.온라인권번호 = a.상품번호
```

게시판 출력

TOP-N 쿼리

```
-- 500만 건의 데이터가 있는  T_ORDER51 테이블이 있다.
-- 정렬전 데이터를 위에서 부터 10개 출력한다.
SELECT CODE , PRODID, CUSTID, ORDERDT
FROM T_ORDER51
WHERE ROWNUM <= 10
ORDER BY ORDER_DT

-- 정렬된 데이터를 위에서 부터 10개 출력한다.
SELECT CODE , PRODID, CUSTID, ORDERDT
FROM (
  SELECT CODE , PRODID, CUSTID, ORDERDT
  FROM T_ORDER51
  ORDER BY ORDER_DT
)
WHERE ROWNUM <= 10

--- OPERATION
-- SELECT STATEMENT
```

```
--  COUNT STOPKEY
--    VIEW
--      SORT ORDER BY STOPKEY
--        COUNT
--          TABLE ACCESS FULL OF T_ORDER51
```

SQL 실행순서

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

TOP-N 쿼리(STOP KEY) 의 부하 경감 원리

ROWNUL <= 10 이면 결과 레코드는 항상 10개이다. 처음 읽은 10개을 레코드를 정렬된 상태로 두고 이후 레코들에 대해서는 맨 우측에 있는 값보다 작을때만 배열 내에서 재 정렬한다. N 개의 레코드만 사용하기 때문에 대부분의 경우 IN-MEMORY SORT 가 가능하다.

```
-- PSEUDO 컬럼 rownum 을 사용하지 않고 대체한 값을 이용한다면 top_n 쿼리가 동작하지 않는다.
SELECT CODE , PRODID, CUSTID, ORDERDT
FROM (
  SELECT X.*, ROWNUM R_NUM
  FROM (
      SELECT CODE , PRODID, CUSTID, ORDERDT
      FROM T_ORDER51
      ORDER BY ORDER_DT
  )
)
WHERE R_NUM <= 10

-- stopkey 미발행
--- OPERATION
-- SELECT STATEMENT
--    VIEW
--      SORT ORDER BY
--        COUNT
--          TABLE ACCESS FULL OF T_ORDER51
```

윈도우 함수 Rank() 및 row_number() 사용시 TOP-N 알고리즘이 작동한다. MAX() 함수는 작동하지 않는다.

수동 PGA 메모리 관리 방식 변경 시 주의사항

병렬 프로세스를 8개 띄우면 16개의 서버 프로세스가 발생한다. 이때 서버당 2GB의 PGA를 갖도록 설정하면 16GB의 메모리를 차지하게 된다.

```
alter session set workarea_size_policy = manual;
alter session set sort_area_size = 2GB

select /*+ full(a) parallel(a *)*/ *
from a
order by 배송일자;
```

게시판 QUERY

```
SELECT *
FROM (
  SELECT ... ,ROWNUM RNUM
  FROM (
    SELECT ...
    FROM T_BBM60 A
    ORDER BY REG_DTM DESC -- 전체 데이터 order by
  )
  WHERE ROWNUM <= 20 -- stopkey 발생
)
WHERE RNUM >= 11; -- 범위 추리기
```

최종 출력 건에 대해서만 조인하기

```
-- 일반적인 게시판 출력 조인 방식
select no, 주문번호, 주문일자, 주문금액, 주문고객명, 판매부서명, 판매사원명
from (
  select rownum no, 주문번호, 주문일자, 주문금액, 주문고객명, 판매부서명, 판매사원명
  from (
    select o.주문번호, o.주문일자, o.주문금액
        , c.고객명 주문고객명
        , d.부서명 판매부서명
        , e.사원명 판매사원명
    from 주문 o, 고객 c, 사원 e,  부서 d
    where o.판매부서번호 = 'D004'
      and c.고객번호 = o.주문고객번호
      and e.사원번호 = o.판매사원번호
      and d.부서번호 = o 판매부서번호
    order by o.주문일자, o.판매사원번호
)
  where rownum <= 41
)
where no between 31 and 40
RLDJQorder by no;

--- OPERATION

SELECT STATEMENT
  SORT ORDER BY
```

```
    VIEW
      COUNT STOPKEY
        VIEW
          SORT ORDER BY STOPKEY
            NESTED LOOPS
              NESTED LOOPS
                NESTED LOPS
                  TABLE ACCESS BY INDEX ROWID OF 주문
                    INDEX RANGE SCAN OF IX_주문_01
                  TABLE ACCESS BY INDEX ROWID OF 고객
                    INDEX UNIQUES SCAN OF IX_고객_01
                TABLE ACCESS BY INDEX ROWID OF 사원
                  INDEX UNIQUE SCAN OF PK_사원
              INDEX UNIQUE SCAN OF PK_부서
            TABLE ACCESS BY INDEX ROWID 부서
```

아래의 두 SQL의 일량은 동일하다

```sql
-- ACCESS TABLE BY USER ROWID
select 주문번호, 주문고객번호, 판매사원번호, 판매부서번호, 주문금액
from (
  select rowid r_id
  from 주문 n
  Where 판매부서번호 = 'D004'
)X , 주문 O
where O.rowid = x.r_id;

-- ACCESS TABLE BY INDEX ROWID
select 주문번호, 주문고객번호, 판매사원번호, 판매부서번호, 주문금액
from 주문
where 판매부서번호 = 'D004'
```

인덱스 구성에 따라 TABLE RANDOM ACCESS 수가 달라진다.

```sql
-- IDX : 판매부서번호 + 주문일자
from
(
select r_id, no
from (
  -- 정렬 조건 컬럼인 판매사원번호가 인덱스에 존재하지 않는다.
  -- 소트연산과 테이블랜덤엑세스가 발생한다.
  select rowid r_id, rownum no
  from 주문 n
  Where 판매부서번호 = 'D004'
  order by 판매사원번호, 주문일자
)
WHERE ROWNUM <= 41
)X , 주문 O
WHERE X.NO BETWEEEN 31 AND 40
```

```
AND O.ROWID = X.R_ID
```

인덱스 최후행에 판매사원번호를 추가하면 (IDX : 판매부서번호 + 주문일자 + 판매사원번호) 정렬을 위한 테이블랜덤엑세스가 발생하지 않는다.

징검다리 테이블 조인을 이용한 튜닝

```
-- T_CUST51 : PK(CUST_NO)
-- T_ORDER51 : PK(ORDER_NO), (CUST_NO)
-- T_ORDER_PROD51
-- PK (ORDER_NO, PROD_NO)
-- (ORDER_NO + ONDER_QTY + PROD_DEL_DT)

SELECT /*+ORDERED USE_NL(O) USE_NL(OP) INDEX(OP)*/
FROM T_CUST51 C, T_ORDER51 O, T_ORDER_PROD51 OP
WHERE C.CUST_NO = 'C000100'
AND O.CUST_NO = C.CUST_NO
AND OP.ORDER_NO = O.ORDER_NO
AND OP.PROD_DEL_DT = '20200809'
AND OP.ORDER_QTY >= 50
AND OP.DEL_TYPE = '2'

-- SELECT STATEMENT
--  NESTED LOOPS
--    NESTED LOOPS -- 1만번의 수직적 탐색
--      NESTED LOOPS  - 1번의 수직적 탐색
--        TABLE ACCESS BY INDEX ROWID OF T_CUST51 -- A_ROW : 1
--          INDEX UNIQUE SCAN OF PK_T_CUST51 -- A_ROW : 1
--        TABLE ACCESS BY INDEX ROWID OF T_ORDER51 -- A_ROW : 10000
--          INDEX UNIQUE SCAN OF IX_T_ORDER51 -- A_ROW : 10000
--     INDEX RANGE SCAN OF IX_T_ORDER_PROD51_01 -- 27개의 만족하는 인덱스
--  TABLE ACCESS BY INDEX ROWID OF T_ORDER_PROD51 -- 13개의 만족하는 테이블(랜덤
엑세스 비효율)
```

부담이 되는 구간은 1만번의 테이블 랜덤엑세스와 1만번의 수직적 탐색 인데 이 부담을 줄이기 위해 T_ORDER51 에 대한 엑세스 필터조건 이나 테이블 필터조건이 없다.

```
-- T_CUST51 : PK(CUST_NO)
-- T_ORDER51 : PK(ORDER_NO), (CUST_NO + ORDER_NO)
-- T_ORDER_PROD51
-- PK (ORDER_NO, PROD_NO)
-- (PROD_DEL_DT + DEL_TYPE +  ONDER_QTY  + ORDER_NO )


SELECT /*+
  NO_QUERY_TRANSFORMATION
  LEADING(C O_BRG OP_BRG O OP) USE_NO(O_BRG) USE_HASH(OP_BRG)
```

```
SWAP_NOIN_INPUTS(OP_BRG)
*/ ...
FROM T_CUST51 C, T_ORDER51 O_BRG, T_ORDER_PROD51 OP_BRG, T_ORDER51 O,
T_ORDER_PROD51 OP

WHERE C.CUST_NO = 'C000100'
AND O_BRG.CUST_NO = C.CUST_NO
AND OP_BRG.ORDER_NO = O_BRG.ORDER_NO
AND OP_BRG.PROD_DEL_DT = '20200809'
AND OP_BRG.ORDER_QTY >= 50
AND OP_BRG.DEL_TYPE = '2'
AND O.ROWID = O_BRG.ROWID
AND OP.ROWID = OP_BRG.ROWID;

-- SELECT STATEMENT
--  NESTED LOOPS
--    NESTED LOOPS
--      HASH JOIN
--        INDEX RANGE SCAN OF IX_T_ORDER_PROD51_02
--        NESTED LOOP
--          TABLE ACCESS BY INDEX ROWID OF T_CUST51
--            INDEX UNIQUE SCAN PK__T_CUST51
--          INDEX RANGE SCAN IX_T_ORDER51
--      TABLE ACCESS BY USER_ROWID OF T_ORDER51
--   TABLE ACCESS BY USER_ROWID OF T_ORDER_PRO51
```

이력관리의 방법

점이력

변경일자만 있는 경우

```
-- 특정상품의 마지막 변경이력
SELECT *
(
  SELECT /*+INDEX_DESC(X PK_상품변경이력_59)*/
  ... , ROW_NUMBER() OVER(PARTITION BY 상품ID ORDER BY 변경일자 DESC 순번 DESC)
R_NUM
  FROM 상품변경이력_59 X
  WHERE 상품ID = '1'
)
WHERE R_NUM = 1;
```

```
-- 특정 상품(상품id 1)의 최종건 찾기
DROP TABLE SANGPUM_UPDATE_RECORD_59;
CREATE TABLE SANGPUM_UPDATE_RECORD_59
       (
        SANGPUMID VARCHAR2(26),
        UPDATEDT VARCHAR2(26),
```

```
        NUMNO NUMBER,
        STATECODE VARCHAR2(23),
        SANGPUMPRICE NUMBER
        );

INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161208', 1, '5',
132298);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161208', 2, '8',
709709);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161209', 1, '9',
957490);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161209', 2, '3',
872877);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161209', 3, '2',
437101);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('2', '20161208', 1, '8',
709709);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('2', '20161209', 1, '9',
957490);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('2', '20161209', 2, '3',
797000);


SELECT *
FROM (
  SELECT SANGPUMID, UPDATEDT, NUMNO, STATECODE, SANGPUMPRICE
  ,ROW_NUMBER() OVER (PARTITION BY SANGPUMID ORDER BY UPDATEDT DESC, NUMNO
DESC) R_NUM
  FROM SANGPUM_UPDATE_RECORD_59
  WHERE SANGPUMID = '1'
)
WHERE R_NUM = 1;

TRUNCATE TABLE SANGPUM_UPDATE_RECORD_59;

INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20170315', 5, '1',
781451);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20170315', 8, '1',
281917);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20170316', 5, '1',
244924);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20170317', 4, '1',
869003);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20170317', 5, '1',
206305);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161208', 4, '2',
709708);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161208', 7, '2',
928181);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161209', 4, '2',
88635);
INSERT INTO SANGPUM_UPDATE_RECORD_59 VALUES ('1', '20161209', 5, '2',
501122);
```

```
SELECT SANGPUMID, STATECODE, UPDATEDT,  NUMNO, SANGPUMPRICE
FROM SANGPUM_UPDATE_RECORD_59;

SELECT *
FROM (
  SELECT SANGPUMID, UPDATEDT, NUMNO, STATECODE, SANGPUMPRICE
  ,MAX(SANGPUMPRICE) OVER (PARTITION BY SANGPUMID, STATECODE)
SANGPUMPRICE_MAX
  ,MIN(SANGPUMPRICE) OVER (PARTITION BY SANGPUMID, STATECODE)
SANGPUMPRICE_MIN
  ,AVG(SANGPUMPRICE) OVER (PARTITION BY SANGPUMID, STATECODE)
SANGPUMPRICE_AVG
  ,ROW_NUMBER() OVER (PARTITION BY SANGPUMID, STATECODE ORDER BY UPDATEDT
DESC, NUMNO DESC) R_NUM
  FROM SANGPUM_UPDATE_RECORD_59
)
WHERE R_NUM = 1;

-- keep 함수 사용
  SELECT SANGPUMID, STATECODE
  ,MAX(SANGPUMPRICE)  SANGPUMPRICE_MAX
  ,MIN(SANGPUMPRICE)  SANGPUMPRICE_MIN
  ,AVG(SANGPUMPRICE)  SANGPUMPRICE_AVG
  ,MAX(UPDATEDT) UPDATEDT
  ,MAX(NUMNO) KEEP(DENSE_RANK LAST ORDER BY UPDATEDT) NUMNO_LAST
  ,MAX(SANGPUMPRICE) KEEP(DENSE_RANK LAST ORDER BY UPDATEDT, NUMNO)
SANGPUMPRICE_LAST
  FROM SANGPUM_UPDATE_RECORD_59
  GROUP BY SANGPUMID, STATECODE;
```

누적이력을 구하는 쿼리에서 누적합은 결과레코드가 작은 테이블의 컬럼을 입력값으로 한다.

선분이력 끊기에서 나의 종료 일자는 기존 시작일자보다 커야하고 나의 시작일 자는 기존 종료일자보다 커드야한다.

최종데이터만 가지고 조인

게시판 쿼리

인덱스만 가지고 ROWID랑 취한다음 rowid로 테이블 액세스를 한다. 인덱스를 최대한 활용해 ROwid를 가져오는 쿼리에서 테이블 랜덤엑세스가 최소한으로 일어나도록 한다.

점이력 : 윈도우 함수로 rank를 구하고 Rowid = 1 조건으로 구한다.

선분 이력

징검다리 조인은 브릿지테이블을 통해 인덱스로 최소범위를 구하고 원 테이블과 다시 조인한다.

조인 관계가 아크일때 구분코드를 통해 상호배타적인 테이블고 조인하지 못하도록 하고 각각의 조인 결과를 구하고 union all 한다.

고급 SQL 활용

CASE문 활용

```
SELECT 고객번호, 납입월
    , SUM(DECODE(납입방법코드, 'A', 납입금액)) 지로
    , SUM(DECODE(납입방법코드, 'B', 납입금액)) 자동이체
    , SUM(DECODE(납입방법코드, 'C', 납입금액)) 신용카드
    , SUM(DECODE(납입방법코드, 'D', 납입금액)) 휴대폰
    , SUM(DECODE(납입방법코드, 'E', 납입금액)) 인터넷
FROM 월별납입방법
WHERE 납입월 = '200903'
GROUP BY 고객번호, 납입월
```

UNION ALL

FULL OUTER JOIN 을 대체할 용도로 UNION ALL 활용가능

FULL OUTER JOIN 하면 다수의 조인이 발생함(카티션 곱)

한 번만 읽고 처리할 수 있도록 UNION ALL 사용

```
select
부서번호, 년월, nvl(max(계획금액), 0) 계획금액, nvl(max(실적금액), 0) 실적금액
from (
  select 부서번호, 년월, 계획금액, to_number(null) 실적금액
  from 계획
  where 부서 = '10'
  and 년월 between '202001' and '202007'

  union all

  select 부서번호, 년월, to_number(null) 계획금액,  실적금액
  from 계획
  where 부서 = '10'
  and 년월 between '202001' and '202007'
)
group by 부서번호, 년월;
```

WITH 구문 활용

```
with t_dept as (select deptno, dname, loc from dept)

select e.empno, ename, d.dname
from emp e, t_dept d
where e.deptno = d.deptno
and deptno in (
  select deptno
  from t_dept
```

```
  where loc = 'NEW YORK'
);
```

같은 테이블을 여러번 읽는 쿼리는 성능상 좋지않은 쿼리다.

- MATERIALIZE 방식 : 내부적으로 임시 테이블을 생성함으로써 반복 재사용 - INLINE 방식 : 참조된 횟수만큼 런타임 시 반복 수행, 코딩량 감소 효과

소트 튜닝

소트와 성능

메모리 소트와 디스크 소트

메모리 소트 : 전체 데이터의 정렬 작업을 메모리 내에서 완료하는 것.

디스크 소트 : 메모리 소트를 넘어 디스크 공간까지 사용

소트를 발생시키는 오퍼레이션

SORT AGGREGATE

전체 로우를 대상으로 집계를 수행할 때 발생

진짜 소트가 발생하지는 않음

```
SELECT MAX(SAL), MIN(SAL), SUM(SAL) FROM SCOTT.EMP;

-- SELECT STATEMENT
--  SORT AGGREGATE
--    TABLE ACCESS FULL OF EMP

-- STATISTICS
-- 0 SORTS (MEMORY)
-- 0 SORTS (DISK)
```

SORT ORDER BY

ORDER BY 절 사용

```
SELECT * FROM EMP ORDER BY SAL;;

-- SELECT STATEMENT
--  SORT ORDER BY
--    TABLE ACCESS FULL OF EMP

-- STATISTICS
```

```
-- 1 SORTS (MEMORY)
-- 0 SORTS (DISK)
```

SORT GROUP BY

SORTING 알고리즘을 사용해 그룹별 집계를 수행할때 발생

```
SELECT DEPTNO, SUM(SAL), MAX(SAL)
FROM SCOTT.EMP
GROUP BY DEPTNO
ORDER BY DEPTNO;

-- SELECT STATEMENT
--  SORT GROUP BY
--    TABLE ACCESS FULL OF EMP

-- STATISTICS
-- 1 SORTS (MEMORY)
-- 0 SORTS (DISK)
```

HASH GROUP BY 방식 도입 HASH GROUP BY는 값이 정렬되지 않기 때문에 반드시 추가 정렬 필요

SORT UNIQUE

UNNESTING 된 서브쿼리가 M쪽 집합이거나, 인덱스가 없거나 세미 조인으로 수행되지 않는다면 메인쿼리와 조인되기 전에 중복 레코드 제거를 위히 sort unique 오퍼레이션 먼저 수행

```
select *
from scott.dept
where deptno in
(
  select /*+unnest*/ deptno from scott.emp
  where sal >= 3000
)

-- SELECT STATEMENT
--  MERGE JOIN SEMI
--    TABLE ACCESS BY INDEX ROWID OF DEPT
--      INDEX FULL SCAN OF PK_DPET
--    SORT UNIQUE
--      TABLE ACCESS FULL OF EMP

-- STATISTICS
-- 1 SORTS (MEMORY)
-- 0 SORTS (DISK)
```

SORT JOIN

소트머지 조인을 수행할 때 발생

```
SELECT /*+ORDERED USE_MERGE(E)*/
*
FROM SCOTT.DPET D, SCOTT.EMP E
WHERE D.DEPTNO = E.DEPTNO

-- SELECT STATEMENT
--  MERGE JOIN
--    TABLE ACCESS BY INDEX ROWID OF DEPT
--      INDEX FULL SCAN OF PK_DPET
--    SORT UNIQUE
--      TABLE ACCESS FULL OF EMP

-- STATISTICS
-- 1 SORTS (MEMORY)
-- 0 SORTS (DISK)
```

WINDOW SORT

윈도우 함수를 사용하면 정렬을 수행한다.

```
select EMPNO, ENAME, COUNT(*) OVER (PARTITION BY DEPTNO) CNT
from scott.EMP

-- SELECT STATEMENT
--  WINDOW SORT
--    TABLE ACCESS FULL OF EMP

-- STATISTICS
-- 1 SORTS (MEMORY)
-- 0 SORTS (DISK)
```

소트가 발생하지 않도록 SQL 작성

UNION을 UNION ALL 로 대체

DISTICT 를 EXISTS로 대체

```
SELECT DISTINCT M.M_CODE, M.M_NM
FROM T_ORDER O, T_PRODUCT P, T_MANUF M
WHERE O.ORDER_DT >= '20090101'
AND O.PROD_ID = P.PROD_ID
```

```
AND M.M_CODE = P.M_CODE

-- SELECT STATEMENT
--  HASH UNIQUE
--    NESTED LOOPS
--      NESTED LOOPS
--        NESTED LOOPS
--          VIEW
--            HASH UNIQUE
--              INDEX FAST FULL SCAN OF IX_ORDER53_01
--            TABLE ACCESS BY INDEX ROWID OF T_PRODUCT53
--              INDEX UNIQUE SCAN OF PK_T_PRODUCT53
--          INDEX UNIQUE SCAN OF PK_T_MANUF53
--      TABLE ACCESS BY INDEX ROWID OF T_MANUUF53

SELECT M.M_CODE, M.M_NM
FROM T_MANUF M
WHERE EXISTS
  (
    SELECT 1
      FROM T_PRODUCT P
      WHERE M_CODE = M.M_CODE
      AND EXISTS
        (
          SELECT 1
          FROM T_ORDER
          WHERE O.ORDER_DT >= '20090101'
          AND PROD_ID = P.PROD_ID
        )
  )

-- SELECT STATEMENT
--  NESTED LOOPS SEMI
--     TABLE ACCESS FULL OF T_MANUF53
--     VIEW PUSHED PREDICATE
--      NESTED LOOPS SEMI
--        INDEX RANGE SCAN OF IX_PRODUCT53_01
--        INDEX RANGE SCAN OF IX_ORDER53_01
```

NESTED LOOPS SEMI 는 조인을 한 번 성공하면 다음 조인은 진행하지 않는다.

불필요한 COUNT 연산 제거

member 테이블에 해상 조건의 데이터가 존재하는지 여부를 점검하기 위해 전체를 count

1건 이상 존재한다면, 더 이상 읽지 않도록 Rownum <= 1을 준다.

```
declare
  v_cnt number
begin
  select count(*) into v_cnt
  from (
```

```
    select 고객번호
    from 고객
    where 고객등급 = 'vip'
    -- 1건만 읽어오도록 한다.
      and rownum <= 1
  ):
```

인덱스를 이용한 소트연산 대체

SORT ORDER BY

인덱스에 정렬 기준 컬럼을 추가하면 별도의 sort operation이 발생하지 않는다.

SORT AREA를 적게 사용하도록 SQL 작성

소트를 완료하고 데이터 가공하기

```
select  lpad(카드번호) || lpad(고객번호, 30)
from (
  select 카드번호, 고객번호
  from 카드
  where 발급일자 between :1 and :2
  order by 배송일자;
)
```

분석함수에서의 TOP-N 쿼리

- window sort 시에도 rank() 및 row_number() 사용 시 top-n 알고리즘 작동 - max() 등의 함수보다 부하경감

옵티마이저

오브젝트 통계정보를 이용해 실행계획의 예상비용을 산정한다.

종류

- 규칙기반 옵티마이저

휴리스틱 옵티마이저

미리 정해 놓은 규칙에 따라 액세스 경로 평가 및 실행계획 선택

경로 별 우선순위로 인덱스 구조, 연산자, 조건절 형태가 순위를 결정짓는 원인

- 비용기반 옵티마이저

쿼리를 수행하는데 소요되는 일량 또는 시간을 비용으러 산정 비용은 테이블, 인덱스에 대한 통계정보 기초하여 산정

최적화 과정

파서가 syntax, semantic 체크

파싱된 sql을 표준형태로 변환

실행계획 별로 실제 실행할 수 있는 코드 형태로 포멧팅

최적화 목표

전체 처리속도 최적화

- 최종 결과집합을 끝까지 읽는 것으로 전제

최초 응답속도 최적화

- 전체 결과집합 중 일부만 읽다가 멈추는 것을 전잰

```
alter system set optimizer_mode = all_rows;
alter system set optimizer_mode = first_rows;

select /*+all_rows*/ * from t where ... ;
```

옵티마이저 행동에 영향을 미치는 요소

MV : 실제 데이터를 갖고 있는 view(일반적으로 sql만 가지고 있음)

옵티마이저의 한계

바인드 변수 사용 시 균등분포 가정

한 데이터가 분포의 큰 부분을 차지하더라도 균등한 데이터로 가정

통계정보를 이용한 비용계산 원리

선택도

1 / DISTINCT VALUE 수

카디널리티

- 특정 엑세스 단계를 거치고 난 후 출력될 것으로 예상되는 결과 건순

총 로우 수 * 선택도 = 총 로우 수 * (1/distinct value)

히스토그램

- 도수분포 히스토그램

DISTINCT 가 적을 때 최대 254개 까지 HASH BUCKET 별 개수를 가진다.

- 높이균형 히스토그램

DISTINCT 가 254개를 넘으면 hash 하나 당 몇 개의 개수를 가진다.

인덱스를 경유한 테이블 엑세스 비용

비용 = Blevel(수직적 탐색 비용)

- (리프 브록 수 * 유효 인덱스 선택도) -- 인덱스 수평적 탐색 비용
- (클러스터링 팩터 * 유효 테이블 선택도) -- 테이블 RANDOM 액세스 비용\

옵티마이저 힌트

```
select /*+LEADING(B@STUDY X) USE HASH(X*/
from (
  select /*+full(a)*/ a.*
  , row_number() over (partition by acno order by trd desc) r_num
  from t1_63 a
  where trd >= '20170617'
  and trd < '20171217'
) X
where X.R_NUM = 1
AND EXISTS (
SELECT /*+UNNEST QB_NAME(STUDY) INDEX (B IX_12)*/ 1
FROM T2_63 B
WHERE ACNO = X.ACNO
AND TRD = '20171217'
);
```

힌트가 무시되는 경우

- 의미적으로 안 맞게 기술한 힌트

서브 쿼리에 unnestdhk push_subq를 같이 기술한 경운 Unnest 되지 않은 서브쿼리만이 push_subq 힌트의 적용 대상

힌트 종류

- all_rows - first_rows - full - index - index_desc - index_ffs - index_ss - no_query_transformation - use_concat : or expand를 해라 - no_expand : or expand를 하지마라 - merge , no_merge - use_merge - unnest, un_unnest - ordered, leading - use_nl, use_hash - parallel, pq_distribute - append - push_pred , no_push_pred - push_subq, no_push_subq - qb_name

쿼리변환

옵티마이저가 사용자 쿼리를 표준 쿼리로 변환하는 것

서브쿼리 UNNESTING

서브쿼리를 풀어서 메인쿼리와 동등한 계층에서 조인하도록하는 것

```
SELECT EMPNO, ENAME, SAL, COMM, DEPTNO
FROM EMP
WHERE DEPTNO IN(
  SELECT /*+NO_UNNEST*/ DEPTNO FROM DEPT
);

-- SELECT STATMENT
--   FILTER
--      TABLE ACCESS FULL OF EMP
--        INDEX UNIQUE SCAN PK_DPET
```

FILTER : 메인쿼리를 먼저 풀고 대응되는 서브쿼리에 하나씩 값을 전달한다.

```
SELECT EMPNO, ENAME, SAL, COMM, DEPTNO
FROM SCOTT.EMP
WHERE DEPTNO IN(
  SELECT  DEPTNO FROM SCOTT.DEPT
);

-- unnesting
SELECT EMPNO, ENAME, SAL, COMM, DEPTNO
FROM SCOTT.EMP e,
(SELECT  DEPTNO FROM SCOTT.DEPT) d
where d.deptno = e.deptno;

--view merging
SELECT EMPNO, ENAME, SAL, COMM, DEPTNO
FROM SCOTT.EMP e, SCOTT.DEPT D
WHERE D.DEPTNO = E.DEPTNO;

-- SELECT STATEMENT
--  NESTED LOOPS -- SUBQUERY UNNESTING
--    TABLE ACCESS FULL EMP
--      INDEX UNIQUE SCAN PK_DEPTNO
;
```

- 서브 쿼리를 unnesting 한 결과가 항상 좋은 것은 아니다. - cost를 산정한 후 결정하는 방향으로 발전

- unnest : 서브쿼리를 Unnesing 하여 조인 방식으로 유도 - no_unnest : 원본 쿼리 변경 없이, fuilter 방식으로 유도

서브쿼리가 M 쪽 집합이거나 NON-UNIQUE 인덱스 일때

```
select *
from emp
where deptno in (select deptno from dept)

-- SELECT STATEMENT
```

```
-- NESTED LOOPS
--  NESTED LOOPS
--    SORT UNIQUE
--      INDEX FULL SCAN OF IX_EMP_01
--    INDEX UNIQUE SCAN OF PK_DEPT
--  TABLE ACCESS BY INDEX ROWID
```

위 예제는 SUB QUERY가 1쪽 집합인 경우로 DEPT TABLE의 DEPTNO 가 PK(NOT NULL, UNIQUE) 이다. 결과집합이 emp 테이블 수준으로 출력된다.

옵티마이저의 고민

SUB QUERY 가 M쪽 집합이거나 서브 쿼리에 PK가 없어 1 쪽 집합임을 옵티마이저가 확신하지 못할 경우 UNNESTING 을 시도하면 결과집합이 서브쿼리 M쪽 수준으로 출력된다.

```
select *
from dept
where deptno in(select deptno from emp)

-- subquery unnesting

select * from
(select deptno from emp) a, dept b
where a.deptno = b.deptno
```

옵티마이저의 선택

1쪽 집합임을 확신할 수 없는 테이블이 드라이빙 테이블이 된다면 sort unique 오퍼레이션을 수행하여 1쪽 집합으로 만들고 조인 한다.

```
-- SELECT STATEMENT
--   NESTED LOOPS
--     NESTED LOOPS
--       SORT UNIQUE
--         INDEX FULL SCAN OF IX_EMP_01
--       INDEX UNIQUE SCAN OF PK_DEPT
--   TABLE ACCESS BY INDEX ROWID OF DEPT
```

메인쿼리 쪽 테이블이 드라이빙 된다면 1쪽 집합임을 확신할 수 없는 서브쿼리 테이블에 대해 세미 조인 방식으로 조인한다.

```
-- SELECT STATEMENT
--   NESTED LOOPS
--     TABLE ACCESS FULL
--     INDEX RANGE SCAN OF IX_EMP_01
```

실행계획과 다른 실행순서

- 스칼라 서브쿼리 - 해시조인 계단식 - push_subq 힌트

push_subq

```
SELECT C.고객번호, C.고객명. C.고객선향코드
FROM 고객 C
where C.고객성향코드 = 'C020'
AND EXISTS (
  SELECT /*+*NO_UNNEST PUSH_SUBQ*/ 1
  FROM 주문 O
  WHERE 주문고객번호 = C.고객번호
  AND   주문유형코드 = '010'
  AND   주문일자 IS NULL
)

-- INDEX 고객(고객성향코드), 주문(고객번호 + 주문유형코드 + 주문일자)

-- SELECT STATEMENT
--    TABLE ACCESS BY INDEX ROWID OF 고객 --- 2
--      INDEX RANGE SCAN OF IX_고객_01  --- 1
--      INDEX RANGE SCAN OF IX_주문_01 --- 3

-- INDEX 고객(고객성향코드 + 고객번호), 주문(고객번호 + 주문유형코드 + 주문일자)
-- 인덱스 끼리 미리 비교한다음 범위를 줄여 TABLE ACCESS
-- SELECT STATEMENT
--  TABLE ACCESS BY INDEX ROWID OF 고객  -- 3
--    INDEX RANGE SCAN OF IX_고객  -- 1
--      INDEX RANGE SCAN OF IX_주문  -- 2
```

no_push_subq

```
-- NO_push_subq로 전통적인 filter 방식으로 쿼링
-- 인덱스 : 고객(고객성향코드 + 고객번호), 주문(고객번호 + 주문유형코드 + 주문일자)
SELECT C.고객번호, C.고객명. C.고객선향코드, H.C11
FROM 고객 C
, 고객취미 H
where C.고객성향코드 = 'C020'
AND H.고객번호 = C.고객번호
AND EXISTS (
  SELECT /*+*NO_UNNEST NO_PUSH_SUBQ*/ 1
  FROM 주문
  WHERE 주문고객번호 = C.고객번호
  AND   주문유형코드 = '010'
  AND   주문일자 IS NULL
)

-- SELECT STATEMENT
```

```
--  FILTER
--    NESTED LOOPS  -- 고객취미데이터와 조인하고
--      TABLE ACCESS BY INDEX ROWID OF 고객 -- 고객 데이터 읽고
--        INDEX RANGE SCAN OF IDX_고객_01
--      TABLE ACCESS BY INDEX ROWID OF 고객취미
--        INDEX RANGE SCAN OF IDX_고객취미
--   INDEX RANGE SCAN OF IX_주문_02 -- 서브쿼리에 한 행씩 전달

-- push_subq 방식으로 쿼링
-- 인덱스 : 고객(고객성향코드 + 고객번호), 주문(고객번호 + 주문유형코드 + 주문일자)
SELECT C.고객번호, C.고객명. C.고객선향코드, H.C11
FROM 고객 C
, 고객취미 H
where C.고객성향코드 = 'C020'
AND H.고객번호 = C.고객번호
AND EXISTS (
  SELECT /*+*NO_UNNEST NO_PUSH_SUBQ*/ 1
  FROM 주문
  WHERE 주문고객번호 = C.고객번호
  AND   주문유형코드 = '010'
  AND   주문일자 IS NULL
)

-- SELECT STATEMENT
--    NESTED LOOPS
--      TABLE ACCESS BY INDEX ROWID
--        INDEX RANGE SCAN OF IDX_고객
--          INDEX RANGE SCAN OF IDX_주문
--      TABLE ACCESS BY INDEX ROWID OF 고객취미
--        INDEX RANGE SCAN OF 고객취미
```

서브쿼리는 pinning 을 못써서 Filter 방식보다 그다지 효율적이지 못하다.

뷰 MERGING

```
SELECT *
FROM (SELECT * FROM EMP WHERE JOB = 'SALESMAN') A, (SELECT * FROM DEPT
WHER LOC = 'CHICAGO') B
WHERE A.DEPTNO = B.DEPTNO;

-- 뷰 MERGING
SELECT *
FROM EMP A, DEPT B
WHERE A.JOB = 'SALESMAN'
AND B.LOC = 'CHICAGO'
AND A.DPETNO = B.DEPTNO
```

```
CREATE OR DP
```