2주차 강의 내용 정리

이전 수업 정리

인덱스를 통해서 데이터를 읽는 과정은 수직적 탐색, 수평적 탐색 그리고 테이블 액세스 위주로 진행된다. 수직적 탐색은 블록 랜덤 엑세스를 통해 수평적 탐색의 시작점을 찾는 과정이다.

수평적 탐색은 시퀀셜 엑세스 찾고자 하는 조건을 만족하는 시작점에서 끝점을 찾아가는 과정이다.

랜덤 엑세스는 하나의 블록을 읽어서 조건을 만족하는 하나의 레코드만 얻고 나머지는 버린다.

시퀀셜 엑세스는 한 블록을 읽어서 차례로 모든 데이터를 읽는다.

인덱스 리프 블록은 정렬되어있어 수평적 스캔을 통해 범위내의 레코드를 차곡차곡 읽을 수 있다.

TABLE FULL SCAN 은 정렬 여부에 상관없이 모든 블록의 레코드를 다 읽어온다.

인덱스 스캔의 궁극적 목적은 테이블 레코드의 위치정보가 담긴 Rowid를 찾는 것이다.

수평적 탐색은 조건에 맞는 인덱스의 rowid를 찾는 기법이다.

테이블 렌덤 액세스는 인덱스 스캔의 결과에 해당하는 레코드의 수만큼 발생한다.

인덱스 튜닝의 핵심 원리는 수평적 스캔을 최소화 하여 선택도(스캔범위에서 실제로 랜덤 액세스까지 이어지는 인덱스의 비율)를 높이고 테이블 랜덤 엑세스(데이터 값을 최대한 인덱스 키에 포함하도록 인덱스 생성, 버퍼피닝등 이용) 를 최소화하는 것이다.

검색조건에 해당하는 모든 컬럼이 인덱스에 포함되어 있다면 테이블 랜덤 액세스 비효율이 발생하지않는다. 이는 테이블에서 필터링이 발생하지 않기 때문이다.

수평적 선택의 선택도를 최적화 하는 방법은 인덱스 매칭도를 100%에 가깝게 만드는 것이다.

인덱스 루트노드의 개수는 하위 노드의 블록 개수만큼 존재한다.

루트블록과 브랜치블록의 의 각 레코드에는 키의 시작과 종료점(키의 범위), 하위 블록의 DBA 가 담겨져있다.

리프블록은 키 + RowId를 가지고 있다.

RowId 는 DBA(오브젝트 번호 + 파일 번호 + 블록 주소) + 로우 번호로 구성된다.

인덱스 레인지 스캔을 위해선 조건절에 인덱스의 선두 컬럼이 반드시 존재해야 한다. 수평적 스캔의 시작점이 결정되어야 하기 때문이다.

인덱스는 소량의 데이터일 경우 빠른 속도를 보장한다.

일반적으로 10% ~ 20% 일반적인 손익분기점을 갖는데 이는 인덱스를 이용하면 테이블 렌덤 액세스가 발생하기 때문이다. 단, 클러스터링 팩터에 따라 손익분기점이 달라진다.

수평적 스캔의 선택도를 높이고 테이블 랜덤 액세스를 최소화 시켜야만 인덱스 레인지 스캔의 효율이 상승한다.

'수평적 스캔의 선택도를 높인다'는 것은 필터 조건 없이 모두 driving 조건으로 인덱스 액세스 하는 것이다.

'테이블 랜덤 액세스'를 최소화 하기 위해서는 검색조건의 모든 컬럼이 인덱스에 존재하면 된다. 즉 인덱스에서 필터링이 완료되어 조건에 해당하지 않는 인덱스에 액세스가 발생하지 않도록 하는 것이다.

'인덱스 풀스캔'은 인덱스 선두컬럼은 조건절에 없지만 나머지 인덱스 컬럼들이 조건절에 존재할 때 모든 인덱스의 키를 읽는 것이다.

'인덱스 유니크 스캔'은 유니크 인덱스를 = 조건으로 읽을 때 발생한다.

'인덱스 스킵 스캔'은 인덱스를 구성하는 선두 컬럼에 대한 조건은 없지만 선두 컬럼의 DISTINCT가 매우 낮다면 옵티마이저는 '인덱스 풀스캔' 대신 '인덱스 스킵 스캔'을 선택한다. 이는 선두컬럼을 Between 이나 like 로 조회했을때도 사용 가능하다.

'인덱스 패스트 풀 스캔'은 '테이블 랜덤 엑세스' 가 필요없는 상황에서 세그먼트 단위로 블록을 읽어와 인덱스를 스캔한다. (MULTI BLOCK I/O) 단, 테이블 랜덤 엑세스(Select list에 인덱스에 없는 컬럼이 포함된 경우 등)을 해야하는 경우 '인덱스 풀 스캔'을 사용한다.

'인덱스 레인지 스캔 DESENDING'은 인덱스 리프노드를 거꾸로 읽어 올라간다.

스캔 범위 내의 리프노드를 통해 테이블을 읽어 나갈때, 다음번 READ 에서 현재 읽은 동일 block을 리드할 경우 대상 현재 읽은 Block이 Age-out 되지 않도록 pin 을 걸어 해당 주소가 가리키는 메모리 번지수를 PGA에 저장하여 바로 찾아가는 기법을 '버퍼 Pinning' 이라고 한다. 리프노드의 키 순서대로 테이블 블록의 레코드가 모여있다면 버퍼피닝의 효과가 극대화 된다. 리프노드의 키 순서대로 테이블 블록의 레코드가 모여있는 정도를 수치로 표현한 것이 클러스터링 팩터이다. 클러스터링 팩터는 가장 많이 쓰는 인덱스에 맞게 최적화 되도록 테이블을 리빌드하면 큰 효과가 있다.

클러스터링 팩터를 높이기 위해 클러스터드 인덱스가 도입됐다. 특정 유니크 칼럼으로 인덱스를 구성하고 칼럽의 각각의 데이터에 해당하는 데이터는 같은 블록에 저장한다.

오라클의 IOT는 리프노드에 필요한 데이터를 모두 인덱스 수직 + 스평 스캔만으로 데이터를 찾아온다. 유니크 칼럼으로 만들어진 인덱스가 아닌 세컨더리 인덱스로 조회하면 성능이 현저히 떨어진다.

IOT는 기존 값들이 모두 정렬되어있어 새로운 값이 추가되면 인덱스 스플릿이 발생해서 저장되는 블록의 주소가 바뀐다. 이로 인해 rowid 가 바뀐다.

rowid가 휘발성이므로 인덱스 리프노드에 Table RowId를 보관하지 않고 key + 유니크필드의 값을 가지고 있다. 그렇기 때문에 세컨더리 인덱스로 조회 시 수직적 탐색이 발생한다.

조건절의 컬럼이 모두 인덱스 컬럼이면 랜덤엑세스 비효율은 없다. 테이블에서 필터링이 발생하면 랜덤액세스 비효율이 발생한다.

수평적 스캔의 선택도를 높이는 방법은 인덱스 선두 테이블을 = 또는 in 조건으로 조회하는 것이다.

Access 조건은 인덱스 스캔과정에서 필요하다. Access 조건은 인덱스 스캔의 범위를 줄이는데 기여한다. Check 조건은 스캔 범위에 속한 인덱스에 대해 랜덤 엑세스를 진행할지 결정하는 역할을 한다.

테이블에서 필터 조건으로 필터링 했다는 것은 테이블 랜덤 액세스 비효율이 발생했음을 의미한다.

결합 인덱스 우선순위

최대한 많이 사용되는 SQL의 컬럼에 대한 인덱스를 만드는 것이 좋다.

인덱스에서 읽어온 레코드의 수보다 테이블 랜덤 액세스한 레코드가 적으면 랜덤액세스 비효율이 있다.

IN-LIST ITERATOR

IN 의 개수만큼 인덱스 수직적 탐색이 발생한다. 수평적 스캔의 비효율 보다 수직적 탐색에 대한 비효율이 더 클 수 있다. 특히 인덱스의 높이가 높을때 비효율 증가한다.

```
-- WHERE C1 = 1 AND C2 BETWEEN 1 AND 3 AND C3 = 'A'
-- 수직적 탐색 1 + 수평적 탐색 27

-- WHERE C1 = 1 AND C2 IN (1, 2, 3) AND C3 = 'A'
-- 수직적 탐색 3 + 수평적 탐색 3

-- BETWEEN 조건의 변별력이 좋은경우
-- WHERE C1 = 1 AND C2 BETWEEN 1 AND 3 AND C3 = 'A'
-- 수직적 탐색 1 + 수평적 탐색 5
```

드라이빙 조건의 변별력이 좋아 검색 구간을 줄일 수 있다면 IN-LIST ITERATOR로 탐색하는 것 보다 BETWEEN 으로 RANGE 탐색을 하는 것이 유리하다.

인덱스 선행 컬럼에 대한 조건절의 좌변을 가공하면 해당 인덱스는 사용할 수 없다. 인덱스 후행 컬럼을 가공하면 드라이빙 조건으로 사용할 수 있는 조건절이라 할지라도 필터 조건으로 사용한다.

인덱스 스킵 스캔을 이용한 비효율 해소

```
-- 한달에 10만건 의 데이터가 존재하는데 판매구분이 a인 데이터가 1만건, b인 데이터는 9만건
-- 판매 구분은 = 조건, 판매월은 between 조건
-- 인덱스가 판매월 + 판매구분 일때
-- BETWEEN 으로 RANGE 스캔하면 각 판매월 당 9만건 만큼의 블록 리드 비효율이 발생한다.
-- 만약 BETWEEN을 IN 으로 바꾸면 수직적 스캔은 늘어나지만 수평적 스캔이 현저히 감소한다.
-- 만약 INDEX SKIP SCAN 으로 읽으면  INLIST ITERATOR 처럼 탐색을 루트에서 다시 시작하지 않기 때문에 근소하게 유리하다.
```

'INDEX SKIP SCAN'의 조건은 인덱스 선두컬럼에 대한 조건절이 없고 선두컬럼의 distinct 가 낮아야 하는데 선두 컬럼이 있어도 = 조건이 아니면 INDEX SKIP SCAN을 활용할 수 있다.

범위검색 조건을 남용할 때 발생하는 비효율

```
-- 주문일자 조건이 있을 경우
SELECT 상품코드, 주문일자, 주문우형
FROM 주문
WHERE 상품코드 = :PORD_CD
AND 주문일자 = :ORDER_DT -- 선택조건
AND 주문유형 = :ORDER_TYPE

-- 주문일자 조건이 없을 경우
SELECT 상품코드, 주문일자, 주문우형
FROM 주문
```

```
WHERE 상품코드 = :PORD_CD
-- AND 주문일자 = :ORDER_DT -- 선택조건
AND 주문유형 = :ORDER_TYPE
```

주문일자가 조건에 포함될 경우 비효율이 0 이지만 주문일자가 조건에 포함되지 않을 경우 주문유형이 필터조건이 되서 인덱스 스캔 비효율 발생

```
SELECT 상품코드, 주문일자, 주문우형
FROM 주문
WHERE 상품코드 = :PORD_CD
AND 주문일자 LIKE :ORDER_DT || '%'
AND 주문유형 = :ORDER_TYPE -- 선두 컬럼의 Like 조건 때문에 DRIVING 에서 check 조건으로 변경됨
```

주문일자가 조건에 포함되는 좌측의 경우에도 주문유형이 필터조건이되서 인덱스 스캔 비효율 발생

```
-- 지역 조건 참여 여부에 따른 SQL 분기
-- 주문일자 조건이 있을 경우
SELECT 상품코드, 주문일자, 주문우형
FROM 주문
WHERE  :ORDER_DT IS NOT NULL
AND 상품코드 = :PORD_CD
AND 주문일자 = :ORDER_DT -- 선택조건
AND 주문유형 = :ORDER_TYPE
UNION ALL
-- 주문일자 조건이 없을 경우
SELECT 상품코드, 주문일자, 주문우형
FROM 주문
WHERE :ORDER_DT IS NULL
AND 상품코드 = :PORD_CD
AND 주문유형 = :ORDER_TYPE
```

주문일자가 포함될 경우 비효율 없음 주문일자가 포함되지 않을 경우 어쩔 수 없는 비효율

```
-- 주문일자 칼럼이 not null 인 경우 sql 단순화 가능
SELECT 상품코드, 주문일자, 주문우형
FROM 주문
WHERE 상품코드 = :PORD_CD
AND 주문일자 = NVL(:ORDER_DT, 주문일자) -- 바인드변수 :ORDER_DT 값이 NULL 이면 '주문일자 = 주문일자' 비교연산의 결과로 항상 'TRUE'
AND 주문유형 = :ORDER_TYPE
```

단, 주문일자가 NULL이 허용컬럼이면 NULL(주문일자) = NULL(바인드변수 :order_dt)이 되는데 이 비교연산의 결과값은 항상 'FALSE' 이다. 이렇게 되면 주문일자의 값이 정말로 NULL인 필드가 결과에 포함되지 않아 결과값에 문제가 생긴다.

인덱스 레인지 스캔을 유도하면서 하나의 컬럼이 check 조건일때 IN-LIST ITERATOR 을 선택할 것이냐 아니면 체크조건으로 읽도록 할 것이냐는 선두 컬럼의 변별력에 달려있다.

같은 컬럼에 두 개의 범위검색 조건 사용시 주의사항

범위가 적은 조건을 드라이빙조건으로 하고 범위가 넓은 조건을 체크 조건으로 삼는 것이 유리하다.

```
select *
from (
  --- 이전 조회와 도서명이 같은 경우
  select /*+ index(도서 도서명_idx) */
  rowid rid, 도서번호, 도서명, 가격, 저자, 출판사, isbn
  From 도서
  where 도서명 = :last_book_nm
  and rowid > :last_rid
  union all
  --- 이전 조회와 도서명이 큰 경우
  select /*+ index(도서 도서명_idx) */
  rowid rid, 도서번호, 도서명, 가격, 저자, 출판사, isbn
  From 도서
  where 도서명 like :last_book_nm || '%'
  and 도서명 > :last_book_nm
)
where rownum <= 10;
```

Like 로 조회한 경우보다 부다라등호로 조회한 경우의 스캔 범위가 적기 때문에 등호 조건의 컬럼을 Driving 조건으로 삼는 것이 유리하다.

```
select *
from (
  --- 이전 조회와 도서명이 같은 경우
  select /*+ index(도서 도서명_idx) */
  rowid rid, 도서번호, 도서명, 가격, 저자, 출판사, isbn
  From 도서
  where 도서명 = :last_book_nm
  and rowid > :last_rid
  union all
  --- 이전 조회와 도서명이 큰 경우
  select /*+ index(도서 도서명_idx) */
  rowid rid, 도서번호, 도서명, 가격, 저자, 출판사, isbn
  From 도서
  where trim(도서명) like :last_book_nm || '%'
  and 도서명 > :last_book_nm
)
where rownum <= 10;
```

BETWEEN과 LIKE 스캔 범위 비교

RANGE 스캔의 체크조건으로 쓰여야하는 범위조건은 BETWEEN 이 유리하다.

```
-- idx : 판매월(D) + 판매구분코드(c)

select count(*)
from 집계
where 판매월 BETWEEN '201901' AND '201902' -- D
AND 판매구분 = 'A' -- C

select count(*)
from 집계
where 판매월 LIKE '2019%' -- D
AND 판매구분 = 'A' -- C

select count(*)
from 집계
where 판매월 >= '201901'
AND 판매월 < '201903'
AND 판매구분 = 'A'
```

BETWEEN으로 읽는 경우 중간에는 체크조건으로 읽혀야 하지만 찾고자하는 시작점과 끝점은 = 조건과 같다. LIKE로 읽는 경우 시작점과 끝점이 = 조건이 아니다. 엑세스 조건의 정확한 정의는 스캔의 범위를 줄이는데 기여를 했는지 여부이다.

조건이 모두 Driving 조건이라면 between 과 like의 차이가 없다. 단, 인덱스 후행 컬럼에 체크조건이 있다면 Between 조건으로 쓰는게 유리하다. BETWEEN의 시작점과 끝점이 엑세스 조건이 되기때문이다.

선분이력의 인덱스 스캔 효율

```
-- INDEX : 고객번호 + 시작일 + 종료일
SELECT * FROM 주소변경이력
WHERE 고객번호 = 'C101'
AND '20190831' BETWEEEN 시작일 and 종료일

-- 튜닝

SELECT  /*+ INDEX_DESC(a idx_x01)*/
* FROM 주소변경이력
WHERE 고객번호 = 'C101'
AND '20190831' BETWEEEN 시작일 and 종료일
AND ROWNUM <= 1;

-- 실제 실행계획
WHERE 고객번호 = 'C101'
AND 시작일 <= '20190831'
AND 종료일 >= '20190831'
```

선분이력에서 시작일과 종료일 중 어느것을 DRIVING 조건으로 두는 것이 좋을까?

검색일이 최근일이라면 종료일자의 범위가 적고 검색일이 예전일이라면 시작일의 범위가 적다. 그러므로 최근 데이터를 주로 읽을 때 인덱스는 종료일자 + 시작일 과거 데이터를 주로 읽을 때 인덱스는 시작일자 + 종료일자로 구성하는 것이 유리하다.

인덱스 수정이 불가하다면 최근 데이터라면 INDEX_DESC 힌트를 통해 뒤에서 부터 읽는 것이 유리하다.

인덱스 관련 응용 정리

- 인덱스 매칭 - between / in 비교 - skip - 범위 c1 + c2 + c3 인덱스 구성에서 c2가 선택조건일때 C2 가 Null 을 허용하지 않는 컬럼이라면 NVL 함수를 통해 간략히 표현 가능하다. C2 가 NULL 을 허용하는 컬럼이라면 UNION ALL을 사용한다. - 같은 컬럼을 조건으로 사용할 때 드라이빙의 조건은 범위가 좁은 조건을 기준으로 하고 넓은 조건은 좌변을 변경하여 체크조건으로 활용한다. - BEETWEEN vs LIKE - 선분이력 조회

인덱스 종류

B*TREE 인덱스

1. unbalanced index

- B-TREE 인덱스에서는 발생하지 않는다.

2. index skew 현상

대량 삭제 후 발생하는 현상으로 빈 블록은 free-list로 등록되지만 반환하지 않는다. 현상 해결을 위해 index-rebuild 가 필요하다.

3. index sparse

대량의 삭제 작업 후 발생한다 인덱스의 밀도가 낮아지는 형상이다. index-rebuild 가 필요하다.

4. 인덱스 재생성

인덱스 분할에 의한 경합이 현저히 높을 때 혹은 NL 조인에서 반복 액세스되는 인덱스 높이가 증가했을 때 필요하다. NL 조인할 때 먼저 읽히는 테이블을 Driving, 후에 읽히는 테이블을 look-up 테이블이라 한다. Loop-up 테이블은 Driving 테이블의 조건에 해당하는 레코드 수만큼 반복 탐색된다.

대량의 delete 작업을 수행하고 재입력되기까지 기간이 소요될때 혹은 일정한 레코드 수인데 인덱스가 계속 커질 때 또한 인덱스 재생성이 필요하다.

BITMAP INDEX

distinct value 개수가 적을 때 유리한 인덱스 구성방식이다. 적은 용량을 차지하므로 인덱스가 여러개 필요한 대용량 테이블에 유용하다. 다양한 분석관점이 필요한 테이블에서 주로 사용한다.

```
-- b.tree 인덱스로 구성할 경우 조회 조건이 많아도 하나의 인덱스를 사용해야한다.
-- bitmap 인덱스의 경우 조건 별로 가벼운 인덱스를 많이 만들어 사용할 수 있다.

select *
from t
where 성별 = '여'
```

```
and 연령대 = '30대'
and 고객등급 = 'vip'
and 지역 = '서울'
```

OR 연산은 UNION ALL을 유발하므로 많이 쓰이지 않는데 비트맵 연산에는 OR 연산에도 부하가 없다.

함수기반 인덱스

```
-- 주문 수량이 null 인 레코드에 0으로 채워진 인덱스 생성
-- 대소문자를 구분해서 입력 받은 데이터를 대,소문자 구분없이 조회할때 흔히 사용
-- 데이터 입력, 수정시 함수를 적용해야하므로 부하 발생가능성
-- 특히 user-defined 함수일 경우 부하 극심

select * from 주문
where nvl(주문수량, 0) < 100

create index emp_x01 주문  (nvl(주문수량, 0));
```

리버스 인덱스

일련번호, 주문일시 등 컬럼에 인덱스를 만들 경우 입력값이 순차적으로 쌓여 새로운 INSERT 시 인덱스 블록 경합으로 초당 트랜잭션 처리량 감소한다.

이때 입력된 값을 거꾸로 저장하여 데이터를 고르게 분포하도록한다. = 검색만 사용가능하며, 부등호, between, like 등의 범위 검색 조건에서는 사용불가하다.

조인

NESTED LOOP 조인

```
-- 사원 테이블(DRIVING)의 인덱스 [이름]
-- 부서 테이블(LOOK UP)의 인덱스 [부서번호]
-- 조인 컬럼 [부서번호]

SELECT * FROM 사원 A, 부서 B
WHERE A.이름 = '홍길동'
AND B.부서번호 = A.부서번호

--- EXECUTION PLAN

-- SELECT STATEMENT
--  NESTED LOOPS
--    TABEL ACCESS BY INDEX ROWID OF 사원(TABEL)
--      INDEX RANGE SCAN OF I_사원(INDEX)
--    TABEL ACCESS BY INDEX ROWID OF 부서(TABEL)
--      INDEX RANGE SCAN OF I_부서(INDEX)
```

룩업 테이블의 인덱스에 조인 컬럼이 반드시 존재해야 한다. 조인 과정에서 인덱스를 사용해 룩업 테이블을 랜덤 엑세스 한다.

조인을 한 레코드씩 순차적으로 진행한다. 인덱스 스캔으로 인해 대용량 처리 시 치명적인 한계점이 발생한다.

NESTED LOOP 조인은 OLTP 에 적합하다.

PREFETCH

```
SELECT *
FROM EMP E, DEPT D
WHERE D.DOC = E.DOC
AND E.DEPTNO = D.DEPTNO

-- nl 조인시 전통적인 실행 계획
--- EXECUTION PLAN

-- NESTED LOOPS
--  TABLE ACCES OF DEPT
--    INDEX RANGE SCAN OF I_DEPT
--  TABLE ACCES OF EMP
--    INDEX RANGE SCAN OF I_EMP

-- PREFETCH 작동시 실행 계획

-- TABLE ACCES OF EMP
--  NESTED LOOPS
--    TABLE ACCES OF DEPT
--      INDEX RANGE SCAN OF I_DEPT
--    INDEX RANGE SCAN OF I_EMP
```

SORT MERGE 조인

조인 연결컬럼을 기준으로 두 테이블을 PGA에서 정렬한다. 정렬 후 nL 조인과 같은 방식으로 진행한다. PGA 에서 동작하므로 경합이 없고 속도가 빠르다. 조인 컬럼에 인덱스 유무와 상관이 없다.

HASH 조인

두 테이블 중 작은 테이블에 있는 레코드를 읽어서 PGA에 있는 Hash AREA(PGA)에 적재(BUILD INPUT) 반대쪽 큰 집합을 읽어 해시 테이블 탐색(PROBE INPUT) 작은 테이블의 조인연결컬럼의 값을 HASH FUNCTION 에 대입해 나온 값과 큰 테이블의 조인연결컬럼을 HASH FUNCTION에 대입한 값이 일치하면 조인한다. 등차(=) 조인만 가능하다. 조인 연결 컬럼의 조건이 각각 = 조건 그리고 > 조건일때 등차 조건만 해시조인에 참여할 수 있다. NL 조인처럼 랜덤 액세스 조인이 없다. 이미 PGA에 올라온 레코드를 비교하기 때문에 빠른 탐색과정이 가능하다. 한쪽 테이블이 HASH AREA 에 담길 정도로 충분히 작아야하한다 overflow 시 디스크 공간인 TEMPORARY SPACE로 이동한다. BUILD INPUT의 해시 키 컬럼(연결컬럼) 중복이 없어야 한다. 조인컬럼에 대한 인덱스가 없어 NL 조인이 비효율적일 때 유용하다. OUTER (DRIVING) 테이블에서 INNER 테이블(LOOPKUP) 테이블로의 조인 엑세스량이 많아 RANDOM 액세스 부하가 심할때 유용하다. 소트머지조인하기에 두 테이블이 너무 커 정렬에 부하가 예상될 때 유용하다. 또한 배치용 대용량 테이블을 조인할 때 유용하다 CPU 와 메모리 사용률이 크게 증가하기 때문에 수행빈도가 높고, OLTP 환경하에 성능향상을 위한 용도로 사용해선 안된다.

```
SELECT  /*+USE_HASH(D E)*/
D.DEPTNO, D.DNAME, E.EMPNO, E.ENAME
FROM DEPT.D, EMP.E
WHERE D.DEPTNO = E.DEPTNO

-- 0  SELECT STATEMENT
-- 1      HASH JOIN
-- 2        TABALE ACCESS FULL DEPT
-- 3        TABLE ACCESS FULL  EMP
```

관련 힌트

USE_HASH(BUILD INPUT, PROBE INPUT)

HASH 조인 실행 계획 - 등호형 & 계단형

```
SELECT
/*+LEADING(APPR CARD MERCH PROD CUST)
        USE_HASH(CARD) USE_HASH(MERCH) USE_HASH(PROD) USE_HASH(CUST)
        SWAP_JOIN_INPUTS(APPR) -- BUILD INPUT
        NO_SWAP_JOIN_INPUTS(CARD) -- PROBE INPUT
        NO_SWAP_JOIN_INPUTS(MERCH) -- PROBE INPUT
        NO_SWAP_JOIN_INPUTS(PROD) -- PROBE INPUT
        NO_SWAP_JOIN_INPUTS(CUST) -- PROBE INPUT
*/
*
FROM T_CUST CUST, T_PROD PROD, T_CARD CARD, T_MERCH MERCH, T_APPR APPR
WHERE APPR.APPR_DT BETWEEN '20200701' AND '20200702'
AND MERCH.MERCH_NO = APPR.MERCH_NO
AND CARD.CARD_NO = APPR.CARD_NO
AND PROD.PROD_NO = CARD.PROD_NO
AND CUST.CUST_NO = CARD.CUST_NO;

-- 등호식
-- 0 SELECT STATEMENT
-- 1   HASH JOIN (B : APPR - CARD - MERCH - PROD P : CUST)
-- 2    HASH JOIN (B : APPR - CARD - MERCH P : PROD)
-- 3      HASH JOIN (B : APPR - CARD P : MERCH)
-- 4        HASH JOIN (B : APPR P : CARD)
-- 5          INDEX RANGE SCAN OF IX_T_APPR_01
-- 6          TABLE ACCESS FULL OF T_CARD
-- 7        TABLE ACCESS FULL OF T_MERCH
-- 8      TABLE ACCESS FULL OF T_PROD
-- 9    TABLE ACCESS FULL OF T_CUST


SELECT
/*+
```

```
            LEADING(APPR CARD MERCH PROD CUST)
            USE_HASH(CARD) USE_HASH(MERCH) USE_HASH(PROD) USE_HASH(CUST)
            SWAP_JOIN_INPUTS(APPR) --  BUILD INPUT
            SWAP_JOIN_INPUTS(MERCH) -- BUILD INPUT
            SWAP_JOIN_INPUTS(PROD) --  BUILD INPUT
            SWAP_JOIN_INPUTS(CUST) --  BUILD INPUT
            NO_SWAP_JOIN_INPUTS(CARD) -- PROBE
*/
*
FROM T_CUST CUST, T_PROD PROD, T_CARD CARD, T_MERCH MERCH, T_APPR APPR
WHERE APPR.APPR_DT BETWEEN '20200701' AND '20200702'
AND MERCH.MERCH_NO = APPR.MERCH_NO
AND CARD.CARD_NO = APPR.CARD_NO
AND PROD.PROD_NO = CARD.PROD_NO
AND CUST.CUST_NO = CARD.CUST_NO;

-- 계단식
-- 0 SELECT STATEMENT
-- 1  HASH JOIN (B : CUST P : APPR – CARD – MERCH – PROD)
-- 2    TABLE ACCESS FULL OF T_CUST
-- 3    HASH JOIN (B : PROD P : APPR – CARD – MERCH)
-- 4      TABLE ACCESS FULL OF T_PROD
-- 5      HASH JOIN (B : MERCH P : APPR – CARD)
-- 6        TABLE ACCESS FULL OF T_MERCH
-- 7        HASH JOIN (B : APPR P : CARD)
-- 8          INDEX RANGE SCAL OF IX_T_APPR_01
-- 9          TABLE ACCESS FULL OF T_CARD
```

실행계획 상 순서와 실제 순서가 다른 경우

1. select 절의 scarla 서브쿼리
2. HASH 조인 - 계단형 실행계획

테이블 실행 우선 순위 지정 힌트

- leading(a, b, c) - order : 테이블에 선언된 순서대로

조인 방식

- USE_NL(LOOKUP_TABLE) - USE_MERGE (PROBE_INPUT) - USE_HASH (PROBE_INPUT)

조인 순서의 중요성

NL 조인

```
-- index_emp sal
select *
from emp E, dept d
```

```
where e.sal > 1500
and e.jbo = 'manager'
and d.deptno = e.deptno;
```

emp 가 드라이빙 될때는 3번의 조인이 발생하고 dept가 드라이빙 될때는 50번의 조인이 발생한다.

결과적으로 NL 조인은 레코드 수가 적은 쪽에서 큰 쪽으로 조인하는 것이 좋다.

SORT MERGE JOIN

- Dist sort 가 필요한 경우 : 큰 테이블 DRIVING이 유리하다. - PGA SORT AREA 안에 담길 경우 : 작은 테이블 DRIVING이 유리하다

HASH JOIN

HASH AREA에 충분히 담길정도로 적은 테이블이 DRIVING 되야 한다.

```
select *
from T1, T2
WHERE T1.C1 = 'A'
AND T2.C11 = T1.C11
AND T2.C2 = 'A'

-- T1.C1 = 'A' 조건에 속하는 레코드 수 100건
-- T2.C2 = 'A' 조건에 속하는 레코드 순 10000건 인 경우
-- T1이 드라이빙 테이블이 되어야함
-- 그 결과 인덱스 구성은
-- IDX_T1[C1]
-- IDX_T2[C11 + C2]
-- IDX_T2의 인덱스 순서는 상관없지만 JOIN 연결컬럼이 선두에 나오는게 범용적이다.

-- T1.C1 = 'A' 조건에 속하는 레코드 수 10000건
-- T2.C2 = 'A' 조건에 속하는 레코드 순 100건 인 경우
-- T2이 드라이빙 테이블이 되어야함
-- 그 결과 인덱스 구성은
-- IDX_T1[C11 + C1]
-- IDX_T2[C2]
-- IDX_T1의 인덱스 순서는 상관없지만 JOIN 연결컬럼이 선두에 나오는게 범용적이다.
```

OUTER 조인

OUTER NL 조인

```
-- OUTER TABLE : EMP, INNER TABLE : DEPT
SELECT *
FROM EMP E, DEPT D
```

```
WHERE E.SAL > 5000
AND D.DEPTNO(+) = E.DEPTNO

-- ANSI
SELECT *
FROM EMP E LEFT OUTER JON , DEPT D
ON E.DEPTNO = D.DEPTNO

-- EXECUTION PLAN
-- SELECT STATEMENT
--    NESTED LOOPS(OUTER)
--      TABLE ACCESS BY INDEX ROWID OF EMP(TABLE)
--        INDEX RANGE SCAN OF IX_EMP(INDEX)
--      TABLE ACCESS BY INDEX ROWID OF DEPT(TABLE)
--        INDEX RANGE SCAN OF IX_DEPT(INDEX)
```

OUTER TABLE은 항상 DRIVING 테이블이 된다. OUTER JOIN 을 위해 연결 조건절의 INNER TABLE 뒤에 (+) 를 표시한다.

```
-- DEPT TABLE이 INNER TABLE로 지정된 상태(조건절 조인 조건문 (+) 위치 기준) 에서
-- DEPT 테이블이 OUTER(DRIVING) 이 되고 EMP 테이블이 INNER(LOOP_UP) 테이블이 되도록 힌트를 통해 강제 지정해도 변경할 수 없다.
SELECT /*+LEADING(D) USE_NL(E)*/
 *
FROM EMP E, DEPT D
WHERE E.SAL > 5000
AND D.DEPTNO(+) = E.DEPTNO
```

OUTER JOIN 일때 OUTER TABLE의 조건은 Filter(where) 에 그냥 선언해도 되지만 INNER TABLE의 조건은 뒤에 (+)를 붙인다. ANSI 표현식의 경우 ON 절에 INNER TABLEDML 조건을 추가한다.

OUTER SORT MERGE 조인

```
-- EMP TABLE이 OUTER TABLE, DEPT TABLE이 INNER TABLE로 지정된 상태(조건절 조인 조건문 (+) 위치 기준) 에서
-- DEPT 테이블이 DRIVING 이 되고 EMP 테이블이 LOOP_UP 테이블이 되도록 힌트를 통해 강제 지정해도 변경할 수 없다.
-- 그 이유는 OUTER TABLE이 LOOKUP TABLE이 되면 조인이 실패한 레코드를 보여줄 방법이 업기 때문이다.
SELECT /*+LEADING(D) USE_MERGE(E)*/
 *
FROM EMP E, DEPT D
WHERE E.SAL > 5000
AND D.DEPTNO(+) = E.DEPTNO
```

OUTER HASH JOIN

```
-- OUTER TABLE : EMP
-- INNER TALE : DEPT
SELECT /*+LEADING(E) USE_HASH(D)*/
*
FROM EMP E, DEPT D
WHERE D.DEPTNO(+) = E.DEPTNO

-- EXECUTION PLAN
-- SELECT STATEMENT
--    HASH JOIN(OUTER)
--      TABLE ACCESS(FULL) OF EMP(TABLE)
--      TABLE ACCESS(FULL) OF DEPT(TABLE)
```

1. EMP(OUTER)를 hash 테이블로 적재

2. DEPT를 읽어 조인 성공한 건 결과 집합 적제

3. DEPT 해시 테이블에 조인 성공한 건 표기

4. EMP 해시 테이블에서 조인 실패건 결과 집합 적재

RIGHT OUTER HASH 조인

```
-- OUTER TABLE : EMP
-- INNER TALE : DEPT
-- LEADING(D) : DEPT 를 드라이빙 테이블로
-- USE_HASH(E) : EMP를 PROBE 테이블로 지정해서 해쉬조인
-- SWAPT_JOIN_INPUTS(D) : DEPT TABLE을 BUILD INPUT으로
SELECT /*+LEADING(D) USE_HASH(E) SWAP_JOIN_INPUTS(D)*/
*
FROM EMP E, DEPT D
WHERE D.DEPTNO(+) = E.DEPTNO

-- EXECUTION PLAN
-- SELECT STATEMENT
--    HASH JOIN(RIGHT OUTER)
--      TABLE ACCESS(FULL) OF DEPT(TABLE)
--      TABLE ACCESS(FULL) OF EMP(TABLE)
```

1. DEPT(INNER)를 해시 테이블로 빌드 -- Build input
2. EMP(OUTER)를 읽어 해시 테이블 검색
3. Join 성공여부와 상관없이 모두 결과 집합에 포함

- hash join 은 INNER 집합인 DEPT를 build input으로 사용할 수 있다 - 즉, RIGHT OUTER JOIN이 가능하다.

스칼라 서브쿼리

select 절에 나온 서브쿼리는 스칼라 서브쿼리는 맞지마 모든 스칼라 서브쿼리가 select절에만 나오는 것은 아니다.

스칼라 서브쿼리는 한 컬럼에 대응되서 나오는 결과 값을 조회하는 서부쿼리이다.

```
select empno, dname, sal, hiredate
, (select dname from dept where deptno = e.deptno) dname
from emp e
where sal >= 2000
```

스칼라 서브쿼리는 캐싱 기능을 활용하기 위해 사용한다.

위 쿼리의 [e.deptno, dname] 가 캐싱된다.

대량 테이블일경우 조인으로 풀어준다.

```
insert into _
select ...
(),
from t1, t2, t3
where
```

위 쿼리의 결과가 수억건일때 최적화

1. 해시 조인
2. scarlar subquery를 Outer 조인으로 푼다.
3. 사용자 정의함수는 OLPT 환경의 경우 Scalar 서브쿼리로 풀고 배치 테이블일 경우 outer join 으로 푼다.

Bypass_ujvc 힌트

1:M 관계를 join 하면 M level로 나오는데 조인후 결과 즉, M 레벨의 테이블에는 1 LEVEL table의 Unique 키가 중복으로 나타난다.

이때 1 LEVEL의 table을 비 키-보존 테이블, M LEVEL의 테이블을 키-보존 테이블이라고 한다.

키-보존 테이블 테이블은 업데이트를 할 수 있지만 비 키-보존 테일은 업데이트를 할 수 없다.

- cannot modify a column which maps to a non key-preserved table 에러 발생

조인 연결 컬럼으로 GROUP BY 한 후 생기는 뷰에 대한 업데이트는 가능하다.

bypass_ujvc (byoass updatable join view check) : 수정가능 뷰 체크 생략

조인 응용

누적매출 구하기

```
SELECT T1.지점, T1.판매월, T1.매출액, T2.판매월, T2.매출액
FROM 누적매출_예제 T1, 누적매출_예제 T2
WHERE T2.지점 = T1.지점
```

```
AND T2.판매월 <= T1.판매월
ORDER BY T1.지점, T1.판매월, T2.판매월;

-- 지점, 판매월, 매출액은 판매월이 큰쪽 테이블(T1)의 컬럼을 활용
-- 누적매출은 판매월이 작은쪽 테이블(T2)의 컬럼을 활용

SELECT T1.지점, T1.판매월, MIN(T1.매출액) 매출액, SUM(T2.매출액)
FROM 누적매출_예제 T1, 누적매출_예제 T2
WHERE T2.지점 = T1.지점
AND T2.판매월 <= T1.판매월
GROUP BY T1.지점, T1.판매월

-- 윈도우 함수 활용 [ORACLE 8i 부터]
SELECT 지점,판매월,매출액
SUM(매출액) OVER (PARTITION BY 지점 ORDER BY 판매월 ROWS BETWEEN UNBOUNDED
PRECEDING AND CURRENT ROW) 누적액
FROM 누적매출_예제;
```

조인을 사용해 누적매출을 구하는 쿼리는 괄호문제로 자주 나온다.

윈도우함수

```
-- 집계함수
-- SUM, MAX, MIN, AVG, COUNT
-- BETWEEN 사용 시
-- ROWS(RANGE)  BETWEEEN UNBOUNDED PRECEDING  AND CURRENT ROW
--                       10 PRECEDING        AND UNBMOUNED FOLLOWING
--                       CURRENT ROW         AND 10 FOLLOWING

select sum(sal) over () -- 총연봉합계
select sum(sal) over (partition by 지점) -- 지점별 총연봉합계
select row_number() over (partitionby deptno order by sal) -- 부서별 연봉 순위
select sum(sal) over (partition by deptno orber by hiredate rows between
unbounded preceding and current row) 누적합계

select 일련번혼, 측정갑, 부서, last_value(상태코드 ignore null) over (partition by
부서 order by 상태코드 rows betweeen unbounded preceding and current row) 상태코
드
from 장비측정
order by 부서, 일련번호
```

ROW_NUMBER 는 동일 순위를 인정하지 않고 값이 같을경우 row_number 순으로 정렬한다.