Inner join은 옵티마이저가 조인의 순서를 결정해서 최적화!

Outer join은 어느테이블을 읽을지를 정해주어야함!(left join일 경우에는 왼쪽에 있는 테이블이 먼저 읽힘! 반대로 right join일 경우에는 오른쪽에 있는 테이블이 먼저 읽힘!)

6.3.6 테이블 조인

INNER JOIN(일반적으로 부르는 JOIN) : 어느 테이블을 먼저 읽어도 결과가 달라지지 않으므로, 옵티마이저가 조인의 순서를 결정해서 최적화 할 수 있다.

 SELECT \* FROM emp e, sal s WHERE e.emp\_no = s.emp\_np;

 SELECT \* FROM emp e INNER JOIN sal s ON s.emp\_no = e.emp\_np;

 SELECT \* FROM emp e INNER JOIN sal s USING (emp\_no)

 문법은 다르지만, 전부 같은 쿼리이다.

OUTER JOIN : 반드시 OUTER가 되는 테이블을 먼저 읽어야 해서 최적화 할 수 없다. (어느 테이블을 먼저 읽느냐는 조인에서 아주 중요한 요소이다)

 outer table은 반드시 먼저 읽어야 하며, 주도적인 역할을 한다고 해서 driving table이라고 한다. inner table은 driven table이라고도 한다.

 INNER JOIN pseudo - IF처럼 매칭되는것만 조인의 결과로 가져온다.

 FOR ( r1 in TABLE1)

    FOR ( r2 in TABLE2 ){

      IF(r1.c == r2.c) {

  found(r1.\* , r2.\*);

      }else{

          not\_found();

     ...

OUTTER joint pseudo  - INNER JOIN과 달리 IF의 매칭이 안되더라도 OUTER TABLE의 결과를 가져온다.

 FOR ( r1 in TABLE1)

    FOR ( r2 in TABLE2 ){

      IF(r1.c == r2.c) {

  found(r1.\* , r2.\*);

      }else{

          found(r1.\*, NULL)

     ...

\* inner 테이블의 조인의 결과에 전혀 영향을 미치지 않고, outer에 따라 결과가 결정된다. (결과는 최소한 outer 의 갯수보다 같거나 크다)

\* left outer join : SELECT \* FROM employees e LEFT OUTER JOIN salaries s ON s.emp\_np = e.emp\_no;

  (outer : employees)

\* right outer join : SELECT \* FROM salaries s  RIGHT OUTER JOIN employees e ON s.emp\_np = e.emp\_no;

  (outer : employees)

\* 둘은 같은 결과를 낸다. 그래서 혼동을 막기 위해 보통 LEFT OUTER JOIN으로 통일해서 사용하는것이 일반적

\* 유의사항

 - 실행계획은 inner join인지 outer join인지 알려주지 않는다.

 - 최적화 계획에 의해 outer -> inner join으로 실행 될 수 있으므로 '반드시 ON 절에 사용되는 조건을 명시적으로 작성하는 것'이 좋다.

   SELECT  \* FROM employees e LEFT OUTER JOIN salaries s ON s.emp\_no = e.emp\_no WHERE s.salary > 5000

   ===> (위를 실행하면 최적화에 의해 inner join이 되므로 아래처럼 명시적으로 작성)

   SELECT  \* FROM employees e LEFT OUTER JOIN salaries s ON s.emp\_no = e.emp\_no AND s.salary > 5000

\* LEFT OUTER JOIN 이 아닌 쿼리는 검색조건,조인조건을 WHERE / ON 어느곳에 명시해도 결과&성능 차이가 나지 않는다.

카테시안 조인 (FULL JOIN , CROSS JOIN)

- 조건 자체가 없이, 2개의 테이블의 모든 조합을 결과로 가져온다. (MySQL을 응답 불능 상태로 만들 수 있다)

- 한건만 조회하는 여러 쿼리를 한번에 실행하기 위해 사용하기 도 한다

  SELECT \* FROM de WHERE dept\_no = 'd0001';

  SELECT \* FROM emp WHERE emp\_no = '1000001';

  ==>

  SELECT d.\*, e.\* from dep d , emp e WHERE dept\_no = 'd0001' AND emp\_no= 1000001;

  (조인 조건이 없는것을 확인 할 수 있다, 그리고 사용에 주의를 해야한다)

NATURAL JOIN

  SELECT \* FROM emp e NATURAL JOIN sal s;

  emp와, sal테이블에 존재하는 칼럼중에서 이름이 같은 칼럼을 모두 조인 조건으로 사용한다!

  조인 조건을 명시 안해도 되는 편리함이 있지만, 칼럼명이 바뀔 이슈가 없는지 꼭 확인을 해야한다.

  이런 방식이 있다는 것 정도로 알고 , 사용은 안해도 될듯하다.

\*\*\* 조인에서 중요한 2가지 \*\*\*

1. 실행 결과의 정렬 순서

2. INNER, OUTER 조인의 선택

 실제로 가져와야하는 레코드가 같다면 성능의 차이는 거의 없다. 용도가 다르므로 적절한 사용을 하는것이 더 중요하다.

6.4. 실행 계획 분석 시 주의사항

6.4.1 Select\_type

  DERIVED : FROM절에 사용된 서브쿼리로부터 발생한 '임시테이블'을 뜻한다. (만약 디스크에 저장된다면 성능 저하)

  UNCACHEABLE SUBQUERY : FROM절 이외에서 사용되는 서브 쿼리는 가능한 MySQL이 재사용할 수 있게 유도하지만, 없을때 이렇게 노출

  DEPENDENT SUBQUERY : 외부쿼리에서 값을 받아올 경우, 이 쿼리는 먼저 실행되지 못하고 의존적이라 전체 성능 저하. (제거하는 것이 좋다)

6.4.2 Type

 index:인덱스 풀 스캔, ALL: 풀 테이블 스캔

6.4.5 Extra

  쿼리 실행 계획이 좋은 경우

    Distinct

    Using index

    Using index for group by

   쿼리 실행 계획이 좋지 않은 경우

    Using filesort

    Using join buffer

    Using temporary

    등이 나오면 좀 더 최적화 할 수 없는지 검토

   쿼리를 검수해봐야 하는 경우

    Full scan on NULL key

    Unique row not found

   쿼리가 요구사항을 제대로 반영했는지 , 버그가 없는지 확인해야한다.

출처: [http://bistros.tistory.com/entry/6장-실행계획-636-테이블-조인](http://bistros.tistory.com/entry/6%EC%9E%A5-%EC%8B%A4%ED%96%89%EA%B3%84%ED%9A%8D-636-%ED%85%8C%EC%9D%B4%EB%B8%94-%EC%A1%B0%EC%9D%B8) [b]

**\*JOIN 원리**

--DBMS 작동원리 : 평소에 데이터는 하드디스크의 데이터 파일에 저장해 두었다가 필요한 시점에 메모리로 복사한다.

--(오라클은 이때 사용하는 메모리를 데이터베이스 버퍼 캐시라고한다)

-- 여기서 중요한 것은 칼럼 100개를 가진 테이블이 하드디스크에 저장되어 있을 때 사용자가 그 중 1개의 칼럼에 SELECT를

-- 수행할 경우 해당 칼럼 1개만 메모리로 복사하는 것이 아니라 일단 100개의 칼럼 모두를 메모리로 복사해 온다는 점이다.

--그래서 불필요한 I/O도 많고 메모리 사용량도 많이 생기기 때문에 정규화를 철저히 해서 부하를 최대한 줄여야한다.

--일단 메모리로 올라온 테이블에서 필요한 칼럼을 가져와서 사용자가 원하는 결과를 만들어야 한다.

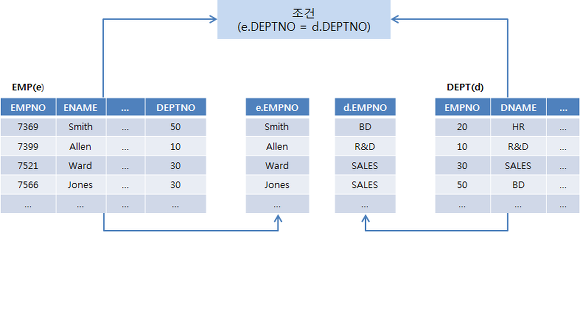


**1) NESTED LOOP JOIN(가장 기본적인 JOIN 기법)**

SELECT E.ENAME, D.DNAME

FROM EMP E, DEPT D

WHERE E.DEPTNO = D.DEPTNO;



**SQL을 수행하면 오라클은 아래와 같은 순서로 작업을 수행한다.**

**--1. 사원 테이블과 부서 테이블을 메모리(데이터베이스 버퍼 캐시)로 복사해온다.**

**--2. 그후에 사원테이블에서 사원이름을 꺼내서 임시 작업공간으로 가져 간다.(인덱스 상황이나 다른 요소에 따라 작업순서가 변경될 수도 있다.)**

**--3. 그 후에 부서 테이블에서 해당부서명을 찾으러 가는데 그때 위 SQL의 3행에 있는 조건을 보고 해당 조건에 맞는 데이터를 찾아서 부서명을 가져온다.**

**--4. 한 행의 작업이 끝나면 다시 처음 테이블로 돌아가서 두 번째 행의 이름을 다시 PGA로 가져온다.**

**--5. 다시 부서 테이블에 가서 사원 테이블에서 가져온 두 번째 행의 부서번호와 동일한 부서번호를 가진 부서명을 꺼내온다.**

앞과 같은 과정을 계속 반복해서 먼저 읽었던 사원테이블의 데이터가 끝이 날 때까지 작업이 반복(LOOP)됩니다.

그래서 이 JOIN을 **NESTED LOOP JOIN**이라고도 하며 모든 JOIN의 기본이 되는 JOIN이다.

또한 JOIN을 수행하는 횟수는 먼저 읽는 테이블의 행 수 만큼 JOIN이 수행된다.

그래서 먼저 읽는 테이블이 JOIN의 성늘을 결정한다고해서 **DRIVING TABLE(선행 테이블)**이라고 부르고 나중에 읽는 테이블을**DRIVEN TABLE(후행 테이블)**이라고 부른다.

여러 개의 테이블을 JOIN해야 할 경우 JOIN의 성능은 어떤 테이블을 선행 테이블로 설정하는가가 아주 중요하다.

이것을 결정해 주는 오라클 내부 구성요소가 옵티마이저인데 **RULE BASED OPTIMIZER(규칙 기반 옵티마이저)**를 사용할 경우에는 쿼리를 수행하는 사람이 선행 테이블을 계산해서 결정해야 했다.

그래서 실력좋은 개발자나 DBA일 경우 JOIN 작업을 아주 빠르게 수행할 수 있지만 실력이나 경험이 부족한 사람이 작성한 JOIN은 성능이 아주 느린 경우가 많다.

그러나 현재 옵티마이저의 주류를 이루는 **COST BASED OPTIMIZER(비용 기반 옵티마이저)**는 이런 부분이 자동화 되어서 인덱스만 잘 만들어져 있다면 아주 좋은 성능을 내도록 해주고 있다.

(**PGA : PROGRAM GLOBAL AREA**의 약자로 단어가 가진 의미 그대로 공유되지 않고 혼자서만 사용하는 공간.

PGA는 프로세스에 대한 데이터와 제어정보가 포함되 비 공유 메모리 영역으로 서버 프로세스가 시작될 때 생성되며 데이터 베이스에 접속하는

모든 사용자에게 할당된 각각의 프로세스가 독자적으로 사용하는 오라클 데이터베이스의 메모리 공간이다.

즉, 한 프로세스 혹은 스레드의 개별적인 메모리 공간으로 다른 프로세스와 스레드는 접근 불가, 사용자마다 공유하지 않고 개별적으로 사용한다.)

(**SGA : SYSTEM GLOBAL AREA** 오라클 프로세스들이 접근하는 하나의 큰 공유 메모리 세그먼트.)

위와 같은 SQL을 수행하게 되면 오라클은 emp 테이블의 Smith를 가져오고 Smith의 DNAME을 찾으로 DEPT 테이블로 가게 된다.

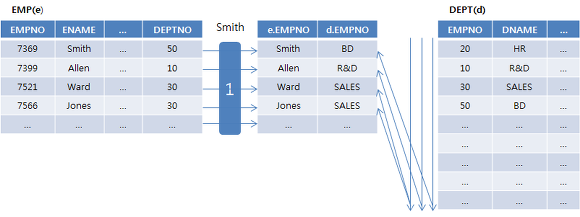
그런데 DEPT 테이블에 데이터가 많을 경우 어떤 DNAME을 가져와야 할지 모르기 때문에 SQL 문자의 WHERE 절에 있는 조건을 보고 그 조건에 맞는 DNAME을 가져오는 것이다.

만약 SQL 문장에서 WHERE 절에 잘못된 조건을 줄 경우나 조건을 안 줄 경우에는 올바른 데이터를 가져오지 못한다.(모든 데이터를 다 가져오게 되며 이를 카티션 곱이라 한다.)

만약 DEPT 테이블에 데이터가 1억건이 있다라고 가정하면 오라클은 Smith의 DNAME을 찾기 위해 1억건을 읽어 본 후 적당한 데이터를 가져와야한다.

이렇게 되면 Join의 성능이 아주 느려지게 된다.

그래서 Join과 더불어 필수적으로 인덱스를 사용하게 된다.



위 그림과 같이 인덱스가 없을 경우 EMP 테이블에서 ENAME이 Smith인 데이터를 한 건 꺼내고 DEPT 테이블 가서 DNAME을 꺼내야 하는데

WHERE 조건대로 SMITH의 DEPTNO와 동일한DEPTNO를 DEPT 테이블에서 검색하게 된다.

그런데 인덱스가 없기 때문에 만약 DEPT 테이블에 데이터가 **1억** 건이 들어 있을 경우 오라클은 DEPT 테이블의 **1억** 건을 다 읽고 난 후 원하던 부서번호

50번의 DNAME인 BD를 가져가게 된다.

위 그림처럼 4번째에서 50번을 찾았다 하더라도 50번이 하나만 있다는 보장을 못하기 때문에 전부를 다 읽게 되는 것이다.

그리고 다시 Allen으로 돌아가서 Smith에서 했던 작업을 반복하게 된다.

만약 emp 테이블에 데이터가 10건 있고 dept 테이블에 데이터가 1억건있다면 dept 테이블은 총 10억 번 읽게 되는 결과가 생긴다는 뜻이며 절대로 성능이 나올 수 가 없는 구조가 된다.

그래서 오라클은 인덱스가 없을 경우 빨리 해당 데이터를 찾아서 결과를 출력하기 위한 방법으로 SORT-MERGE JOIN과 HASH JOIN을 지원한다.

**2) SORT-MERGE JOIN**

- 인덱스가 없을 경우 빨리 해당 데이터를 찾아서 결과를 출력하기 위해 고안한 방법 중 하나이다.

- SORT MERGE JOIN란 용어처럼 SORT한 후 그 결과를 MERGE 해서 데이터를 찾는 방식이다.

- 이 JOIN의 단점은 SORT할 때 시간이 너무 오래 걸린다는 것이다. 그래서 RBO보다 발전된 버전인 CBO에서는

- 이렇게 인덱스가 없을 경우 이 방법을 쓰지 않고 HASH 함수 기반의 HASH JOIN을 사용한다.

- (RBO란 : RULE-BASED OPTIMIZATION 약자로 미리 정해진 규칙에 의한 실행 계획[규칙기반])

- (CBO란 : COST-BASED OPTIMIZATION 약자로 통계정보와 I/O와 CPU 비용을 계산하여 실행계획을 예측[비용기반 최적화])

**3) HASH JOIN(CBO에서만 가능)**

- HASH JOIN은 양쪽 테이블 모두 JOIN 칼럼에 인덱스가 없을 경우에 과거의 SORT-MERGE JOIN이 시간이 너무 오래 걸리는 단점을 보완해서 만들어지 JOIN 방법이다.

- 두개의 테이블에 인덱스가 없는 상황에서 JOIN을 수행한다면 아래와 같은 순서로 진행된다.

- 1. 두 테이블 중에서 범위가 좁을 테이블을 메모리 (PGA 내부의 HASH AREA라는 곳)로 가져온다.

- 2. JOIN 조건 칼럼의 데이터를 HASH 함수에 넣어서 나온 HASH VALUE 값으로 HASH TABLE을 생성한다.

- 3. 후행 테이블의 JOIN 조건을 HASH 함수에 넣어서 HASH VALUE를 생성하고 이 값을 선행 테이블의 HASH TABLE의 값과 비교하여 같은 값이 있으면

- 해당 칼럼의 값을 매칭한다.

**4) SORT-MERGE JOIN과 HASH JOIN**

- 둘 다 모든 테이블을 다 읽는다는 부분은 동일하다.

- 다른 점 : SORT- MERGE JOIN은 정렬을 해서 작업을 수행한다.

-           HASH JOIN은 정렬을 하지 않는다.

- 성능 : 테이블의 특징에 따라 다르겠지만 일반적으로는 HASH JOIN이 SORT-MERGE JOIN 보다 2배 DLTKD TJDSMDDL WHGEK.

- 일반적으로 HASH JOIN은 CBO DB 중 인덱스가 없는 TABLE에 JOIN이 발생할 경우 선택하는 방법이지만 경우에 따라서는 (추출해야할 데이터가 많을 경우 등)

- 인덱스가 있다  하더라도 HASH JOIN을 수행하는 경우도 종종 있다. 그만큼 대용량 데이터를 처리할 경우 HASH JOIN 성능이 좋다는 의미이다.

출처: [http://kshmc.tistory.com/entry/JOIN-원리](http://kshmc.tistory.com/entry/JOIN-%EC%9B%90%EB%A6%AC) [노력하는 개발자 & 꿈꾸는 개발자]  
  
출처: [http://kshmc.tistory.com/entry/JOIN-원리](http://kshmc.tistory.com/entry/JOIN-%EC%9B%90%EB%A6%AC) [노력하는 개발자 & 꿈꾸는 개발자]

설명굿 : <http://kshmc.tistory.com/entry/JOIN-%EC%9B%90%EB%A6%AC>

조인이란 두 개 이상의 테이블을 하나의 집합으로 만드는 연산이다. SQL문에서 FROM 절에 두 개 이상의 테이블이 나열될 경우 조인이 수행된다. 조인 연산은 두 테이블 사이에서 수행된다. FROM 절에 A, B, C라는 세 개의 테이블이 존재하더라도 세 개의 테이블이 동시에 조인이 수행되는 것은 아니다. 세 개의 테이블 중에서 먼저 두 개의 테이블에 대해 조인이 수행된다. 그리고 먼저 수행된 조인 결과와 나머지 테이블 사이에서 조인이 수행된다. 이러한 작업은 FROM 절에 나열된 모든 테이블을 조인할 때까지 반복 수행한다. 예를 들어, A, B, C 세 개의 테이블을 조인할 때를 가정으로 설명하면 다음과 같다. 먼저 A와 B 두 테이블을 먼저 조인하면 해당 조인 결과와 나머지 C 테이블을 조인한다(A → B → C). 만약, A와 C 테이블을 먼저 조인한다면 해당 조인 결과와 나머지 B 테이블을 조인한다(A → C → B). 테이블 또는 조인 결과를 이용하여 조인을 수행할 때 조인 단계별로 다른 조인 기법을 사용할 수 있다. 예를 들어, A와 B 테이블을 조인할 때는 NL Join 기법을 사용하고 해당 조인 결과와 C 테이블을 조인할 때는 Hash Join 기법을 사용할 수 있다. 조인 기법은 두 개의 테이블을 조인할 때 사용할 수 있는 방법이다. 여기서는 조인 기법 중에서 자주 사용되는 NL Join, Hash Join, Sort Merge Join에 대해서 조인 원리를 간단하게 설명한다.

**1. NL Join**

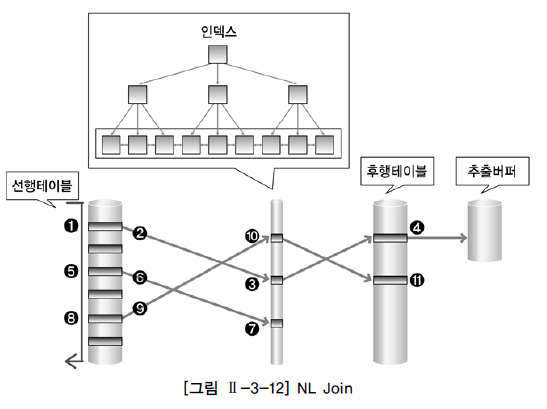
NL Join은 프로그래밍에서 사용하는 중첩된 반복문과 유사한 방식으로 조인을 수행한다. 반복문의 외부에 있는 테이블을 선행 테이블 또는 외부 테이블(Outer Table)이라고 하고, 반복문의 내부에 있는 테이블을 후행 테이블 또는 내부 테이블(Inner Table)이라고 한다.

FOR 선행 테이블 읽음 → 외부 테이블(Outer Table) FOR 후행 테이블 읽음 → 내부 테이블(Inner Table) (선행 테이블과 후행 테이블 조인)

먼저 선행 테이블의 조건을 만족하는 행을 추출하여 후행 테이블을 읽으면서 조인을 수행한다. 이 작업은 선행 테이블의 조건을 만족하는 모든 행의 수만큼 반복 수행한다. NL Join에서는 선행 테이블의 조건을 만족하는 행의 수가 많으면(처리 주관 범위가 넓으면), 그 만큼 후행 테이블의 조인 작업은 반복 수행된다. 따라서 결과 행의 수가 적은(처리 주관 범위가 좁은) 테이블을 조인 순서상 선행 테이블로 선택하는 것이 전체 일량을 줄일 수 있다. NL Join은 랜덤 방식으로 데이터를 액세스하기 때문에 처리 범위가 좁은 것이 유리하다.

NL Join의 작업 방법은 다음과 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음 ② 선행 테이블의 조인 키 값을 가지고 후행 테이블에서 조인 수행 ③ 선행 테이블의 조건을 만족하는 모든 행에 대해 1번 작업 반복 수행



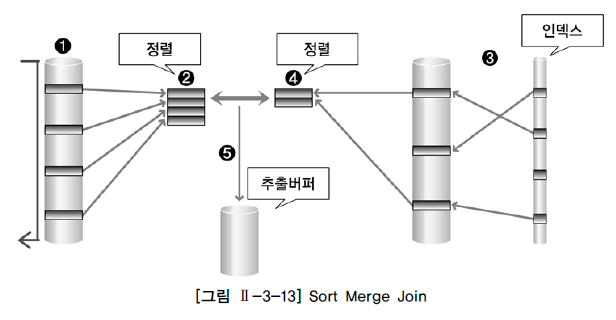
[그림 Ⅱ-3-12]의 예를 이용하여 NL Join의 수행 방식을 알아보도록 하자. [그림 Ⅱ-3-12]에서 인덱스는 B-트리 인덱스의 리프 블록만을 그린 것임을 표현한 것이다.

① 선행 테이블에서 조건을 만족하는 첫 번째 행을 찾음 → 이때 선행 테이블에 주어진 조건을 만족하지 않는 경우 해당 데이터는 필터링 됨 ② 선행 테이블의 조인 키를 가지고 후행 테이블에 조인 키가 존재하는지 찾으러 감 → 조인 시도 ③ 후행 테이블의 인덱스에 선행 테이블의 조인 키가 존재하는지 확인 → 선행 테이블의 조인 값이 후행 테이블에 존재하지 않으면 선행 테이블 데이터는 필터링 됨 (더 이상 조인 작업을 진행할 필요 없음) ④ 인덱스에서 추출한 레코드 식별자를 이용하여 후행 테이블을 액세스 → 인덱스 스캔을 통한 테이블 액세스 후행 테이블에 주어진 조건까지 모두 만족하면 해당 행을 추출버퍼에 넣음 ⑤ ~ ⑪ 앞의 작업을 반복 수행함

추출버퍼는 SQL문의 실행결과를 보관하는 버퍼로서 일정 크기를 설정하여 추출버퍼에 결과가 모두 차거나 더 이상 결과가 없어서 추출버퍼를 채울 것이 없으면 결과를 사용자에게 반환한다. 추출버퍼는 운반단위, Array Size, Prefetch Size라고도 한다. [그림 Ⅱ-3-12]에서 만약 선행 테이블에 사용 가능한 인덱스가 존재한다면 인덱스를 통해 선행 테이블을 액세스할 수 있다. (여기서는 사용할 인덱스가 없음을 가정으로 설명한 것임) NL Join 기법은 조인이 성공하면 바로 조인 결과를 사용자에게 보여 줄 수 있다. 그래서 결과를 가능한 빨리 화면에 보여줘야 하는 온라인 프로그램에 적당한 조인 기법이다.

**2. Sort Merge Join**

Sort Merge Join은 조인 칼럼을 기준으로 데이터를 정렬하여 조인을 수행한다. NL Join은 주로 랜덤 액세스 방식으로 데이터를 읽는 반면 Sort Merge Join은 주로 스캔 방식으로 데이터를 읽는다. Sort Merge Join은 랜덤 액세스로 NL Join에서 부담이 되던 넓은 범위의 데이터를 처리할 때 이용되던 조인 기법이다. 그러나 Sort Merge Join은 정렬할 데이터가 많아 메모리에서 모든 정렬 작업을 수행하기 어려운 경우에는 임시 영역(디스크)을 사용하기 때문에 성능이 떨어질 수 있다. 일반적으로 대량의 조인 작업에서 정렬 작업을 필요로 하는 Sort Merge Join 보다는 CPU 작업 위주로 처리하는 Hash Join이 성능상 유리하다. 그러나 Sort Merge Join은 Hash Join과는 달리 동등 조인 뿐만 아니라 비동등 조인에 대해서도 조인 작업이 가능하다는 장점이 있다.



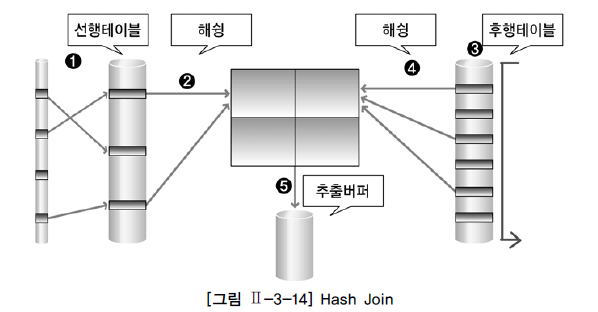
Sort Merge Join의 동작은 [그림 Ⅱ-3-13]과 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음 ② 선행 테이블의 조인 키를 기준으로 정렬 작업을 수행 ① ~ ②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행 ③ 후행 테이블에서 주어진 조건을 만족하는 행을 찾음 ④ 후행 테이블의 조인 키를 기준으로 정렬 작업을 수행 ③ ~ ④번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행 ⑤ 정렬된 결과를 이용하여 조인을 수행하며 조인에 성공하면 추출버퍼에 넣음

Sort Merge Join은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Sort Merge Join에서 조인 작업을 위해 항상 정렬 작업이 발생하는 것은 아니다. 예를 들어, 조인할 테이블 중에서 이미 앞 단계의 작업을 수행하는 도중에 정렬 작업이 미리 수행되었다면 조인을 위한 정렬 작업은 발생하지 않을 수 있다.

**3. Hash Join**

Hash Join은 해슁 기법을 이용하여 조인을 수행한다. 조인을 수행할 테이블의 조인 칼럼을 기준으로 해쉬 함수를 수행하여 서로 동일한 해쉬 값을 갖는 것들 사이에서 실제 값이 같은지를 비교하면서 조인을 수행한다. Hash Join은 NL Join의 랜덤 액세스 문제점과 Sort Merge Join의 문제점인 정렬 작업의 부담을 해결 위한 대안으로 등장하였다.



Hash Join의 동작은 [그림 Ⅱ-3-14]와 같다.

① 선행 테이블에서 주어진 조건을 만족하는 행을 찾음 ② 선행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해쉬 테이블을 생성 → 조인 칼럼과 SELECT 절에서 필요로 하는 칼럼도 함께 저장됨 ① ~ ②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행 ③ 후행 테이블에서 주어진 조건을 만족하는 행을 찾음 ④ 후행 테이블의 조인 키를 기준으로 해쉬 함수를 적용하여 해당 버킷을 찾음 → 조인 키를 이용해서 실제 조인될 데이터를 찾음 ⑤ 조인에 성공하면 추출버퍼에 넣음 ③ ~ ⑤번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해서 반복 수행

Hash Join은 조인 칼럼의 인덱스를 사용하지 않기 때문에 조인 칼럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. Hash Join은 해쉬 함수를 이용하여 조인을 수행하기 때문에 '='로 수행하는 조인 즉, 동등 조인에서만 사용할 수 있다. 해쉬 함수를 적용한 값은 어떤 값으로 해슁될 지 알 수 없다. 해쉬 함수가 적용될 때 동일한 값은 항상 같은 값으로 해슁됨이 보장된다. 그러나 해쉬 함수를 적용할 때 보다 큰 값이 항상 큰 값으로 해슁되고 작은 값이 항상 작은 값으로 해슁된다는 보장은 없다. 그렇기 때문에 Hash Join은 동등 조인에서만 사용할 수 있다. [그림 Ⅱ-3-14]와 같이 Hash Join은 조인 작업을 수행하기 위해 해쉬 테이블을 메모리에 생성해야 한다. 생성된 해쉬 테이블의 크기가 메모리에 적재할 수 있는 크기보다 더 커지면 임시 영역(디스크)에 해쉬 테이블을 저장한다. 그러면 추가적인 작업이 필요해 진다. 그렇기 때문에 Hash Join을 할 때는 결과 행의 수가 적은 테이블을 선행 테이블로 사용하는 것이 좋다. 선행 테이블의 결과를 완전히 메모리에 저장할 수 있다면 임시 영역에 저장하는 작업이 발생하지 않기 때문이다. Hash Join에서는 선행 테이블을 이용하여 먼저 해쉬 테이블을 생성한다고 해서 선행 테이블을 Build Input이라고도 하며, 후행 테이블은 만들어진 해쉬 테이블에 대해 해쉬 값의 존재여부를 검사한다고 해서 Prove Input이라고도 한다.

조인원리 설명 굿 : <http://www.dbguide.net/db.db?boardConfigUid=9&boardIdx=136&boardStep=1&boardUid=148210&categoryUid=216&cmd=view>

**옵티마이저가 아니라 내가 테이블 join 건 순서대로 하고싶으면 select 대신 select straight\_join 을 사용하면된다.**