공 학 석 사 학 위 논 문

# 중첩 루프 조인의 성능분석 및 개선 절차에 관한 연구


2013년 8월

부 경 대 학 교 대 학 원

컴 퓨 터 공 학 과

유신(Chen Liu)

공 학 석 사 학 위 논 문

# 중첩 루프 조인의 성능분석 및 개선 절차에 관한 연구


지도교수 여정모

이 논문을 공학석사 학위논문으로 제출함

2013년 8월

부 경 대 학 교 대 학 원

컴 퓨 터 공 학 과

유신(Chen Liu)

LIU CHEN의 공학석사 학위논문을 인준함

2013년 8월 23일


주 심 공학박사 조우현 (인)

위 원 공학박사 서경룡 (인)

위 원 공학박사 여정모 (인)

# 목차

# 표 목차

# 그림목차

# 중첩 루프 조인의 성능분석 및 개선 절차에 관한 연구

LIU CHEN

부경대학교 일반대학원 컴퓨터공학과

## 요 약

정보시스템의 기반이 되는 관계형 데이터베이스에서는 데이터의 양에 따라 성능 차이가 많이 발생한다. 데이터베이스에 있는 여러 가지 기능에 대한 이해가 부족하여 많은 성능 저하 문제를 유발한다. 그 중에 조인 성능문제가 큰 비중을 차지하고 있다. 아주 드문 경우가 아니라면 대부분의 데이터 처리는 하나 이상의 테이블을 필요하기 때문이다. 조인을 정확하게 사용하면 성능 개선에 큰 이점을 가져 올 수 있다. 조인의 종류에는 세가지가 있는데 본 논문에서는 관계형 데이터베이스에서 가장 기본적인 조인 방식인 중첩 루프 조인에 대해 연구하고자 한다. 기존에 아주 많은 연구들이 중첩 루프 조인의 성능문제를 유발하는 요인에 대해서 개선 방법들을 연구 하였다. 또한 실무에서 일하는 경험자들은 경험에 의존하여 중첩 루프 조인의 성능을 개선하고 있다. 하지만 중첩 루프 조인의 성능을 체계적으로 개선하는 절차가 없었다. 그렇기 때문에 본 논문에서 중첩 루프 조인의 수행과정 중 각 단계에서 성능 문제를 유발할 수 있는 요인들을 실험을 통해서 분석한다. 분석한 결과를 통해 중첩 루프 조인의 성능 문제를 검증하고 개선하는 절차를 제안하고자 한다.

이러한 절차에 따라서 중첩 루프 조인의 성능문제를 체계적으로 개선할 수 있다. 또한 데이터베이스 튜닝 입문자들에게 쉽게 중첩 루프 조인 원리의 이해에 도움이 되고 절차에 따라 성능 영향을 미치는 요인들을 찾아 개설할

수 있다. 또한 데이터베이스를 전문으로 다루지 않는 개발자들에게도 이런 절차를 습득하게 되면 차후의 프로그램 품질도 향상될 것으로 예상된다. 본 논문에서 제안하는 중첩 루프 조인 성능 개선 절차는 실무에서도 큰 도움을 가져올 수 있을 것으로 기대한다.

# A Study on the Performance Analysis and Improvement Procedures of Nested Loops Join

LIU CHEN

Dept. of Computer Engineering of Pukyong National University

## Abstract

In a relational database that is a foundation for the information system, there is a significant performance difference depending on the amount of data. Insufficient understanding about a variety of functions in a database leads to numerous performance degradation problems. Of them, the join performance problem accounts for a great part. It is because most data processing requires one or more tables except in very rare cases. If joins are used correctly, it could bring a great advantage of improving performance. There are three kinds of joins, and this paper would like to study the nested loops join that is the most basic join method in a relational database. To date, a great number of studies have investigated improvement methods for the main cause inducing the performance problem of nested loops join. In addition, the experienced persons working in the field improve the performance of nested loops join by experience. However, there is no procedure to systematically improve the performance of nested loops join. Therefore, this paper analyzes potential factors in causing a performance problem at each step while executing a nested loops join through some experiments. It would like to propose a procedure to verify and improve the performance problem of nested loops join through the analyzed result.

The performance problem of nested loops join could be improved systematically by following this procedure. In addition, it helps the database tuning novices easily understand the principle of nested loops join, and could find factors having an effect on the performance to describe them in accordance with the procedure. Furthermore, if

programmers, who do not specialize in databases, also acquire this procedure, it is expected that the future program quality would also be improved. It is expected that the procedure to improve the performance of nested loops join proposed in this paper could have a great help also in the actual working.

# Ⅰ. 서론

국내에 DBMS(Database Management System)이 도입 되어 이를 이용한 정보시스템의 개발을 한지도 10년 이상의 세월이 흘렀고, 그 결과 DBMS는 정보 시스템에 없어서는 안 될 필수 기반 기술로써 확고히 그 자리를 매김 하게 되었다[9]. 데이터들이 대형화됨에 따라 데이터베이스는 수억 건, 아니 수십억 건을 처리해야 하고 기가 단위 혹은 테라 단위의 오브젝트를 연결하여 원하는 결과를 도출해야 하는 상황에 이르렀다. 시대가 발전할수록 데이터베이스 의존도가 높아지고 있고 이것은 데이터베이스의 처리시간이 응용프로그램의 반응시간과 직결된다는 것을 의미한다. 따라서 시스템 전체의 성능을 향상시키는 가장 빠른 방법은 데이터베이스의 성능 튜닝이다. 데이터베이스 튜닝은 여러 가지 방법을 가지고 수행함으로써 개발비용을 줄일 수 있다.

데이터베이스 튜닝은 응답시간을 빠르게 하며, 처리량을 향상시키므로 소프트웨어의 성능이 향상되고 소프트웨어 사용자의 생산성을 높여 준다. 이처럼 잘 튜닝 된 시스템은 소프트웨어 사용자에게도 편이성을 가져다 준다. 데이터베이스 튜닝은 비용과 성능을 향상시킴으로써 소프트웨어 생산성에 큰 이익을 가져다 주며 개발단계의 전체적인 튜닝은 소프트웨어 개발기간을 단축시킬 수 있다. 이것이 우리가 튜닝을 하는 최종적인 목적이다[9].

데이터베이스 튜닝에 있어서 가장 중요한 것은 시스템 설계가 잘 되어 있어야 한다. 설계가 잘 되어 있지 않으면 데이터를 가져오는데 있어서 내부적으로 많은 시간이 걸리기 때문이다. 그 다음으로는 잘 작성된 SQL 문이야말로 데이터베이스 성능 향상에 주된 역할을 하게 된다. 잘 못 작성된 SQL 쿼리로 인해 최적의 액세스 방법과 경로를 선택할 수 없게 되

고 이로 인해 시스템의 성능 저하를 유발하게 된다. 개발자가 최적의 액세스 경로를 알고 있다면 옵티마이저가 이 최적의 경로로 액세스 할 수 있도록 유도해주기만 하면 되는 것이다. 옵티마이저는 주어진 조건에서 내부적인 계산방법에 의해 최적의 경로를 찾아 유도해 준다. 하지만 인덱스가 없다거나 다른 적절한 도구가 없고 또는 개발자의 잘 못 짜 여진 SQL 쿼리로 인해 옵티마이저가 잘 못된 경로를 선택할 수 있다. 그렇기 때문에 사용자나 개발자가 옵티마이저의 원리를 이해하고 올바른 액세스 경로를 유도해 주어야 할 필요성이 있다.

올바른 액세스 경로를 선택하기 위해서는 중첩 루프 조인, 소트 머지 조인, 해시 조인 방식 중 어떤 조인 방식을 선택하는 것이 효율적인 방식인지를 고려해야 하고, 또한 두 테이블 간에 이루어지는 조인의 순서를 결정해야 한다. 액세스 경로뿐만 아니라 그 밖에 시스템의 속도에 영향을 주는 요인이 많이 있지만 여기에서 특히 다루고자 하는 것은 일반적인 조인 방식 중 가장 많이 사용하고 있는 중첩 루프 조인에 초점을 맞추어 성능 저하를 유발하는 요인을 분석하고 중첩 루프 조인을 사용하기 위해 고려해야 할 사항들에 대한 것이다.

본 논문의 목적에는 두 가지가 있다.

첫째, 중첩 루프 조인이 진행되는 과정 중 조인 성능에 영향을 미치는 요인들을 분석하여 성능 저하를 유발하는 문제점을 찾고자 한다. 둘째, 중첩 루프 조인의 성능에 미치는 요인들의 중요도 순서에 따른 성능 개선 절차를 제안한다.

# II. 관련연구

## 2.1 조인 방식에 대한 고찰

### 2.1.1 조인 방식의 종류

현실 업무에서 필요한 데이터를 출력할 때 대부분은 두 개 이상의 테이블을 조인하여 결과를 보여 준다[15]. 그렇기 때문에 조인은 데이터베이스에서 그만큼 중요한 기능이다. 데이터베이스로 전공하는 사람들뿐만 아니라 프로그래밍 개발자들도 조인을 이해하고 사용해야 하기 때문에 조인에 대해서 알아둘 필요가 있다.

조인의 종류는 각 DBMS에서 모두 같지만 내부 메커니즘이 조금 다를 수 있다. 조인의 종류는 중첩 루프 조인(Nested Loops Join), 소트 머지 조인(Sort Merge Join), 해시 조인(Hash Join) 세 가지로 나눌 수 있다[22].

중첩 루프 조인은 모든 관계형 데이터베이스에 존재한다. 조인을 접하는 개발자들이 가장 처음 듣는 용어가 아마도 중첩 루프 조인일 것이다. 여러 가지 조인 방식 중에 가장 먼저 소개된 방식이며 그만큼 효용 가치가 있는 조인 방식이다. 또한 조인은 데이터베이스 안에서 항상 사용되며 OLTP(On-Line Transaction Processing) 시스템에서 조인을 튜닝 할 때에도 일차적으로 중첩 루프 조인부터 고려한다[20]. 왜냐하면 중첩 루프 조인이 가장 기본적인 조인 방식이며 다른 조인 방식들은 중첩 루프 조인의 성능이 좋지 않을 때 이를 보완하기 위해 탄생되었기 때문이다. 만약 중첩 루프 조인으로 좋은 성능을 얻을 수 없으면 다른 조인 방식을 고려하게 된다.

### 2.1.2 중첩 루프 조인

#### 1. 중첩 루프 조인의 매커니즘 및 수행 과정

중첩 루프 조인[22]은 프로그래밍에서 사용하는 중첩된 반복문과 유사한 식으로 조인을 수행한다. 반복문의 외부에 있는 테이블을 선행 테이블 또는 드라이빙 테이블(Driving Table)이라고 하고, 반복문의 내부에 있는 테이블을 후행 테이블 또는 드리븐(Driven Table)이라고 한다. 본 논문에서 혼란을 일으키지 않도록 드라이빙 테이블 및 드리븐 테이블이라는 용어를 사용하기로 한다.

```
for(i=0; i<100; i++)
{ --Driving Table
        for(j=0; j<100; j++)
        { --Driven Table
                //Do Anything...
        }
}
```

그림 1. 프로그래밍 언어의 중첩 반복문 구조

먼저 드라이빙 테이블의 조건을 만족하는 행을 추출하여 드라이빙 테이블을 읽으면서 조인을 수행한다. 이 작업은 드라이빙 테이블의 조건을 만족하는 모든 행의 수만큼 반복 수행한다. 중첩 루프 조인에서는 드라이빙 테이블의 조건을 만족하는 행의 수가 많으면(처리 주관 범위가 넓

으면), 그 만큼 드라이빙 테이블의 조인 작업은 반복 수행된다. 따라서 결과 행의 수가 적은(처리 주관 범위가 좁은) 테이블을 조인 순서상 드라이빙 테이블로 선택하는 것이 전체 일량을 줄일 수 있다. 중첩 루프 조인은 랜덤 방식으로 데이터를 액세스하기 때문에 처리 범위가 좁은 것이 유리하다.

중첩 루프 조인의 작업 방법은 다음과 같다. 먼저 드라이빙 테이블에서 주어진 조건을 만족하는 행을 찾는다. 그 다음 드라이빙 테이블의 조인 키 값을 가지고 드리븐 테이블에서 조인을 수행한다. 드라이빙 테이블의 조건을 만족하는 모든 행에 대해 1번 작업 반복 수행한다.

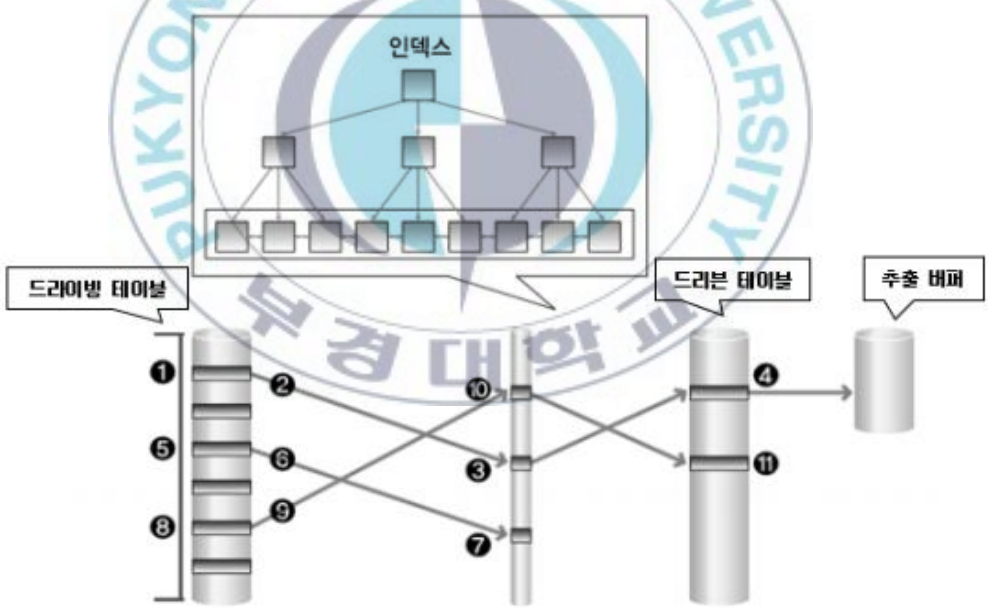



**그림 2. 중첩 루프 조인 수행 과정**

그림2를 이용하여 중첩 루프 조인의 수행 과정을 설명한다[22]. 그림2

에서 인덱스는 B-Tree 인덱스의 리프 블록만을 그린 것이다.

①드라이빙 테이블에서 조건을 만족하는 첫 번째 행을 찾음

→ 이때 드라이빙 테이블에 주어진 조건을 만족하지 않는 경우 해당 데이터는 필터링 됨

②드라이빙 테이블의 조인 키를 가지고 드리븐 테이블에 조인 키가 존재하는지 찾으러 감

→ 조인 시도

③드리븐 테이블의 인덱스에 드라이빙 테이블의 조인 키가 존재하는지 확인

→ 드라이빙 테이블의 조인 값이 드리븐 테이블에 존재하지 않으면 드라이빙 테이블 데이터는 필터링 됨(더 이상 조인 작업을 진행할 필요 없음)

④인덱스에서 추출한 로우 식별자를 이용하여 드리븐 테이블을 액세스

→ 인덱스 스캔을 통한 테이블 액세스

드리븐 테이블에 주어진 조건까지 모두 만족하면 해당 행을 추출버퍼에 넣음

⑤~⑪앞의 작업을 반복 수행함

추출버퍼는 SQL문장의 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.

그림2에서 만약 드라이빙 테이블에 사용 가능한 인덱스가 존재한다면 인덱스를 통해 드라이빙 테이블을 액세스할 수 있다. (여기서는 사용할 인덱스가 없을 가정으로 설명한 것임) 중첩 루프 조인 기법은 조인이 성공하면 바로 조인 결과를 사용자에게 보여 줄 수 있다. 그래서 결과를 가능한 빨리 화면에 보여줘야 하는 온라인 프로그램에 적당한 조인 기법이다.

## 2. 중첩 루프 조인의 활용 방법

### 1) 힌트로 중첩 루프 조인 유도

조인 SQL문장을 실행하여 3가지 조인 방식 중에서 어떤 방식으로 조인하는지는 DBMS 내부에 있는 옵티마이저(Optimizer)가 통계정보를 통해 비용을 계산한 다음 비용이 가장 낮은 조인 방식을 채택한다. 그렇지만 통계정보가 오래되거나 다른 이유 때문에 옵티마이저가 계산한 비용이 부정확하고 비효율적인 조인 방식도 선정되는 경우도 가끔 생긴다. 이런 경우에 사용자가 원하는 대로 조인 방식, 조인 순서 등을 지정하거나 실행계획을 고정시키고자 할 때 힌트를 이용할 수 있다.

```
SELECT /*+ ORDERED USE_NL(E) */ *
  FROM DEPT D, EMP E
 WHERE E.DEPTNO = D.DEPTNO;
```

그림 3. ordered 힌트로 두 개 테이블 조인 순서 지정

그림3에서 나오는 조인 SQL 문장과 같이 ordered 힌트는 from 절에 기

술된 순서대로 조인하라고 옵티마이저에게 지시할 때 사용하고, use_nl 힌트는 중첩 루프 조인 방식으로 조인하라고 지시할 때 사용한다. 위에서는 ordered와 use_nl(e) 힌트를 같이 사용했으므로 dept 테이블(드라이빙 테이블)을 기준으로 emp 테이블(드리븐 테이블)과 조인할 때 중첩 루프 조인 방식으로 조인하라는 뜻이 된다. 위에서는 두 개 테이블을 조인하고 있지만 세 개 이상을 조인할 때는 힌트를 그림4처럼 사용하는 것이 올바른 사용법이다.

```
SELECT /*+ ORDERED USE_NL(B)
           USE_NL(C) USE_HASH(D) */*
  FROM A, B, C, D
 WHERE ……
```

그림 4. ordered 힌트로 두 개 이상 테이블 조인 순서 지정

해석해 보면, A→B→C→D순으로 조인하되 B와 조인할 때 그리고 이어서 C와 조인할 때는 중첩 루프 조인 방식으로 조인하고 D와 조인할 때는 해시 방식으로 조인하라는 뜻이다.

ordered 대신 leading 힌트를 사용해 조인 순서를 제어할 수도 있다. 오라클 9i까지는 leading힌트에 인자를 하나만 입력할 수 있었다. 조인할 때 가장 처음에 읽을 기준 집합(드라이빙 테이블) 하나만 명시하는 것이다. 그러나 그러다 보니 leading힌트만으로는 조인순서를 세밀하게 제어할 수 없어 9i 이전 버전에서는 ordered 힌트를 주로 사용하였다. 하지만, 조인 순서가 바뀌질 때 FROM절의 테이블 순서를 일일이 바꿔 주어야 되는 단점을 가지고 있다.

10g부터는 leading힌트에 2개 이상 테이블을 기술할 수 있도록 기능이 개선돼, 그림5와 같이 FROM 절을 바꾸지 않고도 마음껏 순서를 제어할 수 있게 되었다.

```
SELECT /*+ LEADING(C,A,D,B) USE_NL(A)
           USE_NL(D) USE_HASH(B) */ *
  FROM A, B, C, D
 WHERE ……
```

그림 5. leading 힌트로 두 개 이상 테이블 조인 순서 지정

```
SELECT /*+ USE_NL(A, B, C, D) */*
  FROM A, B, C, D
 WHERE ……
```

그림 6. ordered또는 leading 힌트를 사용하지 않은 경우

그리고 그림6와 같이 ordered나 leading 힌트를 기술하지 않으면 옵티마이저가 스스로 정한 순서에 따라서 조인하라는 의미이다.

### 2) 중첩 루프 조인의 특징 및 적용환경

오라클은 블록 단위로 I/O를 수행하며, 하나의 레코드를 읽으려고 블록을 통째로 읽는 랜덤액세스(Random Access) 방식이기 때문에 가령 메모리 버퍼에서 빠르게 읽더라도 비효율은 존재한다. 중첩 루프 조인의 첫 번째 특징은 랜덤액세스 위주의 조인 방식이라는 점이다. 따라서 인

덱스 구성이 아무리 완벽하더라도 대량의 데이터를 조인할 때 매우 비효율적이다.

두 번째 특징은 조인을 한 로우씩 순차적으로 진행한다는 점이다. 첫 번째 특징 때문에 대용량 데이터 처리 시 매우 치명적인 한계를 드러내지만 반대로 이 두 번째 특징 때문에 부분범위처리가 가능한 상황에서 아무리 대용량 집합이더라도 매우 극적인 응답 속도를 낼 수 있다. 그리고 순차적으로 진행하는 특징 때문에 먼저 액세스되는 테이블의 처리 범위에 의해 전체 일량이 결정된다는 특징도 가지고 있다.

다른 조인 방식과 비교했을 때 인덱스 구성 전략이 특히 중요하다는 것도 중첩 루프 조인의 중요한 특징이다. 조인 컬럼에 대한 인덱스가 있느냐 없느냐 또한 만약, 있다면 인덱스 컬럼이 어떻게 구성되었느냐에 따라 조인 효율이 크게 달라진다.

이런 여러 가지 특징을 종합할 때, 중첩 루프 조인은 소량의 데이터를 주로 처리하거나 부분범위처리가 가능한 온라인 트랜잭션 환경에 적합한 조인 방식이라고 할 수 있다[20].

### 2.1.3 소트 머지 조인

소트 머지 조인은 조인 컬럼을 기준으로 데이터를 정렬하여 조인을 수행한다. 중첩 루프 조인은 주로 랜덤 액세스 방식으로 데이터를 읽는 반면 소트 머지 조인은 주로 스캔 방식으로 데이터를 읽는다. 소트 머지 조인은 랜덤 액세스로 중첩 루프 조인에서 부담이 되던 넓은 범위의 데이터를 처리할 때 이용되던 조인 기법이다. 그러나 소트 머지 조인은 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 소트 머지 조인보다는 CPU작업 위주로 처리하는 해시 조인이 성능상 유리하다. 그러나 소트 머지 조인은 해시 조인과는 달리 동등 조인 뿐만 아니라 비동등 조인에 대해서도 조인 작업이 가능하다는 장점이 있다.

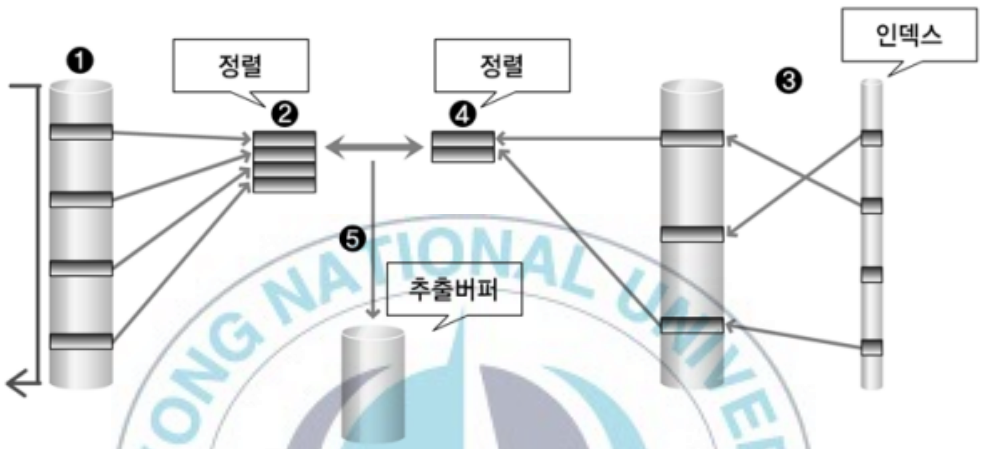



**그림 7. 소트 머지 조인 수행과정**

소트 머지 조인의 수행과정은 그림7과 같다[22].

①선행 테이블에서 주어진 조건을 만족하는 행을 찾음

②선행 테이블의 조인 키를 기준으로 정렬 작업을 수행

③후행 테이블에서 주어진 조건을 만족하는 행을 찾음

④후행 테이블의 조인 키를 기준으로 정렬 작업을 수행

⑤정렬된 결과를 이용하여 조인을 수행하며 조인에 성공하면 추출버퍼에 넣음

소트 머지 조인은 조인 컬럼의 인덱스를 사용하지 않기 때문에 조인

컬럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. 소트 머지 조인에서 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.

### 2.1.4 해시 조인

해시 조인은 해시 기법을 이용하여 조인을 수행한다. 조인을 수행할 테이블의 조인 컬럼을 기준으로 해시함수를 수행하여 서로 동일한 해시 값을 갖는 것들 사이에서 실제 값이 같은지를 비교하면서 조인을 수행한다.

해시조인은 중첩 루프 조인의 랜덤 액세스 문제점과 소트 머지 조인의 문제점인 정렬 작업의 부담을 해결하기 위한 대안으로 등장하였다.

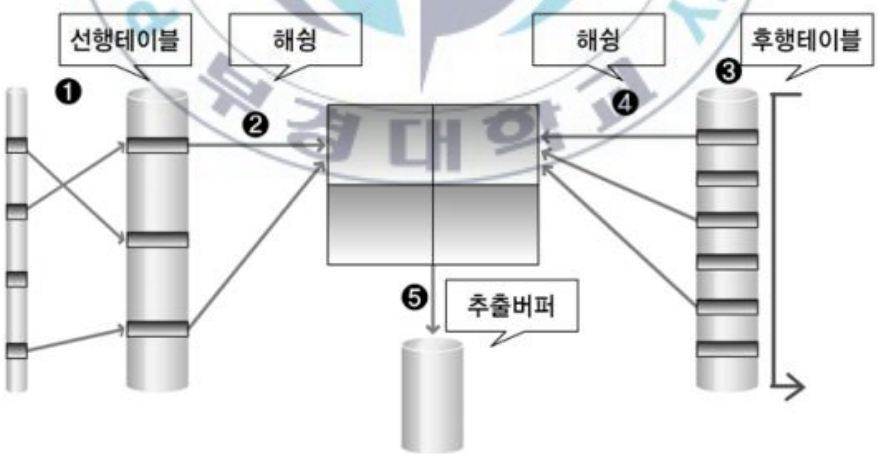



**그림 8. 해시 조인 수행 과정**

해시 조인의 수행과정은 그림8와 같다[22].

①선행 테이블에서 주어진 조건을 만족하는 행을 찾음

②선행 테이블의 조인 키를 기준으로 해시 함수를 적용하여 해시 테이블을 생성

→조인 컬럼과 SELECT 절에서 필요로 하는 컬럼도 함께 저장됨

①~②번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행

③후행 테이블에서 주어진 조건을 만족하는 행을 찾음

④후행 테이블의 조인 키를 기준으로 해시 함수를 적용하여 해당 버킷을 찾음

→ 조인 키를 이용해서 실제 조인될 데이터를 찾음

⑤조인에 성공하면 추출버퍼에 넣음

③~⑤번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해서 반복 수행

해시 조인은 조인 컬럼의 인덱스를 사용하지 않기 때문에 조인 컬럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 조인 기법이다. 해시 조인은 해시 함수를 이용하여 조인을 수행하기 때문에 '=' 로 수행하는 조인 즉, 동등 조인에서만 사용할 수 있다. 해시 함수를 적용한 값은 어떤 값으로 해시될 지 알 수 없다. 해시 함수가 적용될 때 동일 한 값은 항상 같은 값으로 해시됨이 보장된다. 그러나 해시 함수를 적용할 때 보다 큰 값이 항상 큰 값으로 해시되고 작은 값이 항상 작은 값으로 해시된다는 보장은 없다. 그렇기 때문에 해시 조인은 동등 조인에서만 사용할 수 있다.

그림8와 같이 해시 조인은 조인 작업을 수행하기 위해 해시 테이블을 메모리에 생성해야 한다. 생성된 해시 테이블의 크기가 메모리에 적재할 수 있는 크기보다 더 커지면 임시 영역(디스크)에 해시 테이블을 저장한다. 그러면 추가적인 작업이 필요해 진다. 그렇기 때문에 해시 조인을 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.

해시 조인에서는 선행 테이블을 이용하여 먼저 해시 테이블을 생성한다고 해서 선행 테이블을 Build Input이라고도 하며, 후행 테이블은 만들어진 해시 테이블에 대해 해시 값의 존재여부를 검사한다고 해서 Prove Input이라고도 한다.

## 2.2 인덱스에 대한 고찰

인덱스란 데이터를 액세스하는데 있어서 데이터베이스를 도와주는데 사용되는 보조적인 데이터 구조이다. 사용자는 데이터베이스에 저장된 자료를 더욱 빠르게 조회하기 위하여 인덱스를 생성하고 사용한다. 또한, 인덱스는 적절하게 사용하기만 한다면 데이터의 조회 속도를 증가시킬 뿐만 아니라 디스크 I/O를 줄일 수 있는 수단이기도 하다.

인덱스는 그 대상인 테이블과는 별도의 형태로 독립적인 저장 공간을 가지고 존재한다. 테이블에 값이 입력되거나, 수정, 삭제될 경우 인덱스의 수정은 데이터베이스에서 자동적으로 수행하게 된다. 따라서 대량의 데이터 수정이 발생할 때 인덱스를 수정하기 위한 작업이 지나치게 발생하기 때문에 SQL문의 실행 속도가 저하된다.

### 2.2.1 B*-Tree 인덱스 및 관련 개념

#### 1. B*-Tree 인덱스

모든 DBMS는 나름의 다양한 인덱스를 제공하는데, 저장방식과 구조, 탐색 알고리즘이 조금씩 다르긴 해도 원하는 데이터를 빨리 찾도록 돕는다는 근본적인 목적은 같다[22]. 또한 모든 DBMS가 B*Tree 인덱스를 기본적으로 제공하며 추가적으로 제공하는 인덱스 구조는 모두 B*Tree 인덱스의 단점을 보완하기 위해 개발된 것들이다.

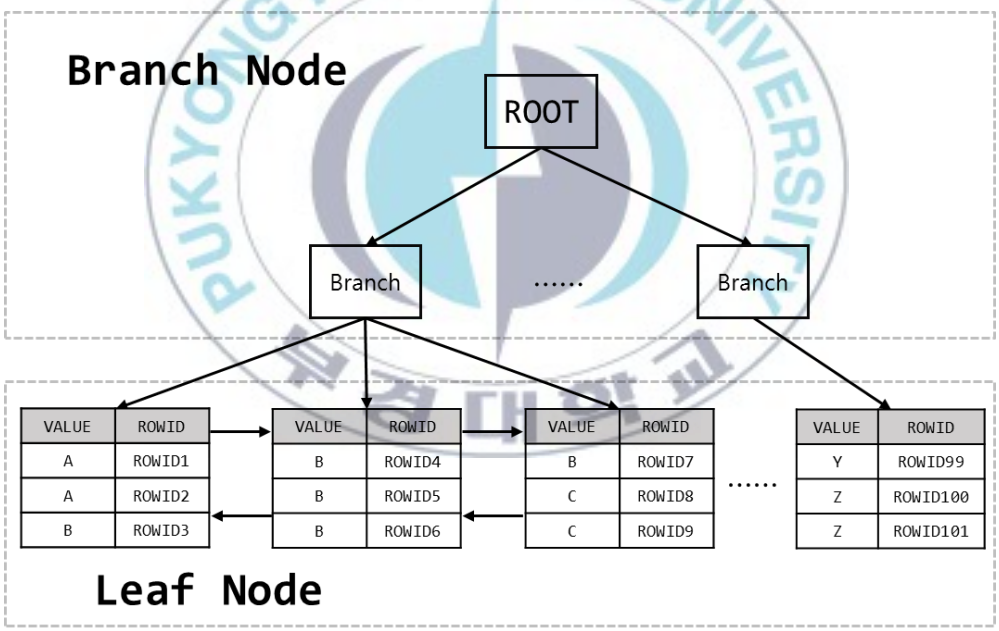



그림9. 개념적인 B*Tree 인덱스 구조도

B*Tree 인덱스는 그림9에서 보는 바와 같이 일반적으로 알고 있는 트리 구조를 가지고 있다. 트리 구조 중에 노드의 종류는 브랜치 노드(Branch Node)와 리프 노드(Leaf Node) 두 가지로 나눌 수 있다. 브랜치

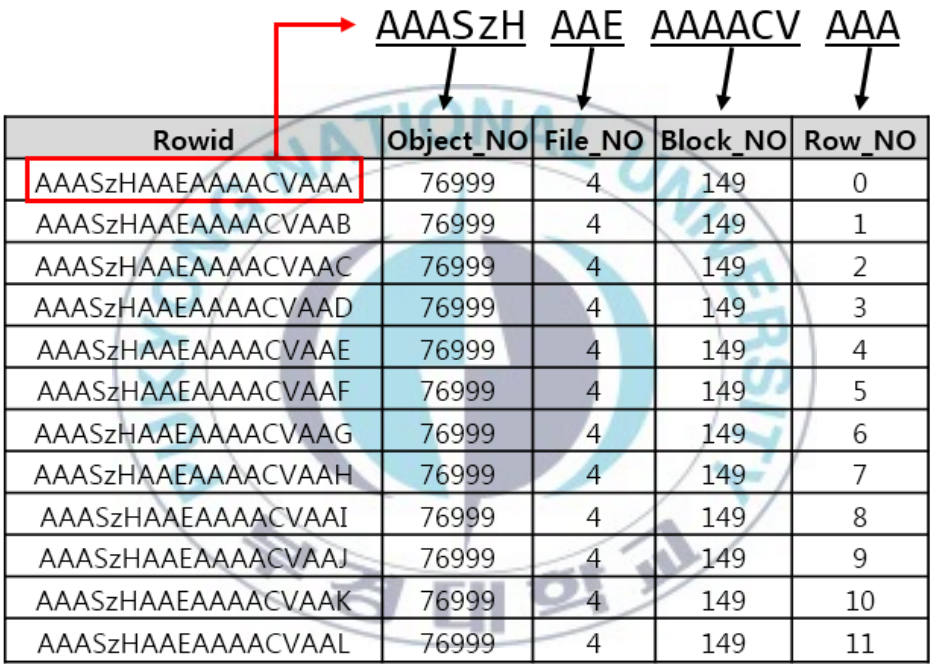
노드의 최상단, 즉 전체 트리 구조의 최상단에 있는 노드를 루트 노드(Root Node)라 한다. 브랜치 노드는 리프 노드에 연결되어 있으며 조회하는 값이 있는 리프 노드까지 검색한다. 리프 노드는 정렬되어 있는 컬럼 값(키 값이라고 하기도 함)과 ROWID 한 쌍으로 저장되어 있다.

이러한 B*Tree 인덱스 구조에서 리프 노드 간에는 서로 연결하는 링크를 가지고 있어서 인덱스를 스캔할 경우 다음 리프 노드로 쉽게 액세스할 수 있다. 논리적으로는 리프 노드 간의 연결고리로 인해 정렬된 리스트의 형태로 보이게 된다. 이때, 중복을 허용하는 Non-Unique 인덱스의 경우 인덱스에서 컬럼값이 같으면 ROWID순서로 정렬이 된다.

SQL문장에서 특정 조건에 만족하는 로우를 인덱스를 통해서 검색할 때 인덱스의 루트 노드에서부터 리프 노드까지 찾게 된다. 리프 노드에서 해당 값을 찾게 되면 이 값과 대응하고 있는 ROWID를 읽고 바로 데이터 블록에 가서 해당 로우를 액세스한다. ROWID를 통해서 데이터를 액세스하는 방식을 랜덤 액세스라고 한다. 왜냐하면 ROWID를 통해서 디스크 임의의 위치에 직접 액세스하여 데이터를 읽어오기 때문이다. 그림10과 같이 ORACLE에서 제공해 주는 DBMS_ROWID 패키지를 사용하여 확인하면 그림11와 같이 ORACLE의 ROWID의 구조는 OBJECT_NO, FILE_NO, BLOCK_NO, ROWNO로 구성되어 있다는 것을 확인할 수 있다[20].

```
SELECT ROWID
     , DBMS_ROWID.ROWID_OBJECT(ROWID) OBJECT_ID
     , DBMS_ROWID.ROWID_RELATIVE_FNO(ROWID) FILE_ID
     , DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) BLOCK_ID
     , DBMS_ROWID.ROWID_ROW_NUMBER(ROWID)   ROW_ID
FROM EMP;
```

그림 10. DBMS_ROWID 패키지를 이용한 ROWID구조 조회 SQL문




| Rowid | Object_NO | File_NO | Block_NO | Row_NO |
|---|---|---|---|---|
| AAASzHAAEAAAACVAAA | 76999 | 4 | 149 | 0 |
| AAASzHAAEAAAACVAAB | 76999 | 4 | 149 | 1 |
| AAASzHAAEAAAACVAAC | 76999 | 4 | 149 | 2 |
| AAASzHAAEAAAACVAAD | 76999 | 4 | 149 | 3 |
| AAASzHAAEAAAACVAAE | 76999 | 4 | 149 | 4 |
| AAASzHAAEAAAACVAAF | 76999 | 4 | 149 | 5 |
| AAASzHAAEAAAACVAAG | 76999 | 4 | 149 | 6 |
| AAASzHAAEAAAACVAAH | 76999 | 4 | 149 | 7 |
| AAASzHAAEAAAACVAAI | 76999 | 4 | 149 | 8 |
| AAASzHAAEAAAACVAAJ | 76999 | 4 | 149 | 9 |
| AAASzHAAEAAAACVAAK | 76999 | 4 | 149 | 10 |
| AAASzHAAEAAAACVAAL | 76999 | 4 | 149 | 11 |

그림11. ORACLE의 ROWID 구조

데이터베이스 안에서 SQL문을 빠르게 실행하기 위한 목적으로 인덱스를 사용하게 되는데 인덱스만 이용하면 언제든지 빠른 속도로 실행되는 착각을 할 수도 있다. 인덱스는 일반적으로 10~15% 이내의 테이블 자료를 액세스할 경우 효율적이라고 알려져 있으며 그 이상의 테이블 자료를

액세스 할 경우에는 전체 테이블 스캔(FULL TABLE SCAN)의 성능이 더 좋을 수 있다. 그 원인은 인덱스를 사용하여 액세스 할 경우 발생하는 랜덤 액세스에 있다. 랜덤 액세스 할 때 원하는 로우와 필요없는 로우가 데이터 블록 안에 같이 저장되어 있다. 따라서, 인덱스를 사용하여 로우의 위치를 직접 읽어 들이는 이점도 있지만 대량의 자료를 읽는 경우에는 생기는 부하가 이점보다 훨씬 크게 된다.

## 2. 분포도와 손익분기점

인덱스를 생성시키고자 하는 컬럼의 분포도는 10~15%를 넘지 않아야 한다[16]. 여기서 말하는 분포도란 어떤 컬럼이 테이블에 평균적으로 분포되어 있는 정도이다. 그림12와 같은 공식으로 평균 분포도를 구할 수 있다.

$$\text{분포도} = \frac{\textit{컬럼값별 평균 로우수}}{\textit{총 로우수}} * 100 = \frac{1}{\textit{컬럼값의 종류}} * 100$$

그림 12. 분포도 계산 공식 [16]

이 분포도는 모든 값에 대한 로우수가 일정하다고 가정한 평균 분포도이다. 실제로 데이터 값에 따라 분포도에 차이가 있으므로 컬럼내의 분포도에 따라 처리 속도에 차이가 난다. 분포도가 일정 기준을 넘는 값을 액세스하는 경우는 전체 테이블을 스캔하는 것보다 나빠진다.

인덱스 컬럼의 분포도가 10~15%가 기준이라는 것은 그 이상인 경우는 차라리 전체 테이블을 스캔하는 것이 유리하다는 것을 의미한다. 왜냐하

면 인덱스를 경유하여 액세스한다는 것은 처리할 범위의 인덱스 로우에 있는 ROWID 정보를 이용하여 실제 테이블에 있는 로우를 일일이 랜덤하게 액세스해야 하므로 스캔방식보다 훨씬 불리하기 때문이다.

인덱스 범위 스캔에 의한 테이블 액세스가 전체 테이블 스캔보다 느려지는 지점을 흔히 손익분기점이라고 부른다. 예를 들어, 인덱스 손익분기점이 10%라는 의미는 1000개 중 100개 로우 이상을 읽을 때는 인덱스를 이용하는 것보다 테이블 전체를 스캔하는 것이 더 빠르다는 것이다[22].

인덱스 손익분기점은 일반적으로 5~20%의 낮은 수준에서 결정되지만 클러스터링 팩터에 따라 크게 달라진다. 클러스터링 팩터가 나쁘면 손익분기점은 5% 미만에서 결정되며, 심할 때는 1%미만으로 떨어진다. 반대로 클러스터링 팩터가 아주 좋을 때는 손익분기점이 90%수준까지 올라가기도 한다.

## 3. 클러스터링 팩터

1) 클러스터링 팩터의 개념

Oracle은 '클러스터링 팩터(Clustering Factor)'라는 개념을 사용해 인덱스 ROWID에 의한 테이블 액세스 비용을 평가한다[22]. SQL Server는 공식적으로 이 용어를 사용하진 않지만 내부적인 비용 계산식에 이런 개념이 포함돼 있을 것이다[22].

클러스터링 팩터는 '군집성 계수(=데이터가 모여 있는 정도)' 쯤으로 번역될 수 있는 용어로서, 특정 컬럼을 기준으로 같은 값을 갖는 데이터가 서로 모여 있는 정도를 의미한다[22].

2) 클러스터링 팩터가 성능에 미치는 영향

그림13에서 보는 바와 같이 클러스터링 팩터가 좋을 때는 컬럼값이 'A' 인 데이터들을 액세스하는 데 테이블을 저장하고 있는 데이터 블록을 하나만 읽어왔다(클러스터링 팩터가 1임). 그렇지만 클러스터링 팩터가 좋지 않을 때는 컬럼값이 'A' 인 데이터들을 액세스하는 데 4개 데이터 블록을 읽어왔다(클러스터링 팩터가 4임). 그렇기 때문에 클러스터링 팩터가 좋은지 나쁜지가 블록 I/O에 큰 영향을 미칠 수 있다는 것이다.

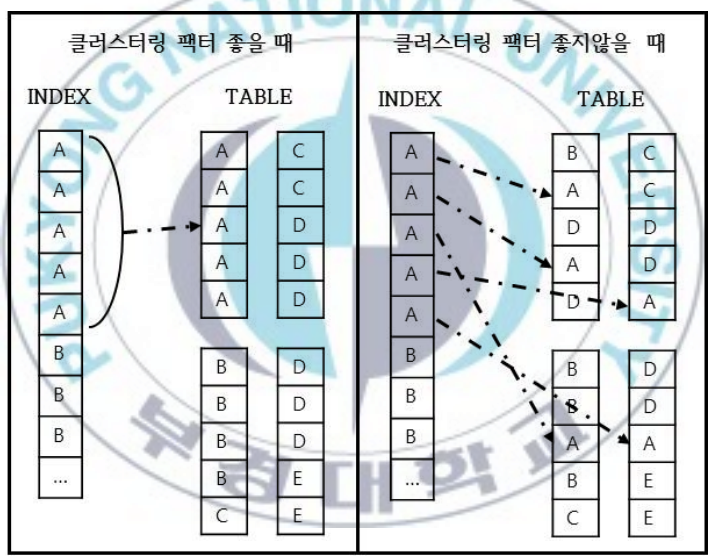



**그림13. 클러스터링 팩터가 좋을 때와 좋지 않을 때의 액세스 비교**

인덱스를 통해서 액세스하면 반드시 클러스터링 팩터의 영향을 받는다. 분포도의 손익분기점도 클러스터링 팩터에 따라 큰 차이가 난다. 또한 클러스터링 팩터는 어떤 하나의 컬럼을 기준으로 계산되는데 어떤 컬럼의 클러스터링 팩터를 좋게 만들어 주면 다른 컬럼의 클러스터링 팩터

는 나빠질 수 있다.

3) 클러스터링 팩터 조회 및 계산 방법

```
CREATE TABLE T
AS
SELECT *
  FROM ALL_OBJECTS
 ORDER BY OBJECT_ID;

CREATE INDEX T_OBJECT_ID_IDX ON T(OBJECT_ID);

CREATE INDEX T_OBJECT_NAME_IDX ON T(OBJECT_NAME);
```

**그림 14. 클러스터링 팩터 조회 및 계산을 위한 테이블/인덱스 생성**

그림14와 같은 SQL로 테이블 레코드가 OBJECT_ID순으로 입력되도록 T 테이블 생성하고, OBJECT_ID 와 OBJECT_NAME 각각에 대한 인덱스를 하나씩 생성하였다.

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(USER, 'T');

SELECT I.INDEX_NAME, T.BLOCKS TABLE_BLOCKS
     , I.NUM_ROWS, I.CLUSTERING_FACTOR
  FROM USER_TABLES T, USER_INDEXES I
 WHERE T.TABLE_NAME = 'T'
   AND I.TABLE_NAME = T.TABLE_NAME;
```

**그림 15. 클러스터링 팩터 조회**

그림15와 같이 통계정보를 생성하고 나면 오라클이 계산한 인덱스 클러스터링 팩터를 뷰에서 확인할 수 있다. 그림16와 같다.

| INDEX_NAME | TABLE_BLOCKS | NUM_ROWS | CLUSTERING_FACTOR |
|---|---|---|---|
| T_OBJECT_NAME_IDX | 1071 | 73414 | 38061 |
| T_OBJECT_ID_IDX | 1071 | 73414 | 1045 |

그림 16. 클러스터링 팩터 조회 결과

클러스터링 팩터 수치가 테이블 블록(TABLE_BLOCKS)에 가까울수록 데이터가 잘 정렬돼 있음을 의미하고, 레코드 개수(NUM_ROWS)에 가까울소록 흩어져 있음을 의미한다. 인덱스 통계를 수집할 때 클러스터링 팩터 계산을 위해 오라클이 사용하는 아래 로직을 따른다.

(1) Counter변수를 하나 선언한다.

(2) 인덱스 리프 블록을 처음부터 끝까지 스캔하면서 인덱스 ROWID로부터 블록번호를 취한다.

(3) 현재 읽고 있는 인덱스 레코드의 블록 번호가 바로 직전에 읽은 레코드의 블록 번호와 다를 때마다 Counter변수 값을 1씩 증가시킨다.

(4) 스캔을 완료하고서, 최종 Counter변수 값을 클러스터링 팩터로서 인덱스 통계에 저정한다.

T테이블을 생성하면서 OBJECT_ID순으로 정렬했고, 이 컬럼에 대해 생성한 인덱스(T_OBJECT_ID_IDX)의 클러스터링 팩터가 테이블 전체 블록수에 근접한 것을 확인할 수 있다.

### 2.2.2 인덱스 활용 방법

## 1. 결합 인덱스

어떤 테이블에서 다양한 형태의 액세스 중에 최소의 인덱스로 모든 경우를 만족할 수 있도록 하기 위해서는 각 인덱스의 활용도를 높여야 한다. 이를 위해서는 두 가지 측면을 고려해야 한다. 첫 번째는 좋은 분포도를 가진 컬럼은 가능하다면 독립적인 인덱스를 만들어 적용의 유연성을 높여야 한다. 두 번째는 그렇지 못한 컬럼들은 유연성은 감소하지만 시너지 효과가 높을 수 있도록 적절한 결합을 하여야 한다[16].

결합 인덱스(또는 복합 인덱스)는 한 테이블의 여러 컬럼을 조합하여 인덱스를 생성하는 것을 말한다. 결합인덱스에 사용되는 컬럼을 테이블에 생성되어 있는 컬럼의 순서와는 무관하게 배열할 수 있다. 그렇지만 정렬순서는 결합인덱스 컬럼의 구성순서에 따라서 정렬된다.

결합인덱스 처리과정은 그림17와 같다. SQL문의 WHERE절에서 COL1 = 'ABC' AND COL2 = '123' 조건이 있고 COL1과 COL2 순서로 결합인덱스가 생성되어 있는 상태이다. 처리과정은 아래와 같다.

① 결합인덱스의 첫 번째 컬럼값이 'ABC'인 데이터들을 찾기 시작한다.

② 첫 번째 컬럼값이 'ABC'인 로우들을 스캔해 내려가면서 두 번째 컬럼 값이 '123'인 로우를 찾아낸다.

③ 찾아낸 로우들의 ROWID로 테이블에 액세스한다.

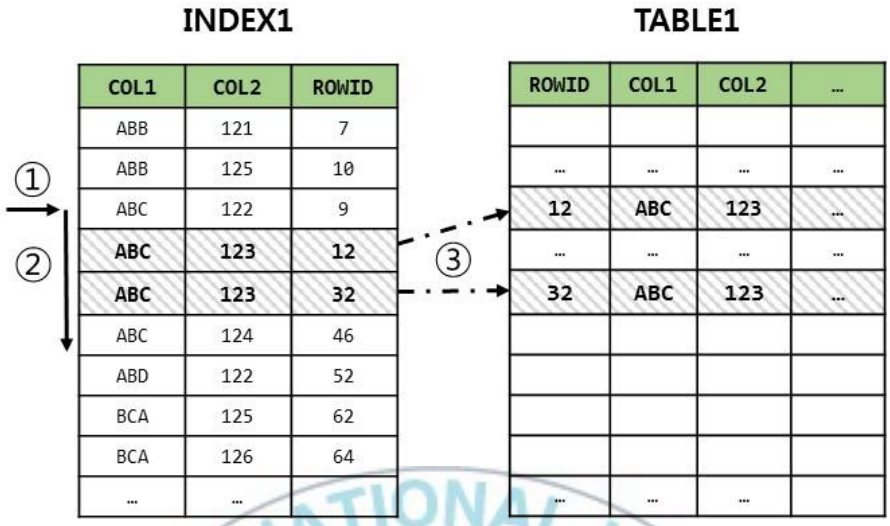



그림17. 결합인덱스 처리과정 흐름도

## 2. 인덱스를 사용하지 못하는 경우

인덱스를 생성해 주었다고 하더라도 인덱스를 사용하지 못하는 경우가 있다.

1) 인덱스 컬럼이 가공될 경우

사용자가 함수(Function)나 연산자(Operator) 등을 사용하여 의도적으로 컬럼값에 변경을 가해 인덱스를 사용하지 못할 수 있다. 또한 조건 양쪽 컬럼 타입이 일치하지 않아서 데이터베이스 내부적으로 발생하는 묵시적인 형변환 때문에 인덱스 사용 못할 수도 있다. 그래서 경험이 부족한 개발자들은 인덱스를 사용하지 못하는 SQL문을 작성하는 경우가 많다. 이런 SQL들을 인덱스를 사용할 수 있도록 표10와 같은 튜닝 방안을 적용할 수 있다[20].

표 1. 인덱스 컬럼이 가공 사례 및 튜닝 방안

| 인덱스 컬럼 가공 사례 | 튜닝 방안 |
|---|---|
| SELECT *<br>FROM 업체<br>WHERE SUBSTR(업체명,1,2) = '대한' | SELECT *<br>FROM 업체<br>WHERE 업체명 LIKE '대한%' |
| SELECT *<br>FROM 사원<br>WHERE 월급여 * 12 = 36000000 | SELECT *<br>FROM 사원<br>WHERE 월급여 = 36000000/12 |
| SELECT *<br>FROM 주문<br>WHERE TO_CHAR(일시,'YYYYMMDD') = :DT | SELECT *<br>FROM 주문<br>WHERE 일시>=TO_CHAR(:DT,'YYYYMMDD')<br>AND 일시<TO_CHAR(:DT,'YYYYMMDD')+1 |
| SELECT *<br>FROM 고객<br>WHERE 연령 \|\| 직업 = '30 공무원' | SELECT *<br>FROM 고객<br>WHERE 연령 = 30<br>AND 직업 = '공무원' |
| SELECT *<br>FROM 회원사지점<br>WHERE 회원번호 \|\| 지점번호 = :STR | SELECT *<br>FROM 회원사지점<br>WHERE 회원번호 = SUBSTR(:STR,1,2)<br>AND 지점번호 = SUBSTR(:STR,3,4) |

2) NOT 연산자 사용

NOT 연산자가 있는 경우에는 B*Tree에서 찾은 인덱스 값을 제외한 모든 값이 결과가 되므로 인덱스를 사용하는 것보다 전체 테이블 스캔이 훨씬 유리하다.

3) IS NULL/IS NOT NULL 사용

B*Tree 구조 인덱스는 '아직 정의되지 않은 미지의 값' 인 NULL 컬럼 값을 인덱스에 저장하지 않기 때문에 IS NULL이나 IS NOT NULL조건은 인덱스를 통해 찾을 수 없고, 전체 테이블 스캔 방식을 통해서 찾는다.

4) 옵티마이저의 취사선택

한 테이블의 여러 인덱스가 WHERE절의 조건에 사용된 경우 RBO (Rule-Baced Optimizer)는 우선 순위에 의해, CBO (Cost-Based Optimizer)는

통계치를 기준으로 선택을 하게 된다. 예를 들어 CBO에서 옵티마이저가 비용을 계산하여 만약 FULL TABLE SCAN방식이 인덱스 스캔 방식보다 비용이 더 낮으면 인덱스 스캔 방식으로 액세스하지 않을 것이다.

5) 결합인덱스를 사용 할 때 선행컬럼이 조건절에 없을 경우

어떤 테이블에서 여러 컬럼을 조합하여 결합인덱스가 구성되었는데, SQL문의 WHERE절에서 이 결합인덱스의 대부분 컬럼이 존재하지만 결합인덱스를 구성하는 첫 번째 컬럼이 존재하지 않는다. 이런 경우에 옵티마이저가 인덱스를 통해서 액세스하지 않을 것이다. 강제적으로 인덱스 스캔을 하도록 유도하려면 INDEX SKIP SCAN힌트를 사용하여야 한다.

## 3. 인덱스 선정 전략

일반적으로 아래와 같은 기준에 의해 인덱스의 사용을 결정한다.

1) 분포도가 좋은 컬럼은 단독적으로 생성하여 활용도를 향상(10%~15% 이내) 시킨다.

2) 자주 조합되어 사용되는 컬럼의 경우에는 결합 인덱스의 생성을 고려한다.

두 개 이상의 컬럼이 결합하여 필터 조건이나 조인 조건으로 사용되는 경우에는, 결합 인덱스에 포함된 컬럼에 개별 인덱스가 생성되어 있다면 실행계획이 의도하지 않는 결과를 생성할 수도 있으므로 주의하여야 한다.

3) 인덱스간의 역할을 정의한다.

생성되는 인덱스는 하나의 목적에 맞게 설계되기 보다는 좀 더 범용적

으로 생성되어야 한다. 접근 경로 조사표를 만들어 표에 기재된 모든 액세스 경로 중 결합 인덱스 또는 클러스터 인덱스로 처리될 수 있는 인덱스는 하나로 생성될 수 있다.

4) 수정이 빈번히 일어나지 않는 컬럼을 인덱스로 사용한다.

인덱스도 테이블과 독립된 객체이므로 인덱스 컬럼이 수정된 경우에는 테이블과 더불어 인덱스도 함께 수정되어야 하기 때문이다.

5) 외부키(Foreign Key)로 사용된 컬럼에 대하여 인덱스를 생성한다.

컬럼이 외부키에 포함되어 있고 그것을 위한 인덱스가 구성되어 있지 않다면 테이블의 무결성을 보장하기 위하여 지나친 락(LOCK)이 발생하게 된다.

6) 정렬기준으로 자주 사용되는 컬럼에 인덱스를 생성한다.

인덱스에 의하여 자동으로 정렬된 순서로 자료가 조회되어 정렬 과정이 필요 없어지므로 속도가 향상되고 부분범위처리가 가능한 SQL문에서는 보다 향상된 성능을 기대할 수 있게 된다.

이렇게 인덱스를 추가로 생성 및 변경할 경우에는 인덱스가 새로 추가되어 기존 프로그램의 동작에 영향을 줄 수 있는지를 주의 깊게 검토해야 한다. 최악의 경우 잘못 생성된 인덱스 하나 때문에 시스템이 마비되는 상황에 이를 수도 있기 때문이다.

# III.중첩 루프 조인 성능 분석

중첩루프조인의 수행비용을 산정할 때 다음과 같은 비용을 생각해 볼 수 있다[4].

1) 드라이빙 테이블로부터 필요한 모든 로우들을 읽기 위해 소요되는 비용이 얼마인가?

2) 드라이빙 테이블을 읽고 나서 얼마나 많은 로우들이 리턴될 것인가?

3) 드라이빙 테이블에서 한 로우 전체를 읽고 나면 드리븐 테이블을 액세스하기 위해 필요한 새로운 정보들을 갖게 된다. 이들 정보를 이용해 드리븐 테이블에서 관련된 로우들을 찾고자 할 때, 한번 조인 액세스를 시도할 때마다 얼마만큼의 비용이 소요되는가?

위 세가지 질문을 염두에 두고 중첩루프 조인 비용을 계산하는 아주 간단한 공식을 생각해 낼 수 있다. 그림18과 같다.

---

COST(NLJ) = COST(DRIVING) + CARD(DRIVING) * COST(DRIVEN)

COST(NLJ): 중첩 루프 조인의 총비용
COST(DRIVING): 드라이빙 테이블로부터 데이터를 가져오는 비용
CARD(DRIVING): 드라이빙 테이블로부터 리턴되는 레코드 개수
COST(DRIVEN): 드리븐 테이블을 한번 방문하기 위해 소용되는 비용

---

**그림 18. 중첩 루프 조인 비용 계산 공식[4]**

### 3.1 비용 분석 환경

본 연구에서는 표2과 같은 시스템 환경에서 실험을 진행하였다.

**표 2. 시스템 환경**

| 구분 | 세부내용 |
|---|---|
| DBMS | Oracle 11g Release 2 (32bit) |
| Operating System | Windows 7 Professional K Service Pack 1 |
| Cpu | Intel® Core™ i7 CPU 860 @ 2.80GHz 2.80GHz |
| Memory | 8.00GB |
| User Interface | SQL Plus |

본 연구의 실험을 하기 위해 각각 크기가 다른 스키마에서 실험 결과를 비교할 것이다. 실험 스키마 환경은 표3,4와 같다.

**표 3. 실험 스키마 환경**

| 실험환경 | HR_DEPT 크기(건) | HR_EMP 크기(만건) |
|---|---|---|
| ENV_1 | 500 | 10 |
| ENV_2 | 1000 | 20 |
| ENV_3 | 1500 | 30 |
| ENV_4 | 2000 | 40 |
| ENV_5 | 2500 | 50 |
| ENV_6 | 3000 | 60 |
| ENV_7 | 3500 | 70 |
| ENV_8 | 4000 | 80 |
| ENV_9 | 4500 | 90 |
| ENV_10 | 5000 | 100 |

표 4. 실험 스키마 구성

| TABLE_NAME | COLUMN_NAME | COLUMN_TYPE |
|---|---|---|
| HR_DEPT | DEPARTMENT_ID | NUMBER (4) |
| | DEPARTMENT_NAME | VARCHAR2 (30) |
| | MANAGER_ID | NUMBER (6) |
| | LOCATION_ID | NUMBER (4) |
| HR_EMP | EMPLOYEE_ID | NUMBER (10) |
| | FIRST_NAME | VARCHAR2 (20) |
| | LAST_NAME | VARCHAR2 (25) |
| | EMAIL | VARCHAR2 (25) |
| | PHONE_NUMBER | VARCHAR2 (20) |
| | HIRE_DATE | DATE |
| | JOB_ID | VARCHAR2 (10) |
| | SALARY | NUMBER (8,2) |
| | COMMISSION_PCT | NUMBER (2,2) |
| | MANAGER_ID | NUMBER (6) |
| | DEPARTMENT_ID | NUMBER (4) |

### 3.2 드라이빙 테이블의 액세스 비용 분석

드라이빙 테이블을 액세스 하는 방식에는 전체 테이블 스캔과 인덱스 스캔 두 가지 방식이 있다. 먼저 전체 테이블 스캔이 발생하는 경우는 첫째, 액세스 조건이 없어서 테이블 전체를 읽어와야 하는 경우가 있고 둘째, 액세스 조건이 있고 컬럼의 인덱스가 생성되어 있지만 인덱스의 분포도나 클러스터링 팩터가 나빠 인덱스 스캔을 통한 액세스 비용이 전체 테이블 스캔 비용보다 커 인덱스 스캔이 불리하여 옵티마이저가 전체 테이블 스캔 방식을 선택하는 경우이다. 두 경우에 모두 전체 테이블 스캔을 수행하지만 두 경우의 차이점은 드라이빙 테이블의 카디날리티가 다르다는 것이다. 이 카디날리티가 미치는 영향은 다음 절에서 분석한

다.

전체 테이블 스캔이 아닌 인덱스 스캔을 유발하고자 한다면 반드시 스캔 범위를 줄이는 액세스 조건이 존재하여야 한다. 또한 액세스 조건 컬럼에 있는 인덱스를 통해서 액세스할 때 발생하는 비용이 전체 테이블 스캔보다 유리해야만 옵티마이저는 인덱스 스캔으로 드라이빙 테이블을 액세스하게 된다. 만약 액세스 조건을 적용한 집합의 크기가 커 전체 테이블 스캔을 하는 비용보다 인덱스 스캔을 하는 비용이 크다면 옵티마이저는 전체 테이블 스캔방식을 선택할 것이다. 인덱스 스캔을 할 때 분포도와 클러스터링 팩터 또한 비용산정에 영향을 미치지만 액세스 조건을 적용한 집합이 인덱스 스캔을 유발 할 수 있는 정도로 작다면 전체 테이블 스캔보다 작은 비용이 들게 된다. 그렇기 때문에 드라이빙 테이블에 액세스 조건이 존재한다면 정확한 인덱스 구성 방법에 따라 인덱스를 생성을 해 주고 인덱스 스캔 비용을 낮춰 전체 테이블 스캔방식보다 비용이 적은 인덱스 스캔을 옵티마이저가 선택할 수 있도록 해야 한다.

### 3.3 드라이빙 테이블의 카디날리티 분석

그림18에 있는 공식에서 확인 할 수 있듯이 만약 드라이빙 테이블과 드리븐 테이블을 액세스하는 비용을 최적화할 수 있다면 드라이빙 테이블의 카디날리티가 작아질수록 중첩 루프 조인의 비용을 더 낮출 수 있다. 그렇기 때문에 조인순서를 결정하는 중요한 요소 중의 하나는 카디날리티가 작은 쪽을 드라이빙 테이블로 선정하는 것이다.

앞선 분석과정에서 드라이빙 테이블의 카디날리티는 액세스 조건의 유무에 따라 달라진다고 하였다. 액세스 조건의 존재여부에 따른 카디날리티 및 드라이빙 테이블의 총 로우 수의 관계는 표5과 같이 나타난다.

**표 5. 액세스 조건 유무에 따른 카디날리티 및 드라이빙 테이블 로우 수 관계**

| 액세스 조건 여무 | 카디날리티 및 드라이빙 총 로우 수 관계 |
|---|---|
| 무 | 카디날리티 = 드라이빙 테이블의 총 로우 수 |
| 유 | 카디날리티 < 드라이빙 테이블의 총 로우 수 |

드라이빙 테이블의 카디날리티를 감소시키려면 반드시 액세스 조건이 존재해야 한다는 것은 명확한 사실이다. 이렇듯 액세스 조건은 드라이빙 테이블의 카디날리티를 줄이는 요소일 뿐만 아니라 드라이빙 테이블의 액세스 비용까지 줄이는 중요한 요소이다. 그렇지만 SQL문장에 액세스 조건이 존재여부는 업무 요구사항이 결정한다. SQL문장은 업무 요구사항에 따라 작성되었기 때문에 액세스 조건을 추가 및 삭제하면 업무 요구사항을 정확하게 반영하지 못해 부정확한 결과를 유발할 것이다. 그렇기 때문에 카디날리티를 줄이기 위해 액세스 조건을 추가하는 것은 성능 개선 수단으로 사용할 수 없다.

만약 조인하는 테이블에 모두 액세스 조건이 존재하지 않는 경우라면 다른 조인 방식과 비교하여 볼 때 비용이 높기 때문에 다른 조인 방식을 사용하는 것을 고려하여야 한다. 조인의 연산자 형태에 따른 다른 조인 방식과의 성능 비교 결과는 그림19,20과 같다.

조인 조건이 EQUAL일 때는 해시조인이 더 좋은 성능을 얻었고 조인 조건이 NON-EQUAL일 때는 해시조인을 사용하지 못하고 소트 머지 조인의 성능이 더 좋은 것으로 나타났다.

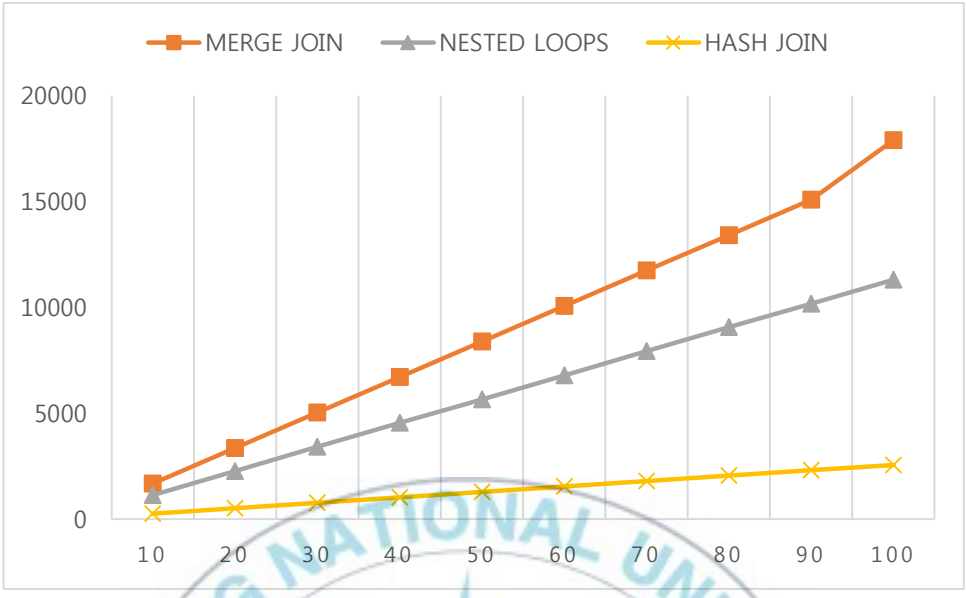



그림 19. 조인 조건이 EQUAL일 때 각 조인 방식의 비용 비교

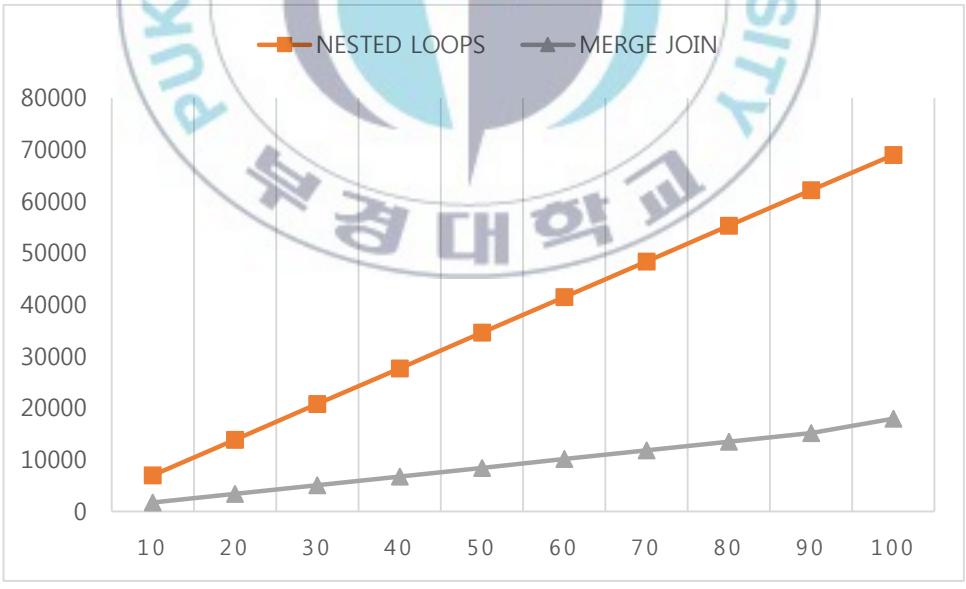



그림 20. 조인 조건이 NON-EQUAL일 때 각 조인 방식의 비용 비교

### 3.4 드리븐 테이블의 액세스 비용 분석

드리븐 테이블의 액세스 비용은 드리븐 테이블을 액세스 하는 방식에 따라 다르다. 드리븐 테이블을 액세스 하는 방식은 연결고리의 상태에 따라 결정되는데 연결고리 상태는 이상 및 정상으로 나눌 수 있다.

#### 3.4.1 연결고리 이상인 경우의 액세스 비용 분석

연결고리 이상인 경우 인덱스를 통해 액세스 하지 못하기 때문에 드리븐 테이블 전체를 스캔하여 해당 데이터를 찾아야 한다. 이러한 경우는 반복적인 전체 테이블 스캔이 발생하게 되어 중첩루프조인의 비용이 커진다.

이런 경우에 발생하는 반복적인 전체 테이블 스캔의 비효율을 증명하기 위하여 그림21와 같은 SQL문장에 대한 실험을 진행하였다.

연결고리 이상:

```
SELECT /*+ LEADING(D) USE_NL(E) */*
  FROM HR_EMP E, HR_DEPT D
 WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
   AND DEPARTMENT_NAME = 'IT';
```

연결고리 정상:

```
SELECT /*+ LEADING(D) USE_NL(E)
           INDEX(E HR_EMP_01B_DEPTID)*/*
  FROM HR_EMP E, HR_DEPT D
 WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
   AND DEPARTMENT_NAME = 'IT';
```

**그림 21. 연결고리 이상 및 정상에 따른 성능 비교 실험 시 사용하는 SQL 문장**

실험 결과는 그림22와 같다.

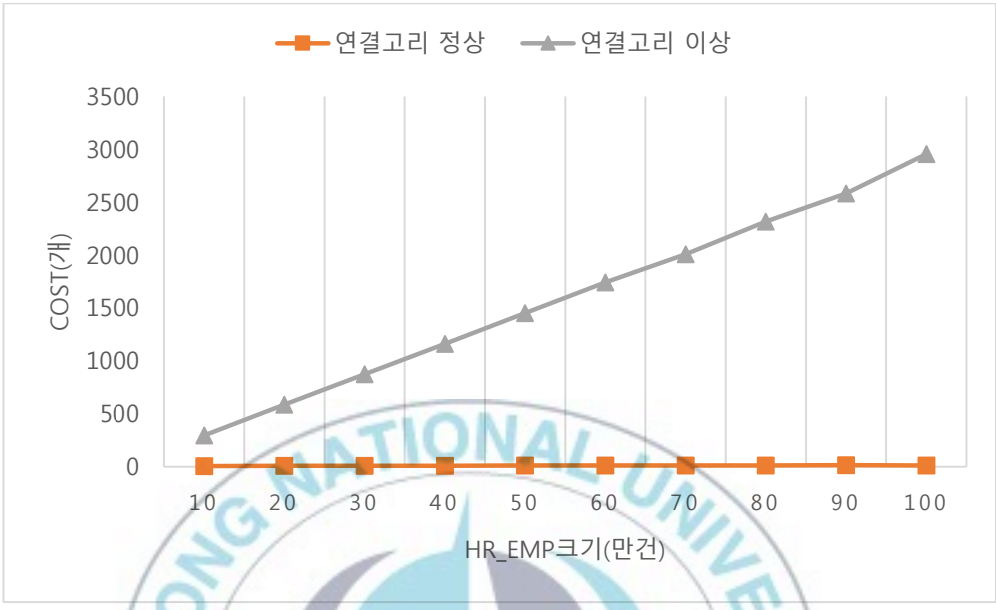



**그림 22. 연결고리 이상 및 정상에 따른 성능 비교 실험 결과**

그림22과 같이 연결고리 정상일 때는 연결고리 이상일 때보다 낮은 비용이 발생하는 것을 알 수 있다. 또한 그림23에서 연결고리 이상인 경우의 실행계획을 확인해 보면 드리븐 테이블에서 전체 테이블 스캔이 발생하였고 연결고리 정상일 때에는 인덱스를 통해서 드리븐 테이블을 액세스하였다.

연결고리 이상인 경우 실행계획

```
-------------------------------------
| Id  | Operation          | Name   |
-------------------------------------
|   0 | SELECT STATEMENT   |        |
|   1 |  NESTED LOOPS      |        |
|   2 |   TABLE ACCESS FULL| HR_DEPT|
|*  3 |   TABLE ACCESS FULL| HR_EMP |
-------------------------------------
```

연결고리 정상인 경우 실행계획

```
-----------------------------------------------------
| Id  | Operation                    | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |              |
|   1 |  NESTED LOOPS                |              |
|   2 |   NESTED LOOPS               |              |
|   3 |    TABLE ACCESS FULL         | HR_DEPT      |
|*  4 |    INDEX RANGE SCAN          | HR_EMP_DEPTID|
|   5 |   TABLE ACCESS BY INDEX ROWID| HR_EMP       |
-----------------------------------------------------
```

그림 23. 연결고리 이상 및 정상에 따른 성능 비교 실험 실행계획

### 3.4.2 연결고리 정상인 경우의 액세스 비용 분석

연결고리 정상 상태라면 인덱스를 통해서 액세스하게 되고 이런 경우 분포도와 클러스터링 팩터처럼 인덱스의 성능에 영향을 미치는 요소들이 드리븐 테이블의 액세스 비용을 결정한다.

또한 조인 순서 및 조인조건의 연산자 형태에 따라서 인덱스 스캔 방식은 INDEX UNIQUE SCAN과 INDEX RANGE SCAN으로 나뉜다.

조인의 순서와 연산자 유형에 따른 인덱스 스캔방식은 표7와 같다.

표 6. 조인의 순서와 연산자 유형에 따른 인덱스 스캔방식

| 조인 순서 | 연산자 유형 | 인덱스 스캔 방식 |
|---|---|---|
| 1→M | Don't Care | INDEX RANGE SCAN |
| M→1 | NON EQUAL | INDEX RANGE SCAN |
| M→1 | EQUAL | INDEX UNIQUE SCAN |

분포도는 1/컬럼값의 종류수로 계산되는데 컬럼값의 종류수는 바꿀 수 있는 값이 아니다. 하지만 드리븐 테이블의 액세스 조건이 존재한다면 결합 인덱스를 생성해 줌으로써 분포도를 개선할 수 있다.

그렇기 때문에 드리븐 테이블의 조인 조건과 액세스 조건이 함께 존재한다면 결합인덱스를 사용하여 분포도를 개선함으로써 드리븐 테이블의 액세스 비용을 줄여 중첩루프 조인의 비용을 줄일 수 있다. 물론 정확한 순서로 결합 인덱스를 구성하여야 한다. 분포도를 향상시킴으로써 액세스 비용을 줄일 수 있는 이유는 인덱스 스캔 과정에서 범위를 줄여 인덱스를 통해 드리븐 테이블을 액세스할 때 발생하는 랜덤 액세스를 줄이기 때문이다.

결합인덱스를 사용하여 분포도를 높임으로 중첩 루프 조인의 성능을 개선할 수 있는지를 실험하기 위하여 그림24와 같은 실험을 하였다.

단일 인덱스: HR_EMP(DEPARTMENT_ID)

```
SELECT /*+LEADING(D) USE_NL(E)
         INDEX(E HR_EMP_DEPTID)*/*
  FROM HR_EMP E, HR_DEPT D
 WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
   AND D.MANAGER_ID > 200
   AND JOB_ID = 'IT_PROG';
```

결합 인덱스: HR_EMP(DEPARTMENT_ID, JOBID)

```
SELECT /*+LEADING(D) USE_NL(E)
         INDEX(E HR_EMP_DEPTID_JOBID)*/*
  FROM HR_EMP E, HR_DEPT D
 WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
   AND D.MANAGER_ID > 200
   AND JOB_ID = 'IT_PROG';
```

그림 24. 결합인덱스로 성능개선 시 사용하는 SQL문장

실험 결과는 그림25와 같다.

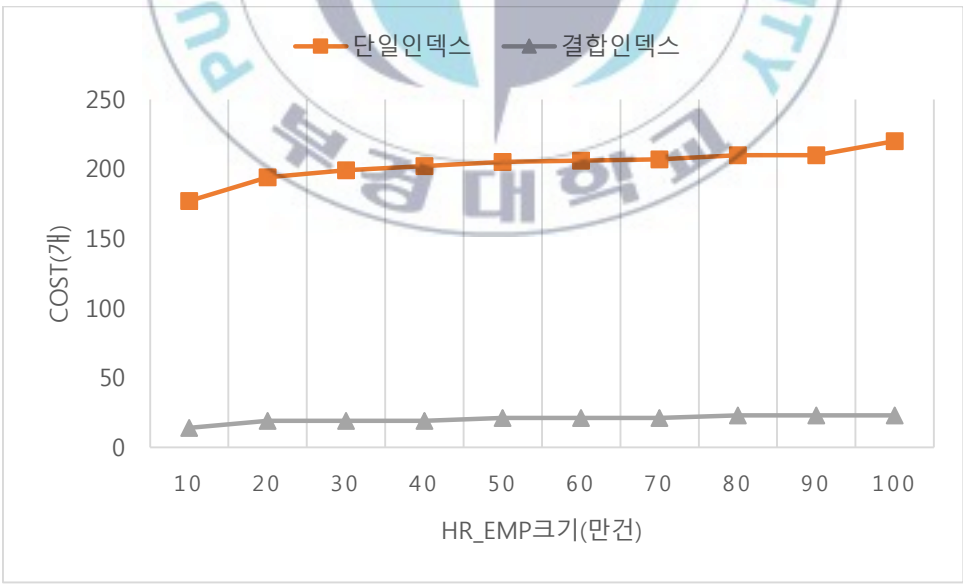



그림 25. 결합인덱스 사용에 따른 성능 변화

그림25와 같이 결합인덱스를 사용할 때 단일인덱스를 통해서 조인한 것보다 좋은 성능이 나타났다.

인덱스의 성능에 영향을 미치는 또 다른 요소인 클러스터링 팩터는 인덱스를 통해 테이블을 액세스 할 때 읽는 테이블의 블록 갯수이다. 클러스터링 팩터가 좋아진다면 인덱스를 통해서 읽어오는 블록의 수가 적어지고 이로 인해 드리븐 테이블의 액세스 비용을 줄일 수 있다.

클러스터링 팩터가 미치는 영향은 인덱스 스캔방식에 따라 다를 수 있다. 만약 INDEX UNIQUE SCAN인 경우는 관계의 기수성이 M쪽인 테이블로부터 1쪽으로 조인이 될 때 발생하는데 M쪽의 각 값마다 조인 되는 로우는 드리븐 테이블에 하나의 값이 존재하기 때문에 조인을 시도할 때 드라이빙 테이블의 모든 로우는 똑같이 하나의 블록만 읽어오면 된다. 그렇기 때문에 INDEX UNIQUE SCAN이 발생할 때는 클러스터링 팩터의 영향을 받지 않는다.

클러스터링 팩터를 개선함으로 중첩 루프 조인의 성능을 개선할 수 있는지를 실험하기 위하여 그림26와 같은 실험을 하였다.

```
SELECT /*+LEADING(D) USE_NL(E)
         INDEX(E HR_EMP_DEPTID)*/*
  FROM HR_EMP E, HR_DEPT D
 WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
   AND D.MANAGER_ID > 200
   AND JOB_ID = 'IT_PROG';
```

**그림 26. 클러스터링 팩터에 따른 성능 비교 시 사용하는 SQL문장**

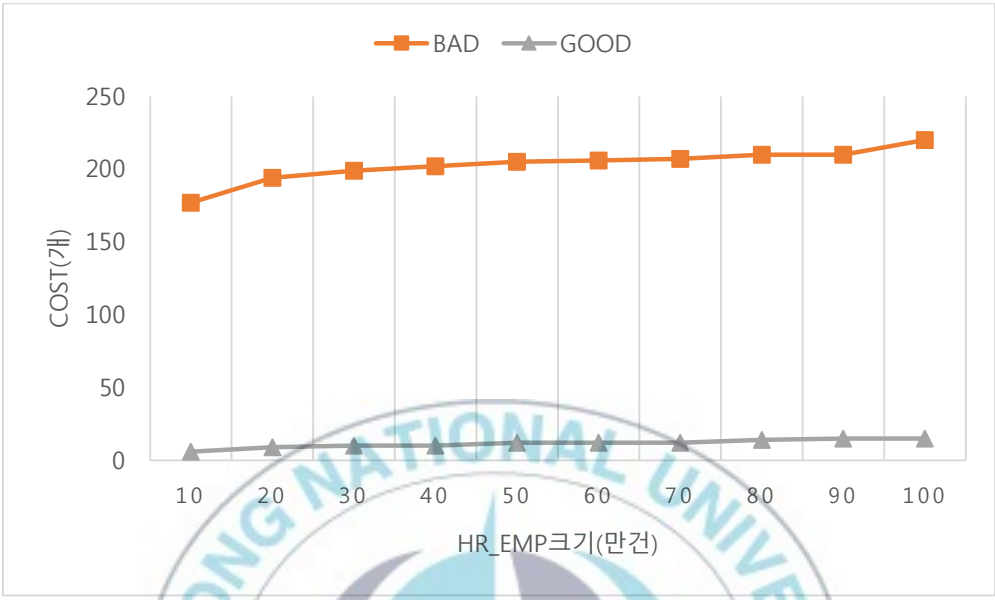



그림 27. 클러스터링 팩터에 따른 성능 비교

그림27에서 나오는 결과와 같이 클러스터링 팩터를 개선시킨 후에 보다 더 낮은 성능으로 조인한 결과가 나타났다.

### 3.5 중첩 루프 조인 성능 분석 결과

앞서 각 단계별 중첩 루프 조인에 영향을 미치는 요소들을 분석하였다. 이 요소들을 중첩 루프 조인이 진행하는 순서에 따라 정리 하여 그림14과 같은 분석결과를 도출 하였다.

먼저 중첩 루프 조인의 성능은 액세스 조건 유무에 따라 달라질 수 있다. 이 사실은 드라이빙 테이블의 카디날리티에 영향이 결정하는 것이다. 또한 드라이빙 테이블의 카디날리티가 중첩 루프 조인의 전체 성능

에 영향을 미치기 때문에 중첩 루프 조인의 성능을 고려할 때 가장 먼저 고려해야 되는 요소이다. 그 다음 중첩 루프 조인의 성능은 연결고리 상태에 따라서 다르다. 연결고리 이상 상태라면 드라이빙 테이블의 카디날리티 만큼 드리븐 테이블에서 반복적인 전체 테이블 스캔이 발생하다. 연결고리 정상이면 인덱스를 통해 드리븐 테이블을 액세스하기 때문에 연결고리 컬럼의 분포도에게 영향을 받는다. 또한 만약 INDEX RANGE SCAN발생한다면 클러스터링 팩터의 수치에 따라 성능이 달라진다.

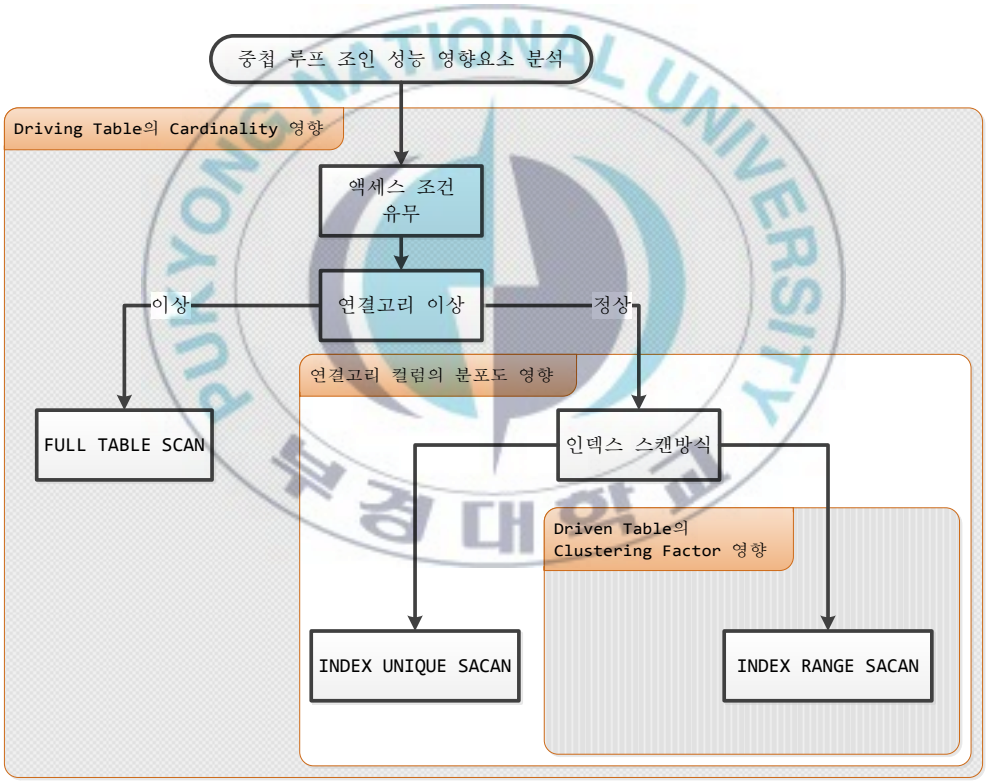



**그림 28. 중첩 루프 조인 성능 분석 결과**

그림28과 같은 분석결과를 통해서 중첩 루프 조인을 하는 과정 중에

어떤 단계에 어떤 요소들의 영향을 받는지 찾을 수 있고 정확하게 성능 저하의 요인을 찾을 수 있다. 이 분석 결과는 중첩 루프 조인의 성능을 개선을 하기 위한 자료로 사용될 것이다.

# IV. 중첩 루프 조인 성능 개선 절차

그림28의 중첩 루프 조인 성능 분석결과에 의해 순차적으로 중첩 루프 조인 성능에 영향을 미치는 요소들을 개선을 시켜 준다면 중첩 루프 조인의 성능을 개선할 수 있다. 그렇기 때문에 실험을 통해서 성능 개선 절차를 찾고자 한다.

## 4.1 중첩 루프 조인 성능 개선 실험

### 4.1.1 실험 환경

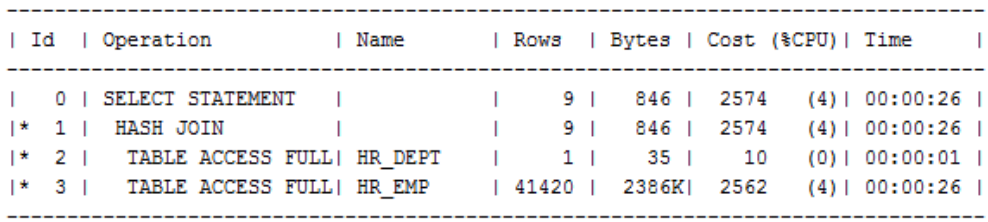
본 연구에서 제안하는 중첩 루프 조인 성능 개선 절차에 따라서 성능을 개선할 수 있는지를 검증 하기 위해 HR_DEPT테이블에 5027건의 데이터를 저장하고 있고 HR_EMP테이블에 100만건의 데이터를 저장하고 있는 환경에서 실험하였다. 성능을 개선하기 위하여 사용하는 SQL문장은 표29과 같다.

```
SELECT *
  FROM HR_EMP E , HR_DEPT D
 WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
   AND D.DEPARTMENT_NAME = 'IT'
   AND E.HIRE_DATE >= TO_DATE('19000101','YYYYMMDD');
```

그림 29. 중첩 루프 조인 성능개선 절차 적용 실험 시 SQL문장

### 4.1.2 성능 개선 절차 실험

성능 개선 절차를 적용하기 전의 SQL문 실행 계획은 그림30와 같다.



| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
|---|---|---|---|---|---|---|
| 0 | SELECT STATEMENT | | 9 | 846 | 2574 (4) | 00:00:26 |
| * 1 | HASH JOIN | | 9 | 846 | 2574 (4) | 00:00:26 |
| * 2 | TABLE ACCESS FULL | HR_DEPT | 1 | 35 | 10 (0) | 00:00:01 |
| * 3 | TABLE ACCESS FULL | HR_EMP | 41420 | 2386K | 2562 (4) | 00:00:26 |

그림 30. 성능 개선 절차 적용 전 실행계획

실행 계획을 보면 중첩 루프 조인을 사용하지 않았다. 이는 옵티마이저가 내부적으로 계산을 통해서 해시조인의 비용이 더 낮다고 판단했기 때문이다. 그래서 본 연구에서 제안하는 성능 개선 절차를 적용하여 성능 개선을 하였다.

### 1. 액세스 조건 유무확인

그림29에 있는 SQL를 확인해 보면 HR_EMP 과 HR_DEPT 테이블 모두 액세스 조건이 존재한다. 그렇기 때문에 다음 단계인 조인 순서 선정 단계에 들어간다.

### 2. 조인 순서 선정

각 테이블에 해당 액세스 조건을 적용하여 카디날리티를 구한 후에 수치를 비교한다.

표 7. 조인 순서 선정 하기 위한 SQL문장으로 카디날리티를 구함

| 적용 SQL문장 | 조회된 결과 |
|---|---|
| SELECT COUNT(*)<br>FROM HR_DEPT D<br>WHERE D.DEPARTMENT_NAME = 'IT'; | 1 |
| SELECT COUNT(*)<br>FROM HR_EMP_10A E<br>WHERE E.HIRE_DATE >= TO_DATE('19000101','YYYYMMDD'); | 107 |

표11와 같이 조회된 결과를 비교하며 카디날리티가 작은 쪽인 HR_DEPT

테이블을 드라이빙 테이블로 선정한다.

### 3. 드라이빙 테이블 엑세스 비용 개선

조인 순서를 선정한 후 HR_DEPT테이블의 액세스 조건 컬럼의 분포도를 조사하여 인덱스를 생성해 주어 더 좋은 성능을 얻을 수 있다면 이 컬럼에 인덱스를 생성해 주어야 한다. 분포도가 양호하고 인덱스를 통한 액세스비용이 낮다고 판단하였기 때문에 HR_DEPT테이블의 DEPARTMENT_NAME 컬럼에 인덱스를 생성해 주었다. 인덱스를 생성해 준 후의 실행계획은 그림31와 같다.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |     9 |   846 |  2566   (4)| 00:00:26 |
|*  1 |  HASH JOIN                   |                   |     9 |   846 |  2566   (4)| 00:00:26 |
|   2 |   TABLE ACCESS BY INDEX ROWID| HR_DEPT           |     1 |    35 |     2   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | HR_DEPT_DEPTNM    |     1 |       |     1   (0)| 00:00:01 |
|*  4 |   TABLE ACCESS FULL          | HR_EMP            | 41420 |  2386K|  2562   (4)| 00:00:26 |
-----------------------------------------------------------------------------------------------
```

**그림 31. 드라이빙 테이블 액세스 비용 개선 후 실행계획**

그림31과 같이 HR_DEPT테이블을 액세스할 때 인덱스를 사용하게 되었고 비용 또한 조금 줄어들었다. 그렇지만 아직도 해시조인을 사용하고 있는 것을 확인할 수 있다. 이는 아직도 해시조인의 성능이 더 좋다는 것이다.

### 4. 연결고리 상태 확인

앞에 단계에서 조인순서를 선정하였고 이 단계에서 드리븐 테이블의 연결고리 컬럼에 인덱스가 생성되어 있는지를 확인하여야 한다. 생성되지 않았다면 연결고리 정상상태로 개선 하기 위해 인덱스를 생성해 주어야 한다. 그래서 HR_EMP테이블의 DEPARTMENT_ID컬럼에 단일 컬럼 인덱스를 생성하여 실행한 후의 실행 계획은 그림32와 같다.

| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
|---|---|---|---|---|---|---|
| 0 | SELECT STATEMENT | | 9 | 846 | 201 (0) | 00:00:03 |
| 1 | NESTED LOOPS | | | | | |
| 2 | NESTED LOOPS | | 9 | 846 | 201 (0) | 00:00:03 |
| 3 | TABLE ACCESS BY INDEX ROWID | HR_DEPT | 1 | 35 | 2 (0) | 00:00:01 |
| * 4 | INDEX RANGE SCAN | HR_DEPT_DEPTNM | 1 | | 1 (0) | 00:00:01 |
| * 5 | INDEX RANGE SCAN | HR_EMP_DEPTID | 199 | | 2 (0) | 00:00:01 |
| * 6 | TABLE ACCESS BY INDEX ROWID | HR_EMP | 9 | 531 | 199 (0) | 00:00:02 |

그림 32. 단일 인덱스로 연결고리 이상을 제거한 후 실행계획

그림 32 를 보면 단일 인덱스를 사용하여 연결고리 정상 상태로 개선한 후의 COST 비용은 전 단계의 2566 에서 201 로 줄어든 것을 확인할 수 있다.

## 5. 드리븐 측 액세스 조건 유무 확인

이 단계에서는 먼저 드리븐 측 액세스 조건이 존재하는지를 판단한다. 드리븐 테이블에 액세스 조건이 존재하기 때문에 연결고리 컬럼과 액세스 조건 컬럼을 정확히 조합하여 결합인덱스를 생성해 준다. 결합인덱스를 생성한 후에 다시 실행한 후의 실행 계획은 그림 33 와 같다.

| Id | Operation | Name | Rows | Bytes | Cost (%CPU) | Time |
|---|---|---|---|---|---|---|
| 0 | SELECT STATEMENT | | 9 | 846 | 14 (0) | 00:00:01 |
| 1 | NESTED LOOPS | | | | | |
| 2 | NESTED LOOPS | | 9 | 846 | 14 (0) | 00:00:01 |
| 3 | TABLE ACCESS BY INDEX ROWID | HR_DEPT | 1 | 35 | 2 (0) | 00:00:01 |
| * 4 | INDEX RANGE SCAN | HR_DEPT_DEPTNM | 1 | | 1 (0) | 00:00:01 |
| * 5 | INDEX RANGE SCAN | HR_EMP_DEPTID_HIREDT | 9 | | 2 (0) | 00:00:01 |
| 6 | TABLE ACCESS BY INDEX ROWID | HR_EMP | 9 | 531 | 12 (0) | 00:00:01 |

그림 33. 결합인덱스를 적용한 후 실행계획

그림 33 의 실행계획에서 확인할 수 있듯이 결합인덱스를 생성해 주고 나서 랜덤 액세스가 더 줄어들었기 때문에 COST 비용이 전 단계보다 낮아진 것을 확인할 수 있다.

### 6. 인덱스 스캔 방식 확인

앞 단계의 실행계획에서 드리븐 테이블을 액세스하였을 때 INDEX RANGE SCAN 방식을 사용한 것을 확인할 수 있다. 더 좋은 성능으로 개선시키기 위해서는 클러스터링 팩터를 개선해야 된다. 클러스터링 팩터가 개선하기 전에 889603 이었고 개선한 후에 32994 이 된다.

클러스터링 팩터를 개선한 후의 실행계획은 그림 34 와 같다.

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                      |     9 |   846 |     5   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                      |       |       |            |          |
|   2 |   NESTED LOOPS               |                      |     9 |   846 |     5   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| HR_DEPT              |     1 |    35 |     2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN         | HR_DEPT_DEPTNM       |     1 |       |     1   (0)| 00:00:01 |
|*  5 |    INDEX RANGE SCAN          | HR_EMP_DEPTID_HIREDT |     9 |       |     2   (0)| 00:00:01 |
|   6 |   TABLE ACCESS BY INDEX ROWID| HR_EMP               |     9 |   531 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

**그림 34. 클러스터링 팩터를 개선 후 실행계획**

앞 단계의 실행계획과 비교하여 전체 비용이 한 층 더 낮아진 것을 확인할 수 있다.

### 4.2 중첩 루프 조인 성능 개선 절차 제안

중첩 루프 조인 성능 분석에 결과를 의하여 성능 개선 실험을 하였다. 실험의 절차를 정리하여 그림35과 같은 중첩 루프 조인 성능 개선 절차를 제안하고자 한다.

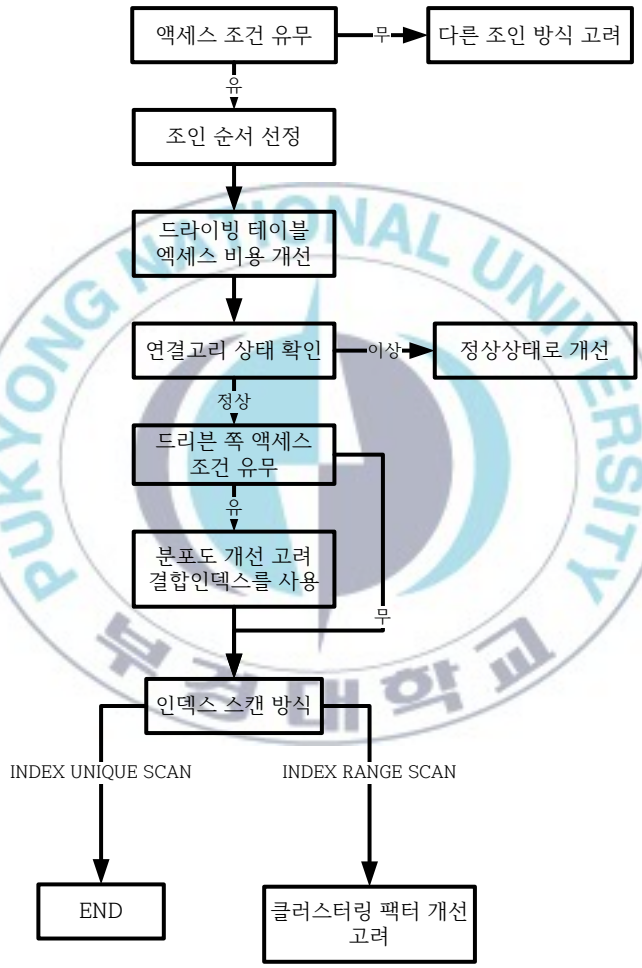



**그림 35. 중첩 루프 조인 성능개선 절차**

먼저 액세스 조건의 유무를 확인함으로써 중첩 루프 조인의 사용 여부

를 판단할 수 있고 중첩 루프 조인을 사용할 때 조인순서를 어떻게 선정해야 하는지를 판단할 수도 있다. 하지만 액세스 조건이 존재하지 않더라도 비용이 더 큰 중첩루프조인을 사용하지 않는 것은 아니다. 부분범위처리를 사용하여 빠른 응답으로 결과를 출력해야 하는 경우라면 액세스 조건이 없더라도 중첩 루프 조인을 사용해야 한다. 비록 조인의 비용이 크더라도 부분범위처리를 이용할 수 있는 중첩루프조인을 사용할지 아니면 다른 조인방식을 사용할지 또한 신중하게 고려해봐야 할 사항이다. 이런 선택에 관한 연구는 본 연구의 범위에서 벗어나므로 더 이상 다루지 않기로 한다.

액세스 조건 유무를 확인한 후 조인순서를 선정한다. 카디날리티가 작은 쪽을 드라이빙 테이블로 선정하여야 하는데 액세스 조건이 있다면 액세스 조건을 적용한 후의 카티날리티가 작은 쪽을 드라이빙 테이블로 선정한다.

다음 단계는 연결고리 상태를 확인하여 연결고리 이상이라면 연결고리 정상으로 개선하여야 성능을 높일 수 있다. 연결고리 정상으로 개선한 후 드리븐 테이블 쪽에 액세스 조건이 존재하는지를 확인 해야 한다. 만약 드리븐 테이블 쪽에 액세스 조건이 존재하지 않는다면 결합인덱스를 통해서 분포도를 개선할 수 없기 때문이다. 이 단계에서 드리븐 테이블 쪽에서 액세스 조건이 없다고 하면 그 다음 단계로 넘어가 인덱스 스캔 방식을 판단하게 되는 것이다. INDEX UNIQUE SCAN이 발생하면 더 이상 성능을 개선할 수 없고 만약 이 때 발생하는 비용이 만족스럽지 않다면 다른 조인 방식을 고려한다. INDEX RANGE SCAN이 발생하면 드리븐 테이블의 연결고리 컬럼에 있는 인덱스의 클러스터링 팩터를 개선할 수 있다면 클러스터링 팩터를 개선하는 것을 고려할 수 있다.

연결고리 정상으로 개선한 후에 드리븐 테이블 쪽에 액세스 조건이 존

재한다는 것이 확인되면 그 다음 단계에서 분포도를 개선하도록 결합인덱스를 정확한 순서대로 생성해 줘야 한다. 생성한 후에 인덱스 스캔 방식을 확인하여 만약에 INDEX UNIQUE SCAN이 발생하면 더 이상 개선할 것이 없다. 만약 INDEX RANGE SCAN이 발생하면 드리븐 테이블의 연결고리 컬럼에 있는 인덱스의 클러스터링 팩터를 개선할 수 있는지를 검토를 한다.

이러한 절차에 따라서 중첩 루프 조인의 성능을 최적으로 개선할 수 있다. 만약에 이런 절차에 따라 개선한 후에도 만족하지 않거나 더 나은 성능을 원한다면 다른 조인 방식을 사용하거나 다른 기능들을 활용하는 것을 고려한다.

# V. 결론

기업의 경영활동에 있어 정보의 수집, 축적, 가공 및 효과적인 활용은 필수적이다. 점차 대형화, 복잡화가 되어가는 기업의 정보시스템에서 관계형 데이터베이스는 데이터의 관리와 활용에 있어 기존의 파일시스템과 비교할 때 매우 간편하고 강력한 기능을 제공한다. 그러나, 데이터베이스의 성능은 활용하는 수준에 따라 구형된 성능의 편차가 심하게 나타나며, 대부분의 성능저하는 사용자들은 관계형 데이터베이스의 원리를 정확히 이해하지 못하기 때문이다. 관계형 데이터베이스의 동작원리 및 메커니즘에 대한 깊은 이해는 데이터 베이스 사용 및 튜닝을 정확하고 효율적으로 진행할 수 있는 바탕이 된다. 특히 데이터 베이스 튜닝을 하는 데에 있어서 많은 이론 지식도 필요하고 체계적은 절차도 필요하다.

본 논문에서는 관계형 데이터베이스의 세가지 조인 방식 중에 중첩 루프 조인을 수행하는 과정에서 비용을 일으키는 요인들을 실험을 통해서 분석하며, 중첩 루프 조인 수행하는 과정 중에서 이러한 요인들이 영향을 미친다는 사실을 실험을 통해 증명하였다. 그 다음 이러한 요소들을 중첩 루프 조인의 수행과정을 맞추어 영향을 미치는 범위 및 중요도를 정리하여 그래프 형태로 나타냈다. 이를 바탕으로 중첩 루프 조인의 성능을 개선 할 수 있는 절차를 제안하였다.

기존에 영향 요소들만 분석하고 부분별 성능을 개선하는 실무 경험적인 튜닝 방법보다 본 논문에서 제안하는 방법은 절차적이고 철저하다. 또한 데이터 베이스 튜닝을 배우고 있는 입문자들에게도 이런 절차에 따라 쉽게 중첩 루프 조인의 수행과정 및 성능 저하요인들을 찾아낼 수 있다. 게다가 데이터베이스를 전문적으로 다루지 않은 개발자들에게도 이러한 절차를 알고 있으면 프로그램을 만들 때부터 성능저하 문제가 유발

되는 요인들을 고려하여 올바른 SQL문장을 사용하는데 도움이 될 것이다. 이로 인해 프로그램의 품질도 향상될 것으로 예상되고 그렇기 때문에 본 논문에서 제안하는 중첩 루프 조인 성능 개선 절차는 실무에서도 큰 도움을 가져올 수 있을 것으로 기대한다.

# VI. 참고문헌


[1] Dan Tow, "SQL Tuning”, O’Reilly, 2003

[2] Dbguide.net, www.dbguide.net

[3] Huaiyuan Tan, "How to Make Your Oracle Fly”, Publishing House of Electronics Industry, 2010

[4] Jonathan Lewis, "Cost-Based Oracle Fundamentals”, 사이텍미디어, 2010

[5] Mishra, Chaitanya, "Join Reordering by Join Simulation”, IEEE 25th International Conference, 2009

[6] Wei Huang, "Oracle High Performance SQL Engine SQL Optimization and Tuning", China Machine Press, 2013

[7] 강남이, 권지윤, 김남훈 김윤성, 박중건, 양용열, 정지혜, 한승란, "실전사례로 살펴보는 SQL튜닝 비법", 인사이트, 2013

[8] 권순용, "실행 계획으로 배우는 고성능 데이터베이스 튜닝”, 러닝스페이스, 2009

[9] 김신일, "SQL의 튜닝 방법에 관한 연구", 한남대학교 정보산업대학원, 2005

[10] 김양진,주복규, "정보시스템 성능 향상을 위한 SQL 튜닝 기법",한국인터넷방송통신학회 눈문지, Vol.10 No3, pp.27-33, 2010

[11] 신명주, 김용성, "관계형 데이터베이스에서 테이블 연산시 효율 향상을 위한 성능 평가 방법”, 한국정보과학회 학술발표논문집,

Vol.30 No.1A, pp.731-733, 2003

[12] 안지현, "웹 기반 시스템 성능 향상을 위한 관계형 데이터베이스 SQL튜닝 연구", 성균관대학교 정보통신대학원, 2007

[13] 이영훈, 김교중, "대용량 데이터베이스의 트레이스분석을 통한 개선된 튜닝방법", 한국정보기술학회눈문지, Vol.3 No.1, 2005

[14] 이원조, "관계형 데이터베이스에서 셀프조인 쿼리를 위한 성능평가", 연구논문집, 제28권 제2호, pp.33-36, 2001

[15] 이화식, "대용량 데이터베이스 솔루션 Ⅰ" 대칭, 1996

[16] 이화식, "새로운 쓴 대용량 데이터베이스 솔루션Ⅰ", 엔코아 컨설팅, 2008

[17] 이화식, 조광원, "대용량 데이터베이스 솔루션 Ⅱ",대칭, 1998

[18] 정미영, "SQL 튜닝을 이용한 처리속도 향상 요소에 고찰 및 성능 측정", 홍익대학교 정보대학원, 2006

[19] 조동욱, "Optimizing Oracle Optimizer", ㈜엑셈, 2008

[20] 조시형, "오라클 성능 고도화 원리와 해법 Ⅱ", 비투엔 컨설팅, 2011

[21] 최송희, "SQL튜닝을 이용한 데이터베이스 시스템 성능개선에 관한 연구", 경희대학교 산업정보대학원, 2003

[22] 한국데이터베이스진흥원, "SQL전문가 가이드", ㈜크리홍보, 2012