제 19 회 한국정보처리학회 춘계학술발표대회 논문집 제 10 권 제 1 호 (2003. 5)

이동체 데이터베이스를 위한 메인 메모리 색인의 성능 결정 요소에 관한 연구

이 창우 O, 안 경환, 홍 봉희
부산대학교 컴퓨터공학과
e-mail : {cwlee, khan, bhhong}@pusan.ac.kr

A Study of Performance Decision Factor for Moving Object Database in Main Memory Index

Chang-Woo Lee O, Kyoung-Hwan Ahn, Bong-Hee Hong
Dept. of Computer Engineering, Pusan National University

요 약

이동 통신 기술의 발달로 인하여 무선 이동 기기의 사용이 보편화되면서 LBS(Location Based System)의 요구가 나날이 증대되고 있다. 이와 같은 위치 기반 서비스에서 클라이언트인 이동체들은 일정한 보고 주기를 가지고 서버에 위치 데이터를 보고하게 되는데, 빈번한 보고 데이터를 실시간으로 처리하기 위해서 서버에서는 메인 메모리 DBMS 를 유지하는 것이 필요하다. 기존에 제시된 메인 메모리 색인으로는 T-tree 가 있는데, 이는 1 차원 데이터를 위한 것이므로 이동체 데이터베이스 환경에 적합하지 못하다. 그리고, 디스크 기반의 다차원 색인으로는 R-tree 계열이 있는데, 이는 메인 메모리에서 효율적인 사용을 보장하지 못한다.

이 논문에서는 이동체 데이터베이스 환경에 적합한 메인 메모리 색인을 고려함에 있어서, 기존의 디스크 기반의 다차원 색인으로 가장 널리 알려진 R-tree 계열의 색인을 메인 메모리에 적재 후 메인 메모리 환경에서 성능에 영향을 주는 요소를 실험을 통하여 제시한다. 실험은 메인 메모리에서는 간단한 알고리즘을 사용하는 것이 성능에 좋고, 삽입 시에는 삽입할 노드를 찾기 위해서 비교하는 엔트리의 수가, 검색 시에는 노드간의 중첩으로 인하여 비교하는 노드의 수와 엔트리의 수가 성능에 영향을 주는 요소임을 보여준다.

1. 서론

최근 이동 통신 기술의 발달로 인하여 휴대폰, PDA 등과 같은 무선 이동 기기의 사용이 보편화되면서 GPS(Global Positioning System)기술을 이용한 위치 기반 서비스(LBS: Location Based Service)의 요구가 증가하고 있다. 휴대폰, PDA 등의 무선 이동 기기를 소지한 개인 또는 차량이 GPS 수신기를 사용하여 서버에 일정한 시간 간격으로 위치 데이터를, 필요 시에 검색을 위한 질의를 보낸다. 서버에서는 다수의 클라이언트가 주기적으로 보내는 이동체의 위치 데이터를 효율적으로 처리하기 위한 색인이 필요하다.

그러나 기존의 시공간 색인은 디스크 기반의 색인이 대부분이므로 이동체 데이터베이스 환경에 적용할 경우, 이동체의 수가 많아지거나 데이터의 보고되는 회수가 많아질 경우 서버의 병목 현상(*bottle neck*)으로 인하여 이를 실시간으로 처리할 수 없다. 과거와는 달리, 메인 메모리의 가격 하락, 크기 증대, 안정성 제공, 그리고 가장 중요한 클라이언트의 요구에 빠르게 응답할 수 있다는 점으로 인해 메인 메모리 DBMS 가 새롭게 제기되고 있다. 따라서 이동체 데이터베이스 환경을 위한 메인 메모리 색인이 필요하다.

메인 메모리 색인으로는 T-tree[1]가 가장 널리 알려져 있는데, 이는 1 차원 데이터를 위한 것으로, 다차원인 이동체 데이터베이스 환경에서는 적합하지 않다. 기존의 디스크 기반의 색인은 디스크 환경을 고려하여 만들어졌기 때문에, 그대로 메인 메모리에 적용할 경우 효율적인 사용을 보장하지 못한다. 따라서 메인 메모리 환경에서 성능에 영향을 주는 중요한 요소를 분석하는 것이 필요하다.

이 논문에서는 GPS 기기를 가진 이동체가 주기적으로 자신의 위치 정보를 서버로 보고하는 환경을 대상으로 하고 있다. 서버는 주기적으로 들어오는 이동체의 위치 데이터를 실시간으로 업데이트 해야 한다. 반면에 검색은 실시간이라기 보다는 최대한 빠르게 처리를 하는 것을 목적으로 한다. 즉 보통의 시공간 색인과는 다르게, 검색 시간과 함께 삽입 시간도 중요하다.

이 논문에서는 이러한 이동체 데이터베이스 환경에 적용하기 위해서 메인 메모리 DBMS 를 사용하는 경우에, 기존의 디스크 기반의 색인을 메인 메모리에 적재 후, 여러 가지 파라미터를 이용하여 성능을 평가한다. 그런 후에, 메인 메모리 기반에서 성능을 결정짓는 요소를 알아보고자 하는 데 있다.

이 논문의 구성은 다음과 같다. 먼저 2 장에서는 관련 연구를 소개하고, 3 장에서는 디스크 기반의 색인과 메인 메모리 기반의 색인의 차이점을 설명한다. 4 장에서는 메인 메모리 환경에서 고려해야 할 요소에 대해서 기술하고, 5 장에서는 실험을 통하여 영향을 주는 요소들을 설명한다. 마지막으로 6 장에서는 결론 및 향후 연구를 기술한다.

2. 관련 연구

기존의 색인에는 크게 메인 메모리 기반의 색인과 디스

크 기반의 색인으로 나눌 수 있다. 가장 널리 알려진 메인 메모리 기반의 색인으로는 T-tree[1]가 있고, 디스크 기반의 색인으로는 R-tree[2], R*-tree[3]가 있다.

[1]에서 T-tree 는 기존의 B-tree 와 AVL-tree 로부터 진화되어 나온 색인이다. T-tree 는 이진 검색과 높이 균형을 가지는 AVL-tree 의 성질을 가지고 있고, 한 노드안에 여러 개의 데이터를 가지는 B-tree 의 성질을 가지고 있다. 이러한 성질로 인하여 빠른 처리 속도와 메모리 사용의 최적화라는 메인 메모리의 특성에 적합한 구조로 알려져 있다. 그러나, T-tree 는 1 차원 데이터에만 적용 가능한 색인으로서 다차원인 이동체 데이터베이스 환경에서는 정렬(sorting)의 어려움으로 인하여 적용하기가 어렵다.

[2], [3]에서 R-tree 와 R*-tree 는 저장되는 데이터 객체를 최소 경계 박스(MBB: Minimum Bounding Box)로 표현하며 B-tree 에 대해서 k 차원으로 확장한 모델이다. 다차원 데이터에 대하여 높은 성능을 나타내며, 디스크에 적합한 구조로 알려져 있다. 그러나 R-tree 계열은 디스크의 특성을 고려한 디스크 기반의 색인으로서 메인 메모리 DBMS 에서는 최적의 성능을 보장할 수 없다.

3. 디스크 기반 색인 VS 메모리 기반 색인

메인 메모리를 위해 설계된 색인과 디스크를 위해 설계된 색인은 차이가 있다. 디스크 기반 색인 구조의 목표는 디스크의 접근 회수와 디스크 공간을 최소화하는 것이다. 그러므로 디스크 I/O 를 가장 큰 비용으로 생각한다. 그러나 메인 메모리 기반의 색인은 디스크 접근 회수가 없다. 따라서 메인 메모리 기반 색인의 목표는 CPU cycle 에 따른 전체 수행 시간을 줄이고, 가능한 한 적은 메모리 공간을 사용하는 것이다. 메인 메모리 기반 색인에서는 노드를 접근하는 비용이 노드의 메모리 주소에 대한 포인터를 할당하고, 획득하는 비용이기 때문에 다른 연산에 비해서 상대적으로 작다[5].

메인 메모리 색인 부분에서 자주 언급되는 부분 중의 하나는 전체 데이터를 저장할 수 있는 큰 버퍼를 쓰는 디스크 기반의 색인과 메인 메모리 기반의 색인의 성능 차이에 관한 것이다. 큰 버퍼를 쓰는 디스크 기반의 색인은 모든 데이터가 메모리에 있어도 노드를 가리키는 포인터는 디스크 주소를 가지고 있기 때문에, 버퍼 관리자를 통하여 디스크 주소를 메모리 주소로 변환하는 시간이 추가된다. 메인 메모리 기반의 색인은 노드를 가리키는 포인터가 메모리 주소를 가지고 있어서 직접 노드를 접근하기 때문에 디스크 기반의 색인보다 성능이 좋다[1].

4. 메인 메모리 환경에서 고려해야 할 요소

3 장에서 언급했듯이, 디스크 기반의 색인을 메인 메모리로 적재할 경우에, 가장 중요한 것은 전체 수행 시간이다. 전체 수행 시간에 영향을 주는 요소에는 여러 가지가 있지만, 이 논문에서는 R-tree 계열의 3 개의 분할 알고리즘과 트리의 깊이를 파라미터로 사용하여 실험을 하였다.

4.1 분할 알고리즘

R-tree 계열의 분할 알고리즘에는 Exhaustive Split, Quadratic Split, Linear Split, R*-tree Split 등이 있다. 이 중에서 Exhaustive Split 은 모든 가능한 분할 가능한 경우를 다 고려하기 때문에 너무 많은 시간이 걸린다. 따라서 여기서는 고려하지 않는다.

Quadratic Split 은 R-tree 에서 가장 많이 쓰이는 분할 알고리즘이다. 이 알고리즘은 사장 공간(dead space)이 가장 큰 두 개의 seed 를 잡는다(pick seed). 그리고, 남아 있는 엔트리 중에서, 한 엔트리를 두 개의 그룹에 삽입함으로써 커지는 넓이의 차가 가장 큰 엔트리를 선택한다(pick next). 그리고 나서, 그 엔트리를 삽입함으로서 영역이 가장 작게 커지는 그룹에 할당하는 방식이다.

Linear Split 은 축 별로 가장 작은 값과 큰 값의 거리를 계산하여, 그 중에서 거리가 최대가 되는 축을 골라서 seed 로 잡은 후(pick seed), 남아 있는 엔트리들을 제일 작게 커지는 그룹에 할당하는 방식이다.

R*-tree Split 은 각 축에 대하여 정렬하고, 최소 용량만큼 배당한다. 그리고, 남아 있는 엔트리들을 모든 분포에 있어서 둘레(perimeter)를 계산하여 더한 후에, 둘레의 합이 최소가 되는 축을 잡는다. 이 축을 기준으로 중첩이 최소가 되는 분포로 분할 한다.

기존 디스크 기반의 R-tree 에서는 성능에 있어서 알고리즘의 복잡도는 크게 고려하지 않기 때문에 Quadratic Split 이나 R*-tree Split 이 선호되지만, 메인 메모리 기반에서는 전체 수행 시간이 성능에 있어서 가장 큰 영향을 주기 때문에 이동체 데이터베이스 환경을 위한 메인 메모리 색인에서는 기존의 분할 알고리즘을 다시 고려할 필요가 있다.

4.2 트리의 깊이

기존의 디스크 기반의 색인에서는 노드의 접근 회수가 디스크의 I/O 수와 같기 때문에, 트리의 깊이는 성능에 매우 큰 영향을 끼쳤다. 삽입 시에는 데이터를 삽입할 단말 노드를 찾기 위해 비교하면서 내려가는 노드의 개수가 성능에 가장 큰 영향을 준다. 그리고 검색 시에는 데이터가 기존의 MBB 와 교차하는지 비교하면서 내려가는 노드의 개수가 가장 큰 영향을 주기 때문에, 얕고 넓게 퍼진 트리를 써서 삽입 시나 검색 시에 최소한의 I/O 비용을 가지게 했다.

메모리 기반에서는 노드를 한 번 접근하는 데 드는 비용은 포인터로 노드의 메모리 주소를 획득하는 비용이기 때문에 크지 않다. 따라서 기존의 디스크 기반에서 선호되는 얕고 넓게 퍼지는 트리 구조는 메모리 기반에서는 더 이상 유용하지 않다. 디스크 기반의 색인에서는 한 번에 읽어 오는 페이지의 크기에 따라 노드의 용량이 정해져 있지만, 메모리 기반에서는 노드의 용량에 아무런 제약이 없다. 그러므로 노드의 용량을 변화시켜 트리의 깊이를 조절하면서 성능을 평가할 수 있다.

5. 성능 평가

4 장에서 언급했듯이, 메모리 기반의 색인에서 성능에 영향을 줄 수 있다고 판단되는 알고리즘의 복잡도와 트리의 깊이에 대해서 실험을 수행했다.

5.1 실험 요소

이동체 데이터베이스 환경을 고려하여 메인 메모리 색인의 성능을 평가하기 위해 다음과 같은 데이터와 질의를 사용하여 삽입 시간과 검색 시간을 비교하였다. 삽입은 50 만개(1000*500: 1000 개의 이동체가 500 번 보고), 100 만개(1000*1000: 1000 개의 이동체가 1000 번 보고), 200 만개(1000*2000: 1000 개의 이동체가 2000 번 보고)의 데이터를 가지고 수행하였고, 검색은 전체 영역의 5%크기를 가진 1000 개의 영역 질의(Range Query)를 수행하였다.

5.2 실험 파라미터

4 장에서도 언급했듯이, 알고리즘의 복잡도에 따른 성능을 실험하기 위해서 분할 알고리즘인 Quadratic Split, Linear

Split, R*-tree Split 으로 삽입과 검색 시간을 실험 하였다. 그리고 트리의 깊이에 따른 성능을 실험하기 위해서 노드의 용량이 3, 5, 10, 20, 30, 50, 100, 200, 300, 500, 1000, 2000, 3000, 5000, 10000 일 때, 이에 따른 삽입과 검색 시간을 실험 하였다.

5.3 실험 세부 사항

각 알고리즘 및 색인은 C 로 구현하였고, 윈도우 XP 에서 640MB 의 메인 메모리, CPU Pentium IV 1.5Ghz 를 가지고 실험하였다. GSTD 생성기[4]를 가지고 데이터 집합을 생성하였다. GSTD 생성기는 시간 간격 마다 이동체 위치인 점 좌표를 생성하기 때문에, 이전 위치를 참조하여 선분을 생성하여 색인에 저장하였다. 이동체 위치 데이터를 생성하기 위한 매개 변수로, 이동체의 초기 분포는 가우시안 분포이며 이동체의 이동은 랜덤(random)이다.

5.4 실험 결과

□ 알고리즘의 복잡도에 따른 성능 평가

그림 1 은 노드의 용량을 50 으로 하여 앞에서 언급한 3 개의 분할 알고리즘에 따라 삽입 시간을 측정해 본 것이다.

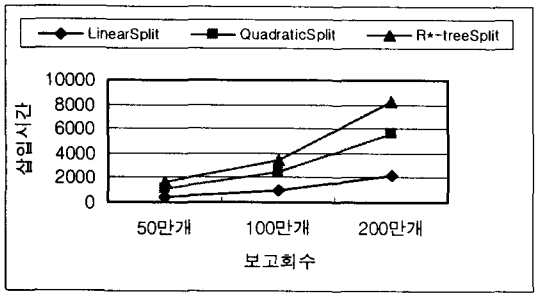

그림 1. 알고리즘의 복잡도에 따른 삽입 시간 측정

알고리즘의 복잡도가 가장 낮은 Linear Split 이 삽입 시간이 가장 빨랐고, 그 다음에 Quadratic Split, R*-tree Split 순이었다. 결과에서 보면, 데이터의 개수가 증가하면서 삽입 시간의 차이가 점점 더 커지는 것을 알 수 있다. 즉, 알고리즘이 간단할수록 더 좋은 삽입 성능을 가지는 것을 알 수 있다.

그림 2 는 분할 알고리즘에 따라 검색 시간을 측정해 본 것이다. 알고리즘의 복잡도가 가장 낮은 Linear Split 이 검색 시간이 가장 늦었고, 그 다음에 Quadratic Split, R*-tree Split 순이었다.

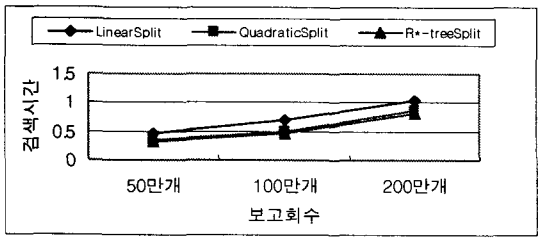

그림 2. 알고리즘의 복잡도에 따른 검색 시간 측정

그림 2 의 결과에서 보면 알고리즘이 간단할수록 삽입 시간에서는 좋은 성능을 보이지만, 검색 시간에서는 좋지 않은 성능을 보여준다. 그러나 앞에서도 본 바와 같이, 이동체 데이터베이스 환경에서는 삽입 시간이 실시간으로 이루어져야 하고, 검색 시간은 그렇지 않다. 그리고 알고리즘의 복잡도에 따라 삽입 시간의 차이에 비해서 검색 시간의 차이는 얼마 나지 않는다. 따라서, 이동체 데이터베이스 환경에서는 삽입 성능을 고려하여 Linear Split 과 같은 간단한 알고리즘을 쓰는 것이 더 적합하다고 할 수 있다.

□ 삽입 알고리즘에서의 함수 별 시간 측정

앞의 실험의 결과를 보면 기존의 알고리즘에서 Linear Split 알고리즘이 이동체 데이터베이스 환경에서 가장 적합한 분할 알고리즘이라는 것을 알 수 있었다. Linear Split 알고리즘을 사용하여 전체 삽입 시간 중에서 함수 별로 시간을 분석해 보았다.

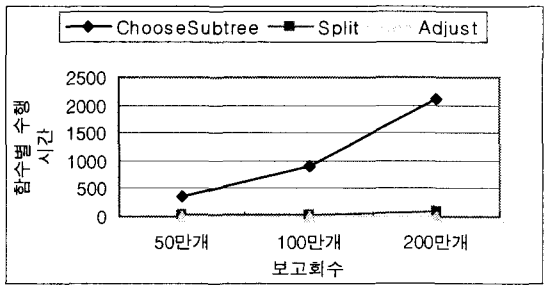

그림 3. 삽입에서 함수 별 수행 시간

ChooseSubtree 함수는 삽입 시에, 삽입할 노드를 찾기 위해서 루트에서 시작하여 단말 노드까지 내려가면서 엔트리들을 비교하는 함수이고, Split 함수는 노드의 용량을 초과할 경우에 분할을 하는 함수이며, Adjust 함수는 삽입 및 분할 후에 트리를 유지시켜 주는 함수이다. 실제로 이 3 개 함수의 시간의 합이 전체 삽입 시간의 약 98%의 비율을 차지하고 있다. 여기서는 ChooseSubtree 함수에서 걸리는 시간이 전체 삽입 시간에서 가장 크다는 것을 알 수 있다.

□ 트리의 깊이에 따른 성능 평가

앞의 실험의 결과에 따라 Linear Split 알고리즘을 사용하여 트리의 깊이에 따라 성능이 어떻게 달라지는지를 살펴보았다. 트리의 깊이를 조절하기 위해서 노드의 용량을 변화시켜서 실험을 하였다.

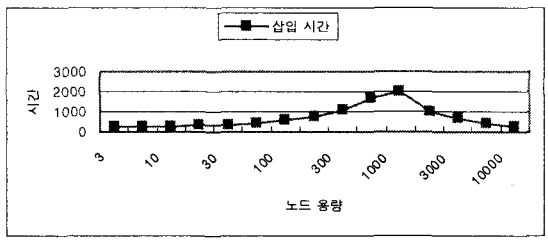

그림 4. 트리의 깊이에 따른 삽입 시간 측정

그림 4 는 트리의 깊이에 따른 삽입 시간을 측정해 본 것이다. 결과에서 보면 노드의 용량이 작을 때와 노드의 용량이 아주 클 때, 삽입 시간이 적게 걸린다. 앞의 실험에서 본 바와 같이, 삽입 시간의 대부분은 삽입할 노드를 찾아가는 데 걸리는 시간이므로, 이 때 비교하는 엔트리의 수가 삽입 시간에 영향을 주는 것을 알 수 있다. 노드의 용량에 따른 비교 엔트리 수를 구하는 공식은 다음과 같다.

$$\text{비교 엔트리 수} = \sum_{n=1}^{l-2}(u*M)+N*(\frac{1}{u*M})^{l-1}$$

(l = 트리의 레벨(= $\log_{uM} N$), u = 노드의 utilization, M = 노드 용량, N = 전체 보고 데이터 수)

수식 1. 삽입할 단말 노드를 찾기 위한 비교 엔트리 수

우선 루트(root)를 제외한 비단말 노드 하나 당 들어 있는 엔트리의 개수, 즉 비교해야 하는 엔트리의 수는 u * M 이라 가정하였다. 노드의 용량에 노드 utilization 을 곱한 값만큼 들어 있다고 가정한 것이다.

수식 1 의 첫번째 항은 단말 노드의 바로 위 레벨의 노드에서 루트(root) 바로 아래 레벨의 1-2 레벨 노드까지의 노드별 엔트리의 비교 회수를 더한 것이다. 두번째 항은 루트에서의 엔트리의 비교 회수로, 단말노드로부터 엔트리의 비교 수를 루트까지 계산한 것이다. 루트 아래에 있는 비단말 노드에서는 분할이 적어도 한 번 이상 일어났기 때문에 노드의 utilization 과 비슷하다고 가정할 수 있고, 루트 노드에서는 노드의 utilization 을 보장할 수 없기 때문이다. 그림 5 는 앞에서 제시한 수식에 따른 이론적인 엔트리의 비교 회수를 나타낸 것이다.

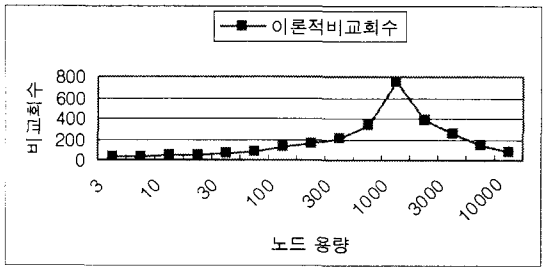

그림 5. 이론적 엔트리의 비교 회수

그림 4 와 그림 5 의 그래프를 보면 같은 모양임을 알 수 있다. 즉, 노드의 용량이 작을 때에는 단말 노드로 내려가기 위해 비교해야 하는 엔트리의 수가 작고 노드의 용량이 아주 클 때도 비단말 노드에서 비교하는 엔트리의 수가 작기 때문에, 삽입 성능이 좋아짐을 알 수 있다. 삽입 시간은 비교하는 엔트리의 수와 비례하는 것을 알 수 있다.

그림 6 은 앞의 실험에서 생성된 색인에 1000 개의 영역 질의를 수행해서, 노드의 용량에 따른 검색 시간을 측정해 본 것이다.

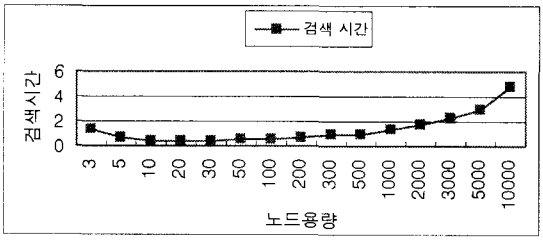

그림 6. 트리의 깊이에 따른 검색 시간 측정

실험에서, 노드의 용량이 아주 작을 때와 클 때는 검색 성능이 좋지 않은 것을 알 수 있다. 노드의 용량이 30 에서 100 사이일 때, 검색 성능이 좋은 것으로 나타났다. 검색 비용은 접근한 노드 수와 비교한 엔트리 수에 의해 영향을 받기 때문에 다음과 같은 수식으로 나타낼 수 있다.

검색 비용 = 접근한 노드 수 + **K***비교한 엔트리 수
(K: 한 엔트리를 비교하는 비용 / 한 노드를 접근하는 비용)

수식 2. 검색 비용

3 장에서도 언급했듯이 노드 접근 비용이 엔트리의 비교 비용보다 작지만, 실제적으로 구현 시에는 검색 알고리즘에서 명시되어 있는 것과 같이 재귀 호출이 들어 가야 하므로 노드의 접근 비용이 엔트리의 비교 비용보다 낮다고 말할 수 없다. 따라서 정확한 K 의 값은 프로그램마다 다를 수 있기 때문에 여기서는 K 의 값을 1 로 하여 비교하는 노드 수와 엔트리 수를 더해 보았다.

그림 7 은 검색 시에, 비교하는 노드 수와 엔트리 수의 합을 나타낸 것이다. 그림 7 에서 알 수 있듯이, 노드간의 중첩으로 인하여, 노드의 용량이 아주 작으면 비교하는 노드의 수와 엔트리의 수가 많아지기 때문에 검색 성능이 나빠지고, 반대로 노드의 용량이 크면 비교하는 노드의 수는 적으나, 엔트리의 수가 많아지기 때문에 검색 성능이 나빠진다.

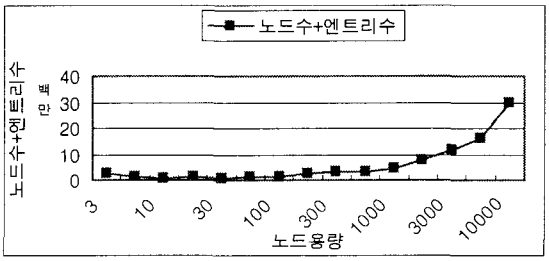

그림 7. 검색 시에 비교하는 노드 수와 엔트리 수의 합

위의 실험에서 알 수 있는 것처럼, 이동체 데이터베이스를 위한 메인 메모리 색인에서 고려할 요소로는 삽입 시에는 삽입할 단말 노드를 찾기 위해서 비교하는 엔트리 수가, 검색 시에는 노드간의 중첩으로 인하여 비교해야 하는 엔트리 수와 노드 수임을 알 수 있다.

6. 결론 및 향후 연구

이 논문에서는 이동체 데이터베이스 환경을 위한 메인 메모리 색인에서 성능에 영향을 주는 요소에 대하여 알아보았다. 이동체 데이터베이스 환경에서는 간단한 알고리즘을 가진 색인이 좋은 성능을 내며, 삽입 시에는 삽입할 노드를 찾기 위해서 비교하는 엔트리의 수, 그리고 검색 시에는 노드간의 중첩으로 인한 비교하는 엔트리의 수와 노드 수가 성능에 영향을 준다. 따라서 이동체 데이터베이스 환경을 위한 메인 메모리 색인에서는 삽입할 노드를 선택하기 위해 비교해야 하는 엔트리의 수와 노드 간의 중첩을 최소화하는데 그 목적을 두어야 한다.

향후 연구로서는 이러한 실험 결과를 바탕으로 이동체 데이터베이스 환경을 위한 메인 메모리에서 삽입과 검색에 있어서 좋은 성능을 낼 수 있는 색인을 구현하는 것이다.

참고문헌

[1] Tobin J. Lehman, Michael J. Carey "A Study of Index Structures for Main Memory Database Systems", in Proceedings 12th Int'l. conf. on Very Large Databases, Kyoto, Aug. 1986, pp204~230

[2] A. Guttman, " R-trees: A dynamic index structure for spatial searching" , ACM SIGMOD Conference, p47-54, 1984.

[3] N. Beckmann and H. P. Kriegel, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", In Proc. ACM SIGMOD, p332-331, 1990.

[4] Y. Theodoridis, J. R. O Silva and M.A Nascimento, " On the Generation of Spatiotemporal Datasets" , SSD, Hong Kong, LNCS 1651, Springer, p147-164, 1999.

[5] Hongjun Lu, Yuet Yeung Ng, Zengping Tian, " T-tree or B-tree: Main Memory Database Index Structure Revisited" , Australasian Database Conference2000: 65-73

[6] Hector Garcia-Molina, Kenneth Salem, " Main Memory Database Systems: An Overview" . TKDE4: 509-516