메모리 기반의 인덱스 기법에 관한 연구

A Study of Index Method Based on Main Memory

홍기채(G.C. Hong) 정보유통연구팀 책임기술원
문병주(B.J. Moon) 정보유통연구팀 선임연구원, 팀장

본 고에서는 디스크 기반의 정보검색시스템의 성능을 높이는 것을 목표로, 주기억장치 상주형 정보검색시스템에 적합한 주기억장치 기반의 인덱싱 기법을 비교 평가하고자 한다. 인덱스는 인덱스를 구성하는 키의 순서가 유지되는지의 여부에 따라 크게 두 종류로 나눌 수 있는데, 키가 일정한 순서로 유지되는 트리 계열과 키의 순서와 관계없이 무작위로 유지되는 해시 계열로 구분할 수 있다. 트리 계열 인덱스는 일정한 범위가 주어지는 연산을 처리할 때 유용하게 사용될 수 있으며, 해시 계열 인덱스는 특정한 키에 의한 빠른 데이터 접근을 제공한다. 트리 계열 인덱스로는 AVL 트리, B+ 트리, T 트리 등이 있으며, 해시 계열 인덱스로는 체인 버켓 해싱(Chained Bucket Hashing: CBH), 확장 해싱(Extendible Hashing: EH), 선형 해싱(Linear Hashing: LH), 수정된 선형 해싱(Modified Linear Hashing), 다중 디렉토리 해싱(Multi-directory Hashing) 및 확장된 체인 버켓 해싱(Extendible Chained Bucket Hashing: ECBH) 등이 있다.

I. 서 론

최근 컴퓨터 기술과 인터넷 등 통신기술의 발달로 인터넷 기반에서 정보검색시스템의 응용분야가 점차 늘어나고 있다. 따라서 정보검색시스템은 핵심인 정보의 구축 및 검색 성능 향상을 통해 더 많은 사용자에게 빠른 응답시간으로 반응할 수 있는 능력을 가지고 있어야 한다. 이를 위해서는 고객의 요구를 실시간으로 처리할 수 있어야만 하는데 기존의 Fulcrum이나 Dialog, BRS와 같은 디스크 기반의 범용 검색 엔진을 사용해서는 수시로 업데이트 되어지는 대용량 정보의 색인 및 검색 성능 요구조건을 만족시킬 수 없다.

이와 같이 대용량의 정보를 빠른 시간 안에 처리하기 위해서 검색 시스템의 응용 분야에 대한 많은 연구가 있어 왔는데 그 중 가장 주목할 만한 것이 주기억장치 상주형 DBMS이다. 주기억장치 상주형 DBMS는 디스크 기반의 범용 DBMS와는 달리 메모리에 DBMS를 상주시켜 트랜잭션의 처리를 가능하게 하는 DBMS로서, 그것 자체의 실시간적 요소로 인해 앞으로 폭발적으로 이용 분야가 늘어날 정보검색 분야에 유용하게 쓰일 수 있다.

또한 주기억장치 상주형 DBMS는 주기억장치에 모든 데이터를 상주시키는 DBMS로서, 디스크 입출력으로 인한 부하를 없애고 주기억장치에 적합한 기법들을 적용함으로써 디스크 기반의 범용 DBMS 시스템에서는 기대할 수 없었던 대용량 데이터의 고속처리 응용을 위한 성능 향상 도구로서 활용된다.

이처럼 인터넷 상에서 기하 급수적으로 증가하는 데이터에 대한 다양한 접근점과 제한점의 처리를 위하여 디스크 기반인 고가의 정보검색 엔진과 필요

이상의 동시 이용자 수를 지원하는 고가의 상용 DBMS를 구입하여 검색 속도를 보장 받아 온 것이 현실이다.

그러나 이러한 디스크 기반의 정보검색시스템이나 주기억장치 상주형 DBMS는 대용량의 정보 구축 시 색인 및 검색 성능에 많은 제한적인 요소를 가지고 있으며, 정보검색시스템에서 지원하는 색인 파일 구성 및 검색 기능의 제한성 때문에 주기억장치 상주형 검색 엔진이 필요하게 된다. 따라서 본 고에서는 정보검색시스템에 적용될 수 있는 주기억장치 기반의 인덱스 기법을 비교 평가하여 정보검색시스템에 적용하는 방안을 제시하고자 한다.

II. 트리 계열 인덱스

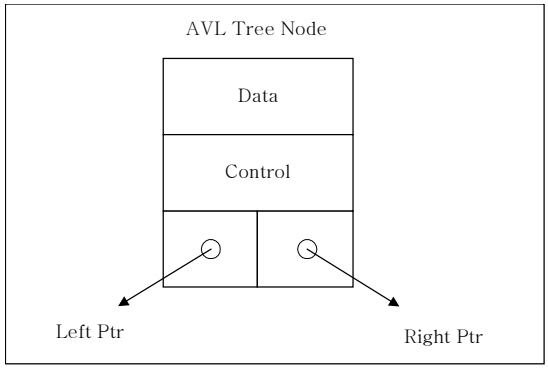
1. AVL 트리

AVL 트리는 서브 트리의 깊이 관점에서 (그림 2)와 같이 항상 균형이 유지되는 이진 트리 구조를 갖게 되며, 모든 서브 트리의 깊이를 일정하게 유지함으로써 임의의 데이터에 대한 최적의 탐색이 가능하도록 고안된 인덱스 기법이다[1].

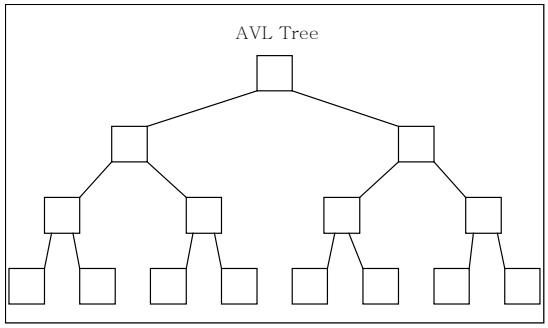
AVL 트리에서의 모든 삽입은 트리의 리프 노드에서 이루어지며, 노드의 삽입과 삭제로 인해 발생하는 트리의 불균형은 회전 규칙을 적용하여 해결하고 AVL 트리에 가해지는 모든 연산 후에는 항상 트리의 균형이 유지되도록 한다.

(그림 1)에서와 같이 AVL 트리의 노드는 데이터, 노드 자체의 제어 정보, 왼쪽 포인터, 오른쪽 포인터로 구성되며, 데이터는 키에 대한 정보로서 레코드가 위치한 메모리 공간의 주소이며, 이 주소를 이용하여 실제 키 값을 얻을 수 있다.

노드의 제어 정보에는 왼쪽, 오른쪽 서브 트리의 깊이, 균형 인자 등이 포함될 수 있는데, 여기서 균형 인자는 오른쪽 서브 트리와 왼쪽 서브 트리의 깊이의 차이와 깊이가 더 깊은 서브 트리의 방향을 나타내도록 구성된다.



(그림 1) AVL 트리의 노드 구성



(그림 2) AVL 트리의 구성

◈ AVL 트리의 연산

1) 탐색

① 트리의 루트 노드부터 탐색을 시작하며, 노드의 데이터가 찾고자 하는 키와 같은지 비교한 후 같으면 데이터를 반환하고 탐색을 마치고, 작으면 왼쪽 포인터가 가리키는 노드로 이동한 후 다시 비교한다.

② 또한 클 경우에는 오른쪽 포인터가 가리키는 노드로 이동한 후 다시 비교한다.

2) 삽입

① 탐색에 의해 데이터가 삽입될 위치를 결정한 후, 데이터를 삽입한다.

② 데이터를 삽입한 후, 데이터가 삽입된 리프 노드에서 루트 노드까지의 탐색 경로를 따라 각각의 노드마다 서브 트리의 깊이를 조정하고, 노드의 균형 인자를 변경한다.

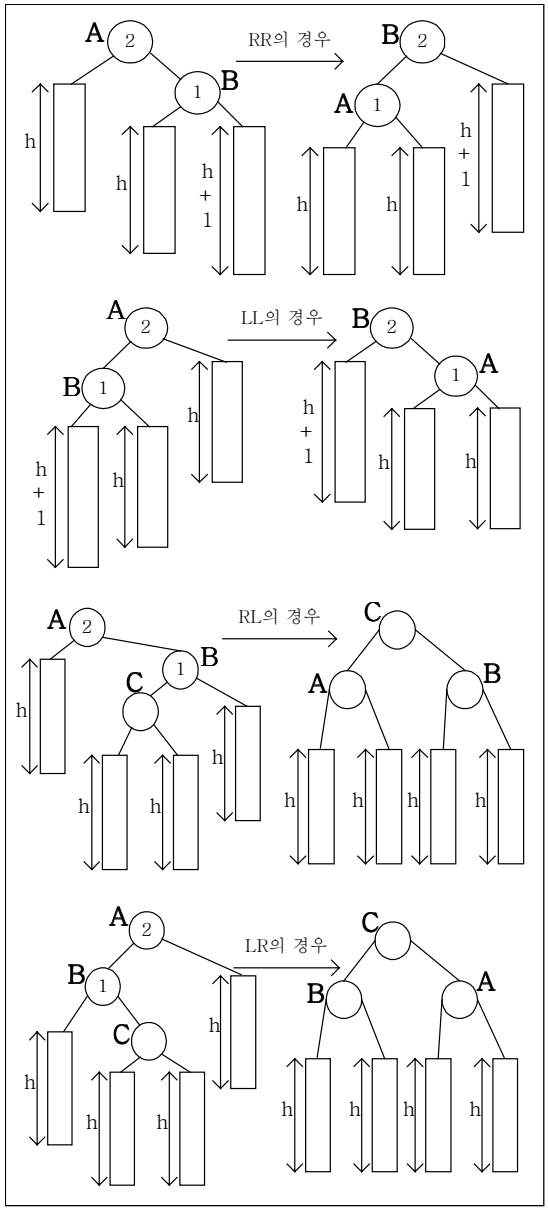
③ 각각의 노드에 서브 트리의 깊이를 조정하는 과정에서 노드의 균형 인자가 2 이상이면 이 노드에 대해 회전 규칙을 적용하여 트리의 균형을 맞춘다.

3) 삭제

① 탐색에 의해 삭제하려는 데이터를 갖는 노드 A를 결정하며, 노드 A가 리프이면 노드를 삭제한다.
② 노드 A가 리프가 아니면 삭제하려는 데이터의 바로 다음 크기의 데이터를 포함하는 노드 B와 맞바꾼 후 삭제한다. 이때 맞바뀌는 노드 B는 삭제될 노드 A의 오른쪽 서브 트리 중에서 가장 작은 데이터를 갖는 노드이다.
③ 데이터를 삭제한 후, 리프 노드에서 루트 노드까지의 탐색 경로를 따라 각각의 노드마다 서브 트리의 깊이를 조정하고, 노드의 균형 인자를 변경한다.
④ 각각의 노드에 대해 서브 트리의 깊이를 조정하는 과정에서 노드의 균형 인자가 2 이상이면 이 노드에 대해 회전 규칙을 적용하여 트리의 균형을 맞춘다.

4) 트리의 균형

① 트리에 노드가 추가되거나 삭제될 때마다 다음과 같이 트리의 균형을 맞추도록 한다.
② 리프에서 루트까지의 탐색 경로에 위치한 모든 노드에 대해 각각의 노드에 유지되는 균형 인자를 조사하여 2 이상 차이가 나면 방향 정보에 따라 적용할 회전 규칙의 종류를 결정한다.
③ 회전 규칙의 종류는 (그림 3)과 같이 RR, LL, RL, LR로 구분될 수 있으며, 트리의 불균형을 일으키는 노드의 서브 트리 방향을 이용하여 만들어지는데, LL은 회전이 적용될 노드의 Left Left 서브 트리의 깊이가 더 깊다는 것을 의미한다.
④ RR과 LL은 단순 회전에 의해 트리의 재균형을 유지할 수 있으나 RL과 LR의 경우에는 트리의 재균형을 유지하기 위해 회전이 두 번 적용된다.



(그림 3) AVL 트리의 회전 규칙

5) 회전 규칙

① RR의 경우는 균형 인자가 2인 노드 A를 균형 인자가 1인 노드 B의 왼쪽 자식 노드가 되도록 하고, 노드 B의 왼쪽 서브 트리를 노드 A의 오

른쪽 서브 트리가 되도록 한다.

② LL의 경우는 균형 인자가 2인 노드 A를 균형 인자가 1인 노드 B의 오른쪽 자식 노드가 되도록 하고, 노드 B의 오른쪽 서브 트리를 노드 A의 왼쪽 서브 트리가 되도록 한다.

③ RL의 경우는 균형 인자가 1인 노드 B에 대해 LL 회전 규칙을 적용한 후 균형 인자가 2인 노드 A에 대해 RR 규칙을 적용한다.

④ LR의 경우는 균형 인자가 1인 노드 B에 대해 RR 회전 규칙을 적용한 후 균형 인자가 2인 노드 A에 대해 LL 규칙을 적용한다.

AVL 트리는 탐색 연산이 간단하고, 빠르다는 장점을 갖고 있으나 노드마다 데이터 하나에 포인터가 두 개씩 유지되므로 인덱스를 구성하는 데 필요한 메모리의 낭비 문제가 발생한다.

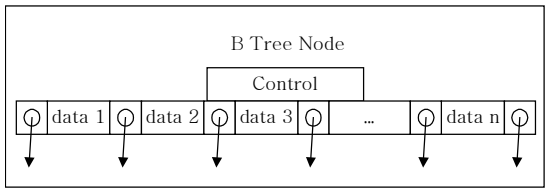
중복되는 키를 처리하는 것에 대한 해결책이 제시되어 있지 않으므로 중복 키를 허용하는 인덱스를 생성할 때는 사용할 수 없다.

2. B 트리

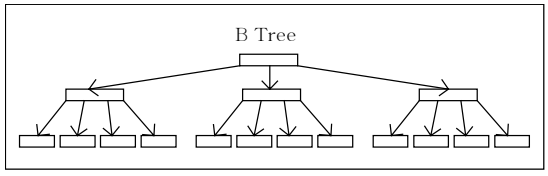
B 트리는 (그림 5)와 같이 d가 노드에 포함되는 데이터의 최소 개수일 때 한 노드에는 최대로 2d개의 데이터와 2d+ 1개의 포인터가 포함되는 노드들로 구성된 이진 탐색 트리이다. B 트리의 노드는 내부 노드와 리프 노드로 구분할 수 있고, 내부 노드에는 트리의 루트 노드가 포함된다[2].

내부 노드에 유지되는 데이터의 개수는 노드마다 각기 다를 수 있으나 적어도 d개의 데이터와 d+ 1개의 포인터는 반드시 유지하여 노드의 반 이상이 채워져야 한다. 이때 내부 노드의 데이터는 서브 트리에 대한 분리자로 사용된다. 리프 노드는 공간이 허용하는 한 포함되는 데이터 개수의 제한이 없으며 서브 트리에 대한 포인터를 유지할 필요가 없다.

B 트리 노드를 구성하는 데이터는 (그림 4)와 같이 키 값과 키가 위치한 레코드의 식별자로 구성된 것이나 다음의 탐색, 삽입, 삭제 연산의 설명 부분에서는 키 값의 의미로 사용된다. B 트리는 한 노드에



(그림 4) B 트리 노드의 구성



(그림 5) B 트리의 구성

많은 데이터를 포함할 수 있으므로 깊이가 낮고 넓게 펼쳐지는 트리 구조를 가지며, 따라서 몇 개의 노드만 접근하여 필요한 데이터를 얻을 수 있으며, 삽입이나 삭제 연산 후에도 서브 트리의 깊이가 일정하게 유지된다.

◈ B 트리의 연산

1) 탐색

① 트리의 루트 노드부터 탐색을 시작한다.

② 노드에 포함된 각각의 데이터들과 비교한다.

③ 탐색하는 데이터가 있으면 데이터를 반환하고 마친다.

④ 그렇지 않으면 계속 탐색할 서브 트리를 결정하여 이동한 후 ②부터 수행한다.

2) 삽입

① 삽입할 데이터가 위치할 노드 A를 탐색에 의해 찾는다.

② 데이터가 들어갈 노드 A에 여유 공간이 있으면 데이터를 삽입한 후 수행을 마친다.

③ 데이터를 삽입하려는 노드 A에 여유 공간이 없으면 노드를 A, B 둘로 분리한다.

④ 삽입하려는 데이터를 포함하여 노드 A의 데이터 중에서 가장 작은 것부터 커지는 순서로 d

개를 원래 노드 A에, 가장 큰 것부터 작아지는 순서로 d개를 새로 할당된 노드 B로 이동한다.

⑤ 나머지 데이터 한 개를 부모 노드로 이동한다. 이때 이동된 데이터는 서브 트리에 대한 분리자로 사용된다.

⑥ 노드의 분리 동작은 리프 노드에서 루트 노드까지 확산될 수도 있으며, 루트 노드가 분리될 때마다 트리의 깊이가 하나씩 증가된다.

3) 삭제

① 삭제될 데이터 d1이 위치한 노드 A를 탐색에 의해 찾는다.

② 결정된 노드 A가 내부 노드이면 삭제될 데이터 d1 바로 다음 크기의 데이터 d2를 찾아서 데이터 d1을 삭제한 후 노드 A 내에서 데이터의 순서를 유지하여 d2를 위치시킨다.

③ 노드 A가 리프 노드이면 노드 A에서 데이터 d1을 삭제한 후 남아있는 데이터의 개수가 d개 이상 유지되는지를 검사한다.

④ 노드 A에 유지되는 데이터의 개수가 d 보다 작으면 이웃하는 다른 노드 B를 이용하여 노드의 데이터 개수를 조절한다.

⑤ 노드 A와 이웃하는 노드 B에 포함된 데이터 개수의 합이 2d 보다 작으면 두 노드 A, B의 데이터를 모두 한 노드로 이동하고, 빈 노드는 반환한다.

⑥ 노드 A와 이웃하는 노드 B에 포함된 데이터 개수의 합이 2d 보다 크면 두 노드의 데이터를 반씩 나누어 각각의 노드에 위치시킨다.

이와 같은 B 트리의 확장, 변형된 형태로는 B+ 트리, B* 트리 등이 있으며, 특히 B+ 트리는 디스크를 기반으로 하는 DBMS의 인덱스를 생성할 때 가장 많이 사용되는 기법이다. B+ 트리는 B 트리 형태인 인덱스 부분과 실제 데이터들이 유지되는 리프 부분으로 구성되는데, 인덱스인 내부 노드의 데이터는 단지 빠르게 리프 노드를 찾아가는 길잡이 정도의 역할만 수행한다. 모든 데이터를 리프 노드에 유지하고 리프 노드끼리 연결 리스트를 구성함으로써 범위가 주어진 연산을 수행할 때 데이터에 대한 빠른 순차적 접근을 제공한다.

그러나 주기억 상주 검색시스템의 인덱스를 생성할 때는 데이터에 대한 빠른 접근을 제공하는 주기억장치의 특성으로 인해 모든 데이터를 리프 노드에 유지하는 장점이 감소되고, 오히려 많은 데이터의 중복으로 인한 메모리의 낭비가 심각하므로 B+ 트리보다 B 트리가 더 적합하다.

3. T 트리

T 트리는 AVL 트리와 B 트리의 장점을 이용하여 주기억 상주 데이터를 접근하는 데 적합하도록 고안되었다[3]. T 트리는 한 노드에 많은 수의 데이터를 갖는 이진 탐색트리로 구성되어 빠른 데이터 접근을 제공하며 메모리의 사용상에도 장점을 갖는다.

T 트리의 노드에 포함되는 데이터는 (그림 6)과 같이 실제 키 값이 아닌, 키를 포함하는 레코드의 메모리 주소를 나타낸다.

키를 포함하는 레코드의 메모리 주소를 데이터로 사용함으로써 키 값을 구하는 연산으로 인한 부하가 전체 인덱스의 처리 속도에 거의 영향을 미치지 않는 상태에서 고정 길이 키에 대해서 뿐만 아니라 가변 길이 키를 처리할 때 발생하는 메모리 사용의 낭비 문제도 해결할 수 있다. 단, 다음의 T 트리 연산을 기술할 때 사용되는 데이터는 키 값의 의미로 사용한다.

T 트리의 노드 내의 데이터는 항상 크기순으로 유지되며, 부모 노드를 가리키는 포인터 한 개와 왼쪽과 오른쪽 자식 노드를 가리키는 포인터가 각각 한 개씩 유지된다.

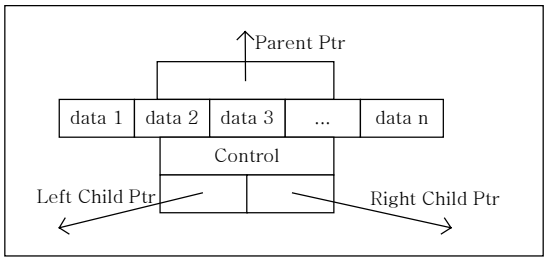
T 트리의 노드는 다음과 같이 세 종류로 구분할 수 있다.

- 내부 노드: 오른쪽, 왼쪽의 서브 트리를 가지며, 내부 노드에 포함될 수 있는 데이터의 개수는 MinT 이상, MaxT 이하를 유지해야 한다. 여기

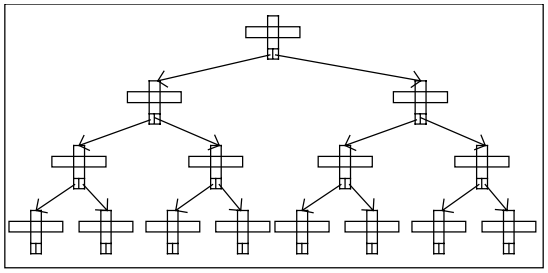
서 MinT와 MaxT는 각각 T 트리의 내부 노드가 가질 수 있는 데이터의 최소 개수와 최대 개수를 의미하며, 그 값은 T 트리를 생성할 때 결정하는 파라미터이다.

- 반쪽 리프 노드: 방향에 상관없이 한 쪽 서브 트리만을 갖는다.
- 리프 노드: 서브 트리를 갖지 않는다.

T 트리는 (그림 7)과 같이 서브 트리의 균형이 항상 유지되는 트리이며, AVL 트리와 같은 방법의 회전 연산에 의해 트리의 균형을 유지한다. 트리의 균형을 유지하기 위해 사용되는 정보인 균형인자는 AVL 트리에서의 균형인자와 동일한 의미를 갖는다.



(그림 6) T 트리의 노드



(그림 7) T 트리의 구성

◈ T 트리의 연산

1) 탐색

① 루트에서부터 탐색을 시작한다.
② 찾는 값이 노드의 최소 데이터보다 작으면 왼쪽 서브 트리로 이동하여 탐색을 계속한다.
③ 찾는 값이 노드의 최대 데이터보다 크면 오른쪽 서브 트리로 이동하여 탐색을 계속한다.
④ 그렇지 않으면 노드에 포함된 데이터들과 비교하여 탐색하는 키를 찾는다.

2) 삽입

① 데이터가 삽입될 노드를 탐색에 의해 찾는다.
② 루트에서부터 리프에 이르는 탐색 경로를 따라 트리의 깊이와 방향을 스택에 유지한다.
③ 데이터를 삽입할 노드 A에 여유공간이 있는지 검사한다.
④ 공간이 있으면 노드 A에 데이터를 삽입하고 마친다. 데이터를 삽입할 때는 노드에 포함되는 데이터들 사이에 순서가 유지되도록 한다.
⑤ 공간이 없으면 노드 A에 포함된 데이터 중에서 최소 데이터를 삭제하고, 그 공간을 이용하여 데이터를 삽입한다.
⑥ 삭제한 데이터를 새로운 삽입 데이터로 하여 노드 A의 왼쪽 서브 트리에서 가장 큰 데이터를 가진 노드로 이동하여 ③부터 수행을 계속한다.
⑦ 트리를 리프 노드까지 탐색할 동안 데이터를 삽입할 노드가 발견되지 않으면, 탐색 경로의 리프 노드 B에 여유 공간이 있는지 검사한다.
⑧ 노드 B에 여유 공간이 있으면 데이터를 삽입하고 마친다.
⑨ 노드 B에 여유 공간이 없으면, 새로운 리프 노드를 생성하여 노드 B의 왼쪽 자식 노드로 만든 후 데이터를 삽입한다.
⑩ 리프 노드에서 루트 노드까지의 탐색 경로를 따라 각각의 노드마다 서브 트리의 깊이를 조정하고, 노드의 균형 인자를 변경한다.
⑪ 각각의 노드에 대해 서브 트리의 깊이를 조정하는 과정에서 노드의 균형 인자가 2 이상이면 이 노드에 대해 회전 규칙을 적용하여 (그림 8)과 같이 트리의 균형을 맞춘다.

3) 삭제

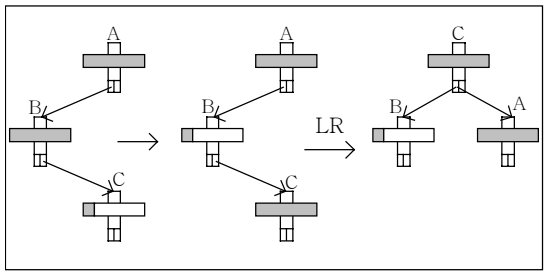
① 삭제할 데이터를 갖고 있는 노드 A를 탐색한다.
② 결정된 노드 A의 형태에 따라 다음과 같이 수행한다.

③ 노드 A가 내부 노드이면, 데이터를 삭제하기 전에 노드 A에 포함된 데이터의 개수가 최소값 MinT 보다 큰 지를 조사한다.
④ 노드 A에서 데이터를 삭제해도 노드 A의 개수가 MinT 이상이면, 노드 A에서 데이터를 삭제한 후 수행을 마친다.
⑤ MinT 보다 작으면, 노드 A의 최소 데이터 개수 MinT를 유지하기 위하여 노드 A의 최소값의 바로 이전 값을 왼쪽 서브트리에서 찾아 삭제한 후, 이를 현재 노드 A에 삽입하고 수행을 마친다. 여기서 왼쪽 서브트리에서 노드 A의 최소값의 바로 이전 값을 삭제하는 것은 다음의 과정을 따른다.
⑥ 노드 A가 반쪽 리프 노드이면 데이터를 삭제한 후, 노드 A가 텅빈 노드인지를 검사하여 데이터가 남아 있으면 그대로 수행을 마친다.
⑦ 노드 A가 텅비었으면 자식 노드인 리프 노드 B와 병합하고 ⑩을 수행한다.
⑧ 노드 A가 리프 노드이면 데이터를 삭제한다. 삭제한 후 노드 A가 텅비지 않으면 그대로 수행을 마친다. 리프 노드 A에서 데이터를 삭제한 후 노드 A가 텅비게되면 노드 A를 반납하고 다음을 수행한다.
⑨ 리프 노드에서 루트 노드까지의 탐색 경로를 따라 각각의 노드마다 서브 트리의 깊이를 조정하고, 노드의 균형 인자를 변경한다.
⑩ 각각의 노드에 대해 서브 트리의 깊이를 조정하는 과정에서 노드의 균형 인자가 2 이상이면 이 노드에 대해 회전 규칙을 적용하여 트리의 균형을 맞춘다.

◈ T 트리의 회전 규칙

회전 규칙은 다음의 특수한 경우를 제외하면 AVL 트리의 회전 규칙을 그대로 적용한다.

- 적용될 회전 규칙의 종류가 RL이거나 LR이고,
- 노드 A와 노드 B가 반쪽 리프 노드이고,
- 노드 C는 리프 노드이고 포함되는 데이터가 오직 한 개인 경우,



(그림 8) 특수한 T 트리 재균형 연산

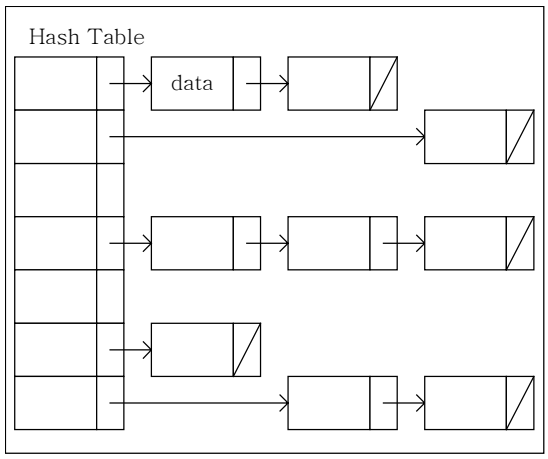
우선 노드 B의 데이터를 노드 C가 꽉 차도록 이동한 후 회전 규칙을 적용한다.

III. 해시 계열 인덱스

1. 체인 버켓 해싱

해시 함수에 의해 키를 해시하였을 때 얻어지는 동일한 해시 값들을 연결 리스트에 의해 유지하는 방법으로 인덱스를 생성한다[1].

CBH는 (그림 9)와 같이 인덱스를 생성할 때 키의 개수에 의해 해시 테이블의 크기가 결정되어 변하지 않는 정적 구조를 갖는다. 따라서 초기에 테이블의 크기를 너무 작게 결정하면 처리하는 데이터의 양이 동적으로 변화할 때 해시 테이블을 재구성해야 하므로 시스템 전체에 대한 성능이 저하되게 되고,



(그림 9) 체인 버켓 해싱의 구조

해시 테이블의 크기를 너무 크게 하면 사용되지 않는 빈 버켓들이 많게 되어 메모리 공간의 낭비가 발생한다.

CBH는 찾고자 하는 데이터에 대해서 매우 빠른 탐색을 제공할 수 있으나, 데이터의 양이 변화함에 따라 해시 테이블의 크기가 동적으로 변화해야 하는 환경에는 부적합하다.

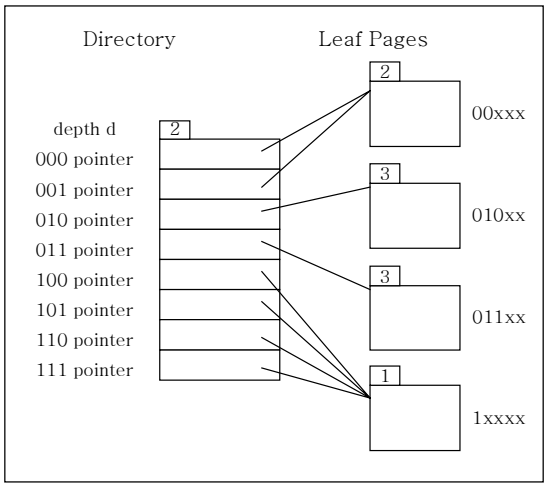
2. 확장 해싱

EH는 데이터의 양이 변화함에 따라 해시 테이블의 크기를 확장할 수 있게 함으로써 동적인 환경에 적합하도록 고안된 해싱 기법이다[4].

확장 해싱의 구조는 (그림 10)과 같이 해시된 데이터를 유지하는 리프 페이지와 리프 페이지의 페이지 식별자를 유지하는 디렉토리로 구성된다.

디렉토리에 유지되는 정보인 깊이 정보 d는 전역 깊이라 하며, 키를 해시하여 얻어진 해시 값을 서로 구분할 수 있는 비트 수로서 맨오른쪽 또는 맨왼쪽 d 비트를 나타낸다. 리프 페이지의 깊이 정보 d'는 지역 깊이라 하며, 같은 리프 페이지에 포함되는 데이터들을 묶어줄 수 있는 비트 수로서 d' 비트가 동일하게 시작되는 데이터임을 나타낸다. 리프 페이지의 깊이 d'는 디렉토리의 깊이 d 보다 클 수 없다.

리프 페이지는 더이상 데이터가 들어갈 공간이 없을 경우 두 페이지로 분리되며 분리되는 리프 페이지는 깊이가 1 증가한다. 페이지의 분리로 인해 리프 페이지가 디렉토리에 유지될 수 있는 개수보다 많아지면 디렉토리의 크기가 두 배로 늘어나며 디렉토리의 깊이 d가 1 증가한다.



(그림 10) 확장 해싱의 구조

3. 선형 해싱

선형 해싱은 데이터의 양이 동적으로 변하는 환경에 적합하도록 고안된 해싱 기법으로 데이터의 삽입과 삭제 시에 해시 함수를 적절하게 변경함으로써 데이터가 유지될 버켓이 결정된다[5].

선형 해싱의 해시 테이블은 데이터가 삽입될 버켓에 여유 공간이 없을 때 버켓이 분리되며 미리 정해진 순서에 의해 선형적으로 증가하게 된다.

1) 선형 해싱의 기본 원리

① 데이터를 해시 함수 h0에 의해 해시하여 구한 버켓에 여유 공간이 있으면 그 버켓에 그대로 삽입한다.

② 데이터를 삽입할 버켓에 여유 공간이 없어 충돌이 발생하면 새로운 해시 함수 h1을 결정하고 충돌이 발생한 버켓의 모든 데이터를 재해시하여 적절히 배치한다.

2) 버켓의 분리

① 해시된 데이터가 들어갈 버켓에 더이상의 여유 공간이 없어 충돌이 발생할 때 해시 함수가 변경되고 새로운 버켓이 할당되는 것을 버켓의 분리라 한다.

② 새로운 해시 함수는 초기의 해시 테이블의 크기를 N이라 할 때 N 만큼이 새로 추가된 크기인 2N의 해시 값 영역을 갖는 해시 함수로 변경된다.

③ 해시 함수가 변경될 때마다 해시 테이블의 크기는 초기의 해시 테이블의 크기 N 만큼이 선형적으로 증가한다.

④ 충돌이 발생한 버켓의 모든 데이터는 변경된 해

시 함수에 의해 재해시되어 충돌이 발생했던 버켓이나 새로 할당된 버켓으로 배정된다.

⑤ 충돌이 발생한 버켓의 바로 다음 버켓에는 다음 분리가 발생할 버켓이라는 의미의 포인터가 위치한다. 마지막 버켓에서 충돌이 발생했으면 이 포인터는 맨처음 버켓으로 이동된다.

(그림 11)을 이용하여 버켓의 분리를 설명하면 다음과 같다.

① 초기의 해시 테이블 크기 N은 100이고 해시 함수는 h0이다.

② 해시 함수 h0에 의해 해시된 데이터가 버켓 0부터 버켓 99까지 고르게 분포한다. 다음에 분리가 발생할 버켓 포인터는 버켓 0을 가리키고 있다.

③ 데이터 4900을 해시 함수 h0로 해시한 결과에 의해 버켓 0에 삽입하려 할 때 버켓에 여유 공간이 없어 충돌이 발생한다.

④ 충돌이 발생하였으므로 새로운 해시 함수 h1이 만들어지고 버켓 0이 분리되어 새로운 버켓 100이 할당되고 해시 테이블의 크기가 선형적으로 증가한다.

⑤ 버켓 0의 모든 데이터를 해시 함수 h1에 재해시하여 버켓 0와 버켓 100에 재배치 한다.

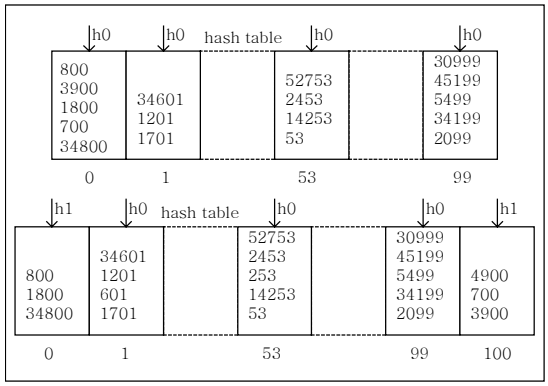
⑥ 다음에 분리될 버켓에 대한 포인터는 버켓 1을 가리키도록 이동한다.

⑦ 데이터 753을 h0에 의해 해시하여 버켓 53에 삽입하려 할 때 버켓에 여유 공간이 없어 충돌이 발생한다.

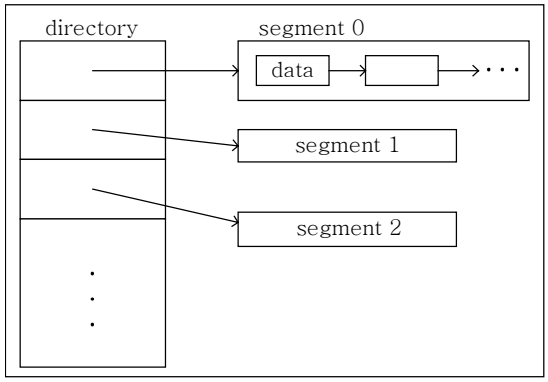
⑧ 충돌에 의해 버켓 53이 분리될 때, 다음에 분리가 발생할 버켓 포인터가 버켓 54를 가리킬 때까지 버켓 1부터 버켓 53까지의 버켓이 대응되는 버켓 101부터 153까지로 분리되어 해시 테이블이 선형적으로 증가된다. 이때 버켓 1부터 버켓 53까지의 모든 데이터는 해시 함수 h1에 의해 재해시된다.

4. 수정된 선형 해싱

디스크의 페이지를 버켓으로 하여 인덱스를 구성



(그림 11) 선형 해싱의 구조



(그림 12) 수정된 선형 해싱의 구조

하도록 고안된 선형 해싱을 주기억장치의 데이터에 적용될 수 있도록 변경한 해싱 기법이다[6].

수정된 선형 해싱의 구조는 (그림 12)와 같이 고정 길이의 노드들이 연결 리스트 형태로 유지되는 세그멘트와 사용중에 있는 세그멘트 각각의 시작 주소(세그멘트에 유지되는 노드 중 첫번째 노드)를 유지하는 디렉토리로 구성된다. 세그멘트에 유지되는 각각의 노드는 데이터와 다음 노드에 대한 포인터로 구성된다. 여기서 데이터는 주기억장치에서의 레코드 포인터를 의미한다.

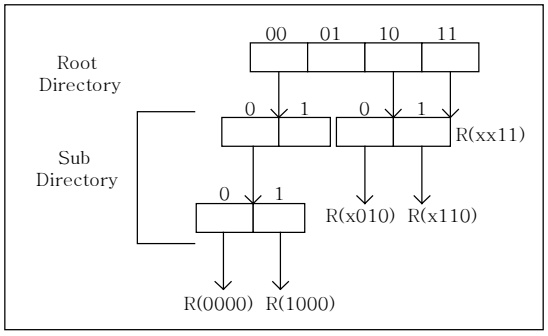
수정된 선형 해싱의 디렉토리의 크기는 세그멘트에 유지되는 데이터의 양이 기준 데이터의 양을 초과하여 메모리로부터 새로운 세그멘트를 할당받을 때마다 선형적으로 증가한다. 이는 LH에서 해시 테이블이 버켓의 분리에 의해 선형적으로 증가하는 것과 같다.

수정된 선형 해싱의 모든 연산은 주기억장치의 특성을 고려하여 두 단계로 유지되는 구조적 차이를 제외하고는 LH에서와 동일하게 수행된다. 예를 들면 데이터를 삽입하는 경우, LH에서는 해시 테이블에서 임의의 버켓을 결정하여 그 버켓에 데이터를 삽입하고, 수정된 선형 해싱에서는 디렉토리에서 임의의 엔트리를 결정하여 이 엔트리가 가리키는 세그멘트에 데이터를 표현하는 노드가 삽입된다.

5. 다중 디렉토리 해싱

주기억 상주 데이터에 대한 빠른 접근을 제공하기 위해 고안된 해싱 기법이다. 이 기법은 EH에서 리프 페이지의 분리가 발생할 때 디렉토리의 크기가 2배로 증가함으로써 메모리를 낭비하게 되는 문제를 보완하고 있다[7].

다중 디렉토리 해싱은 (그림 13)과 같이 EH의 디렉토리를 여러 개의 서브 디렉토리로 나누고, 이들 전체 디렉토리를 트리 구조로 유지한다.



(그림 13) 다중 디렉토리 해싱의 구조

◈ 다중 디렉토리 해싱의 연산

1) 탐색

① 루트 디렉토리부터 탐색을 시작한다.
② 디렉토리 S의 깊이 d를 구한다.
③ 해시 값인 H의 오른쪽 d 비트에 의해서 탐색할 디렉토리 S의 엔트리 E를 구한다.
④ 엔트리 E가 서브 디렉토리 S'를 가리키면, H에서 오른쪽 d 비트를 제외한 H'를 H로, 서브 디렉토리 S'를 S로 하여 다시 ②부터 수행한다.
⑤ 엔트리 E가 찾고자 하는 키를 포함하는 데이터를 가리키면 데이터를 반환한다.

2) 삽입

① 루트 디렉토리부터 탐색을 시작한다. 전역 변수 t를 0으로 초기화한다.
② 디렉토리 S의 깊이 d를 구한다.
③ 해시 값인 H의 오른쪽 d 비트에 의해서 탐색할 디렉토리 S의 엔트리 E를 구한다.
④ 엔트리 E가 서브 디렉토리 S'를 가리키면 전역 변수 t를 d만큼 증가시키고, 해시 값 H의 오른쪽 d 비트를 제외한 H'를 H로, 서브 디렉토리 S'를 S로 하여 데이터 R을 삽입하기 위해 다시 ②부터 재시도 한다.
⑤ 엔트리 E가 비어 있으면 삽입하려는 데이터 R을 엔트리 E가 가리키도록 한다.
⑥ 엔트리 E가 이미 다른 데이터 R'를 가리키고 있어 충돌이 발생하면, 디렉토리 S의 서브 디렉토리 개수를 조사한다.
⑦ 디렉토리 S의 서브 디렉토리 개수가 디렉토리 S의 엔트리 수의 반보다 작으면 크기가 2인 서브 디렉토리 S'를 만들어 엔트리 E가 서브 디렉토리 S'를 가리키도록 한다.
⑧ 전역 변수 t를 d만큼 증가시키고, 해시 값 H의 오른쪽 d 비트를 제외한 H'를 H로, 서브 디렉토리 S'를 S로 하여 데이터 R을 삽입하기 위해 다시 ②부터 재시도 한다. 그리고 데이터 R'에 대해서는 해시 값 H의 오른쪽 t 비트를 제외한 H"를 H로 서브 디렉토리 S'를 S로 하여 다시 ②부터 재시도 한다.
⑨ 디렉토리 S의 서브 디렉토리 개수가 디렉토리 S의 엔트리 수의 반보다 크거나 같으면 디렉토리 S의 크기를 두 배로 확장한다.
⑩ 데이터 R에 대해서 ②부터 다시 수행한다. 그리고 데이터 R'에 대해서는 해시 값 H의 오른쪽 t 비트를 제외한 H"를 H로 서브 디렉토리 S'를 S로 하여 다시 ②부터 재시도 한다.

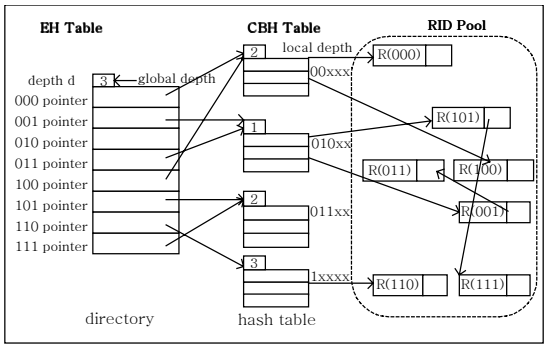
6. 확장된 체인 버켓 해싱

ECBH는 (그림 14)와 같이 확장 해싱과 체인 버켓 해싱을 결합해서 만든 해시 인덱스로, ECBH 인덱스는 EH 테이블, CBH 테이블, RID 풀(pool)의 3 부분으로 구성되며, EH에서의 리프 페이지를 CBH의 해시 테이블과 RID 리스트로 대치한 형태를 갖는다.

ECBH의 EH 테이블은 본래의 EH에서의 디렉토리와 마찬가지로 처리되며, CBH 테이블은 본래의 CBH에서와는 달리 해시 테이블에 지역 깊이 정보가 유지된다. 디렉토리와 해시 테이블에서의 깊이 정보는 본래 EH에서와 동일한 의미로 사용된다.

EH 테이블의 각 엔트리는 CBH 테이블에 대한 포인터를 유지하고, CBH 테이블의 각 엔트리는 RID 풀에 저장된 그 엔트리와 관련된 RID 리스트에 대한 포인터를 값으로 갖는다.

ECBH에서는 처리하고자 하는 데이터를 해시 함수에 의해 해시하여 구한 해시 값 H를 <H1:H2> 두 부분으로 나누어 H2는 해시 테이블을 찾기 위해서 디렉토리의 엔트리를 구하는 데 사용하고, H1은 디렉토리의 엔트리가 가리키는 해시 테이블 내에서 찾고자하는 RID 리스트를 유지하는 엔트리를 결정하는 데 사용한다. 구조적 차이 이외의 기타 연산은 EH에서와 동일하게 수행된다.



(그림 14) 확장된 체인 버켓 해싱의 구조

IV. 결 론

최근 들어 인터넷을 기반으로 하는 전자상거래 및 이동통신을 위한 PCS(Personal Communication System)에서의 HRL(Home Register Location), 이동 컴퓨팅, 포탈 사이트 등 신속한 처리를 요구하는 정보검색시스템의 응용분야가 점차 확대되고 있다.

현재 널리 사용되고 있는 디스크 기반 정보검색 시스템에서는 데이터를 다루기 위한 디스크 액세스의 오버헤드가 지나치게 크므로 빠른 처리를 요구하는 응용에는 적합하지 않다. 더구나 주기억장치의 용량이 커지고 가격이 많이 하락함에 따라 컴퓨터 시스템 내의 주기억장치 용량은 점점 증가하는 추세에 있다. 이에 따라 정보검색 분야에서는 늘어나는 주기억장치의 용량을 최대한 활용하여 디스크 내에 저장된 데이터를 모두 주기억장치로 상주시켜 정보검색시스템의 성능을 개선하는 주기억장치 기반의 검색시스템(memory resident retrieval system)에 관한 연구가 진행중에 있다.

이에, 지금까지 살펴본 인덱스 기법은 검색시스템의 하부 구조로 사용되어 지며 데이터를 물리적 저장매체에 저장하고 색인/검색하는 데 이용된다.

우선 트리 계열 인덱스를 보면, AVL 트리는 노드마다 데이터 하나에 포인터 두 개씩 유지되므로 메모리의 낭비 문제가 발생하고, 중복키를 처리하는 데 어려움이 있으며, B 트리는 트리의 깊이가 일정치 않음으로 worst execution time이 길어지는 단점이 있는 반면에 T 트리는 AVL 트리와 B 트리의 장점을 이용하여 주기억장치에 적합하도록 고안된 기법으로 메모리 사용의 낭비가 적고, 데이터에 대한 빠른 접근이 가능한 장점이 있다.

또한 해시 계열 인덱스의 경우 체인 버켓 해싱(CBH)은 삽입 데이터가 많을 때 체인의 길이가 길어지고, 삽입 데이터가 적을 때 공간 이용률이 낮기 때문에 다이나믹 파일에서 확장 또는 축소 시 해시 테이블 전체를 재구성해야 하는 단점을 가지고 있으며, 확장 해싱(EH)은 리프 노드의 사이즈가 상대적으로 작을 경우 디렉토리의 크기가 커지는 단점이 있다. 이외에도 선형 해싱(LH)은 데이터량이 동적으로 변하는 환경에 적합한 기법인 반면에 삽입 및 탐색이 느린 단점을 가지고 있으며, 확장 선형 해싱

(MLH)은 주소 공간이 선형적으로 확장되기 때문에, 키를 주소로 전환하는 시간이 일정하지 않고, 모든 hash value가 단일해야 하는 단점을 가지고 있다. 하지만 확장된 체인 버켓 해싱(ECBH)은 EH과 CBH를 결합해서 만든 해싱 기법으로서, EH 테이블, CBH 테이블, RID 풀의 3 부분으로 구성되고, EH에서의 리프 페이지를 CBH의 해시 테이블과 RID 리스트로 대치한 형태를 갖기 때문에 ECBH는 CBH에서의 고성능을, EH에서의 단계적인 확장의 용이성이 있다는 것을 볼 수 있다. 따라서 주기억장치 기반의 정보검색시스템에서 활용하기 위한 인덱싱 기법으로 트리 계열 인덱스로는 T-트리가 적합하고, 해시 계열 인덱스로는 ECBH가 적합하다는 것을 제안하고자 한다.

참 고 문 헌

[1] A. Aho, J. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[2] Douglas Comer, "The Ubiquitous B-Tree," Computing Surveys, Vol. 11, No. 2, 1979.

[3] Tobin J. Lehman and Michael J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," *Proc. of Int'l Conf. on VLDB*, 1986.

[4] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, H. Raymond Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files," *ACM Trans. on Database Sys.*, Vol. 4, No. 2, 1979.

[5] Witold Litwin, "Linear Hashing: A New Tool for File and Table Addressing," *Proc. of Int'l Conf. on VLDB*, 1980.

[6] Per-Ake Larson, "Dynamic Hash Tables," *Communications of the ACM*, Vol. 31, No. 4, 1988.

[7] Anastasia Analyti and Sakti Pramanik, "Fast Search in Main Memory Databases," *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1992.