데이터베이스: 강의노트 08

A. Silberschatz, H. Korth, S. Sudarshan
Database System Concepts,
Fourth Edition, McGraw-Hill, 2002.

Part IV. Data Storage and Querying

12 색인과 해싱

12.1 기본 개념

- 파일에 대한 색인은 일반 책에서 사용하는 색인과 같은 원리로 동작한다.
- 색인은 정렬되어 있고, 검색을 빠르게 할 수 있도록 도와주며, 책에 비해 상대적으로 적은 양이다.
- 색인의 종류
  - 순서 색인: 정렬된 값을 기반한 색인
  - 해시 색인: 해시함수를 이용한 색인
- 색인 기법을 평가하는 기준
  - 접근 타입: 효율적 접근이 지원되는 접근의 유형
  - 접근 시간: 특정 데이터 항목을 찾는데 소요되는 시간
  - 삽입 시간: 새 데이터 항목을 삽입하는데 소요되는 시간을 말하며, 실제 데이터를 삽입하는 시간과 색인을 갱신하는데 걸리는 시간을 합친 시간이다.
  - 삭제 시간: 기존 데이터 항목을 삭제하는데 소요되는 시간을 말하며, 실제 데이터를 삭제하는 시간과 색인을 갱신하는데 걸리는 시간을 합친 시간이다.
  - 공간 오버헤드: 색인을 차지하는 추가적 공간을 말하며, 적당한 수준의 공간 오버헤드는 성능 향상을 위해 희생하는 것은 가치가 있다.

12.2 순서 색인

- 색인 구조는 특정 검색키와 연관된다.
- 순서 색인은 검색키의 값을 정렬하여 유지하며, 각 값과 그 값이 포함되어 있는 레코드의 위치를 연관해 놓는다.
- 파일은 여러 개의 색인을 가질 수 있다.
- **주 색인(primary index)**: 색인으로 사용하는 검색키가 파일에 저장되어 있는 레코드의 순서를 정의하는 색인을 말하며, 다른 말로 클러스터 색인(clustering index)이라 한다.
- **보조 색인(secondary index)**: 색인으로 사용하는 검색키가 파일에 저장되어 있는 레코드의 순서와는 다른 순서를 정의하는 색인을 말하며, 다른 말로 비클러스터 색인(non-clustering index)이라 한다.

12.2.1 주 색인

- 파일에 레코드가 어떤 검색키를 기준으로 정렬되어 저장되어 있고 이 검색키를 사용하는 주 색인이 있으면, 이 파일을 색인 순차 파일(index-sequential file)이라 한다.
- 색인 레코드 또는 색인 항은 검색키 값과 이 값을 포함하는 레코드를 가라키는 하나 이상의 포인터로 구성된다. 이 포인터는 파일 내에 레코드가 저장되어 있는 블록의 식별자와 그 블록 내의 오프셋으로 구성된다.
- 주 색인의 색인 레코드는 하나의 포인터만 유지하거나 이 색인 레코드와 같은 값을 가지는 레코드에 대한 모든 포인터를 유지한다. 하나의 포인터만 유지하는 경우에는 추가로 순차 검색이 필요할 수 있다.

12.2.1.1 전체와 일부 색인

- 순서 색인의 종류
  - **전체 색인(dense index)**: 사용되고 있는 모든 검색키 값을 색인으로 사용하는 순서 색인을 말한다. 전체 주 색인(dense primary key)의 색인 레코드는 검색키 값과 그 값을 포함하는 첫 레코드를 가리키는 포인터로 구성된다. 같은 값을 가지는 레코드는 첫 레코드 이후에 순차적으로 저장된다.
  - **일부 색인(sparse index)**: 사용되고 있는 검색키 값 중 일부만 색인으로 사용하는 순서 색인을 말한다. 이 환경에서 검색키 값이 a인 특정 레코드를 찾고 싶으면 a와 같거나 작지만 가장 큰 색인을 이용한다.
- 접근 시간 측면에서는 전체 색인이, 공간 오버헤드 측면에서는 일부 색인이 유리하다. 또한 삽입과 삭제 시 색인 갱신 측면에서도 일부 색인이 유리하다.
- 일반적으로 일부 색인을 사용하며, 한 블럭마다 하나의 색인을 사용한다. 이것은 블록을 주기억장치에 적재한 후에 그 블록을 순차 검색하는 것은 빠르게 할 수 있기 때문이다.

12.2.1.2 다단계 색인

- 일부 색인 방식을 사용하더라도 효율적으로 처리하기에는 너무 클 수 있다.
- 예) 한 파일에 100,000 개의 레코드가 저장되어 있으며, 각 블럭에 10 개의 레코드가 저장된다고 가정하자.

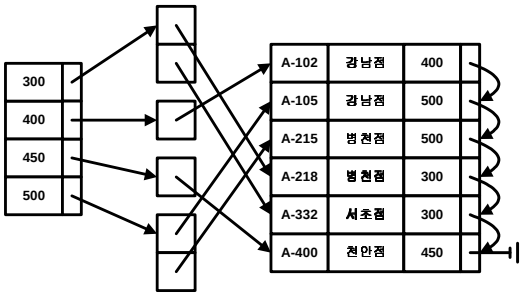
  - 블럭마다 하나의 색인을 유지하면 10,000 개의 색인이 필요하다.
  - 색인의 크기가 작아 한 파일 블록에 100 개의 색인 블록이 저장된다고 하자.
  - 그러면 색인을 저장하기 위해서는 100 개의 블록이 필요하다.

- 색인이 작아 주기억장치에 모두 유지할 수 있으면 빠르게 검색할 수 있다. 그러나 위 예처럼 색인이 커서 여러 블록에 나누어 디스크에 저장되어 있다면 많은 디스크 접근이 필요하다.
- 색인이 b개 블록에 저장되어 있는 경우에 이진 트리 검색을 하더라도 $\lceil \log_2(b) \rceil$[1] 블록을 읽어야 한다.
- 다단계 색인: 색인의 색인을 사용하는 방식으로 디스크 입출력을 많이 줄일 수 있다.

12.2.1.3 색인의 갱신

- 어떤 형태의 색인을 사용하더라도 레코드가 삽입되고 삭제될 때마다 색인은 갱신되어야 한다.
- 삽입: 삽입할 레코드 r의 검색키 값 a을 이용하여 색인을 검색한다.
  - 전체 색인
    1. a가 색인에 존재하지 않으면 r를 적절한 위치에 삽입하고, 색인에 a를 추가한다.
    2. a가 색인에 존재하면
       a. r를 같은 a를 가지는 레코드들 뒤에 삽입한다.
       b. 색인 레코드가 같은 검색키 값을 가지는 모든 레코드의 포인터를 유지한다면 이 레코드에 포인터를 추가한다.
  - 일부 색인: 각 블럭마다 하나의 색인을 유지한다고 가정한다.
    - 삽입 때문에 새 블럭을 생성해야 한다면 새 블럭의 첫 레코드의 해당하는 검색키 값을 색인에 추가한다.
    - 삽입이 블럭의 첫 위치에 삽입되어야 하면 블럭의 색인값을 a로 갱신한다.
- 삭제: 삭제할 레코드 r의 검색키 값 a을 이용하여 색인을 검색한다.
  - 전체 색인
    1. a 값을 가지는 레코드가 r 밖에 없으면 a를 색인에서 삭제한다.
    2. a 값을 가지는 레코드가 r 외에 더 있으면
       1) 색인 레코드가 같은 검색키 값을 가지는 모든 레코드의 포인터를 유지한다면 r의 포인터를 삭제한다.
       2) 색인 레코드가 첫번째 레코드의 포인터만을 유지할 때 r이 첫번째 레코드이면 색인 레코드의 포인터를 두번째 레코드를 가리키도록 갱신한다.
  - 일부 색인: 각 블럭마다 하나의 색인을 유지한다고 가정한다.
    1. 색인에 a 값을 가지는 색인 레코드가 없으면 r만 삭제한다.
    2. 색인에 a 값을 가지는 색인 레코드가 있으면
       1) a 값을 가지는 레코드가 r 밖에 없으면 r를 삭제하고, a 값을 가지는 색인 레코드는 그 다음 값으로 갱신한다. 이 때 그 다음 값에 해당하는 색인 레코드가 있으면 a 값을 가지는 색인 레코드를 삭제한다.
       2) 색인 레코드가 첫번째 레코드의 포인터만을 유지할 때 r이 첫번째 레코드이면 색인 레코드의 포인터를 두번째 레코드를 가리키도록 갱신한다.

12.2.2 보조 색인



<그림 12.1> 보조 색인

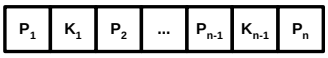
- 주 색인은 일부 색인을 유지할 수 있지만 보조 색인은 전체 색인을 유지해야 한다. 레코드는 보조 색인을 기준으로 정렬되어 있지 않기 때문에 전체 색인을 사용하지 않으면 검색에 도움이 되지 않는다.
- 주 색인은 하나의 포인터만 유지할 수 있지만 보조 색인은 같은 값을 가지는 모든 레코드에 대한 포인터를 유지해야 한다.
- 이를 위해 추가적인 간접 단계를 사용할 수 있다. 즉, 색인 레코드가 바로 각 레코드에 대한 포인터를 유지하지 않고, 포인터들을 유지하는 배열을 가리킨다. 그림 12.1 참조.
- 삽입과 삭제 시 보조 색인에 대한 갱신은 주 색인에서 전체 색인을 사용하는 경우와 같다.
- 보조 색인을 사용하면 검색키가 아닌 다른 것을 이용해야 하는 질의를 빠르게 처리할 수 있다. 그러나 데이터베이스 변경에 대한 상당한 오버헤드를 강요한다.

[1] $\lceil a \rceil$: a보다 크거나 같은 가장 작은 정수

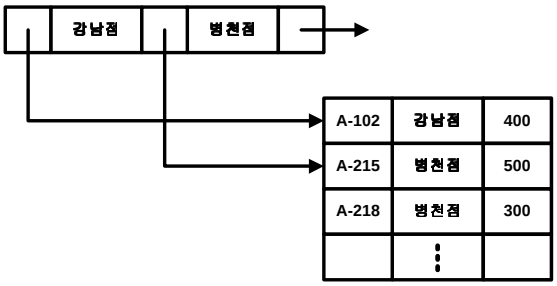
12.3 B$^+$-트리 색인 파일

- 색인 순차 파일은 파일의 크기가 커지면 커질 수록 성능이 떨어진다.
- B$^+$-트리는 현재 가장 널리 사용되는 색인 구조이며, 레코드를 삭제하거나 삽입하더라도 그 성능을 계속 유지한다.
- B$^+$-트리는 균형 트리(balanced tree)이다. 즉, 루트 노드에서 모든 단말 노드까지의 경로의 길이가 같다.
- B$^+$-트리 색인 구조를 사용하면 삽입과 삭제 때 오버헤드가 있고, 공간 측면에서도 오버헤드가 있지만 자주 갱신되는 파일의 경우에는 파일 재정리가 필요없기 때문에 전체적 성능은 오히려 좋다.

12.3.1 B$^+$-트리의 구조



<그림 12.2> B$^+$-트리의 노드 구조

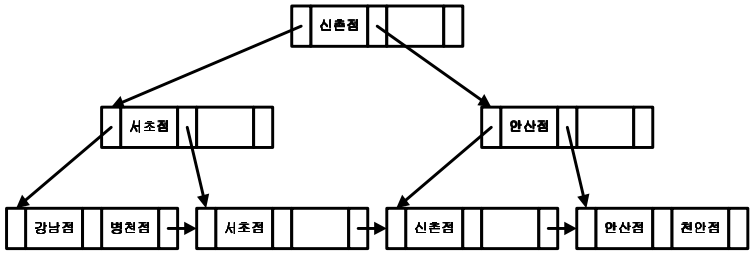


<그림 12.3> $n = 3$인 B$^+$-트리의 단말 노드의 예

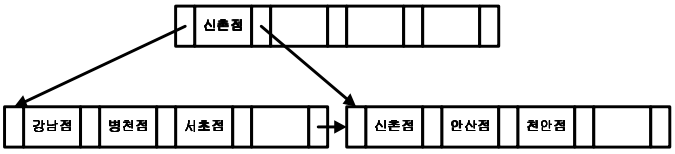
- B$^+$-트리의 노드 구조: 그림 12.2
  - 한 노드는 최대 $n-1$개의 검색키 값을 가진다. (K_1, K_2, ..., K_{n-1})
  - 한 노드는 최대 n개의 포인터를 가진다. (P_1, P_2, ..., P_n)
  - 노드의 포인터 수를 그 노드의 팬아웃(fan-out)이라 한다.
  - 노드 내의 검색키 값은 오름차순으로 정렬되어 있다. 즉, $i < j$이면 $K_i < K_j$이다.
- 단말 노드의 구조: 그림 12.3
  - 단말 노드의 포인터 P_i는 검색키 값 K_i를 가지는 레코드를 가리키거나 검색키 값 K_i를 가지는 모든 레코드에 대한 포인터로 구성된 버켓을 가리킨다.
  - P_n은 오른쪽에 있는 단말 노드를 가리키며, 순차 검색을 할 때 사용된다.
- 노드의 검색키 할당 방법
  - **단말 노드**
    - 한 단말 노드에 최대 $n-1$개의 검색키 값을 할당할 수 있다.
    - 한 단말 노드에는 최소 $\lceil (n-1)/2 \rceil$개의 검색키 값을 가지고 있어야 한다.
    - 각 단말 노드에 있는 검색키 값은 중첩되지 않는다.
    - P_n를 이용하여 단말 노드를 연결 리스트 형태로 유지한다. 이 리스트를 이용하여 효율적으로 순차 검색을 할 수 있다.
  - **비단말 노드**: 단말 노드에 대한 다단계 일부 색인 역할을 한다.
    - 비단말 노드의 구조는 단말 노드와 같다. 다만, 비단말 노드의 포인터는 파일의 레코드를 가리키지 않고 모두 다른 노드를 가리킨다.
    - 비단말 노드는 최대 n개의 포인터를 가질 수 있으며, 최소 $\lceil n/2 \rceil$개의 포인터를 가져야 한다.
    - m개의 포인터를 가진 비단말 노드를 고려하여 보자.
      - 포인터 $P_i (1 < i < n)$는 K_i보다 작고 K_{i-1}보다는 같거나 큰 검색키 값을 포함하는 노드를 가리킨다.
      - 포인터 P_m은 K_{m-1}보다 크거나 같은 검색키 값을 포함하는 노드를 가리킨다.
      - 포인터 P_1은 K_1보다 작은 검색키 값을 포함하는 노드를 가리킨다.
  - **루트 노드**
    - 다른 비단말 노드와 달리 $\lceil n/2 \rceil$개 보다 적은 수의 포인터를 가질 수 있다. 그러나 최소 두 개의 포인터는 가져야 한다.
- B$^+$-트리의 예
  - 예) $n = 3$: 그림 12.4
  - 예) $n = 5$: 그림 12.5

12.3.2 B$^+$-트리에 대한 질의

- B$^+$-트리를 이용하여 검색키가 V인 모든 레코드를 찾는 방법
  - 단계 1. 루트 노드를 검색하여 V보다 크지만 가장 작은 검색키 값 K_i를 찾는다.
    - 찾으면 P_i를 이용하여 자식 노드로 이동한다.
    - 없으면 P_m(노드의 마지막 포인터)를 이용하여 자식 노드로 이동한다.
  - 단계 2. 이동한 노드에 대해 단계 1를 반복한다. 단말 노드까지 이동할 때까지 이것을 반복한다.



<그림 12.4> $n = 3$인 B$^+$-트리의 예



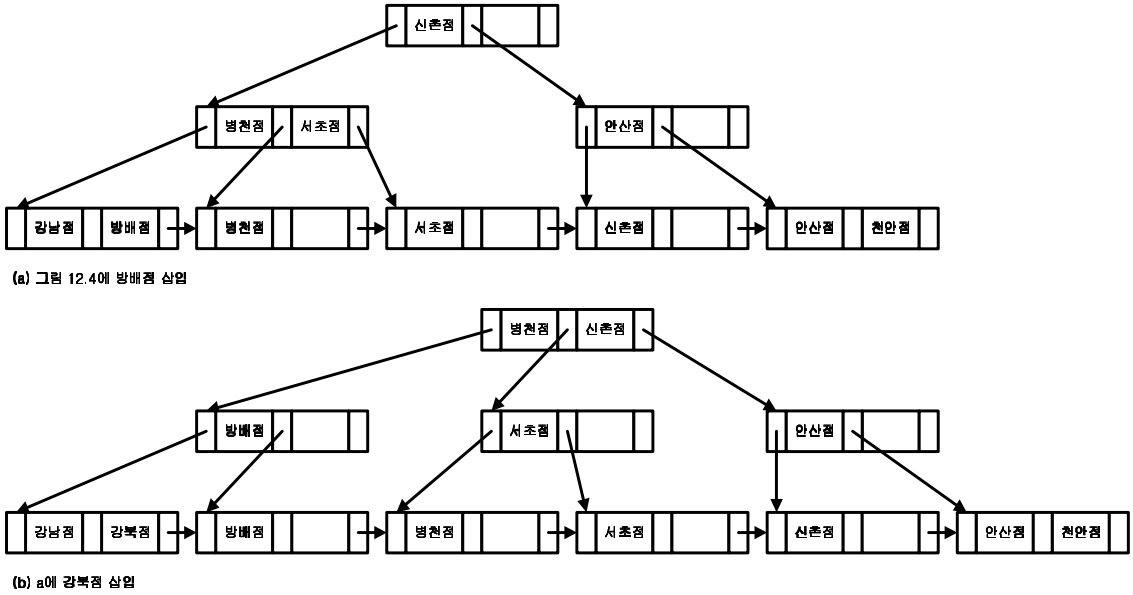
<그림 12.5> $n = 5$인 B$^+$-트리의 예

  - 단계 3. 단말 노드에서 V와 같은 검색키 값 K_i를 찾는다.
    - 찾으면 P_i를 이용하여 레코드를 찾을 수 있다.
    - 없으면 V 값을 가지는 레코드가 없다는 것을 의미한다.

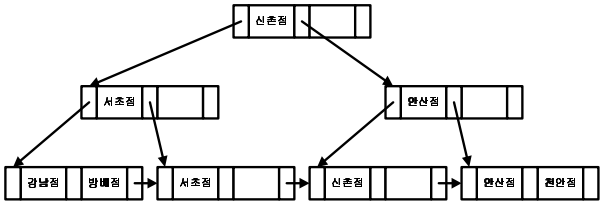
- 파일에 K개의 검색키가 존재하면 루트에서 단말노드까지의 경로는 최대 $\lceil \log_{\lceil n/2 \rceil} K \rceil$이다.
- 전형적으로 한 노드는 디스크 블록의 크기와 같도록 만든다.
- 보통 디스크 블록의 크기는 4 KB이다.
- 검색키 값의 크기가 32 바이트라 하고, 디스크 포인터의 크기가 8 바이트라 하면 n은 대략 100이 된다.
- 파일에 1,000,000 개의 검색키 값이 있다고 하자. 그러면 $\lceil \log_{50} 1,000,000 \rceil = 4$ 노드만 접근하면 된다. 즉, 최대 4 블럭만 접근하면 찾을 수 있다. 더구나 루트 노드는 항상 버퍼에 있으므로 3 블럭 이하의 디스크 접근만 필요하다.

12.3.3 B$^+$-트리에 대한 갱신

- B$^+$-트리에서 삽입과 삭제는 검색보다 어렵다. 삽입할 노드에 공간이 없으면 노드를 둘로 나누어야 하고, 노드에서 한 값을 삭제하여 포인터의 수가 $\lceil n/2 \rceil$개 이하로 내려가면 두 노드를 결합하거나 포인터를 재분배해야 한다. 삭제할 때에는 재분배보다는 결합이 우선이며, 결합은 반드시 형제 노드 간에만 이루어진다.
- 결합이나 분할이 필요없는 경우
  - 삽입: 검색을 하여 삽입할 검색키 값이 포함되어야 하는 단말 노드를 찾는다. 이미 검색키 값이 있으면 파일에 새 레코드만 추가한다. 없으면 단말 노드에 검색키 값을 새로 추가한다. 이 때 검색키 값이 정렬된 순서를 유지하도록 추가한다. 그 다음 파일에 새 레코드를 추가하고 포인터를 갱신한다.
  - 삭제: 검색을 하여 삭제할 레코드를 찾아 파일에서 삭제한다. 삭제된 레코드에 해당하는 검색키 값을 가진 다른 레코드가 없으면 이 검색키 값을 단말 노드에서 제거한다.
- 분할이 필요한 경우: 예1) 그림 12.4에 "방배점"을 삽입
  - 검색을 하면 "방배점"은 "강남점"과 "병천점"이 있는 단말 노드에 삽입되어야 한다는 것을 알 수 있다. 그러나 삽입할 공간이 없다.
  - 따라서 이 노드를 분할한다. 단말 노드를 분할할 때에는 n 개(기존 $n-1$개의 검색키 값과 새 값)의 검색키 값 중에서 $\lceil n/2 \rceil$ 개를 기존 노드에 두고 나머지를 새 노드로 옮긴다.
  - 분할을 한 다음에는 새 노드를 트리에 추가해야 한다. 새 노드의 가장 작은 검색키 값이 "병천점"이다. 그러므로 이 키 값을 부모 노드에 추가해야 한다. 만약 부모 노드에 추가할 공간이 없으면 부모 노드도 분할되어야 한다.
  - 결과: 그림 12.6의 (a)
- 분할이 필요한 경우: 예2) 그림 12.6의 (a)에 "강북점"을 삽입
  - 이 경우에는 단말 노드간 분할된 이후 "방배점"이 부모 노드에 추가로 삽입되어야 한다. 그러나 부모 노드에 충분한 공간이 없이 부모 노드도 분할되어야 한다.
  - 비단말 노드를 분할할 때에는 n 개의 검색키 값 중에서 $\lceil n/2 \rceil$ 개를 새 노드로 옮긴다. 그 다음 이 중 가장 작은 값은 새 노드에서 제거하고 부모 노드로 옮긴다.
  - 결과: 그림 12.6의 (b)

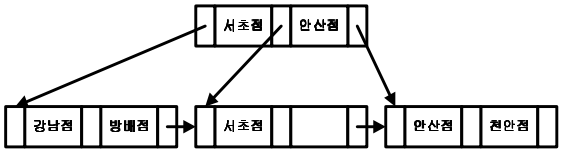


<그림 12.6> 그림 12.4에 “방배점” 삽입



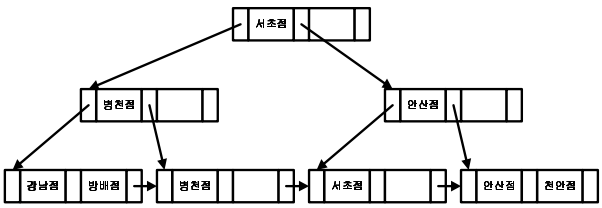
<그림 12.9> 그림 12.6에서 “병천점” 삭제

- $n = 4$인 B^+ 트리에 대한 삽입의 또 다른 예: 그림 12.7와 그림 12.8 참조.
- 결합이 필요없는 경우: 예1) 그림 12.6의 (a)에서 “병천점”을 삭제
  - 검색을 하여 “병천점”이 있는 노드에서 “병천점”을 삭제하면 그 노드는 빈 상태가 된다.
  - 따라서 이 노드는 트리에서 제거되어야 한다. 이 노드를 제거한 다음에는 부모 노드의 포인터를 제거해야 한다. 여기서 해당 포인터를 제거해도 노드에 충분한 포인터가 있으므로 이것으로 제거는 완료된다.
  - 결과: 그림 12.9 참조.



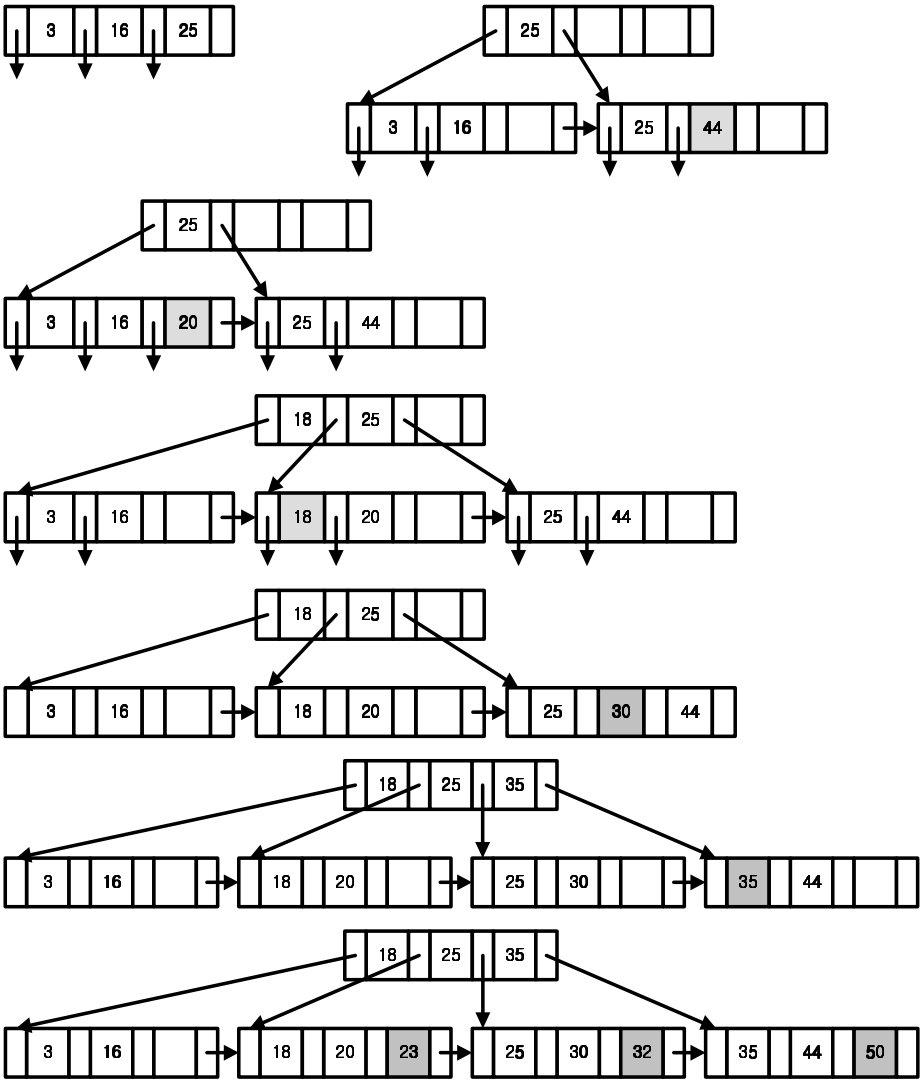
<그림 12.10> 그림 12.9에서 “신촌점” 삭제

- 결합이 필요한 경우: 예2) 그림 12.9에서 “신촌점”을 삭제
  - 검색을 하여 “신촌점”이 있는 노드에서 “신촌점”을 삭제하면 그 노드는 빈 상태가 된다.
  - 따라서 이 노드는 트리에서 제거되어야 한다. 이 노드를 제거한 다음에는 부모 노드의 포인터를 제거해야 한다. 이 때 포인터를 제거하면 해당 노드의 포인터 수가 충분하지 않다.
  - 이 경우에는 형제 노드를 검사한다. 형제 노드에 수용할 공간이 있으면 남은 검색키 값을 형제 노드로 옮기고, 기존 노드는 삭제한다. 이 삭제에 대해 부모 노드의 포인터가 삭제되고 필요하면 같은 방법으로 결합한다.
  - 결과: 그림 12.10 참조.



<그림 12.11> 그림 12.6에서 “신촌점” 삭제

- 결합이 필요한 경우: 예3) 그림 12.6의 (a)에서 “신촌점”을 삭제
  - 검색을 하여 “신촌점”이 있는 노드에서 “신촌점”을 삭제하면 그 노드는 빈 상태가 된다.
  - 따라서 이 노드는 트리에서 제거되고, 그것의 부모 노드의 포인터도 제거된다. 이 결과 부모 노드의 포인터 수가 충분하지 않다.



<그림 12.7> $n = 4$인 B$^+$-트리에 대한 삽입

- 그러나 형제 노드을 검사하여도 수용할 공간이 없다. 형제 노드가 수용할 수 없으면 각 형제 노드가 같은 수의 포인터를 가지도록 포인터를 재분배하거나 하나의 값만 형제 노드로부터 가지고 온다. 이런 재분배는 부모 노드의 검색키 값의 변경을 요구한다.
- 결과: 그림 12.11 참조.

12.3.4 B$^+$-트리 파일 조직

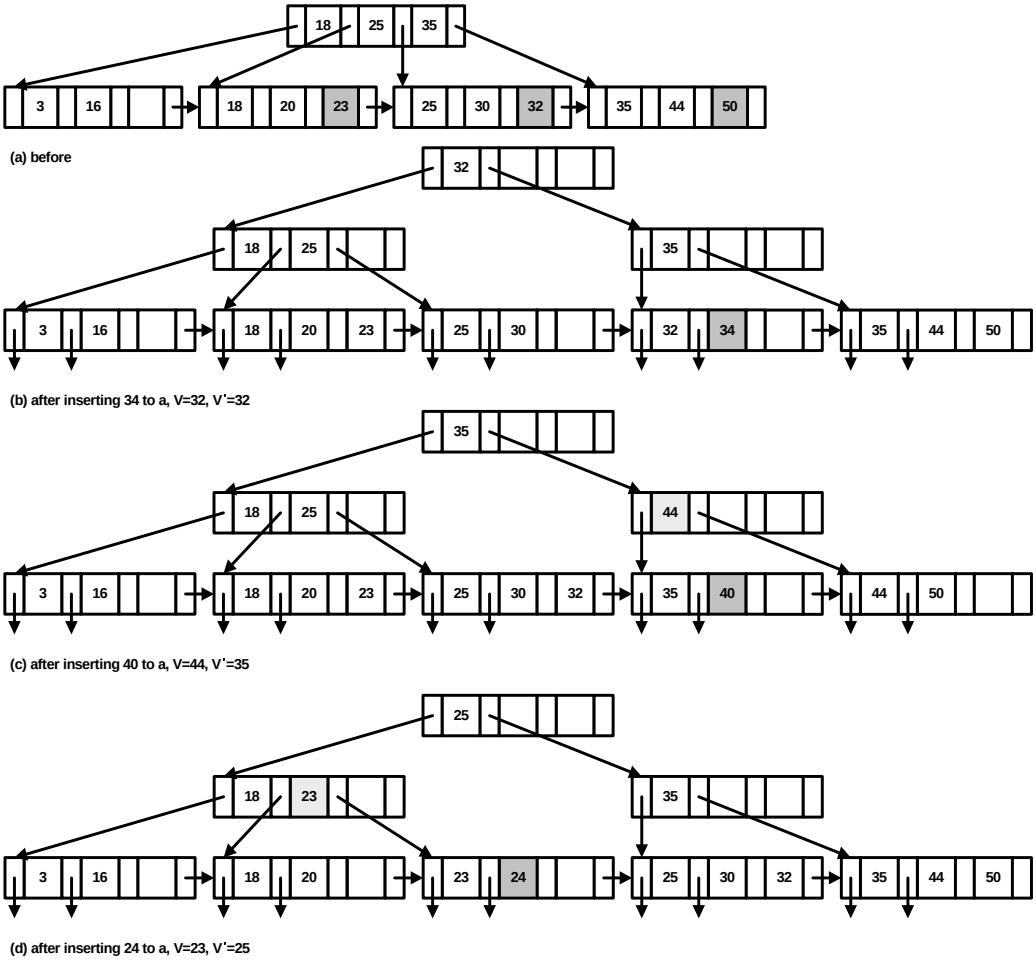
- 색인 뿐만 아니라 실제 레코드도 B$^+$-트리 형태로 파일에 저장하고 관리할 수 있다.
- B$^+$-트리 파일 조직에서는 트리의 단말 노드에 포인터 대신에 레코드를 저장한다.
- 보통 레코드는 포인터보다 크기 때문에 단말 노드에 저장할 수 있는 레코드의 수는 포인터의 수보다는 적다.
- 레코드의 삽입과 삭제는 B$^+$-트리 색인 구조와 같다.
- 공간 활용을 보다 효율적으로 하기 위해 삽입과 삭제시 재분배해야 하면 두 노드가 같은 수의 레코드를 가지도록 한다. 하지만 이동되는 레코드가 많아지면 비용도 그만큼 증가한다.

12.4 정적 해싱

- 순차 파일 조직의 단점은 데이터를 검색하기 위해 색인 구조를 접근해야 하므로 디스크 입출력이 추가로 소요된다는 것이다. 해싱은 색인 구조에 대한 접근 없이 데이터를 검색할 수 있다.

12.4.1 해시 파일 조직

- 해시 파일 조직에서는 레코드의 검색키 값에 대한 해시값을 계산하여 그 레코드가 있는 디스크 블록의 주소를 바로 얻을 수 있다.
- 버켓(bucket)이란 하나 이상의 레코드를 저장할 수 있는 저장 단위를 말한다.



<그림 12.8> $n = 4$인 B$^+$-트리에 대한 삽입

- K가 모든 검색키 값의 집합이고, B가 모든 버켓 주소의 집합이면 해시 함수 h는 K에서 B로의 함수이다. 즉, 처음부터 고정된 수의 버켓을 사용한다.
- 검색키 값이 K_i인 레코드를 삽입하기 위해서는 먼저 $h(K_i)$를 계산하여 버켓의 주소를 얻은 다음 그 버켓에 레코드를 삽입한다.
- 검색키 값이 K_i인 레코드를 검색하기 위해서는 먼저 $h(K_i)$를 계산하여 버켓의 주소를 얻은 다음 그 버켓에서 레코드를 검색한다.
- 검색키 값이 K_i인 레코드를 삭제하기 위해서는 먼저 $h(K_i)$를 계산하여 버켓의 주소를 얻은 다음 그 버켓에서 레코드를 검색하여 삭제한다.

12.4.1.1 해시함수

- 해시함수의 출력의 분포는 다음과 같아야 한다.
  - 분포가 **균일**(uniform)해야 한다. 즉, 각 버켓마다 같은 수의 검색키를 할당해야 한다.
  - 분포가 **랜덤**해야 한다. 즉, 검색키 값의 분포와 상관없이 평균적으로 각 버켓에 같은 수의 값이 할당되어야 한다.
- 예1) *account* 관계에서 "지점명" 속성을 검색키로 사용할 때 26개의 버켓을 사용하여 i번째 영문자를 i번째 버켓에 사용하는 해시함수를 사용하기로 결정하였다. 이것은 균일 분포가 아니다. 'X'로 시작되는 지점명은 거의 없기 때문이다.
- 예2) "잔액" 속성을 검색키로 사용할 때 잔액의 가능 범위가 1부터 100,000이라 하자. 이 범위를 정확하게 10 개 영역으로 나누면 분포 자체는 균일 분포이지만 잔액이 1에서 10,000 사이인 경우가 90,001에서 100,000 사이인 경우보다 많으므로 랜덤이 아니다.
- 키 값이 같은 레코드가 많으면 해시함수와 상관없이 특정 버켓이 다른 버켓보다 상대적으로 많은 레코드를 할당받게 된다.

12.4.1.2 버켓 넘침의 처리

- 버켓 넘침이 발생하는 이유
  - 버켓의 부족: n_B가 버켓의 총 수이고, n_r이 저장될 총 레코드 수이며, f_r이 버켓 당 저장될 수 있는 레코드의 수이면 $n_B > n_r/f_r$이어야 한다. 그런데 총 저장될 레코드 수를 예측하기가 쉽지 않다.

  - 비대칭(skew): 어떤 버켓에 다른 버켓보다 많은 수의 레코드가 할당될 수 있다. 즉, 다른 버켓은 아직 공간이 남아있지만 어떤 버켓은 넘침이 발생할 수 있다. 비대칭이 발생할 수 있는 이유는 다음과 같다.
    - 같은 검색키 값을 가진 레코드가 여러 개 있을 수 있다.
    - 해시함수가 비균일 분포일 수 있다.

- 버켓 넘침 가능성을 최소화하기 위해 버켓의 수는 보통 다음과 같이 결정된다.

$$(n_r/f_r) \times (1+d)$$

  여기서 d는 초과 요인(fudge factor)이라 한다. 보통 0.2 정도의 값을 사용한다.

- 버켓 넘침은 넘침 버켓을 이용하여 해결한다. 버켓이 꽉 차면 넘침 버켓을 연결하여 사용한다. 이것을 **넘침 연결 방법(overflow chaining)**이라 한다.

- 넘침 연결 방법을 사용하는 해싱 기법을 **폐쇄 해싱(closed hashing)**이라 한다. 이것과 비교되는 **개방 해싱(open hashing)**은 버켓에 공간이 없으면 다른 버켓을 사용한다. 보통 그 다음 버켓을 사용하는데 이 방법을 **선형 조사(linear probing)** 방법이라 한다. 이 외에 해시 함수를 다시 또 계산하는 방법도 있다.

- 데이터베이스는 폐쇄 해싱을 선호한다. 이것은 개방 해싱에서는 레코드 삭제 비용이 매우 클 수 있다.

- 지금까지 설명한 해싱 방법의 문제는 시스템을 구현할 때 해시함수를 선택해야 하며, 이 함수를 변경하기가 어렵다는 것이다. 이렇게 한 번 설정한 해시함수를 변경할 수 없는 경우를 **정적 해싱**이라 한다.

12.4.2 해시 색인

- 해싱 기법은 파일 조직뿐만 아니라 색인 구조에 사용될 수 있다.

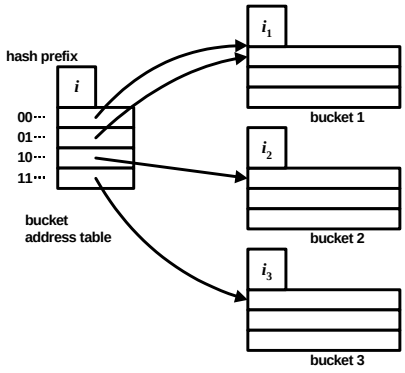
- 해시 색인 구조는 보조 색인 구조에만 사용된다. 파일 조직자체가 해시함수를 이용하여 조직되어 있으면 별도의 색인이 필요없다.

12.5 동적 해싱

- 정적 해싱의 문제점
  - 파일 크기를 바탕으로 해시함수를 선택하였다면 데이터베이스가 커짐에 따라 성능이 나빠진다.
  - 기대한 크기를 바탕으로 해시함수를 선택하였다면 초기에 공간 낭비가 많다.
  - 파일 크기가 증가함에 따라 주기적으로 해시 구조를 바꾼다. 이런 재구성은 해시함수를 새롭게 선택해야 하며, 파일에 있는 모든 레코드에 대해 해시값을 계산하여 파일을 새롭게 구성해야 한다. 이것은 시간이 많이 소요되며, 이 기간 동안에는 파일 접근을 배제해야 한다.

- 동적 해싱은 파일의 재구성없이 데이터베이스의 크기 변화를 수용할 수 있는 해싱 기법이다.

12.5.1 데이터구조



<그림 12.12> 일반 확장가능 해시 구조

- 동적 해싱에서 해시함수는 정적 해싱과 마찬가지로 균일해야 하며, 랜덤해야 한다. 그러나 동적 해싱에서 해시함수는 상대적으로 매우 큰 범위 값을 생성한다. 예를 들어 b 비트 정수를 생성하며, 이 비트의 값은 전형적으로 32 비트이다.

- 가능한 모든 해시함수의 값마다 버켓을 만들지 않는다.

- b 비트 해시값 중 그것의 일부만 사용한다. 데이터베이스의 크기가 변함에 따라 사용하는 비트 수가 변한다. 어떤 순간에 사용하는 비트 수를 i라 하며, i 비트 값은 버켓 주소 테이블에 대한 오프셋으로 사용된다. 그림 12.12 참조.

- i비트만 있으면 버켓 주소 테이블의 항을 찾을 수 있지만 일련의 테이블 항들이 같은 버켓을 가리킬 수 있다. 이렇게 같은 버켓을 가리키는 항은 모두 같은 해시 접두사(prefix)를 가지며, 이 접두사는 i보다 길이가 적을 수 있다.

- 각 버켓과 사용된 공통 접두사의 길이를 나타내는 정수값 j을 연관한다.

- 정수 j와 연관되는 버켓을 i_j라하면 버켓 j와 연관되는 버켓 주소 테이블의 항 수는 다음과 같다.

$$2^{i-i_j}$$

12.5.2 질의와 갱신

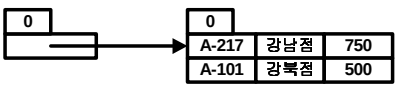
- 검색키 값이 K_l인 레코드를 검색하기 위해서는 먼저 $h(K_l)$에 상위 i 비트를 취한다. 그 다음 해당하는 버켓 주소 테이블의 항에 있는 포인터를 이용하여 버켓을 찾아간다.
- 검색키 값이 K_l인 레코드를 삽입하기 위해서는 검색과 같은 절차를 이용하여 레코드가 삽입될 버켓을 찾는다. 이 버켓에 공간이 있으면 레코드는 이 버켓에 삽입된다. 버켓에 공간이 없으면 새 버켓을 하나 만들어 기존 버켓에 있는 레코드를 재분배한다. 이 때 사용하는 i비트를 증가시킬 필요가 없는지 검사해야 한다.
  - $i = i_j$: 해시 비트의 길이를 늘리고 버켓을 분할한다.
  - $i > i_j$: 해시 비트의 길이를 늘릴 필요가 없고, 버켓만 분할하면 된다. 둘 이상의 버켓 주소 테이블 항이 같은 버켓을 가리키는 경우이다.
- 예) 다음과 같은 *account* 관계가 있다.

계좌번호	지점명	잔액
A-217	강남점	750
A-101	강북점	500
A-110	강북점	600
A-215	병천점	700
A-102	신촌점	400
A-201	신촌점	900
A-218	신촌점	700
A-222	잠실점	700
A-305	천안점	700

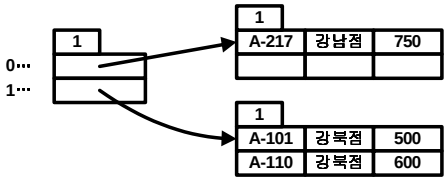
지점명에 대한 해시값은 다음과 같다.

지점명	h(지점명)
강남점	0010 1101 1111 1011 ... 0000
강북점	1010 0011 1010 0000 ... 1111
병천점	1100 0111 1110 1101 ... 1010
신촌점	1111 0001 0010 0100 ... 1101
잠실점	0011 0101 1010 0110 ... 1011
천안점	1101 1000 0011 1111 ... 0001

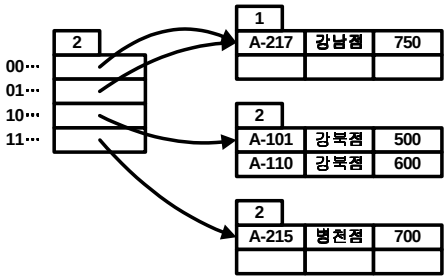
이 때 버켓에 두 개의 레코드만 저장할 수 있다고 가정할 때(실제 데이터베이스에서는 한 버켓에 많은 수의 레코드를 저장한다), 관계의 각 투플을 삽입하는 과정은 다음과 같다. 먼저 "A-217"과 "A-101"을 삽입하면 버켓이 꽉 찬다.



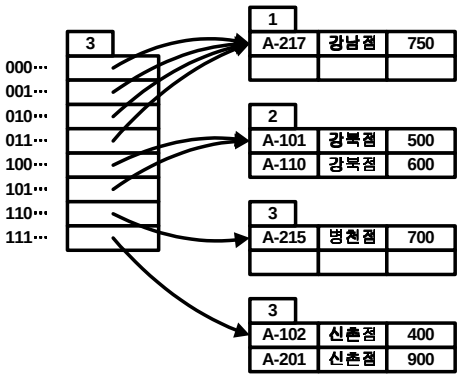
"A-110"을 삽입할 공간이 없으며, $i = i_0$이므로 버켓을 분할하여야 한다. 이를 위해 해시값의 사용되는 비트 수를 늘려야 한다. "강남점"에 해당 해쉬값의 첫 비트가 0이고, "강북점"에 해당 해쉬값의 첫 비트가 1이므로, 이것을 기준으로 분할한다. 결과는 다음과 같다.



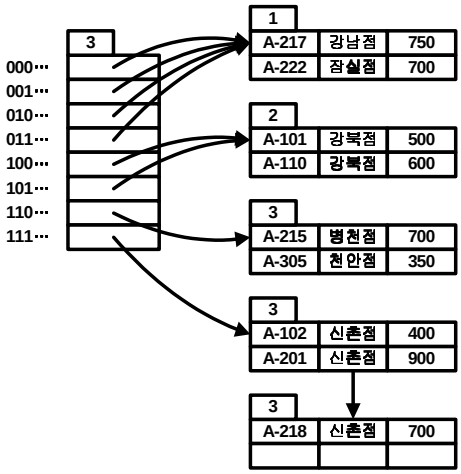
"A-215" 레코드의 해시값의 첫 비트가 1이므로 이 레코드는 "강북점" 레코드들이 들어있는 버켓에 삽입되어야 한다. 그러나 공간이 없고 $i = i_1$이므로 다시 분할하여야 한다. 결과는 다음과 같다.



"A-102" 레코드의 해시값의 첫 두 비트가 11이므로 "병천점" 레코드가 들어있는 버켓에 삽입된다. "A-201"도 "A-102"와 마찬가지로 같은 버켓에 들어가야 하지만 공간이 없고 $i = i_2$이므로 버켓을 다시 분할한다. 결과는 다음과 같다.



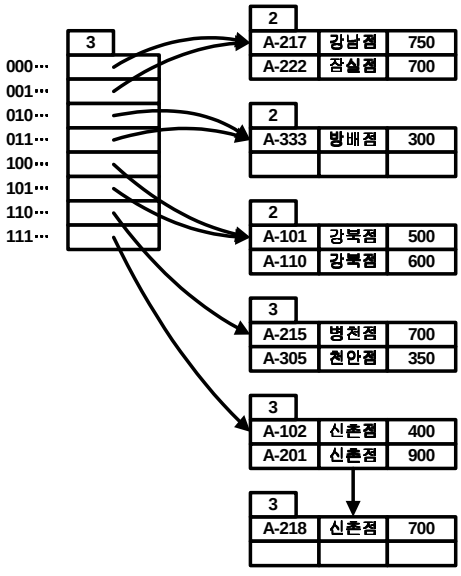
"A-218"은 삽입될 공간이 없다. 그러나 이 경우에는 모두 같은 지점명을 가지므로 해시의 비트를 늘려도 소용이 없다. 이 경우에는 넘침 버켓을 사용해야 한다. "A-218"과 나머지 레코드를 모두 삽입한 결과는 다음과 같다.



지점명이 "방배점"인 다음 레코드를 여기에 추가로 삽입할 경우을 생각하여 보자.

("A-333","방배점",300)

또한 h("방배점")=0101 ... 1111 이라 하자. 그러면 "A-217"이 들어있는 첫번째 버켓에 들어가야 하는데 공간이 없고 $i > i_j$이므로 해시 길이의 증가 없이 버켓만 분할한다.



12.5.3 다른 방식과 비교

- 확장 가능 해싱의 장점
  - 파일 크기가 증가하여도 성능이 떨어지지 않는다.
  - 미래를 위해 버켓 공간을 확보해 놓을 필요가 없이 그 때마다 동적으로 할당할 수 있다.
- 버켓 주소 테이블을 유지하기 위한 공간이 필요하지만 사용하는 해시값의 가능한 접두사 값마다 하나의 포인터만 유지하므로 비교적 크지 않다.
- 버켓을 접근하기 전에 버켓 주소 테이블을 접근해야 한다. 정적 해싱에서는 이런 추가 참조가 필요없다. 그러나 이런 추가 참조의 필요성은 성능에 큰 영향을 미치지 않는다.

12.6 정렬 색인과 해싱의 비교

- 파일 조직과 색인 기법을 선택할 때 고려 사항
  - 주기적인 색인 또는 해시 조직의 재구성 비용이 수용할만한가? B^+ 트리 색인 구조와 동적 해싱 파일 구조는 재구성할 필요가 없다.
  - 삽입과 삭제의 상대적 빈도는?
  - 최악 접근 시간을 희생하면서 평균 접근 시간을 최적화하는 것이 좋은가?
  - 어떤 종류의 질의가 많이 질의되는가?
- 예1) 다음과 같은 질의를 많이 한다고 하자.

$$\begin{array}{l}\text{select } A_1, A_2, \ldots, A_n \\ \text{from } r \\ \text{where } A_i = c\end{array}$$

A_i의 값이 c인 레코드를 찾아야 한다. 이 경우에는 해시 조직이 정렬 색인 구조에 비해 우수하다. 해시 조직에서는 평균 검색 시간은 데이터베이스 크기와 독립적이지만 정렬 색인 구조는 크기에 로그 비례한다. 다만 정렬 색인 구조는 최악 시간이 r 관계에 있는 A_i 값의 수에 로그 비례하지만 해시 구조는 그냥 비례한다. 그러나 해시 구조에서는 최악 시간은 거의 발생되지 않으므로 전체적으로 해시 구조와 이와 같은 질의에 더 우수하다.

- 예2) 다음과 같은 질의를 많이 한다고 하자.

$$\begin{array}{l}\text{select } A_1, A_2, \ldots, A_n \\ \text{from } r \\ \text{where } A_i \leq c_2 \text{ and } A_i \geq c_1\end{array}$$

이 경우에는 어떤 범위 내에 있는 모든 값을 찾아야 한다. 정렬 색인 구조의 경우에는 c_2를 찾은 다음에는 순차적으로 그 다음 값들을 쉽게 찾을 수 있다. 그러나 해시 조직에서는 c_2를 찾은 다음에 그 다음 값을 찾기가 어렵다. 연결 리스트를 추가로 유지할 수 있다고 생각할 수 있지만 해시 함수는 랜덤하므로 널리 퍼져있어 많은 버켓을 읽어야 한다.

12.7 SQL에서 색인 정의

- 데이터베이스 시스템이 자동으로 어떤 색인들을 유지할지 결정할 수 있지만 그러나 올바른 결정을 하는 것은 쉽지 않다.
- SQL 표준에는 색인의 생성과 제거 기능을 제공하지 않지만 대부분의 데이터베이스는 이 기능을 제공한다.

- 색인의 생성

  create index 색인이름
  on 관계이름(< 속성목록 >)

  여기서 "속성목록"은 검색키를 구성한 관계의 속성들을 말한다. 검색키가 후보키임을 선언하기 위해서는 색인을 생성할 때 unique를 사용한다.

  create unique index 색인이름
  on 관계이름(< 속성목록 >)

12.8 다중키 접근

12.8.1 다중 단일키 색인

- 예) *account* 관계가 "지점명"과 "잔액"에 대한 색인을 각각 유지한다고 할 때, 다음 질의의 처리를 생각하여 보자.

  select 계좌번호
  from *account*
  where 지점명="Perryridge" and 잔액 $= 1000$

  이 질의는 다음과 같은 세 가지 방법으로 처리할 수 있다.

  - 방법 1. "지점명"에 대한 색인을 이용하여 지점명이 "Perryridge"인 모든 레코드를 찾은 다음에 잔액이 1000인 것을 찾는다.
  - 방법 2. "잔액"에 대한 색인을 이용하여 잔액이 1000인 모든 레코드를 찾은 다음에 지점명이 "Perryridge"인 모든 레코드를 찾는다.
  - 방법 3. "지점명"에 대한 색인을 이용하여 지점명이 "Perryridge"인 모든 레코드의 포인터를 찾고, "잔액"에 대한 색인을 이용하여 잔액이 1000인 모든 레코드의 포인터를 찾아 두 포인터 집합의 교집합을 찾는다.

  위 방법 중 방법 3만이 다중 색인의 존재를 활용한다. 이 방법도 다음 세 가지 조건이 모두 성립하면 효과적이지 못하다.

  - 조건 1. 지점명이 "Perryridge"인 레코드가 매우 많다.
  - 조건 2. 잔액이 1000인 레코드가 매우 많다.
  - 조건 3. 지점명이 "Perryridge"이고 잔액이 1000인 레코드가 극소수이다.

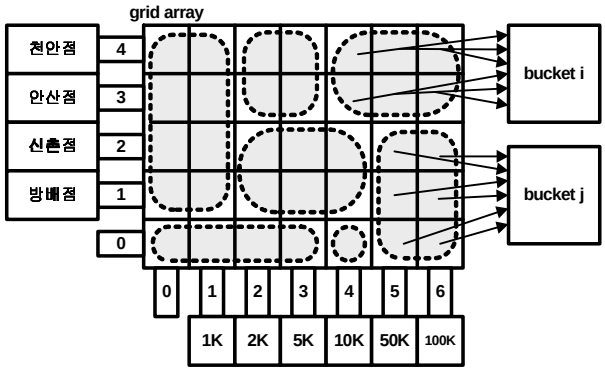
12.8.2 다중키에 대한 색인

- "지점명"과 "잔액"에 대한 색인을 각각 유지하지 않고, (지점명,잔액)에 대한 색인을 유지할 수 있다. 이런 색인의 정렬은 사전 방식을 사용한다.
- 예)

  select 계좌번호
  from *account*
  where 지점명 < "Perryridge" and
  잔액 $= 1000$

  이 질의를 (지점명,잔액) 색인을 이용하여 찾고자 하면 지점명이 "Perryridge"보다 작은 각 지점마다 잔액이 1000인 계좌를 찾아야 한다. 비교가 아니라 등호 있었으면 (지점명,잔액) 색인을 이용하여 쉽게 찾을 수 있지만 이 경우에는 많은 디스크 입출력이 필요하다.

12.8.3 그리드 파일



<그림 12.13> 그리드 파일의 예

- **그리드 파일(grid file)**은 그리드 배열이라고 하는 2차원 배열과 행과 열의 눈금 역할을 하는 두 개의 선형 눈금으로 구성된다.
- 예) (지점명,잔액) 색인에 대한 그리드 배열은 그림 12.13과 같다. 행은 지점명에 대한 눈금 역할을 하며 열은 잔액에 대한 눈금 역할을 한다. 검색키 값은 셀로 사상되며, 여러 셀이 하나의 버켓을 가리킬 수 있다. 그림 12.13에서 점선으로 표시된 부분은 같은 버켓을 가리키는 셀들을 나타낸다.
- ("A-217","강남점",750)의 삽입은 (0,0) 셀로 사상된다. 강남점은 행 1에 할당된 방배점보다 작으므로 행 0에 사상되고, 잔액이 750이므로 열 0에 할당된다.
- 이런 그리드 파일을 사용하면 다음과 같은 검색 조건에 대한 질의를 쉽게 처리할 수 있다.

  지점명 < "신촌점" and 잔액 $= 1000$

  지점명이 신촌점보다 작은 것은 행 0과 1에 해당되며, 잔액이 100인 것은 열 1에만 해당된다. 따라서 (0,1)과 (1,1) 셀만 검색하면 된다.
- 레코드들이 셀들로 균등하게 할당되도록 눈금 값을 적절하게 선택하여야 한다.
- 버켓이 꽉 차있고 이 버켓을 가리키는 셀이 여러 개 있으면 새 버켓을 만들어 셀들이 다른 버켓을 가리키도록 분할한다. 반대로 버켓이 꽉 차있고 이 버켓을 가리키는 셀이 하나면 열 또는 행을 늘려 그리드 파일을 재조정해야 한다.

- 그리드 파일은 단일 키를 사용하는 질의도 효과적으로 처리할 수 있다. 질의 조건이 다음과 같으면

  지점명 = “천안점”

  행 4만이 이 조건을 만족할 수 있다. 따라서 행 4의 셀들만을 이용하여 검색하면 된다.

- 그리드 파일의 문제점
  - 공간 오베헤드(그리드 파일이 매우 클 수 있다.)
  - 삽입과 삭제 때 성능 오버헤드가 있으며, 삽입이 빈번하면 그리드 파일을 자주 재조정해야 한다.
  - 눈금을 결정하는 것이 쉽지 않다.

12.9 비트맵 색인

12.9.1 비트맵 색인 구조

- 비트맵은 단일 키에 대한 색인이지만 다중 키에 대한 질의를 효과적으로 처리할 수 있는 색인 구조이다.
- 가장 단순한 형태의 비트맵 색인 구조는 속성 A가 가질 수 있는 모든 값에 대해 하나의 비트맵을 유지한다.
- 비트맵은 관계에 있는 레코드 수만큼의 비트를 가진다.
- 예) 다음과 같은 *customer-info* 관계가 있다고 하자.

	성명	성별	주소	소득수준
0	John	남	Perryridge	L1
1	Diana	여	Brooklyn	L2
2	Mary	여	Jonestown	L1
3	Peter	남	Brooklyn	L4
4	Kathy	여	Perryridge	L3

“성별” 속성에 대한 비트맵은 다음과 같으며,

남	10010
여	01101

“소득수준” 속성에 대한 비트맵은 다음과 같다.

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- 비트맵은 단일 키에 대한 질의에 대해서는 전혀 도움이 되지 않는다. 그러나 질의의 조건이 다음과 같은 다중 키에 대한 것이면

  성별 = “여” and 소득수준 = “L2”

  비트맵을 이용하여 효율적으로 찾을 수 있다. 이 질의는 “여”에 대한 비트맵과 “L2”에 대한 비트맵을 논리곱하여, 그 결과인 01000을 이용하여 만족하는 레코드를 찾을 수 있다. 그러나 두 비트맵의 논리곱의 결과에 많은 비트가 1이면 자체적으로 검색하는 것이 오히려 효과적일 수 있다.

- 비트맵은 특정 조건을 만족하는 레코드의 수를 계산할 때에도 효과적이다.
- 비트맵의 또 다른 장점은 작은 공간을 차지한다는 것이다.
- 보통 레코드의 삭제를 다루기 위해 존재 비트맵을 유지하며, null 값을 다루기 위해 널값 비트맵을 유지한다.
- 비트 수준의 연산을 이용하여 여러 질의를 효과적으로 처리할 수 있다.
  - 논리곱: 두 조건을 모두 만족하는 모든 레코드
  - 논리합: 두 조건 중 하나만 만족하는 모든 레코드
  - 논리부정: 어떤 조건을 만족하지 않는 모든 레코드