論文98-35C-3-4

분할 시그너춰 화일을 위한 효율적인 디렉토리 관리 기법

(An Efficient Method for Directory Management of the Partitioned Signature File)

金 尙 煜 *, 黃 煥 圭 *, 崔 晃 奎 **, 尹 龍 翼 ***

(Sang-Wook Kim, Whan-Kyu Whang, Hwang-Kyu Choi, and Yong-Ik Yoon)

요 약

분할 시그너춰 화일은 같은 키값을 갖는 시그너춰들이 저장된 블럭들의 집합과 이러한 키값을 저장하는 디렉토리로 구성되는 시그너춰 화일의 개선된 형태이다. 디렉토리는 각 블럭을 액세스하기 전에 이곳의 키값들을 먼저 조사함으로써 불필요한 블럭 액세스를 피하도록 해 주는 메타 정보를 관리한다. 디렉토리는 데이타베이스 크기에 비례하므로 대용량 데이타베이스 환경에서의 효율적인 디렉토리의 관리는 매우 중요하다. 본 논문에서는 먼저 기존의 분할 시그너춰 화일에서 나타나는 디렉토리 관리 기법의 문제점들을 지적하고, 이러한 문제점들을 해결하는 새로운 기법을 제안한다. 제안된 기법에 의한 디렉토리 구조는 대용량 데이타베이스 환경으로의 적응성, 동적인 환경으로의 적응성, 디렉토리를 위한 저장 공간의 오버헤드의 세가지 측면에서 좋은 성능을 제공한다. 뿐만 아니라, 제안된 기법은 기 개발된 다목적 저장 엔진내의 구성 요소로서 자연스럽게 통합될 수 있다. 이러한 특징은 제안된 기법이 하이퍼미디어 시스템, 전자 도서관 시스템, 멀티미디어 다큐먼트 시스템, 멀티미디어 의료 정보 시스템, 멀티미디어 메일링 시스템 등 다양한 멀티미디어 응용 분야의 내용 기반 검색을 위한 액세스 구조로서 적절히 사용될 수 있음을 보이는 것이다.

Abstract

A partitioned signature file is an enhancement of the signature file that divides all the signatures into blocks in such a way that each block contains the signatures with the same key. Its directory stores all the keys as meta information for avoiding unnecessary block accesses by examining them first before the actual searching of the blocks. Efficient directory management is very important in large database environments since its size gets larger proportionally to that of the database. In this paper, we first point out the problems in the directory management methods of the previous partitioned signature files, and then present a new one solving them. Our method offers good features in the following three aspects: (1) suitability for large database environments, (2) adaptability to dynamic situations, and (3) storage overhead for the directory. Moreover, we can seamlessly integrate it as a subcomponent into previously-developed general-purpose storage engines. These features show that our method is applicable to signature-based access structures for the content-based retrieval in various multimedia applications such as hypermedia systems, digital library systems, multimedia document systems, multimedia medical information systems, multimedia mailing systems, and so on.

* 正會員, 江原大學校 情報通信工學科

(Department of Information and Telecommunications Engineering Kangwon National University)

** 正會員, 江原大學校 컴퓨터工學科

(Department of Computer Engineering Kangwon National University)

*** 正會員, 淑明女子大學校 電算學科

(Computer Science Department Sookmyung Women's University)

※ 본 연구는 96학년도 한국과학재단 핵심전문과제 연구비(과제 번호 : 961-0903-019-1)의 부분적인 지원과 강원대학교 멀티미디어 특화 센터를 통한 정보통신부 정보통신 분야 대학원 특성화 사업의 부분적인 지원에 의한 결과임.

接受日字:1997年1月14日, 수정완료일:1998年3月3日

Ⅰ. 서 론

텍스트, 그래픽스, 이미지, 오디오, 비디오 등의 멀티미디어 데이타는 하이퍼미디어 시스템, 전자 도서관 시스템, 멀티미디어 다큐먼트 시스템, 멀티미디어 홈쇼핑 시스템, 멀티미디어 의료 정보 시스템, 멀티미디어 메일링 시스템 등 다양한 멀티미디어 응용에서 사용된다. 이러한 멀티미디어 데이타는 기존의 은행 업무, 사무 업무, 행정 업무 등에서 주로 다루어 왔던 정형 데이타(formatted data)와는 그 특성이 크게 다르므로 B^+ 트리, 해쉬 등 기존의 단순한 인덱싱 기법을 통한 검색은 적합하지 않다.

이러한 비정형 데이타(unformatted data)를 위한 내용 기반 검색(content-based retrieval) 기법으로서 시그너춰 화일(signature file)이 널리 사용되고 있다 [2] [3] [5] [6] [7] [8] [15] [16] [18]. 시그너춰 화일은 시그너춰(signature)라 불리우는 객체[1]의요약 정보들을 저장하는 화일 구조이다. 사용자의 검색 질의는 필터링 단계(filtering step)와 폴스 드롭 해결 단계(false-drop resolution step)를 차례로 거치게 된다. 필터링 단계에서는 시그너춰 화일을 이용하여 사용자 질의의 검색 조건을 만족할 수 있는 후보 객체들을 미리 파악하게 되며, 이 결과 폴스 드롭 해결 단계에서 직접 액세스해야 하는 객체의 수를 크게 줄여준다. 이러한 시그너춰 화일의 필터링 작업으로 인한 성능 개선의 효과는 매우 크다.

검색 조건을 만족할 가능성이 있는 후보 객체들을 파악하기 위해서는 필터링 단계에서 전체 시그너춰 화일을 액세스 해야 한다. 시그너춰 화일의 크기는 데이타베이스의 크기에 비례하므로 대용량의 데이타베이스 환경에서는 시그너춰 화일을 액세스 하는 비용 자체가 큰 오버헤드가 된다. 이를 해결하기 위한 기법으로서 같은 키(key)값을 갖는 시그너춰들을 블럭 단위로 나누어 저장하는 분할 시그너춰 화일(partitioned signature file)이 제안되었다[12] [13] [22]. 이 기법에서는 시그너춰들을 블럭 단위로 나누고, 각 블럭내의 시그너춰들이 질의를 만족하는가의 여부를 그 블럭과 대응되는 키값을 통하여 판단한다. 즉, 키값이 일치하지 않는 블럭내의 시그너춰들은 결코 질의 조건을 만족할 수 없으므로 이 블럭들에 대한 디스크 액세스의 발생을 피할 수 있으며, 이 결과 검색시의 오버헤드를 크게 줄일 수 있다.

본 논문에서는 분할 시그너춰 화일을 위한 효율적인 디렉토리 관리 기법에 관하여 논의하고자 한다. 디렉토리(directory)란 분할 시그너춰 화일에서 키값과 일치하는 블럭들을 찾기 위한 메타 정보를 관리하는 자료 구조를 의미한다. 참고 문헌 [12] [13] [22]에서 제안된 기법들은 분할 시그너춰 화일의 디렉토리 관리를 위한 설계 요건인 (1) 대용량 데이타베이스 환경에서의 고속의 검색, (2) 객체의 삽입과 삭제가 빈번하게 발생하는 동적 환경으로의 적응, (3) 저장 공간의 효율 등의 측면에서 각각의 문제점을 가지고 있다. 본 논문에서는 이러한 문제점들을 지적하고, 이러한 문제점들을 해결하는 새로운 디렉토리 관리 기법을 제안한다.

특히, 제안하고자 하는 기법은 이 외에도 다목적 저장 엔진(general-purpose storage engine)과의 통합을 염두에 두고 설계된 것이 특징이다. 즉, 다목적 저장 엔진의 디스크 할당 단위인 블럭을 화일 구조의 기본 요소로서 사용함으로써 WiSS[1], Exodus[15], KAOSS[14], MIDAS[11] 등의 기 개발된 다목적 저장 엔진에 시그너춰 화일을 추가할 때 성능 저하 현상이나 다목적 저장 엔진의 기본 아키텍춰 변화 없이 자연스러운 통합이 가능하다. 반면, 참고 문헌 [12] [13] [22]에서 제안된 기법들은 대부분 운영 체제에서 제공하는 화일 시스템을 기반으로 하는 특정한 환경만을 대상으로 하고 있다. 이 결과, 대용량 데이타베이스를 다루는 실제 상황에서 심사 숙고되어야 하는 디스크 관리, 버퍼링, 객체 관리, 클러스터링, 일반 속성에 대한 인덱싱 등과의 통합을 거의 고려하지 않고 있다. 그러나 이러한 기능들은 시그너춰 화일이 기 개발된 다목적 저장 엔진내에 내용 기반 검색을 위한 구성 요소로서 자연스럽게 통합되기 위하여 반드시 고려되어야 하는 사항들이다.

본 논문의 구성은 다음과 같다. 제 2장에서는 시그너춰 화일의 기본 개념을 간략히 소개한다. 제 3장에서는 관련 연구로서 기존의 분할 시그너춰 화일들을 소개하고, 각 기법의 디렉토리 관리 기법이 갖는 성능상의 문제점들을 지적한다. 제 4장에서는 제안하는 디렉토리 관리 기법의 기본 아이디어, 연산 알고리즘, 주요 자료 구조, 장단점에 대하여 자세히 기술한다. 끝으로, 제 5장에서는 결론을 내리고, 향후 연구 방향을

1) 멀티미디어 응용에서 객체는 하나의 멀티미디어 데이타에 해당된다.

제시한다.

Ⅱ. 시그너춰 화일

시그너춰 화일[2] [3] [5] [6] [7] [8] [15] [16] [18]은 비정형 객체 검색을 효율적으로 지원하는 액세스 기법의 하나로서 역 화일(inverted files)에 비해 부가적인 저장 공간의 오버헤드가 훨씬 작다는 장점이 있다[2]. 시그너춰 화일의 구성 방법은 크게 superimposed coding 기법, word signature 기법, compression based 기법 등으로 분류된다[6]. 본 장에서는 이들 중 가장 널리 사용되는 superimposed coding 기법을 중심으로 시그너춰 화일의 구성 방법과 시그너춰 화일을 이용한 질의 처리 방법에 대하여 간략히 소개한다.

객체 시그너춰(object signature)란 객체의 요약 정보이며, 해당 객체를 구성하는 모든 단어의 시그너춰(word signature)들을 비트별 논리합(OR-ing)함으로써 구해진다. 각 단어 시그너춰는 사전에 결정된 b개의 비트들 중에서 k개의 비트를 1로 할당하고, (b-k)개의 비트를 0으로 할당하는 해쉉 함수(hashing function)를 해당 단어에 적용시킴으로써 생성된다. 그림 1은 객체가 multimedia, hypermedia, database의 세 단어들로 구성되고, 해쉉 함수를 위한 b가 12, k가 4인 경우, 단어 시그너춰들의 생성과 비트별 논리합을 통한 객체 시그너춰의 생성 과정을 보여준다.

multimedia	001 100 001 010	
hypermedia	000 101 011 000	
database	001 000 001 110	
		bitwise OR-ing
object signature	001 101 011 110	

그림 1. 단어 시그너춰들을 이용한 객체 시그너춰의 생성

Fig. 1. Creation of an Object Signature Using Word Signatures.

시그너춰 화일을 이용한 검색 질의의 처리 방법을 살펴본다. 검색 조건을 갖는 질의가 주어지면, 먼저 질의 시그너춰(query signature)를 생성한다. 질의 시그너춰는 해당 질의의 요약 정보이며, 객체 시그너춰와 유사한 방법으로 생성된다. 즉, 검색 조건에서 나타난 단어들에 대하여 해쉉 함수를 적용하여 만들어진 단어 시그너춰들에 대하여 비트별 논리합을 적용함으로써 생성된다. 다음은 시그너춰 화일을 접근하여 질의 시그너춰내에서 1로 세트되어 있는 비트들과 대응되는 모든 비트가 1로 세트되어 있는 객체 시그너춰들을 찾아낸다. 이들은 질의 조건내의 단어들을 포함할 가능성이 있는 후보 객체들이며, 이러한 처리를 필터링 단계에서 수행하게 된다.

필터링 단계에서 후보로 선정된 객체들 중에는 실제 객체내에는 검색 조건에서 나타난 단어들이 존재하지 않지만, 해당 객체 시그너춰는 질의 시그너춰를 만족시키는 폴스 드롭(false-drop)이 존재하게 된다. 따라서 후보 객체들의 스캔을 통해 검색 조건내의 단어들이 실제로 나타나는가의 여부가 확인하는 절차가 요구되며, 이러한 처리를 폴스 드롭 해결 단계에서 수행하게 된다. 즉, 시그너춰 화일은 전체 객체 집합에 대한 필터 역할을 하며, 이 결과 질의 검색 조건을 만족할 가능성이 전혀 없는 객체들을 사전에 제외시킴으로써 전체 질의 처리 시간을 단축시키는 효과를 갖는다.

Ⅲ. 분할 시그너춰 화일

검색 조건을 만족할 가능성이 있는 후보 객체들을 파악하기 위해서는 필터링 단계에서 전체 시그너춰 화일을 액세스해야 한다. 시그너춰 화일의 크기는 데이타베이스의 크기에 비례하므로 대용량의 데이타베이스 환경에서는 시그너춰 화일을 액세스 하는 비용 자체가 큰 오버헤드가 된다. 분할 시그너춰 화일[12] [13] [22]에서는 이러한 성능상의 문제점을 개선하기 위하여 같은 키값을 갖는 시그너춰들을 블럭 단위로 나누어 저장하고, 각 블럭내의 시그너춰들을 대표하는 키값들을 디렉토리내에서 관리한다.

검색 질의의 처리시에는 각 블럭을 디스크로부터 액세스하기 전에 디렉토리내에서 이 블럭과 대응되는 키값을 먼저 조사함으로써 블럭내의 시그너춰들이 질의 조건을 만족할 가능성이 있는가의 여부를 미리 파악한다[22]. 따라서 블럭과 대응되는 키값 자체가 질의 조건을 만족하지 않는 블럭들은 액세스하지 않으며, 이 결과 검색시의 디스크 액세스 수를 크게 줄일 수 있다.

본 장에서는 분할 시그너춰 화일에 대한 관련 연구로서 기존에 제안된 두가지 기법인 Lee의 기법[12] [13]과 퀵 필터(quick filter) 기법[22]의 특징을 소개하고, 각각의 디렉토리 관리 기법이 가지는 성능상의 문제점을 지적한다.

1. Lee의 기법

Lee의 기법[12][20]은 고정 접두어 방식(fixed prefix method), 확장 접두어 방식(extended prefix method), 부동 키 방식(floating key method)의 세 가지 디렉토리 관리 방식을 제공한다. 그러나 이들은 블럭과 대응되는 키를 추출하는 전략에 차이가 있을 뿐, 기본 자료 구조와 알고리즘에는 큰 차이가 없다. 본 절에서는 설명의 편의를 위하여 이들 중 가장 간단한 고정 접두어 방식을 중심으로 논의를 전개한다.

Lee의 기법에서는 객체 시그너춰의 전위 n개의 비트들로 구성되는 접두어(prefix)를 키로 선정하고, 키 값이 서로 같은 시그너춰들을 동일한 블럭내에 저장시킨다. 또한, 디렉토리를 위한 자료 구조로서 트라이(trie)[10]를 사용한다. 그림 2는 n이 3이고, 블러킹 인수(blocking factor)[21]가 3인 경우, Lee 기법을 이용한 분할 시그너춰 화일의 구성 예를 나타낸 것이다. 블럭 2는 키값이 010인 시그너춰들이 저장된 블럭이며, 이는 트라이의 왼쪽에서 두번째 경로를 통하여 찾을 수 있다. 또한, 키값이 000인 시그너춰들과 001인 시그너춰들은 블럭 1에 함께 저장되어 있으며, 이들은 공통 접두어인 00으로 표현된 트라이의 첫번째 경로를 통하여 찾을 수 있다. 이것은 블럭의 저장 이용율을 높이기 위한 것이다. 이후의 삽입으로 오버플로우가 발생하면, 블럭 1은 키값이 각각 000과 001인 두 개의 블럭으로 분할된다. 한편, 키값이 011인 시그너춰들은 이미 5개로 오버플로우된 상태이지만, n이 3이므로 더 이상의 분할이 불가능하여 오버플로우 체인으로 연결된 블럭 3과 블럭 4내에 저장한다.

Lee의 기법의 근본적인 문제점은 디렉토리 관리를 위한 트라이 구조에서 기인한다. 트라이는 비균형 트리 형태를 가지며, 주기억 장치내의 상주를 가정하는 자료 구조이다. 그러나 대용량의 데이타베이스 환경에서는 대량의 객체들을 다루게 되며, 따라서 디렉토리 자체를 주기억 장치내에 저장할 수 없는 경우가 빈번하게 발생한다. 따라서 디스크내에 저장할 수 있는 블럭을 기본 단위 요소로 하는 새로운 트라이 구조로의 수정이 불가피하다.

이러한 수정 작업을 통하여 디스크를 기반으로 하는 새로운 트라이 구조를 고안하는 경우에도 트라이 고유의 비균형성의 문제점은 그대로 남게된다. 즉, 디렉토리가 비균형 트리 형태를 가지므로 이러한 비균형의 정도가 심화될수록 특정 블럭내의 시그너춰를 조사하기 위해서는 많은 노드들을 탐색해야 한다. 이러한 경우, 노드들은 블럭 단위로 디스크내에 존재하게 되므로 빈번한 디스크 액세스를 야기하게 되어 결국 검색 성능을 저하시키게 된다. 따라서 트라이와 같은 비균형 트리의 디렉토리 구조는 비균형의 정도가 심화될수록 사용자 질의에 대한 응답 시간의 차가 매우 커지게 되므로 대용량의 데이타베이스 환경에는 적합하지 않다.

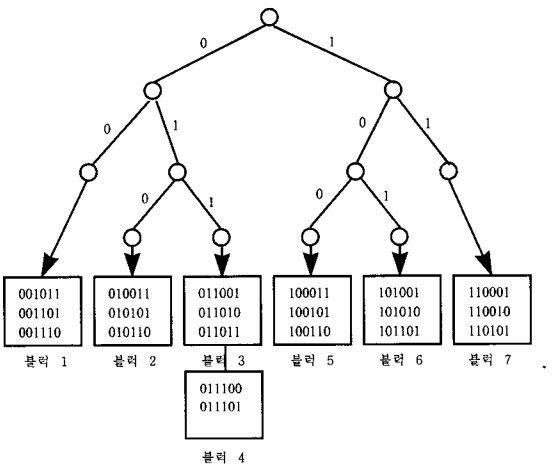


그림 2. Lee의 기법을 이용한 시그너춰 화일 구성
Fig. 2. Construction of a Signature File Using Lee's Method.

Lee의 기법의 또 하나의 문제점은 키 추출 방법에서 기인한다. 즉, 키의 길이를 미리 정해진 상태로 고정시키므로 같은 키값을 갖는 시그너춰들을 하나의 디스크 블럭에 저장할 수 없는 경우 그림 2의 블럭 3, 4와 같이 오버플로우 체인을 발생시키게 된다. 이러한 오버플로우 체인의 발생은 검색 성능을 저하시키는 중요한 원인이 된다.

2. 퀵 필터 기법

퀵 필터 기법[22]에서는 키로서 각 객체 시그너춰의 후위 비트들로 구성되는 접미어(suffix)들을 사용한다. 키를 선정하는데 있어 이와 같이 접미어를 쓰는 것과 Lee의 기법에서와 같이 접두어를 쓰는 것은 성능상에 영향을 미치지 않으므로 큰 의미가 없으나, Lee의 기법에서는 접두어의 길이를 고정시키는 반면, 퀵 필터 기법에서는 접미어의 길이에 제한을 두지 않는 것이 차이점이다. 디렉토리를 위한 자료 구조로는 선형 해싱(linear hashing)[17]을 사용한다.

선형 해싱을 위한 해싱 함수 g는 각 시그너춰를 임

의의 정수 h에 대하여 ($2^{h-1} < n <= 2^h$)의 조건을 만족하는 총 n개의 연속적인 주소 공간에 매핑시켜주는 역할을 하며, 그 매핑 함수의 정의는 아래와 같다[17]. 여기서 S는 시그너춰, n은 할당된 블럭의 수, h는 해싱 단계, b_r은 시그너춰의 r번째 비트값, f는 시그너춰의 전체 비트수이다. h가 0이고, n이 1인 경우, g(S, 0, 1)은 0으로 초기화된다. 그림 3은 퀵 필터 기법을 이용하여 그림 2와 동일한 시그너춰들을 대상으로 블러킹 인수가 3인 분할 시그너춰 화일을 구성한 예를 나타낸 것이다. 본 예는 n이 9이며, h가 4인 경우를 나타낸다.

$$g(S,h,n) = \begin{cases} \sum_{r=0}^{h-1} b_{f-r}\,2^r, & \text{if } \sum_{r=0}^{h-1} b_{f-r}\,2^r < n \\ \sum_{r=0}^{h-2} b_{f-r}\,2^r, & otherwise \end{cases}$$

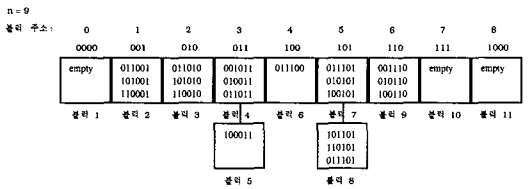


그림 3. 퀵 필터 기법을 이용한 시그너춰 화일 구성
Fig. 3. Construction of a Signature File Using the Quick Filter Method.

퀵 필터 기법의 문제점은 디렉토리의 기본 자료 구조로 사용되는 선형 해싱에서 기인한다. 선형 해싱은 계산을 통한 주소 지정 방식을 사용하므로 전용 디스크(dedicated disk)의 사용이 불가피하다[22]. 즉, 일정량의 크기를 갖는 디스크 공간을 미리 확보하고, 이것을 선형 해싱을 위한 디스크 블럭 풀(block pool)[2]로서 사용하는 것이다. 하나의 새로운 블럭이 요구되는 경우 이 전용 디스크내에서 미리 정해진 순서대로 블럭을 하나씩 할당시켜 준다.

선형 해싱으로 인한 첫 번째 문제점은 이러한 전용 디스크의 사용으로 인하여 객체의 삽입과 삭제가 빈번히 발생하는 동적인 환경에 적합하지 않다는 것이다. 전용 디스크의 크기는 초기 시스템 셋업시 결정되어야 하는데, 이러한 요건은 동적인 환경의 특성과는 부합되지 않는 것이다. 만일, 확보된 전용 디스크의 극히 일부만을 사용하게 되면, 나머지 공간은 다른 용도로 사용될 수 없으므로 디스크 공간의 낭비가 발생한다. 또한, 데이타가 확보된 전용 디스크의 용량을 초과하게 되는 경우에는 더욱 크기가 큰 전용 디스크를 새로이 할당하고 기존의 내용을 이곳에 반영해야 한다. 이러한 동작은 전용 디스크 크기와 비례하는 많은 수의 디스크 액세스를 발생시키므로 오버헤드가 크며, 특히 이 경우 전체 시스템의 사용이 중지된다는 문제점을 갖는다.

선형 해싱으로 인한 두 번째 문제점은 저장 이용률이 저하된다는 것이다. 객체 시그너춰를 생성하는 방식의 특성으로 인하여 특정 블럭들에 대해서는 그 블럭과 대응되는 키의 특성상 해당 시그너춰들이 거의 없는 경우가 발생한다[22]. 또한, 선형 해싱의 특성상 블럭의 오버플로우가 발생할 때 미리 정해진 순서에 따라 이 블럭이 아닌 다른 불필요한 블럭의 분할이 발생하게 된다[17]. 따라서 선형 해싱을 디렉토리로 사용하는 경우에는 저장 이용률이 크게 저하된다. 또한, 저장 공간 이용률의 저하로 인하여 액세스해야 하는 전체 블럭의 수가 상대적으로 많아지므로 이로 인한 검색 효율도 떨어지게 된다.

선형 해싱으로 인한 세 번째 문제점은 오버플로우 체인이 발생할 수 있다는 것이다. 미리 지정된 순서에 의한 선형 해싱의 분할 방식은 오버플로우가 발생하여 분할을 요구하는 블럭을 적시에 분할하지 않고, 그림 3의 블럭 4, 5 및 블럭 7, 8과 같은 오버플로우 체인을 생성한다. 삽입된 시그너춰들이 도메인상에 균일하게 분포하게 되는 경우에는 큰 문제가 발생하지 않으나, 그렇지 않은 대부분의 상황에서는 오버플로우 체인이 빈번하게 발생한다. 이러한 오버플로우 체인의 발생은 질의 처리 성능을 떨어뜨리는 주요 원인으로 작용한다.

Ⅳ. 제안하는 기법

본 장에서는 전술한 문제점들을 극복할 수 있는 새로운 분할 시그너춰 화일의 디렉토리 관리 기법을 제안한다. 먼저, 제 4.1절에서는 제안된 기법의 기본 아이디어에 대하여 설명하고, 제 4.2절에서는 이 기법을 위한 연산 알고리즘을 제시한다. 제 4.3절에서는 제안

2) 일반적으로 블럭 풀이란 저장 시스템의 디스크 관리자가 관리하는 블럭들의 집합을 의미한다. 화일의 크기가 증가하는 경우에는 디스크 관리자는 새로운 블럭을 블럭 풀로부터 해당 화일에 할당하고, 화일의 크기가 축소되는 경우에는 디스크 관리자는 불필요한 블럭을 해당 화일로부터 받아 블럭 풀로 반환한다. 블럭 풀내에서 블럭들을 관리하는 방식은 저장 시스템마다 차이가 있다.

된 기법의 구현을 위한 주요 자료 구조에 관하여 기술하고, 끝으로 제 4.4절에서는 제안된 기법의 장점에 관하여 논의한다.

1. 기본 아이디어

제안하는 기법은 참고 문헌 [12] [13] [22]에서 제안하는 기법에서와 같이 유사한 시그너춰들을 같은 블럭내에 저장하는 분할 시그너춰 화일을 대상으로 한다. 이때, 유사성의 정도를 판별하는 방법으로서 공통 접두어(common prefix)의 길이를 사용한다. 즉, 두 시그너춰의 공통 접두어의 비트 수가 많을수록 두 시그너춰간의 유사성이 크다고 정의한다. 블럭에 대한 요약 정보가 저장된 디렉토리는 (hash_value, block_ptr)의 쌍으로 구성되는 엔트리들의 집합이다. block_ptr은 이 엔트리와 대응되는 시그너춰들이 저장된 블럭의 주소를 나타낸다. 또한, hash_value는 해쉬값으로 표현되는 키값이며, block_ptr이 가리키는 블럭내 시그너춰들의 공통 접두어를 나타낸다. 아래는 그림 2와 동일한 시그너춰들을 대상으로 하는 디렉토리 D1의 예를 나타낸 것이다. 예를 들어, 블럭 1내에는 00으로 시작되는 접두어를 가지는 시그너춰들이 저장되어 있고, 블럭 4에는 0111로 시작되는 접두어를 가지는 시그너춰들이 저장되어 있다. 블럭내의 시그너춰들의 유사성이 클수록 이와 대응되는 엔트리의 해쉬값의 길이가 커짐을 볼 수 있다.

D1 = {(00, 블럭 1), (010, 블럭 2), (0110, 블럭 3),
(0111, 블럭 4), (100, 블럭 5), (101, 블럭 6),
(11, 블럭 7)}

검색 질의의 처리시 각 엔트리가 가리키는 블럭이 질의 시그너춰를 만족하는 객체 시그너춰를 포함하는가의 여부를 조사하기 위하여 엔트리내에서 해쉬값으로 표현되는 키값을 기반으로 하는 접두어 만족성 연산(prefix-qualifying operation)을 사용한다. 접두어 만족성 연산은 엔트리의 해쉬값을 질의 시그너춰 중 이 해쉬값과 같은 크기의 접두어와 비교함으로써 이 접두어내의 1로 세트되어 있는 부분과 대응되는 위치의 모든 비트들이 엔트리의 해쉬값에도 1로 세트되어 있는가를 점검하는 연산이다. 예를 들면, 질의 시그너춰 101001에 대한 접두어 만족성 연산의 결과 D1내의 여섯 번째와 일곱 번째 엔트리만이 이 질의 시그너춰를 만족하는 후보들이 저장된 블럭들을 가리킴을 알 수 있다. 따라서 질의 처리시 이외의 엔트리들이 가리키는 다섯 개의 블럭들은 액세스하지 않아도 된다.

만일, 전체 엔트리들이 화일내에서 단순히 연속적으로 저장된다면, 질의 처리시 모든 엔트리들에 대하여 질의 시그너춰와의 접두어 만족성 연산을 수행해야 한다. 제안된 기법에서는 이러한 오버헤드를 피하기 위하여 디렉토리를 계층 구조로서 관리한다. 즉, 최하위 디렉토리 D1내에서 공통 접두어가 같은 엔트리들을 동일한 블럭내에 저장하고, 이 엔트리들의 공통 접두어와 이 블럭의 주소로 엔트리를 구성하여, 상위 단계 디렉토리 D2내에 유지시킨다. 만일, D2의 엔트리들을 하나의 블럭내에 저장할 수 없는 경우, 같은 방식으로 그 상위에 디렉토리 D3를 두게 되며, 이것은 최상위 디렉토리가 하나의 블럭내에 유지될 때까지 반복된다. 따라서 전체 구조는 균형 트리의 형태가 된다[3]. 이러한 구조적 특성은 제 4.2절에서 기술하는 삽입 및 삭제 알고리즘에 의하여 자동적으로 유지된다. 질의 처리시에는 질의 시그너춰에 대하여 루트 디렉토리로부터 최하위 디렉토리 D1까지 내려오면서 접두어 만족성 연산의 결과가 참이 되는 엔트리들에 대해서만 그 엔트리의 block_ptr이 가리키는 하위 단계의 블럭을 액세스하게 된다.

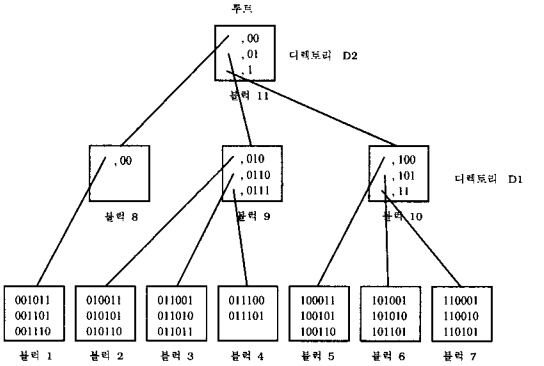


그림 4. 제안된 기법을 이용한 시그너춰 화일의 구성

Fig. 4. Construction of a Signature File Using the Proposed Method.

그림 4는 그림 2에서 나타나는 객체 시그너춰들에 대하여 제안된 기법으로 시그너춰 화일을 구성한 예를

3) 이러한 다단계 균형 트리의 구조는 다차원 동적 화일 구조(multidimensional dynamic file organization)로서 제안된 바 있는 계층 그리드 화일(multilevel grid file)[19] [20]의 일차원 구조와 유사하다.

나타낸 것이다. 모든 정보들이 블럭 단위로 유지되며, 전체 구조는 균형 트리 형태를 가짐을 볼 수 있다. 또한, 그림 2와 3에서와는 달리 오버플로우 체인은 발생되지 않음을 볼 수 있다. 본 논문에서는 이러한 트리 구조내에서 객체 시그너춰를 포함하는 최하위 단계의 블럭들을 시그너춰 블럭(signature block)이라 정의하고, 그 상위의 디렉토리를 위한 블럭들을 디렉토리 블럭(directory block)이라 정의한다.

2. 연산 알고리즘

본 절에서는 제안된 시그너춰 화일을 위한 검색, 삭제, 삽입 등 세가지 연산 알고리즘을 재귀 함수(recursive function) 형태로 기술한다[4]. 먼저, 알고리즘 기술을 위하여 사용되는 변수를 설명한다.

- $Signature_{query}$: 질의 시그너춰를 나타낸다.
- $Signature_{object}$: 객체 시그너춰를 나타낸다.
- CurrentBlock: 현 재귀 함수에서 처리 중인 블럭을 나타낸다.
- NextBlock: 현 재귀 함수에서 호출하고자 하는 하위 단계 재귀 함수내에서 CurrentBlock으로 사용할 블럭을 나타낸다.
- CurrentEntry: NextBlock과 대응되는 상위 단계 디렉토리 엔트리를 나타낸다.
- HashVal: 디렉토리 엔트리의 요소로서 하위 단계 블럭내의 시그너춰들의 공통 접두어인 해쉬값을 가진다.
- NextPtr: 디렉토리 엔트리의 요소로서 시그너춰 블럭의 엔트리인 경우에는 이와 대응되는 객체의 주소를 가지고, 그렇지 않은 경우에는 하위 단계 디렉토리에서 이 엔트리와 대응되는 블럭의 주소를 가진다.
- UnderflowFlag: 현 재귀 함수에서 처리하는 CurrentBlock에서 언더플로우가 발생하였음을 이를 호출한 상위 단계 재귀 함수로 알려주기 위한 플래그로서 사용된다.
- OverflowFlag: 현 재귀 함수에서 처리하는 CurrentBlock에서 오버플로우가 발생하였음을 이를 호출한 상위 단계 재귀 함수로 알려주기 위한 플래그로서 사용된다.
- ToBeInsertedEntry: 현 재귀 함수에서 CurrentBlock에 삽입하고자 하였으나, 오버플로우가 발생하여 삽입에 실패한 디렉토리 엔트리를 나타낸다. 이것은 현 재귀 함수를 호출한 상위 단계 재귀 함수에 전달된다.
- UnderflowFlag_Local: 현 재귀 함수에서 호출한 하위 단계 재귀 함수에서 처리하는 NextBlock에서 언더플로우가 발생하였음을 현 재귀 함수로 알려주기 위한 플래그로서 사용된다.
- OverflowFlag_Local: 현 재귀 함수에서 호출한 하위 단계 재귀 함수에서 처리하는 NextBlock에서 오버플로우가 발생하였음을 현 재귀 함수로 알려주기 위한 플래그로서 사용된다.
- ToBeInsertedEntry_Local: 현 재귀 함수에서 호출한 하위 단계 재귀 함수에서 NextBlock에 삽입하고자 하였으나, 오버플로우가 발생하여 삽입에 실패한 디렉토리 엔트리를 나타낸다. 이것은 하위 단계 재귀 함수로부터 현 재귀 함수에 전달된다.

1) 검색 알고리즘

검색 알고리즘은 디렉토리의 루트 블럭으로부터 시작하여 각 디렉토리 단계를 따라 내려가면서 자신을 재귀적으로 호출함으로써 주어진 질의에 대한 검색 연산을 수행한다. 각각의 재귀 호출에서는 제 4.1절에서 기술한 접두어 만족성 연산을 통하여 현재 처리중인 블럭의 엔트리들이 가리키는 하위 단계 블럭들 중 질의 조건을 만족할 가능성이 있는 것들을 파악한다. 구체적인 세부 알고리즘은 다음과 같다.

A. 검색 알고리즘

Query($Signature_{query}$, CurrentBlock)
IF CurrentBlock이 시그너춰 블럭인 경우, **THEN**
  FOR 이 블럭내에 존재하는 (HashVal, NextPtr)로 구성되는 각 엔트리에 대하여
    IF HashVal이 $Signature_{query}$에 대하여 접두어 만족성 연산을 만족하는 경우, **THEN**
      NextPtr이 가리키는 객체에 대하여 폴스 드롭 해결 단계를 수행한다.
    END_IF
  END_FOR
ELSE

4) 제안된 알고리즘은 계층 그리드 화일의 알고리즘과 유사하다. 단, 다차원 도메인 공간을 다루는 계층 그리드 화일의 알고리즘은 삽입 및 삭제의 복잡도가 더 크며, 일차원 도메인 공간을 다루는 제안된 알고리즘은 보다 간결한 형태를 갖는다. 한편, 검색에 있어서 접두어 만족성 연산을 사용하는 제안된 알고리즘은 접두어 일치성 연산을 사용하는 계층 그리드 화일의 알고리즘과 검색 대상이 되는 엔트리들을 찾는 방법에 있어 차이가 있다.

FOR 이 블럭내에 존재하는 (HashVal, NextPtr)로 구성되는 각 엔트리에 대하여
IF HashVal이 $Signature_{query}$에 대하여 접두어 만족성 연산을 만족하는 경우, **THEN**
NextPtr이 가리키는 블럭을 NextBlock에 읽어 들인다.
Query($Signature_{query}$, NextBlock)
END_IF
END_FOR
END_IF_ELSE

2) 삭제 알고리즘

삭제 알고리즘은 디렉토리의 루트 블럭으로부터 시작하여 각 디렉토리 단계를 따라 내려가면서 삭제될 시그너춰가 존재하는 블럭을 찾고, 해당 블럭에서 시그너춰를 삭제하는 절차를 걸쳐 수행된다. 이러한 삭제로 인하여 해당 블럭이 임계치(threshold) 이하의 공간만을 사용하게 되는 언더플로우가 발생하면 병합을 통하여 해결한다. 임계치는 시스템 초기화시 미리 지정되며, 응용분야의 특성에 따라 조정 가능하다. 임계치를 높게 주는 경우, 저장 공간의 오버헤드를 줄일 수 있으나 동적인 상황에서는 빈번한 분할과 병합으로 인하여 삽입 및 삭제 성능이 떨어진다. 반면, 임계치를 낮게 주는 경우에는 분할과 병합의 발생 빈도가 적으므로 삽입 및 삭제시 좋은 성능을 가져올 수 있으나 상대적으로 저장 공간의 오버헤드가 커진다.

구체적인 세부 알고리즘은 아래와 같다. 여기서 시그너춰가 저장되어 있는 블럭을 찾기 위하여 접두어 일치성(prefix-matching) 연산을 사용한다. 접두어 일치성 연산은 한 엔트리의 해쉬값이 한 시그너춰의 접두어인가를 점검하는 연산으로서 삭제될 시그너춰가 존재하는 블럭을 식별하기 위하여 사용된다. 예를 들어, 삭제될 시그너춰가 101101일 때, 이 시그너춰는 그림 4의 루트 블럭에서는 세 번째 엔트리와 접두어 일치되며, 이 엔트리가 가리키는 블럭 10에서는 두 번째 엔트리와 접두어 일치된다. 따라서 이 시그너춰는 블럭 6에 저장되어 있음을 파악할 수 있다. 한 블럭내에서 한 객체 시스너춰와 접두어 일치성 연산을 만족하는 엔트리는 항상 유일하다.

B. 삭제 알고리즘

Delete($Signature_{object}$, CurrentBlock, UnderflowFlag)
IF CurrentBlock이 디렉토리 블럭이면, **THEN**
이 블럭내에 존재하는 (HashVal, NextPtr)로 구성되는 각 엔트리에 대하여 HashVal이 $Signature_{object}$에 대하여 접두어 일치성 연산을 만족하는 엔트리를 찾아 CurrentEntry로 지정한다.
NextPtr이 가리키는 블럭을 NextBlock에 읽어들인다.
Delete($Signature_{object}$, NextBlock, UnderflowFlag_Local);
IF UnderflowFlag_Local이 TRUE이면, **THEN**
IF CurrentEntry와 병합이 가능한 엔트리 ToBeMergedEntry가 존재하는 경우[5], **THEN**
NextBlock과 ToBeMergedEntry가 가리키는 블럭내의 모든 엔트리들을 NextBlock내에 병합시킨다.
CurrentEntry의 HashVal을 CurrentEntry와 ToBeMergedEntry의 두 HashVal의 공통 접두어로 대체시킨다.
ToBeMergedEntry를 CurrentBlock내에서 삭제하고, 이것이 가리키는 블럭도 블럭 풀에 반환시킨다.
IF CurrentBlock내에 언더플로우가 발생하면, **THEN**
UnderflowFlag의 값을 TRUE로 지정한다.
END_IF
END_IF
END_IF
ELSE
이 블럭내에 존재하는 (HashVal, NextPtr)로 구성되는 각 엔트리에 대하여 HashVal이 $Signature_{object}$에 대하여 접두어 일치성 연산을 만족하는 엔트리를 찾아낸다.
이 엔트리를 CurrentBlock으로부터 삭제시킨다.
IF CurrentBlock내에 언더플로우가 발생하면, **THEN**
UnderflowFlag의 값을 TRUE로 지정한다.
END_IF
END_IF_ELSE

3) 삽입 알고리즘

삽입 알고리즘은 디렉토리의 루트 블럭으로부터 시

5) 한 엔트리 E와 병합 가능한 엔트리 M이란 E가 저장된 블럭내에서 E와의 공통 접두어의 길이가 가장 길고, E와 M이 가리키는 블럭들의 저장 이용률의 합이 100% 미만인 엔트리로 정의된다. 단, 같은 블럭내에 E와의 공통 접두어의 길이가 가장 긴 엔트리가 두 개 이상인 경우에는 병합 가능한 엔트리가 존재하지 않음을 의미한다. 예를 들어, 블럭내에 존재하는 엔트리들의 해쉬값들이 100, 101, 1101, 111이고, 이들 중 하위 단계에서 언더플로우된 블럭과 대응되는 엔트리 E의 해쉬값이 111인 경우, 이 111과 공통 접두어의 길이가 가장 긴 유일한 엔트리의 해쉬값은 1101이므로 이것이 병합 가능한 엔트리가 될 수 있다. 그러나 같은 상황에서 블럭내에 존재하는 엔트리들의 해쉬값들이 100, 101, 1100, 1101, 111인 경우에는 111과 공통 접두어의 길이가 가장 긴 엔트리의 해쉬값은 1101과 1100 두 개 이므로 해쉬값이 111인 엔트리와 병합 가능한 엔트리는 존재하지 않는다. 그 이유는 1101 및 1100과 각각 대응되는 엔트리들은 111과 대응되는 엔트리와 병합되기 이전에 자신들간의 병합이 선행되어야 하기 때문이다.

작하여 각 디렉토리 단계를 따라 내려가면서 객체 시그너춰가 삽입될 블럭을 찾고, 이 블럭에 객체 시그너춰를 삽입하는 절차를 거쳐 수행된다. 이러한 삽입의 결과 해당 블럭에 오버플로우가 발생하면 분할을 통하여 해결한다. 구체적인 세부 알고리즘은 아래와 같다.

C. 삽입 알고리즘

Insert(Signature$_{object}$, CurrentBlock, ToBeInserted Entry, OverflowFlag, UnderflowFlag)
IF CurrentBlock이 디렉토리 블럭이면, **THEN**
이 블럭내에 존재하는 (HashVal, NextPtr)로 구성되는 각 엔트리에 대하여 HashVal이 Signature$_{object}$에 대하여 접두어 일치성 연산을 만족하는 엔트리를 찾아내어 이를 CurrentEntry로 지정한다.
IF 접두어 일치성 연산을 만족하는 엔트리가 없는 경우[6], **THEN**
블럭 풀로부터 새로운 블럭 NewBlock을 할당한다.
Signature$_{object}$를 HashVal로 가지고 NewBlock의 주소를 NextPtr로 가지는 엔트리를 구성하고, 이를 CurrentEntry로 지정한다.
CurrentEntry를 CurrentBlock에 삽입시킨다.
IF CurrentBlock내에 오버플로우가 발생하면, **THEN**
CurrentEntry를 ToBeInsertedEntry로 지정한다.
OverflowFlag의 값을 TRUE로 지정한다.
END_IF
END_IF
NextPtr이 가리키는 블럭을 NextBlock에 읽어들인다.
Insert(Signature$_{object}$, NextBlock, ToBeInserted-Entry_Local, OverflowFlag_Local, Under-flowFlag_Local)
IF OverflowFlag_Local의 값이 TRUE이면, **THEN**
CurrentEntry의 HashVal의 뒤에 새로운 비트들을 추가시킴으로써 새로운 두 개의 HashVal들을 생성한다[7].
블럭 풀로부터 새로운 블럭 NewBlock을 할당한다.
앞의 단계에서 새롭게 생성한 두 HashVal을 기반으로 CurrentBlock내의 모든 엔트리들과 ToBe-InsertedEntry_Local을 NewBlock과 Current-Block내에 분산시킨다. 이 결과, 각 엔트리는 두 HashVal들 중 자신과 접두어 일치되는 것과 대응되는 블럭내에 저장된다.
CurrentBlock내에서 CurrentEntry를 삭제하고, 앞의 단계에서 새롭게 생성한 두 HashVal들과 NewBlock 및 NextBlock의 두 주소값으로 각각 구성되는 새로운 두 엔트리를 차례로 삽입한다.
IF CurrentBlock내에 오버플로우가 발생한 경우, **THEN**
OverflowFlag의 값을 TRUE로 지정한다.
삽입되지 못한 엔트리를 ToBeInsertedEntry로 지정한다.
END_IF
END_IF
IF UnderflowFlag_Local의 값이 TRUE이면[8], **THEN**
IF CurrentEntry와 병합이 가능한 엔트리 To-BeMergedEntry가 존재하는 경우, **THEN**
NextBlock과 ToBeMergedEntry가 가리키는 블럭내의 모든 엔트리들을 NextBlock내에 병합시킨다.
ToBeMergedEntry를 CurrentBlock내에서 삭제하고, 이것이 가리키는 블럭도 블럭 풀에 반환시킨다.
CurrentEntry의 HashVal을 CurrentEntry와 ToBeMergedEntry의 두 HashVal들의 공통 접두어로 대체시킨다.
IF CurrentBlock내에 언더플로우가 발생하면, **THEN**
Underflow의 값을 TRUE로 지정한다.
END_IF
END_IF
END_IF
ELSE
Signature$_{object}$를 HashVal로 가지고 해당 객체의 주소를 NextPtr로 가지는 엔트리를 구성하고, 이를 CurrentBlock에 삽입시킨다.
IF CurrentBlock내에 오버플로우가 발생한 경우[8], **THEN**

---

6) 이러한 경우가 발생하는 이유는 단계 5.1에서 블럭 오버플로우에 의한 분할 요구시 객체 시그너춰를 가지지 않는 도메인내의 구간에 대해서는 대응되는 엔트리를 생성하지 않도록 하기 때문이다. 이는 제 4.4.3절에서 설명하는 바와 같이 제안된 기법에서 디렉토리를 위한 저장 공간의 오버헤드를 최소화하기 위한 전략이 반영된 것이다.

7) 이때 추가되는 비트들의 수는 블럭내의 엔트리들을 성공적으로 양분할 수 있는 최소값을 선택한다. 예를 들어, 엔트리의 해쉬값이 00이고, 이 엔트리가 가리키는 하위 단계 블럭내에 엔트리들이 00101100, 00101101, 00100111, 00100011의 해쉬값을 가질 때, 새로 구성되는 해쉬값은 00101, 00100이 된다. 이 결과, 시그너춰 도메인상에서 000과 0011 등 객체 시그너춰가 존재하지 않는 두 구간은 디렉토리에서 엔트리로 나타나지 않는다. 따라서 00011111과 같은 객체 시그너춰가 삽입되면, 이와 접두어 일치성 연산을 만족하는 엔트리가 시그너춰 블럭 단계내에 존재하지 않으므로 삽입 알고리즘의 두 번째 IF 문 이하와 같은 과정을 거치게 되는 것이다.

8) 이와 같은 삽입 알고리즘에서의 언더플로우는 두 번째 IF 문 이하와 같이 디렉토리 상에 새로운 시그너춰가 속하는 구간을 표현하는 엔트리가 없어 각각 하나의 엔트리만을 가지는 블럭들로 구성되는 경로가 생성되는 경우에 발생한다. 이 경우, 하나의 엔트리만을 가지는 최상위 블럭은 가능하면 이웃한 블럭과 병합하도록 시도함으로써 저장 공간 이용률을 높일 수 있다.

OverflowFlag의 값을 TRUE로 지정한다.
삽입하고자 했던 엔트리를 ToBeInsertedEntry로 지정한다.
END_IF
IF CurrentBlock내에 언더플로우가 발생한 경우, THEN
UnderflowFlag의 값을 TRUE로 지정한다.
END_IF
END_IF_ELSE

3. 구현을 위한 자료 구조

본 절에서는 제안된 디렉토리 관리 기법의 구현을 위하여 필요한 기본 자료 구조에 관하여 논의한다. 먼저, 디렉토리 엔트리의 구조에 대하여 설명하고, 이들을 유지하기 위한 디렉토리 블럭의 구조에 대하여 설명한다.

디렉토리 엔트리는 그림 5에 나타난 바와 같이 세 개의 필드로 구성된다. NumBits 필드는 해당 엔트리의 해쉬값의 비트 수를 가지며, HashVal 필드는 그 해쉬값을 저장한다. NumBits 필드가 사용되는 이유는 제 4.1절에서 설명한 바와 같이, 각 엔트리마다 해쉬값의 길이가 서로 다르기 때문이다. 그림에서 빗금친 부분은 HashVal 필드내에서 현재 사용되지 않는 비트들을 의미한다. 이것은 디스크 및 메모리가 비트 단위가 아닌 바이트 단위로 할당되기 때문에 나타나는 현상이며, 항상 8비트 미만이 된다. BID 필드는 이 엔트리와 대응되는 블럭의 주소(block identifier)를 나타낸다. HashVal 필드가 차지하는 공간의 길이는 (NumBits+1)/8 - 1 바이트로서 엔트리마다 다르므로 전체 엔트리는 가변길이를 갖는다.

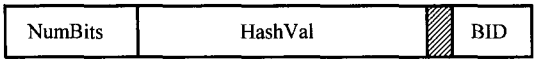


그림 5. 디렉토리 엔트리의 구조
Fig. 5. The Structure of a Directory Entry.

디렉토리 블럭은 엔트리들을 저장하기 위한 고정길이의 저장 단위이다. 엔트리가 가변길이를 가지므로 엔트리들의 저장과 검색을 위한 블럭내의 매핑 구조가 필요하다. 이를 위한 가장 단순한 방법은 블럭내에 저장된 엔트리 수 만큼의 요소를 갖는 배열을 블럭내에 할당하고, 각 요소가 해당 엔트리의 시작 주소를 가리키게 하는 것이다. 그러나 고정길이의 자료 구조인 배열에 대하여 블럭내에서 저장 가능한 최대 엔트리 수 만큼의 요소를 갖도록 해야 하므로 이들중 일부밖에 사용하지 않는 대부분의 경우 저장 공간의 낭비를 초래한다.

그림 6은 이러한 문제를 해결하기 위한 전형적인 해결책으로 널리 사용되는 방법이다[9]. 엔트리들은 블럭의 앞에서부터 뒤로 삽입되며, 매핑 구조의 요소인 슬럿(slot)은 뒤에서부터 앞으로 성장하게 된다. 여기서 Entry_i에는 전술한 구조를 갖는 엔트리가 저장되며, Slot_j는 대응되는 엔트리의 시작 주소를 갖는다. NumSlots 필드는 현재 블럭내에서 사용중인 슬럿의 수를 나타내며, NumUsedBytes 필드는 현재 블럭내에서 사용중인 총 바이트 수를 나타낸다. 빗금친 영역은 추후의 엔트리 삽입을 위하여 사용 가능한 공간을 의미하며, FreeSpace가 이 부분의 시작 위치를 가리킨다.

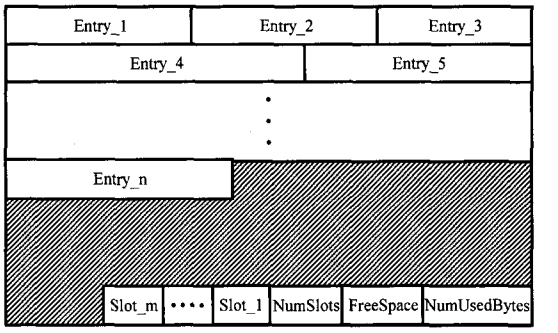


그림 6. 디렉토리 블럭의 내부 구조
Fig 6. The Internal Structure of a Directory Block.

4. 제안된 기법의 장점

본 절에서는 본 논문에서 제안한 분할 시그너춰 화일을 위한 디렉토리 관리 기법의 장점에 대하여 논의한다.

1) 대용량 데이타베이스로의 적용 가능성

시그너춰 화일 기법을 실제 응용에 적용할 수 있도록 하는 가장 중요한 요건은 수십만에서 수백만에 이르는 대용량의 데이타베이스로의 적용 가능성이다. 즉, 데이타베이스가 대형화 하는 경우에도 고속의 검색 성능을 보장할 수 있어야 한다.

제안하는 기법은 분할 기법을 기반으로 하므로 검색시 불필요한 블럭의 액세스를 방지할 수 있다. 또한, 트라이와는 달리 균형 트리 형태를 취하므로 비균형 구조에서 발생하는 성능 저하의 문제를 피할 수 있으며, 시그너춰의 전체 길이를 키로 사용할 수 있어 오버플로우 체인의 발생으로 인한 성능 저하가 나타나지 않는다. 따라서 이러한 특성으로 인하여 제안하는 기

법은 대용량 데이타베이스 환경에서도 좋은 성능을 나타낸다.

2) 동적인 환경으로의 적용 가능성

기존의 시그너춰 화일 기법에 관한 대부분의 연구들은 도서관에서 사용되는 텍스트 데이타베이스와 같이 동적인 변화가 자주 발생하지 않는 정적인 환경을 가정하고 있다. 그러나 많은 응용 분야들이 이러한 가정을 수용할 수 없으며, 특히, 멀티미디어 데이타베이스와 같은 새로운 응용 분야에서는 객체에 대한 삽입 및 삭제 연산이 빈번하게 발생된다. 따라서 이러한 응용에 적용하기 위해서는 동적인 환경에 적응할 수 있어야 한다.

제안하는 기법은 삽입, 삭제 알고리즘을 적용함으로써 동적인 환경에서도 최소의 오버헤드만으로 적용 가능하다. 특히, 삽입 및 삭제시 오버플로우 혹은 언더플로우가 발생되는 경우에도 액세스가 요구되는 블럭들은 루트에서 리프까지의 경로상에 나타나는 블럭들 뿐이므로, 이러한 연산으로 인한 오버헤드가 작다. 따라서 제안하는 기법은 동적인 환경에서 사용하기에 적합하다.

3) 저장 공간 이용의 효율성

위에서 제시한 두가지 요건을 지원하기 위한 별도의 디렉토리 정보를 위한 오버헤드가 지나치게 커진다면, 이 정보 저장을 위한 저장 공간의 오버헤드 자체가 문제가 될 뿐만 아니라, 이 정보들을 액세스하기 위한 검색의 성능 또한 문제가 될 수 있다. 따라서, 디렉토리 크기에 대한 오버헤드를 최소화하고, 블럭내의 저장 이용율(storage utilization)을 극대화할 수 있는 방법이 필요하다.

제안하는 기법에서 각 엔트리는 하나의 시그너춰 블럭과 일대일 대응되며, 삽입 알고리즘에서 보인 바와 같이 도메인내의 시그너춰가 존재하지 않는 구간에 대해서는 별도의 엔트리를 생성하지 않는다. 따라서 최하위 단계 디렉토리의 엔트리 수는 시그너춰 블럭 수와 같으며, 그 증가 형태는 O(n/a)가 된다. 여기서 n은 삽입된 시그너춰의 수이며, a는 시그너춰 블럭의 평균 블럭킹 인수이다. 유사한 방식으로 i번째 단계 디렉토리 엔트리 수의 증가 형태는 $O(n/(a*b^{i-1}))$이 된다. 여기서 b는 디렉토리 블럭의 평균 블럭킹 인수이다. 상위 단계 디렉토리 엔트리의 수는 최하위 단계 디렉토리 엔트리 수와 비교하여 매우 작으므로 제안된 기법을 위한 디렉토리 엔트리의 증가 형태는 삽입되는 시그너춰의 수에 선형적으로 비례한다. 따라서 제안된 기법의 저장 공간 오버헤드는 시그너춰들의 분포 특성에 영향을 받지 않고 일정하게 작음을 알 수 있다.

또한, 제안된 기법은 시그너춰의 삭제시 병합을 통하여 불필요한 블럭들을 적절한 시기에 블럭 풀로 반환함으로써 시그너춰 블럭 및 디렉토리 블럭의 저장 이용률을 높게 유지시킬 수 있다. 구현을 통한 실험한 결과에 의하면, 제안된 기법에서 시그너춰 블럭의 저장 이용률은 시그너춰의 도메인에서의 분포 특성과 객체 시그너춰의 수에 거의 영향을 받지 않고, 약 65% 내외의 안정된 값을 가짐을 볼 수 있었다. 이것은 제안된 기법의 저장 공간 오버헤드가 항상 삽입된 객체 시그너춰의 수에 비례하여 일정하게 나타남을 의미하는 좋은 결과이다.

예를 들어, 시그너춰의 크기가 32 바이트, 평균 디렉토리 엔트리의 크기가 16 바이트, 시그너춰 블럭과 디렉토리 블럭의 크기가 각각 4K 바이트인 환경에서 1,000,000개의 객체 시그너춰들을 제안된 기법을 이용하여 저장하는 경우에 요구되는 블럭의 수를 추정하면 다음과 같다. 먼저, 평균 저장 이용률이 65% 이므로 시그너춰 블럭의 평균 블러킹 인수는 ⌊(4K * 0.65) / 32⌋ = 83이며, 디렉토리 블럭의 평균 블러킹 인수는 ⌊(4K * 0.65) / 16⌋ = 166이다. 따라서 요구되는 시그너춰 블럭의 수는 ⌈1,000,000 / 83⌉ = 12,049가 된다. 유사한 방식으로 최하위 단계 디렉토리 블럭의 수는 ⌈12,049 / 166⌉ = 73이 되며, 이 블럭들과 대응되는 상위 단계 디렉토리 엔트리들은 한 블럭내에 저장될 수 있으므로 이것이 루트 디렉토리가 된다. 따라서 이 경우 요구되는 전체 블럭의 수는 12,049 + 73 + 1 = 12,123이 되며, 시그너춰 블럭 단계까지를 포함한 전체 트리의 깊이는 3이 된다.

4) 기존의 다목적 데이타 저장 엔진과의 자연스러운 통합 가능성

다목적 저장 엔진(general-purpose storage engine)이란, 다양한 응용에서 사용되는 데이타 객체들을 저장 및 관리할 수 있도록 개발된 저장 하부 구조를 의미하며, WiSS[1], Exodus[4], KAOSS[14], MIDAS[11] 등이 그 대표적인 예이다. 이러한 다목적 저장 엔진들은 B^+ 트리, 해쉬(hashing)등을 이용하여 주로 정형 객체에 대해서만 인덱싱을 제공하고 있다. 따라서 멀티미디어와 같은 비정형 객체의 인덱싱 위해서는 시그너춰 화일 등 적절한 액세스 기법의 지원이

요구된다.

한편, 시그너춰 화일 기법은 일종의 화일 처리 기법에 해당되므로 종합적인 데이타 관리를 위해서는 다목적 저장 엔진이 자연스럽게 제공하는 객체의 효율적인 저장, 버퍼링, 정형 객체에 대한 인덱싱, 다사용자를 위한 동시성 제어, 시스템 파손시의 회복 등 필수 기능들을 함께 제공해야 한다. 이러한 기능들을 처음부터 개발 할 수도 있으나, 이것은 엄청난 개발 시간 및 비용이 소요된다. 따라서 다목적 저장 엔진 기능과 시그너춰 화일의 장점을 갖는 시스템 개발을 위해서 기 개발된 다목적 저장 엔진에 시그너춰 화일을 통합함으로써 개발 노력을 최소화시킬 수 있다.

제안하는 기법에서는 사용되는 모든 정보들을 디스크 블럭내에서 관리하며, 선형 해쉬의 전용 디스크와 같은 별도의 블럭 할당 제약을 요구하지 않는다. 이 결과, 디스크 할당 단위로 블럭을 사용하는 기 개발된 대부분의 다목적 저장 엔진과의 통합시, 디스크 블럭의 할당, 버퍼링, 동시성 제어, 파손 회복 등에서 사용되던 기존의 전략들을 큰 변경없이 수용하여 구현할 수 있으며, 따라서 통합을 위한 오버헤드가 매우 작다.

V. 결 론

분할 시그너춰 화일은 시그너춰들을 같은 키값을 갖는 것들끼리 블럭 단위로 나누어 저장함으로써 질의 처리시 각 블럭내의 시그너춰들이 질의를 만족하는가의 여부를 디렉토리내에서 관리되는 키값의 비교를 통하여 판단하는 개선된 형태의 시그너춰 화일이다. 디렉토리는 원하는 시그너춰들이 저장된 블럭의 위치를 파악하는 메타 정보 역할의 하므로 불필요한 디스크 블럭 액세스를 방지함으로써 성능 개선 효과를 제공한다. 특히, 대용량 데이타베이스 환경에서는 디렉토리 자체가 매우 커지게 되므로 효율적인 디렉토리의 관리는 매우 중요하다.

본 논문에서는 분할 시그너춰 화일을 위한 효과적인 디렉토리 관리 기법에 관하여 논의하였다. 먼저, 기존에 제안된 Lee의 기법과 퀵 필터 기법의 디렉토리가 갖는 구조적 문제점들을 차례로 지적하고, 이들을 해결할 수 있는 새로운 디렉토리 관리 기법을 제안하였다. 제안된 기법은 대용량 데이타베이스 환경에서도 고속의 검색을 지원하며, 객체의 삽입과 삭제가 빈번한 동적인 환경에서도 잘 적응할 수 있다. 또한, 디렉토리를 위한 저장 공간의 오버헤드 및 저장 이용률에서도 좋은 성능을 제공한다. 뿐만 아니라, 제안된 기법은 기 개발된 다목적 저장 엔진내에 내용 기반 검색을 위한 구성 요소로서 자연스럽게 통합될 수 있다.

제안된 기법을 기반으로 하는 분할 시그너춰 화일이 현재 라이브러리 형태로 구현되어 있으며, 향후 연구 방향으로는 개별적으로 구현되어 있는 이 시스템을 다목적 저장 엔진과 통합하는 것을 고려하고 있다.

감사의 글

※ 논문의 심사 과정에 참여한 두 분의 심사 위원의 다양한 조언들이 본 논문의 질과 구성을 개선시키는데 큰 도움이 되었음을 밝힙니다. 또한, 제안된 기법의 구현 및 실험에 참여한 강원대학교 정보통신공학과 염 상민 군과 문 현수 양에게 감사를 드립니다.

참 고 문 헌

[1] Chou, H. T. et al., "Design and Implementation of the Wisconsin Storage System," *Software Practice and Experience*, vol. 15, no. 10, pp. 943-962, Oct. 1985.

[2] Christodoulakis, S. and Faloutsos, C., "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," *ACM Trans. on Information Systems*, vol. 2, no. 4, pp. 267-288, Oct. 1984.

[3] Du, D. H. et al., "An Efficient File Structure for Document Retrieval in the Automated Office Environment," *IEEE Trans. on Knowledge and Data Engineering*, vol. 1, no. 2, pp. 258-273, June 1989.

[4] Exodus Project Implementation Team, Using the Exodus Storage Manager, Exodus Project Document, University of Wisconsin, 1992.

[5] Faloutsos, C., "Access Methods for Text," *ACM Computing Surveys*, vol. 17, no. 1, pp. 49-74, Mar. 1985.

[6] Faloutsos, C., "Signature Files: Design and Performance Comparison of Some Signature Extraction Methods," In *Proc. Intl. Conf. on Management of Data,* pp. 63-82, ACM SIGMOD, 1985.

[7] Faloutsos, C. and Christodoulakis, S., "Description and Performance Analysis of Signature File Methods for Office Filing," *ACM Trans. on Office Information Systems,* vol. 5, no. 3, pp. 237-257, July 1987.

[8] Grandi, F. et al., "Frame-Sliced Partitioned Parallel Signature Files," In *Proc. Intl. Conf. on Information Retrieval,* ACM SIGIR, pp. 286-297, 1992.

[9] Gray, J. and Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann Publishers Inc., 1993.

[10] Horowitz, E., Sahni, S., and Anderson-Freed, S., *Fundamentals of Data Structures in C,* Computer Science Press, 1993.

[11] Kim, P. C. et al., "Design and Implementation of Multiuser Index-based Data Access System," In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications,* pp. 156-164, Tokyo, 1991.

[12] Lee, D. L. and Leng, C., "Partitioned Signature Files: Design Issues and Performance Evaluation," *ACM Trans. on Office Information Systems,* vol. 7, no. 2, pp. 158-180 Apr. 1989.

[13] Lee, D. L. and Leng, C., "Partitioned Signature File Structure for Multiattribute and Text Retrieval," In *Proc. Intl. Conf. on Data Engineering,* IEEE, pp. 389-397, 1990.

[14] Lee, Y. K. et al., "KAOSS: A General-Purpose Multiuser Database Storage System Supporting New Database Applications," *Database Review,* vol. 11, no. 2, pp. 91-110, July 1995.

[15] Leng, C. R. and Lee, D. L., "Optimal Weight Assignment for Signature Generation," *ACM Trans. on Database Systems,* vol. 17, no. 2, pp. 346-373, 1992.

[16] Lin, Z. and Faloutsos, C., "Framed-Sliced Signature Files," *IEEE Trans. on Knowledge and Data Engineering,* vol. 4, no. 3, pp. 281-289, 1992.

[17] Litwin, W., "Linear Hashing: A New Tool for Files and Table Addressing," In *Proc. Intl. Conf. on Very Large Data Bases,* VLDB, pp. 212-223, 1980.

[18] Sacks-Davis, R. and Ramamohanarao, K., "Multikey Access Methods Based on Superimposed Coding Techniques," *ACM Trans. on Database Systems,* vol. 12, no. 4, pp. 655-696, Dec. 1987.

[19] Whang, K. and Krishnamurthy, R., "The Multilevel Grid File: A Dynamic Hierarchical Multidimensional File Structure," In *Proc. Intl. Conf. on Database Systems For Advanced Applications,* pp. 449-459, Apr. 1991.

[20] Whang, K. Y., Kim, S. W., and Wiederhold, G., "Dynamic Maintenance ofData Distribution for Selectivity Estimation," *The VLDB Journal,* vol. 3, no. 1, pp. 29-51, 1994.

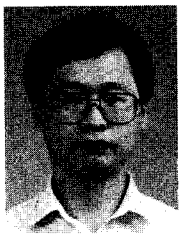
[21] G. Wiederhold, *Database Design,* 2nd Ed., McGraw-Hill Book Company, 1983.

[22] Zezula, P., Rabitti, F., and Tiberio, P., "Dynamic Partitioning of Signature Files," *ACM Trans. on Information Systems,* vol. 9, no. 4, pp. 336-369, Oct. 1991.

저 자 소 개

金 尙 煜(正會員)

1966년 8월 8일생. 1989년 서울대학교 컴퓨터공학과 공학사. 1991년 한국과학기술원 전산학과 공학석사. 1994년 한국과학기술원 전산학과 공학박사. 1991년 7월 ~ 8월 미 스탠포드 대학 방문 연구원. 1994년 3월 ~ 1995년 2월 정보전자연구소 Post-Doc. 1995년 3월 ~ 현재 강원대학교 정보통신공학과 조교수. 주관심분야는 데이타베이스 시스템, DBMS, 트랜잭션 프로세싱, 다차원 동적 화일, 공간 데이타베이스/GIS, 실시간 데이타베이스, 객체지향 데이타베이스, 멀티미디어 데이타베이스 등임

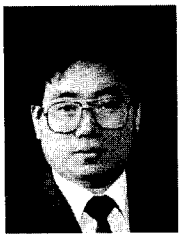


崔 晃 奎(正會員)

1962년 3월 10일생. 1984년 경북대학교 전자공학과 공학사. 1986년 한국과학기술원 전기및전자공학과 공학석사. 1989년 한국과학기술원 전기및전자공학과 공학박사. 1990년 3월 ~ 1993년 2월 강원대학교 전자공학과 조교수. 1993년 3월 ~ 현재 강원대학교 컴퓨터공학과 부교수. 1994년 7월 ~ 1995년 7월 Univ. of Florida, Database R&D Center 방문 교수. 관심 분야는 병렬 데이터베이스 시스템, 멀티미디어 저장 시스템, 병렬 I/O 시스템, 실시간 데이터베이스 시스템 등임

黃 煥 圭(正會員)

1952년 5월 26일생. 1976년 서울대학교 공업교육학과(전자전공) 공학사. 1987년 플로리다 대학 전기공학과 공학석사. 1992년 플로리다 대학 전기공학과 공학박사. 1994년 3월 ~ 현재 강원대학교 정보통신공학과 조교수. 주관심분야는 데이타베이스 시스템, 분산 데이타베이스 등임



尹 龍 翼(正會員)

1960년 11월 3일생. 1983년 동국대학교 통계학과 이학사. 1985년 한국과학기술원 전산학과 공학석사. 1994년 한국과학기술원 전산학과 공학박사. 1985년 3월 ~ 1997년 9월 한국전자통신연구원 책임연구원. 1997년 9월 ~ 현재 숙명여자대학교 전산학과 조교수. 주관심분야는 정보통신, 분산시스템, 실시간 처리 시스템, 멀티미디어, 분산 데이타베이스 시스템, 정보 검색 등임