GOVP1199705561

최종보고서

V.
150

정보검색을 위한 효율적인 저장시스템 개발

The Development of an Efficient Storage System for Information Retrieval

연구수행기관
한국과학기술연구원
연구개발정보센터

과 학 기 술 처

제 출 문

과학기술처 장관 귀하

본 보고서를 "정보검색을 위한 효율적인 저장시스템 개발" 과제의 최종 보고서로 제출합니다.

1996. 12. 31.

주관연구기관명 : 연구개발정보센터

총괄연구책임자 : 김진형 (연구개발정보센터 책임연구원)

연구개발책임자 : 이준호 (〃 선임연구원)

연 구 원 : 조영화 (〃 책임연구원)

박현주 (〃 선임연구원)

서정현 (〃 연구원)

최기석 (〃 〃)

박혁로 (〃 〃)

전준구 (〃 〃)

최윤수 (〃 〃)

최광남 (〃 〃)

안정수 (〃 〃)

여 백

요 약 문

I. 제 목

정보 검색을 위한 효율적인 저장 시스템 개발

II. 연구 개발의 목적 및 중요성

과학 기술 정보 유통의 핵심 중의 하나는 복잡, 다양해지고 기하급수적으로 증가하는 정보들을 효율적으로 저장하고, 사용자에게 신속히 제공해 줄 수 있는 시스템의 연구 및 개발이다. 지금까지 국내에서는 DBMS용 저장 시스템에 대한 연구 및 시제품 개발은 진행되어 왔으나, 비정형 데이터의 저장, 근접도 연산 등을 지원해야 하는 정보 검색용 저장 시스템의 연구 개발은 미비한 실정이다. 일반적으로 정보 검색용 저장 시스템은 사용 목적과 환경에 따라 그 기능을 확장 또는 축소하는 것이 바람직하다. 그러나 외국의 시스템을 도입하여 사용할 경우, 별도로 원시 코드가 제공되지 않는다면 기능의 확장 또는 축소를 위한 시스템의 변경이 근본적으로 불가능하다. 따라서 이러한 문제점을 궁극적으로 해결하기 위해서는 효율적인 정보 검색용 저장 시스템의 개발이 시급하다.

III. 연구 개발의 내용 및 범위

본 연구는 정보 검색을 위한 효율적인 저장 시스템의 개발을 목표로 하며, 세

부적인 연구 개발 내용은 다음과 같다.

1. 정보 저장 지원기 개발

- 입출력 관리
- 버퍼 관리
- 화일 디렉토리 관리
- 레코드 관리
- 대용량 객체 관리

2. 정보 저장 관리기 개발

- 문서 관리
- 색인 관리
- 카탈로그 관리

3. 정보 압축 복원기 개발

- 영상 정보의 압축 복원
- 문서 식별자의 압축 복원

IV. 연구 개발 결과 및 활용 방안

4.1 연구 개발 결과

본 연구에서는 정보 검색을 위한 정보 저장 시스템을 개발하였다. 개발된 정보 저장 시스템은 입출력 관리, 버퍼 관리, 화일 디렉토리 관리, 레코드 관리, 대용량 객체 관리를 담당하는 정보 저장 지원기, 문서 관리, 색인 관리, 카탈로그 관리를 담당하는 정보 저장 관리기, 그리고 정보 압축 및 복원기로 구성

된다.

4.2 활용 방안

본 연구의 최종 목표로서 개발된 정보 검색용 저장 시스템은 다음과 같이 활용될 수 있다.

- 연구개발정보센터에서 개발 중인 정보 검색 시스템의 하부 저장 시스템으로 사용한다.
- 정보 시스템을 구축하고자 하는 모든 분야에 보급하여 하부 저장 시스템으로 사용하도록 한다.

여 백

SUMMARY

I. Title

The Development of an Efficient Storage System for Information Retrieval

II. Objective

Information storage and retrieval systems help to provide users with useful information, and therefore the research and development on the systems is essential to effective distribution of science/technology information. A storage system for Information Retrieval (IR) should be designed to support unformatted data, proximity operation, et al. Though some research efforts have been made in Korea to develop storage systems for DBMS, little has been done for the development of the IR-oriented storage system. To use an IR system efficiently, it is desirable to modify the system according to the purpose and environment of its uses. Such modification requires source codes, which is not available for the systems purchased from foreign companies. In order to overcome these problems, it is crucial and urgent to develop an efficient IR-oriented storage system.

III. The Contents and Scope of the Study

The final objective of this project is to develope an efficient storage system for information retrieval. The detailed research items are as follows:

1. Development of an Information Storage Support System

- Input/output management
- Buffer management
- File Directory management
- Record management
- Large object management

2. Development of an Information Storage Management System

- Document management
- Index management
- Catalog management

3. Development of an Information Compression/Decompression System

- Image Compression/Decompression
- Document ID Compression/Decompression

IV. Result of the Study and Application Schemes

4.1 Result of the study

In this project, we finished the development of a storage system for IR. The system consists of an information storage support system, an information storage management system and an information compression/decompression system. The information storage support system is responsible for input/output management, buffer management, file directory management, record management and large object management. The information storage management system performs document management, index

management and catalog management. The information compression/decompression system deals with image compression/decompression and document ID compression/ decompression.

4.2 Application scheme

The final goal of this project is to develope an efficient storage system for IR, and the final product can be used as follows.

- The system can be used as a low-level storage system of the IR system that has been developed by KORDIC to provide retrieval services on science and technology information.
- The system can be distributed to various areas, and can be used to construct information systems.

여 백

목 차

Table of Contents

제 1 장 서 론

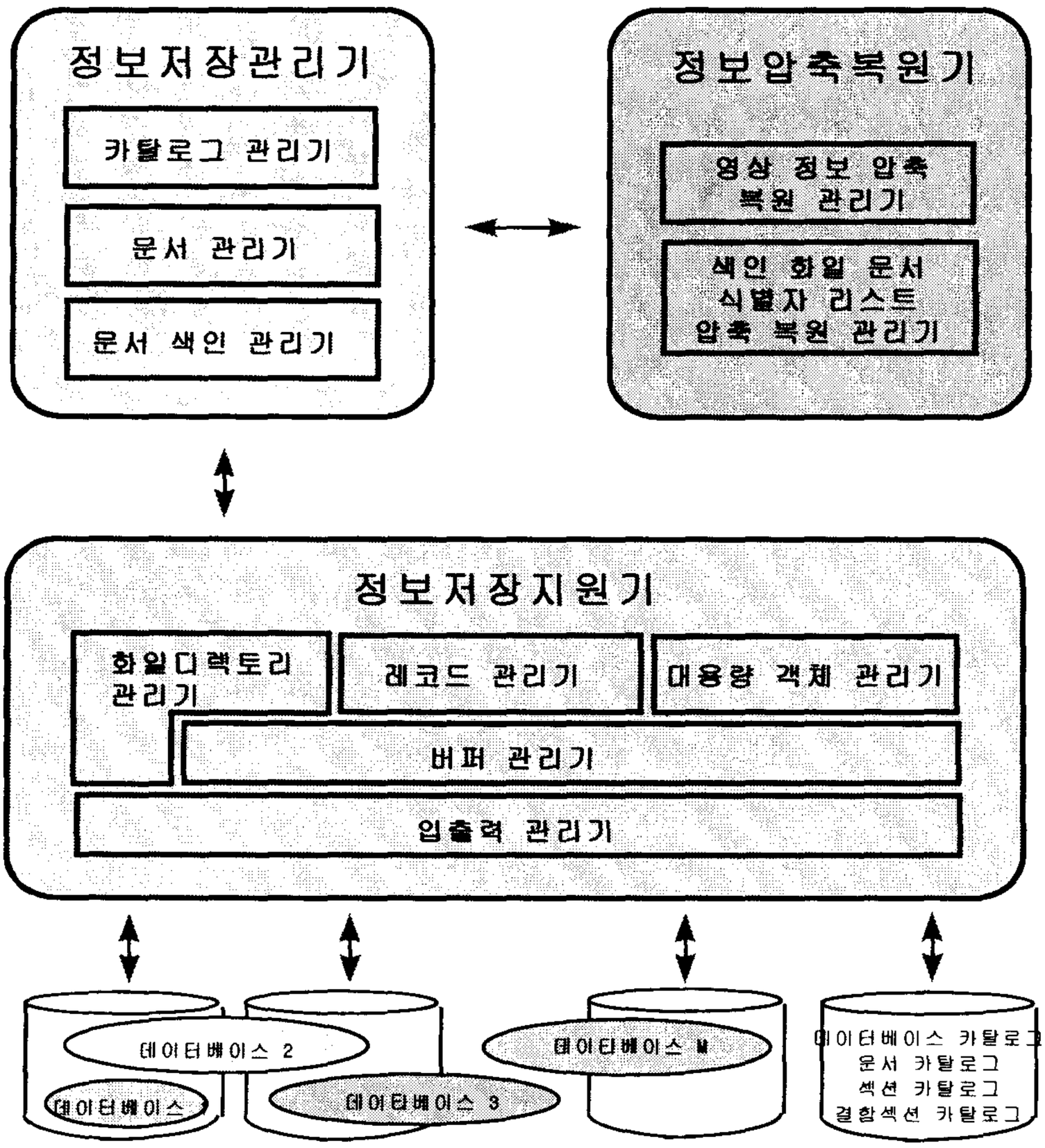
1950년대 초에 제 1 세대 컴퓨터가 출현한 이후로 컴퓨터를 이용하여 대용량의 문서를 효율적으로 검색할 수 있는 정보 검색 시스템에 관한 많은 연구가 이루어져 왔다. 정보 검색 시스템의 사용은 원하는 정보에 대한 접근을 용이하게 함으로써 여러 분야에서의 정보수집에 대한 시간과 노력을 단축시킨다. 특히 관리할 정보의 양이 기하급수적으로 증가하고 있는 정보화시대인 오늘날에는 효율적인 정보 검색 시스템에 대한 요구는 더욱 절실하다.

이미 외국에서는 1960년대 초에 일괄처리 방식에 의한 MEDLARS 시스템이 구축된 이후로 DIALOG, ORBIT, BRS, STAIRS 시스템과 같은 많은 정보 검색 시스템들이 상용화되어 현재까지 사용되고 있다. 그러나 국내에서는 아직까지 정보 검색 시스템에 대한 연구와 인식이 부족한 실정이다. 정보 검색에 관한 연구는 크게 사용자 인터페이스, 자동 색인, 검색 모델, 저장 시스템에 관한 연구로 구분된다. 여기서 저장 시스템은 실제 검색의 대상이 되는 정보들을 저장하고 관리하기 위한 구성 요소로서, 정보 검색 시스템의 성능에 중요한 영향을 미친다.

지금까지 데이터베이스 분야에서 저장 시스템에 관한 많은 연구가 진행되어 왔고, 이러한 연구 결과들을 기반으로 많은 저장 시스템들이 상용화되었다. 그러나 이러한 저장 시스템들은 정형화된 정보들을 주로 다루는 데이터베이스 시스템에 적합하도록 설계되었기 때문에 비정형화된 텍스트, 이미지, 수식 등의 정보를 다루는 정보 검색 시스템을 위한 저장 시스템으로 사용하기에는 적합하지 않은 것으로 인식되고 있다. 따라서 텍스트를 위주로 하는 복잡하고 다양한 형태의 정보를 효율적으로 처리할 수 있는 저장 시스템의 개발이 요구된다.

효율적인 정보검색을 위한 저장 시스템의 개발을 목적으로 하는 본 과제는 3년 여에 걸쳐 수행되었다. 먼저 1차년도에는 실 정보의 표본 분석을 통해 과학 기술 정보의 특성을 파악하였다. 그리고 기존의 다양한 저장 시스템들을 조사하여 본 과제의 특성에 적합하다고 판단된 저장 시스템 WiSS(Wisconsin Storage System)의 소스 코드를 분석하였고, 이를 바탕으로 정보 검색용 저장 시스템의 기본 구조를 설계하였다. 2차년도에는 1차년도의 기본 설계를 바탕으로 프로토타입 저장 시스템을 구현하였고, 실제 과학 기술 정보를 사용하여 구현된 프로토타입 저장 시스템의 성능을 평가하였다. 최종적으로 3차년도에는 프로토타입 시스템을 보완하기 위해 기본 설계를 확장 수정하였으며, 속도 향상 및 저장 공간의 절약을 위한 개선 노력이 이루어졌다.

[그림1.1]은 본 과제에서 개발한 저장 시스템의 전체적인 구조를 도시하고 있다. 정보 저장 지원기는 임의의 정보 저장 구조를 지원하기 위해 공통적으로 요구되는 기본적인 기능들을 제공한다. 최하위 층에 위치하는 입출력 관리기는 효율적인 정보의 입출력을 위해 직접적인 raw device 접근 방식의 독자적인 디스크 관리를 수행한다. 버퍼 관리기는 LRU(Least Recently Used) 방식의 버퍼 교환 알고리즘을 채택하고 있으며, 디스크에 존재하는 모든 화일들의 관리는 화일 디렉토리 관리기에 의해 관리된다. 레코드와 대용량 객체 관리기는 바이트 스트림으로 간주되는 정보 항목을 저장하고 접근 관리하는 기능을 수행하는 부분으로, 작은 크기의 레코드로부터 수백 메가 바이트에 이르는 대용량의 자료를 처리할 수 있다.



[그림 1.1] 저장 시스템의 전체 구성도

정보 저장 관리기는 하위의 정보 저장 지원기를 바탕으로 구현되며, 정보 검색의 주된 대상이 되는 비정형화된 정보 또는 문서들을 저장하고 관리하기 위한 기능들을 지원한다. 그리고 이들 정보의 검색을 위해 역화일 구조의 색인어 접근 방식을 지원하며, 빠른 속도로 역화일을 생성한다. 일반적으로 정보 검색 시스템은 사용자 또는 관리자로 하여금 데이터베이스를 보다 편리하게 처리할 수 있도록 하기 위해서 비정형화된 정보 또는 문서들의 구조에 관

한 정보를 제공하여야 한다. 이를 위해 본 과제에서는 데이터베이스의 목록을 관리하기 위한 목록 관리기를 설계 구현하였다.

최근에는 문서들에 단순한 텍스트 정보뿐만 아니라 컬러나 흑백 그림들이 포함되어 있다. 따라서 이러한 영상 정보들을 효율적으로 저장하기 위해 정보를 압축했다가 다시 복원할 수 있는 기능이 요구된다. 이를 위해 본 과제에서의 정보 압축 관리기는 CCITT의 표준안 JBIG(Joint Bi-level Image experts Group)을 기반으로 하는 이진 영상 압축 복원 관리기를 구현하고 있으며, 컬러 및 그레이 영상을 압축 복원하는 방법인 JPEG(Joint Phtographic Experts Group)을 보다 효과적으로 활용하는 방법으로서 인간의 시각 특성을 고려한 압축 복원 관리기를 구현하고 있다. 한편 이 정보 압축 복원기에서는 역화일 구조의 문서 색인 화일을 압축 복원하기 위한 방법론을 제공하고 있다. 대개의 경우 역화일 접근 방식의 색인 화일은 많은 저장 공간을 문서 식별자 리스트를 유지하는데 소모하고 있다. 따라서 본 과제에서 개발된 정보 압축 복원기는 비손실 압축 기법에 기반하여 문서 식별자의 리스트를 효율적으로 압축 복원할 수 있는 기능을 구현하고 있다.

본 보고서의 구성은 다음과 같다. 2장에서 정보 저장 지원기의 설계 및 구현 내용을 기술하고, 3장에서 정보 저장 관리기의 설계 및 구현 내용을 설명한다. 그리고 4장에서 정보 압축 복원기에 대해 설명하고, 끝으로 5장에서 결론을 제시한다.

제 2 장 정보 저장 지원기

정보 저장 지원기는 개발된 저장 시스템의 최하위에 위치하며, 입출력 관리기, 버퍼 관리기, 화일 디렉토리 관리기, 레코드 관리기, 대용량 객체 관리기로 구성된다. 본 장에서는 정보 저장 지원기의 구성 요소에 대하여 기술한다.

2.1 입출력 관리기

기존의 UNIX 운영체제를 기반으로 하는 저장 시스템은 UNIX가 지니고 있는 제약 때문에 좋은 성능을 갖지 못하였다. 즉, UNIX의 화일 시스템은 작은 크기의 블럭을 사용하고 있고 결집 개념(clustering)을 지원하고 있지 않기 때문에 디스크에서 주기억 장치로 정보를 이동시키는데 있어서 전송 시간(transfer time)보다 찾음 시간(seek time)에 많은 시간을 소모한다.

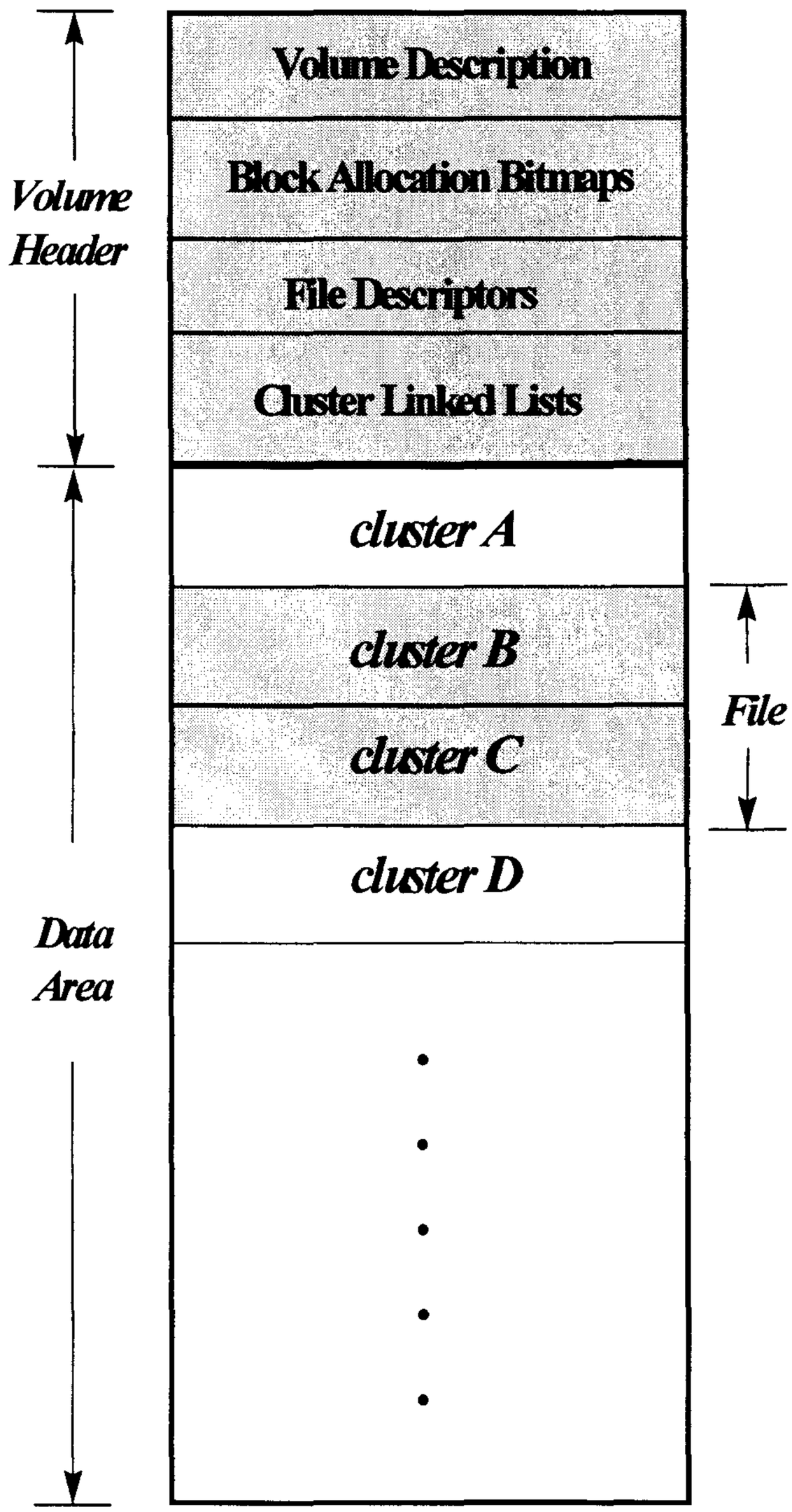
UNIX를 기반으로 할 때의 이러한 문제점을 피하기 위해서 본 과제에서 개발한 저장 시스템은 Raw Device 접근 방식을 통해 직접적으로 디스크를 관리하는 방법을 채택하였다. 즉, 디스크와 주기억 장치 사이의 정보의 입출력의 단위인 블럭의 크기를 시스템의 효율성을 고려하여 확장할 수 있도록 하였으며, 하나의 화일에 대해 물리적으로 연속적인 기억장소를 할당함으로써 찾음 시간을 줄이도록 하였다. 또한 개발된 저장 시스템은 실제 디스크를 직접 접근함으로써 UNIX의 버퍼 관리층을 우회하여 사용자의 주소공간에 정보를 이동시키는 시간을 줄인다.

2.1.1 입출력 관리기의 설계

입출력 관리기에서는 물리적 디스크를 볼륨이라고 부른다. 그리고 각각의 볼륨은 일련의 연속된 블럭들로 구성된 클러스터의 집합이다. 블럭(block)은 스크와 사용자 버퍼 사이에 정보의 입출력을 위해 사용되는 기본 입출력 단위로서, 여러 개의 연속된 디스크 블럭은 하나의 클러스터를 형성한다. 블럭의 크기는 cBLOCKSIZE의 상수로 정의되며, 현재 4K 바이트의 기본값을 갖는다. 볼륨의 블럭들을 구분하기 위해 각각의 블럭은 유일한 식별자를 갖는다.

클러스터(cluster)는 물리적으로 연속된 블럭들의 집합으로, 입출력 관리자는 클러스터 단위로 화일에 저장 영역을 할당한다. 클러스터의 크기는 블럭의 수로 표현되며, 사용자는 새로운 볼륨을 생성할 때 클러스터의 크기를 지정할 수 있다.

볼륨(Volume)은 사용자의 화일들을 저장하기 위한 논리적인 저장 공간을 의미하는 것으로, 물리적 디스크나 테이프 등이 여기에 포함된다. 볼륨은 여러 개의 클러스터들로 나누어지며, 같은 볼륨 상에서 클러스터의 크기는 일정하다. 그러나 서로 다른 볼륨 사이의 클러스터 크기는 사용자에 의해 다르게 지정될 수 있다. 볼륨은 시스템 전체에 걸쳐 유일한 식별자를 부여 받는다. 그리고 각 볼륨의 헤더에 볼륨 관리를 위한 제어 정보를 갖는다. 이 볼륨 제어 정보에는 클러스터 할당을 위한 비트맵, 블럭 할당을 위한 비트맵, 화일 관리를 위한 화일 디스크립터 리스트 등이 포함된다. 볼륨 헤더는 다시 각각 일련의 연속된 블럭들로 구성된 4개의 영역으로 구분된다. [그림 2.1]의 상층부에 볼륨 헤더의 구조가 도시되어 있다.


Volume Description
Block Allocation Bitmaps
File Descriptors
Cluster Linked Lists
cluster A
cluster B
cluster C
cluster D
Volume Header
Data Area
File

[그림 2.1] 볼륨의 구조

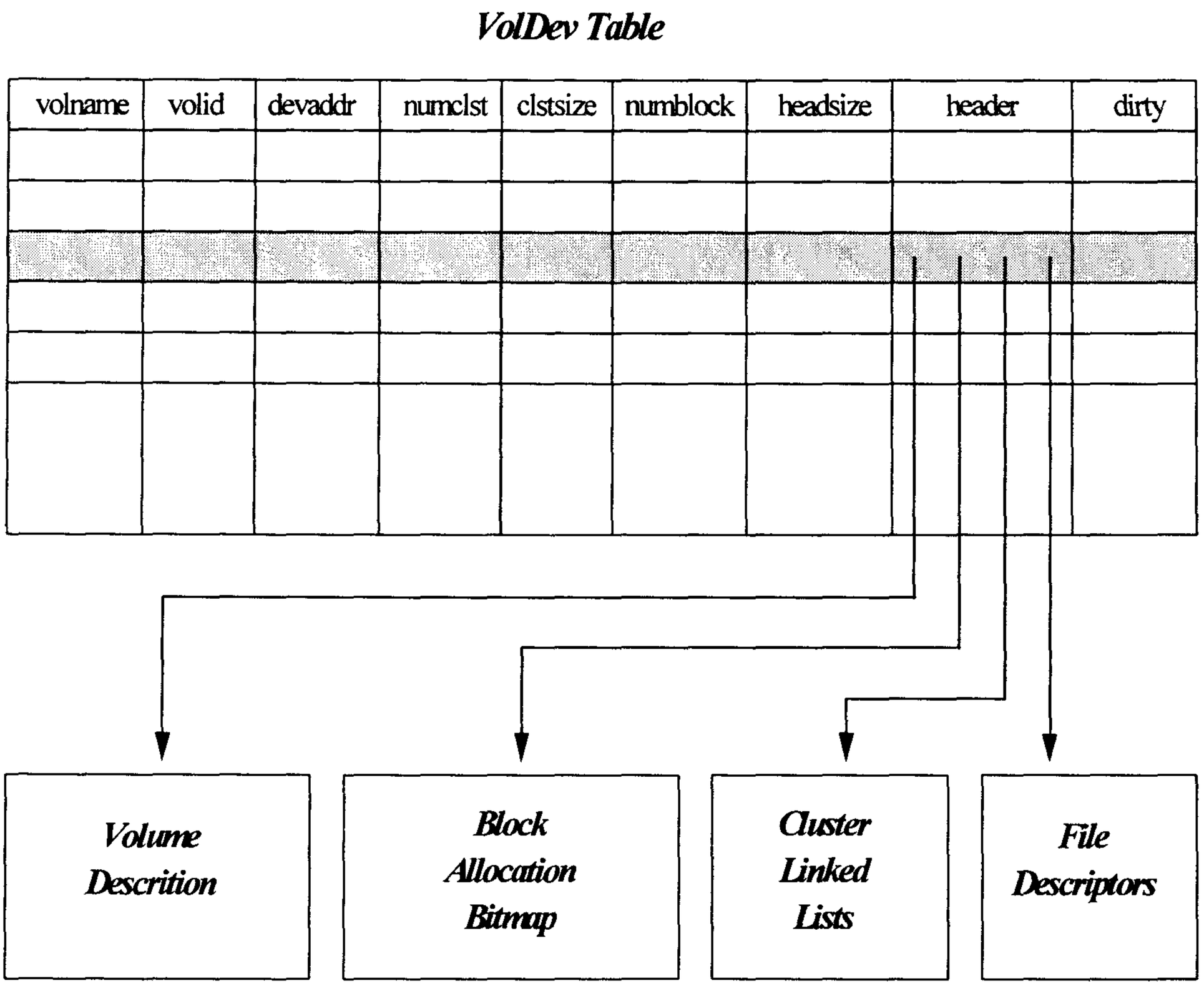
볼륨 헤더의 첫번째 영역은 볼륨에 대한 정보와 실제 물리적 장치의 구조에 대한 정보를 포함한다. 다음은 이 영역에서 유지되고 있는 제어 정보들을 나타낸다.

- 볼륨 식별자
- 볼륨의 제목
- 클러스터의 크기(블럭들의 수)
- 볼륨에 있는 모든 클러스터들의 수
- 볼륨에 있는 자유 클러스터들의 수
- 볼륨에 생성할 수 있는 화일들의 최대 수
- 현재 생성되어 있는 화일들의 수
- 클러스터 할당 여부를 표시하는 비트맵

볼륨 헤더의 두번째 영역은 볼륨 내에 있는 모든 클러스터들에 대해서 각 클러스터를 구성하는 블럭들의 할당 여부를 나타내는 비트맵을 포함하고 있다. 그리고 세번째 영역에는 볼륨에 존재하는 모든 화일들에 대한 디스크립터들이 포함되어 있다. 이 화일 디스크립터는 해당 화일에 할당되어 있는 모든 클러스터들의 리스트에 대한 시작 포인터를 유지한다. 마지막의 네번째 영역은 볼륨 상의 모든 화일에 할당된 클러스터들의 리스트들을 유지하며, 각 리스트는 클러스터 번호의 배열로써 구현되었다.

[그림 2.2]는 사용자의 볼륨에 대한 접근을 관리하기 위해 설계된 입출력 관리기의 내부 자료구조를 보여준다. 이 중에서 가장 중요한 것은 VolDev라고 불리우는 볼륨 테이블로서, 이 테이블은 현재 시스템 상에 온라인으로 마운트되어 있는 모든 볼륨들에 대한 정보를 유지한다. 시스템이 시작될 때 입출력 관리기는 VolDev 테이블을 초기화하고, 지정된 볼륨을 마운트할 때 그 볼륨의 헤더에 있는 내용을 볼륨 테이블로 읽어 들인다. 그리고 프로그램의 실행 중에 이 테이블에 있는 내용을 바탕으로 입출력 관리기는 볼륨에 대한 접근을

관리한다. VolDev 테이블의 각 엔트리는 (1) 볼륨의 이름, (2) 볼륨의 물리적인 주소, (3) 마운트된 볼륨의 식별자, (4) 볼륨 헤더를 저장하고 있는 버퍼들로 구성된다. 시스템은 프로그램의 수행 중에 각각의 볼륨을 식별하기 위해서 볼륨 식별자를 사용하며, 이 볼륨 식별자를 통해 VolDev 테이블을 탐색함으로써 해당 볼륨에 대한 정보를 얻을 수 있다.



[그림 2.2] 입출력 관리기의 내부 자료 구조

2.1.2 입출력 관리기의 구현

io_init( )

입출력 관리기를 기동시키는 함수로서, 볼륨 관리를 위한 VolDev 테이블을 주기억 장치에 할당하고 초기화한다.

가) 주기억 장치에 VolDev 테이블을 위한 저장 영역을 할당한다.

나) VolDev 테이블을 초기화한다. 즉, 테이블의 모든 엔트리에 대해 Volume ID와 Volume Address를 NOVOL로, Volume Name을 NULL로 세팅한다.

io_final( )

입출력 관리기를 종료하는 함수로서, 모든 마운트된 볼륨들의 사용을 중지시키고, 할당된 VolDev 테이블 영역을 반환한다.

가) 모든 마운트된 볼륨들에 대해 다음을 수행한다.

나) VolDev 테이블에 있는 변경된 볼륨 헤더 정보를 볼륨 헤더에 기록한다. 그리고 볼륨의 사용을 중지시키고, VolDev 테이블의 내용을 삭제한다.

io_mount(volname)
char *volname;

volname에 의해 지시된 볼륨을 마운트하는 기능을 수행하는 함수로서, 볼륨에 관한 정보를 볼륨 헤더 영역으로부터 VolDev 테이블로 읽어들이고 볼륨 식별자를 반환한다.

가) volname에 의해 지시된 볼륨이 이미 마운트되어 있는지를 확인한다.

나) 만일 마운트되어 있다면, 그 볼륨의 식별자를 반환하고 종료한다.

다) 그렇지 않을 경우 내부 함수 IO_Mount()를 이용하여 VolDev 테이블로부터 볼륨의 정보를 저장하기 위한 빈 테이블 엔트리를 할당받고, 볼륨의 헤더 정보를 읽어 들인다.

io_dismount(volname)
char *volname;

volname에 의해 지시된 볼륨의 사용을 중지시키고, VolDev 테이블로부터 그 볼륨과 관련된 엔트리를 제거한다.

가) volname에 의해 지시된 볼륨이 이미 마운트되어 있는지를 확인한다.
나) 만일 마운트되어 있지 않다면 에러를 반환하고 함수를 종료한다.
다) 그렇지 않은 경우 내부 함수 IO_Dismount()를 이용하여 변경된 볼륨 헤더 정보를 VolDev 테이블로부터 볼륨 헤더 영역으로 기록하고, 볼륨의 사용을 중지시킨다. 그리고 해당 VolDev 테이블 엔트리를 삭제한다.

io_createfile(volid, numblocks, clusterfillfactor, fileid)
FOUR volid;
FOUR numblocks;
FOUR clusterfillfactor;
FID *fileid;

volid에 의해 지시된 볼륨에 최소한 numblocks 개의 블럭을 갖는 새로운 화일을 생성하고, 그 화일의 식별자를 반환한다. 이 과정에서 생성된 새로운 화일에 대한 정보가 볼륨 헤더에 기록되고, 클러스터들이 화일에 할당된다.

가) 화일을 생성하려는 볼륨이 이미 마운트되어 있는지를 확인한다.
나) clusterfillfactor, numblocks의 인자들이 적절한지를 확인한다.
다) 생성되는 화일에 할당해야 할 클러스터의 갯수를 계산한다.
라) 볼륨에 화일과 클러스터를 할당할 자원이 충분히 있는지를 확인한다.
마) 볼륨에 앞에서 계산된 수 만큼의 클러스터를 갖는 화일을 할당한다.
바) 새롭게 생성된 화일의 식별자를 fileid의 매개변수를 통하여 반환한다.

io_destroyfile(fileid)
FID fileid;

볼륨으로부터 fileid에 의해 지시되는 화일을 제거하고, 그 화일이 차지하고 있던 모든 클러스터들을 볼륨의 자유 영역으로 반환한다.

가) fileid의 적절성 여부를 검사한다.
나) VolDev 테이블을 탐색하여 볼륨이 이미 마운트되어 있는지를 확인한다.
다) 내부 함수 IO_FreeFile()을 호출하여 볼륨으로부터 화일을 제거한다.

io_allocblocks(fileid, nearblockid, numblocks, blockIDs)
FID *fildid;
BID *nearblockid;
FOUR numblocks;
BID blockIDs[];

fileid에 의해 지시되는 화일에 numblocks 갯수 만큼의 블럭들을 할당하고, 할당된 블럭들의 식별자를 blockIDs[]를 통해 반환한다. 만일 nearblockid의 값이 명시되면, 새로운 블럭들을 nearblockid에 의해 지시되는 블럭 근처에 할당한다. 블럭들을 할당할 때 필요한 경우에 새로운 클러스터들이 할당될 수 있다.

가) fileid에 의해 지시되는 화일을 저장하고 있는 볼륨이 이미 마운트되어 있는지를 확인한다.
나) 만일 nearblockid의 인자 값이 명시되었다면, 가능한 범위 내에서 nearblockid에 의해 지시되는 블럭의 근처에 새로운 블럭들을 할당한다. 이를 위해 nearblockid를 이용해 새로운 블럭을 할당할 클러스터를 계산하고, 내부 함수 IO_AllocBlockInCluster()를 호출하여 앞에서 계산된 클러스터에 가능한 갯수 만큼의 블럭들을 할당한다. 이 과정에서 새로 할당된 블럭들의 식별자는 blockIDs[]에 기록된다.
다) 앞의 과정에서 numblocks 만큼의 모든 블럭들이 할당되었다면 함수를 종료하고, 그렇지 않으면 다음을 계속한다.

라) fileid에 의해 지시되는 화일에 이미 할당되어 있는 클러스터들의 리스트를 찾아 이 리스트에 있는 각각의 클러스터를 좇아 가면서 나머지 블럭들을 할당한다. 앞에서와 같이 IO_AllocBlockInCluster()가 이용된다. 그리고 모든 블럭이 할당되면 함수를 종료하고, 그렇지 않으면 다음을 계속한다.

마) 아직도 할당되지 않은 블럭들이 있을 경우 새로운 클러스터들을 할당하고, 할당된 클러스터 내에서 모자라는 블럭들을 할당한다. 그리고 클러스터 리스트에 새로 할당된 클러스터들을 추가한다.

io_freeblock(fileid, blockid)
FID *fildid;
BID *blockid;

fileid에 의해 지시되는 화일에서 blockid에 의해 지정되는 블럭을 화일의 자유 블럭 리스트로 반환한다.

가) fileid와 blockid의 인자 값들의 타당성 여부를 검사한다.

나) fileid에 의해 지시되는 화일을 저장하고 있는 볼륨이 이미 마운트되어 있는지를 확인한다.

다) 내부 함수 IO_FreeBlockInCluster()를 이용하여 blockid에 의해 지시되는 블럭을 포함하고 있는 클러스터를 계산하고, 그 클러스터에서 블럭을 자유 영역으로 반환한다. 이 과정에서 볼륨 헤더 영역의 블럭 할당 비트맵의 내용이 갱신된다.

io_readblock(blockid, buffer)
BID *blockid;
char *buffer;

blockid에 의해 지시되는 블럭의 내용을 buffer로 지시되는 메모리 버퍼로 읽어 들인다.

가) blockid 인자 값의 적절성 여부를 검사한다.

나) blockid에 의해 지시되는 블럭을 포함하고 있는 볼륨이 이미 마운트되어 있는지를 확인한다.

다) 내부 함수 IO_ReadBlock()을 호출하여 디스크 블럭을 buffer에 의해 지시되는 메모리 버퍼로 읽어 들인다.

io_writeblock(blockid, buffer, type, flag)
BID *blockid;
char *buffer;
FOUR type;
FOUR *flag;

buffer에 의해 지시되는 메모리 버퍼 블럭을 blockid에 의해 지시되는 디스크 블럭으로 기록한다.

가) blockid에 의해 지시되는 블럭을 포함하고 있는 볼륨이 이미 마운트되어 있는지를 확인한다.

나) 내부 함수 IO_WriteBlock()를 호출하여 buffer에 의해 지시되는 메모리 버퍼의 내용을 디스크 블럭으로 기록한다.

io_volid(volname)
char *volname;

volname에 의해 지시되는 볼륨의 식별자를 반환한다.

가) VolDev 테이블을 탐색하여 볼륨이 이미 마운트되어 있는지를 확인한다.

나) 만일 볼륨이 마운트되어 있다면 해당 볼륨의 식별자를 반환한다.

2.2 버퍼 관리기

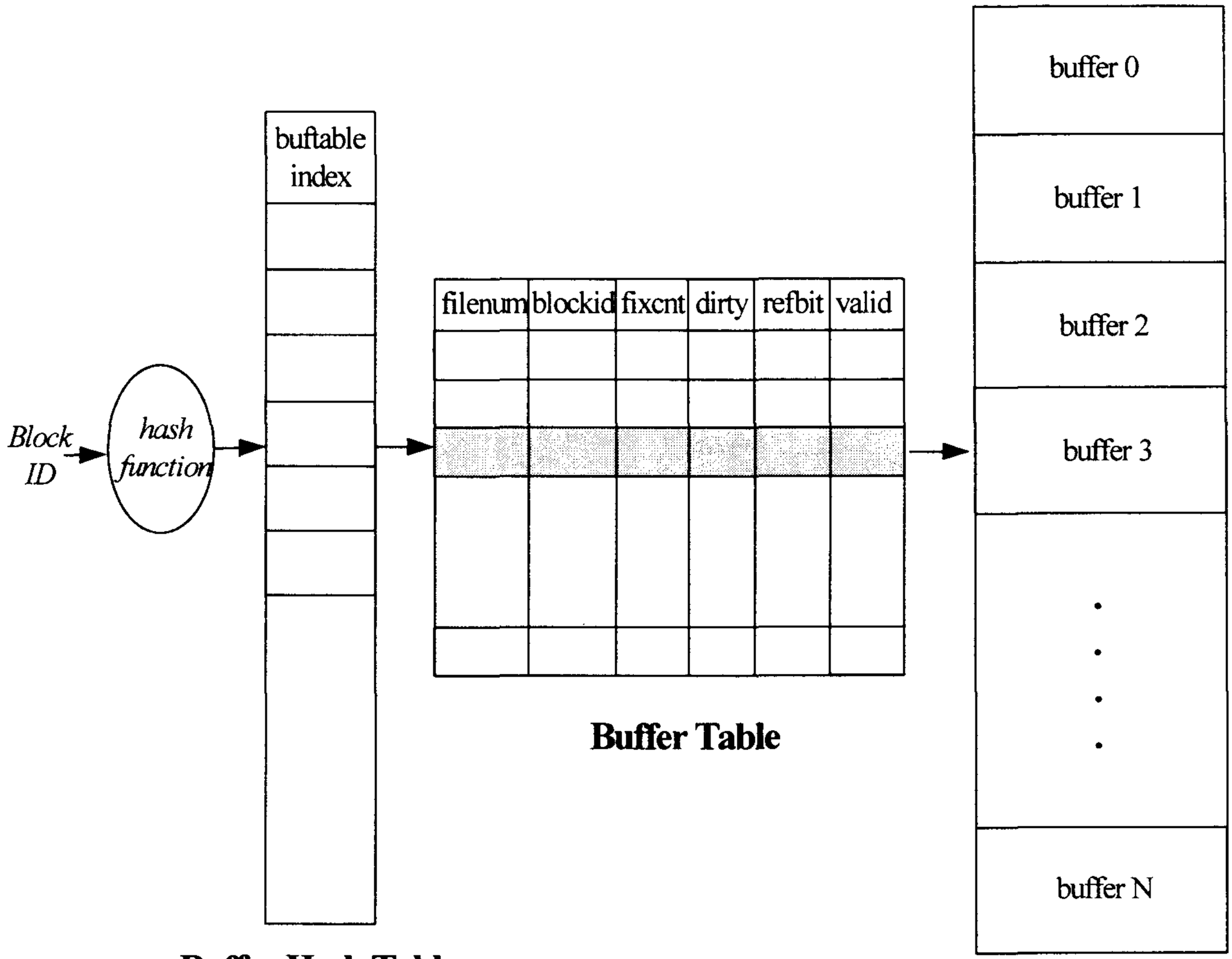
버퍼 관리기는 자주 사용되는 정보를 메모리에 캐쉬함으로써 디스크 입출력을 최소화하는 모듈로서, 일정한 수 만큼의 버퍼 블럭들로 이루어진 버퍼 풀을 관리한다. 이 관리기는 버퍼 블럭의 내용을 볼륨으로부터 읽거나 혹은 다시 볼륨에 기록하기 위해 하위의 입출력 관리기를 호출한다.

본 과제에서 개발한 저장 시스템의 버퍼 관리기는 LRU(least recently used) 방식과 hint의 개념을 결합한 버퍼 블럭 교환 알고리즘을 사용한다. Hint란 어떤 블럭이 시스템 상에서 갖는 중요성의 정도를 나타낸다. 다시 말해 저수준의 힌트(low hint)를 갖는 버퍼 블럭은 시스템의 성능에 별 영향을 미치지 않고 즉시 교체될 수 있다는 것을 의미하며, 중간 수준의 힌트(mid hint)를 갖는 버퍼 블럭은 자신보다 중요도가 적은 버퍼 블럭들이 더 이상 버퍼 풀에 존재하지 않을 경우에 버퍼 블럭 교환을 위해 이용될 수 있다는 것을 의미한다. 그리고 고수준의 힌트(high hint)를 갖는 버퍼 블럭은 최후의 버퍼 블럭 교환을 위한 후보로서 이용된다는 것을 의미한다.

2.2.1 버퍼 관리기의 설계

[그림 2.3]은 버퍼 관리기의 내부 자료 구조를 보여준다. 버퍼 풀은 볼륨의 블럭과 동일한 크기를 갖는 메모리 버퍼 블럭들의 집합을 나타내며, 볼륨의 블럭들이 여기에 캐쉬된다. 사용자는 시스템의 특성에 따라 적정한 수의 버퍼를 상수 cMAXBUFFERS를 통해 지정할 수 있다.

버퍼 테이블은 버퍼 풀에 있는 모든 버퍼 블럭들의 상태 및 정보를 유지하기 위한 자료 구조로서, 각각의 버퍼 블럭에 대해 하나의 엔트리를 갖는다. 각각의 엔트리의 내용은 다음과 같다.



[그림 2.3] 버퍼 관리기의 내부 자료 구조

- filenum : 화일 번호
- blockid : 버퍼가 포함하고 있는 블럭의 볼륨 내에서의 식별자
- fixcnt : 버퍼 블럭의 사용자 수
- dirty : 버퍼 블럭이 변경되었는지를 표시하는 플래그
- refbit : 버퍼 블럭이 최근에 참조되었는지를 표시하는 플래그
- valid : 버퍼 블럭 내용의 적절성 여부를 표시하는 플래그

버퍼 관리기는 블럭 식별자를 통한 버퍼 블럭의 빠른 접근을 위해 버퍼 해쉬 테이블을 유지한다. 이 테이블은 선형 해쉬 구조를 이용하여 구현되었다. 버퍼 해쉬 테이블의 각 엔트리는 버퍼 테이블에 대한 인덱스를 포함하며, 해쉬 값의 충돌을 줄이기 위해 해쉬 테이블 엔트리는 버퍼 테이블 엔트리의 2배로 할당되었다.

2.2.2 버퍼 관리기의 구현

bf_init( )

버퍼 관리기를 시동하는 모듈로서, 버퍼 테이블과 버퍼 해쉬 테이블을 초기화한다.

가) 버퍼 풀의 모든 버퍼 블럭들을 자유 영역으로 설정한다.
나) 버퍼 테이블의 모든 엔트리들을 초기화한다.
다) 버퍼 해쉬 테이블을 초기화한다.

bf_final( )

버퍼 관리기를 종료시키는 모듈로서, 입출력 관리기의 io_writeblock()을 호출하여 버퍼 풀에 있는 모든 버퍼 블럭들의 내용을 볼륨에 저장하고, 버퍼 테이블과 버퍼 해쉬 테이블의 모든 내용을 삭제한다.

가) 버퍼 풀에 있는 모든 변경된 블럭들을 볼륨에 저장한다.
나) 버퍼 테이블의 모든 엔트리의 내용을 삭제한다.
다) 버퍼 해쉬 테이블의 모든 내용을 삭제한다.

bf_mount(volname)

volname에 의해 지시되는 볼륨을 마운트할 때 호출되며, 현재 아무런 작업을 수행하지 않는다.

bf_dismount(volname)
char *volname;

volname에 의해 지시되는 볼륨의 블럭들을 저장하고 있는 모든 버퍼 블럭들을 자유 버퍼 영역으로 반환한다. 이때 만일 버퍼 블럭이 변경되었다면, 먼저 버퍼 블럭의 내용을 볼륨에 기록한다.

가) 버퍼 테이블의 정보를 이용하여 버퍼 풀의 모든 버퍼 블럭들에 대해 다음의 작업을 수행한다.
나) 버퍼 블럭의 블럭 식별자를 통해 그 버퍼 블럭이 volname에 의해 지시되는 볼륨에 속하는지를 검사한다.
다) 버퍼 블럭의 내용이 변경되었는지를 확인하고, 만일 변경된 경우에는 입출력 관리기의 io_writeblock()을 호출하여 버퍼 블럭의 내용을 볼륨에 기록한다.
라) 버퍼 블럭을 버퍼 풀의 자유 영역으로 반환하며, 해쉬 테이블로부터 그 블럭에 대한 내용을 삭제한다.

bf_openfile(filenum, mode)
int filenum;
int mode;

filenum에 의해 지시되는 화일을 개방할 때 호출되는 모듈로서, 현재 아무런 작업도 수행하지 않는다.

bf_closefile(filenum)
int filenum;

filenum에 의해 지시되는 화일에 할당되어 있는 모든 버퍼 블럭들을 버퍼 풀의 자유 버퍼 영역으로 반환한다. 이 모듈은 변경된 버퍼 블럭들의 내용이 모두 볼륨에 기록되었다고 가정한다. 따라서 사용자는 이 모듈을 호출하기 전에 bf_flushbuf()를 수행함으로써 변경된 버퍼 블럭들의 내용이 볼륨에 기록되도록 해야 할 책임이 있다.

가) 버퍼 테이블의 모든 엔트리에 대해 다음을 수행한다.
나) 각 엔트리의 해당 버퍼 블럭이 filenum에 의해 지시되는 화일에 속할 경우, 엔트리의 내용을 삭제함으로써 버퍼 블럭을 자유 영역으로 반환한다.

bf_getbuf(filenum, blockid, returblock)
int filenum;
BID *blockid;
BLOCKreturnblock;**

filenum과 blockid에 의해 지시되는 화일의 데이터 블럭을 위해 버퍼 풀로부터 하나의 버퍼 블럭을 할당하고, returnblock을 통해 할당된 버퍼 블럭의 메모리 주소를 반환한다. 만일 버퍼 블럭이 이미 할당되어 있다면, 별도의 새로운 버퍼 블럭의 할당 없이 이미 할당되어 있는 버퍼 블럭의 메모리 주소를 반환한다.

가) blockid에 의해 지시되는 블럭 식별자를 통해 해쉬 값을 계산하여 이미 버퍼 블럭이 할당되어 있는지를 검사한다.
나) 만일 이미 버퍼 블럭이 할당되어 있다면, 그 버퍼 블럭에 대한 사용자 수(fixcnt)를 1 만큼 증가시키고, 버퍼 블럭의 메모리 주소를 반환한다.
다) 그렇지 않고 버퍼 풀에 할당되어 있지 않다면, 새로운 버퍼 블럭을 할당하고, 버퍼 블럭 할당에 관한 정보를 버퍼 테이블에 기록한다.

bf_freebuf(filenum, blockid, bufblock)
int filenum ;
BID *blockid ;
BLOCK*bufblock;

filenum과 blockid에 의해 지시되는 블럭을 위해 할당되어 있는 버퍼 블럭을 자유 버퍼 영역으로 반환한다.

가) 버퍼 테이블로부터 버퍼 블럭에 대한 엔트리를 찾는다.
나) 버퍼 테이블 엔트리의 fixcnt의 값을 1 만큼 감소시킴으로써 사용자의 수가 줄었음을 표시한다.
다) 만일 감소시킨 fixcnt의 값이 0 이면, 자유 버퍼 블럭의 수를 1 만큼 증가시키고, 버퍼 참조 플래그인 refbit를 TRUE로 설정함으로써 버퍼 블럭이 최근에 참조된 것임을 표시한다.

bf_readbuf(filenum, blockid, returnblock)
int filenum ;
BID *blockid;
BLOCK **returnblock;

filenum과 blockid에 의해 지시되는 화일 블럭을 주기억 장치의 버퍼 풀로 캐쉬한다. 그리고 캐쉬된 버퍼 블럭의 메모리 주소를 반환한다. 만일 화일 블럭에 대한 버퍼 블럭이 이미 버퍼 풀에 캐쉬되어 있다면, 블럭을 읽기 위해 볼륨을 접근할 필요가 없으므로 입출력 시간을 단축할 수 있다. 그렇지 않고 화일 블럭에 대한 버퍼 블럭이 할당되어 있지 않다면, 버퍼 풀에서 새로운 버퍼 블럭을 할당받고, 화일 블럭의 내용을 할당된 버퍼 블럭으로 읽어 들인다.

가) 버퍼 해쉬 테이블을 탐색함으로써 filenum과 blockid에 의해 지시되는 화일 블럭에 대한 버퍼 블럭이 이미 버퍼 풀에 할당되었는지를 검사한다.
나) 만일 버퍼 블럭이 버퍼 풀에 존재하지 않다면, 새로운 버퍼 블럭을 할당하고, 입출력 관리기의 io_readblock()를 호출하여 filenum과 blockid에 의해

지시되는 화일 블럭의 내용을 버퍼 블럭으로 읽어들인다. 그리고 읽어 들인 화일 블럭에 대한 정보(filenum, blockid, fixcnt 등)를 버퍼 테이블에 삽입한다. 마지막으로 그 버퍼 테이블 엔트리에 대한 인덱스를 버퍼 해쉬 테이블에 추가한다.

다) 만일 버퍼 블럭이 이미 버퍼 풀에 할당되어 있다면, 그 버퍼 블럭에 대한 버퍼 테이블 엔트리의 fixcnt 항목을 1 만큼 증가시켜 사용자 수가 증가하였음을 표시한다.

라) 마지막으로 returnblock을 통해 버퍼 블럭에 대한 메모리 주소를 반환함으로써 함수를 종료한다.

bf_flushbuf(filenum)
int filenum;

filenum에 의해 지시되는 화일을 위해 할당된 모든 버퍼 블럭의 내용을 볼륨에 저장한다.

가) 버퍼 풀에 있는 모든 버퍼 블럭들에 대해 버퍼 테이블의 정보를 이용하여 다음의 작업을 수행한다.

나) 먼저 버퍼 블럭이 filenum에 의해 지시되는 화일에 속해 있고 버퍼 블럭의 내용이 적절한지를 확인한다. 만일 그러한 경우에 버퍼 블럭의 내용이 변경된 것이라면, io_writeblock()를 호출하여 변경된 버퍼 블럭의 내용을 볼륨의 화일 블럭으로 기록한다.

다) 그리고 해당 버퍼 테이블 엔트리의 내용을 적절하게 재설정한다.

bf_setdirty(filenum, blockid, bufblock)
int filenum ;
BID *blockid ;
BLOCK *bufblock;

filenum과 blockid에 의해 지시되는 화일 블럭을 저장하고 있는 버퍼 블럭의 내용이 변경되었음을 표시한다.

가) 화일 번호 filenum과 블럭 식별자 blockid의 값이 적절한지를 확인한다.
나) 버퍼 블럭의 정보를 유지하고 있는 버퍼 테이블 엔트리를 찾는다.
다) 버퍼 테이블 엔트리의 dirty 항목을 TRUE로 설정한다.

bf_discard(filenum, blockid, bufblock)
int filenum ;
BID *blockid ;
BLOCK *bufblock;

filenu과 blockid에 의해 지시되는 화일 블럭을 저장하고 있는 버퍼 블럭의 내용이 더 이상 적절하지 않음을 표시한다.

가) 화일 번호 filenum과 블럭 식별자 blockid가 적절한지를 확인한다.
나) 버퍼 블럭에 대한 정보를 유지하고 있는 버퍼 테이블 엔트리를 찾는다.
다) 버퍼 테이블 엔트리의 내용을 다음과 같이 수정한다. 즉, fixcnt 항목을 1 만큼 감소시키고, valid 항목을 FALSE로 설정한다. 이때 만일 fixcnt의 값이 0이면 자유 버퍼 블럭의 수를 1 만큼 증가시키고, 버퍼 해쉬 테이블로부터 해당 버퍼 테이블 엔트리에 대한 인덱스를 삭제한다.

bf_invalidate(numblocks, blockids)
int numblocks;
BID *blockids;

blockids에 의해 지시되는 일련의 블럭들의 내용이 더 이상 적절하지 않음을 표시한다.

가) blockids에 의해 지시는 일련의 블럭들에 대해 다음의 작업을 수행한다.
나) 각 블럭에 해당하는 버퍼 테이블 엔트리를 찾아 valid 항목의 값을 FALSE

로 설정한다.

다) 버퍼 해쉬 테이블에서 버퍼 테이블 엔트리에 대한 인덱스를 삭제한다.

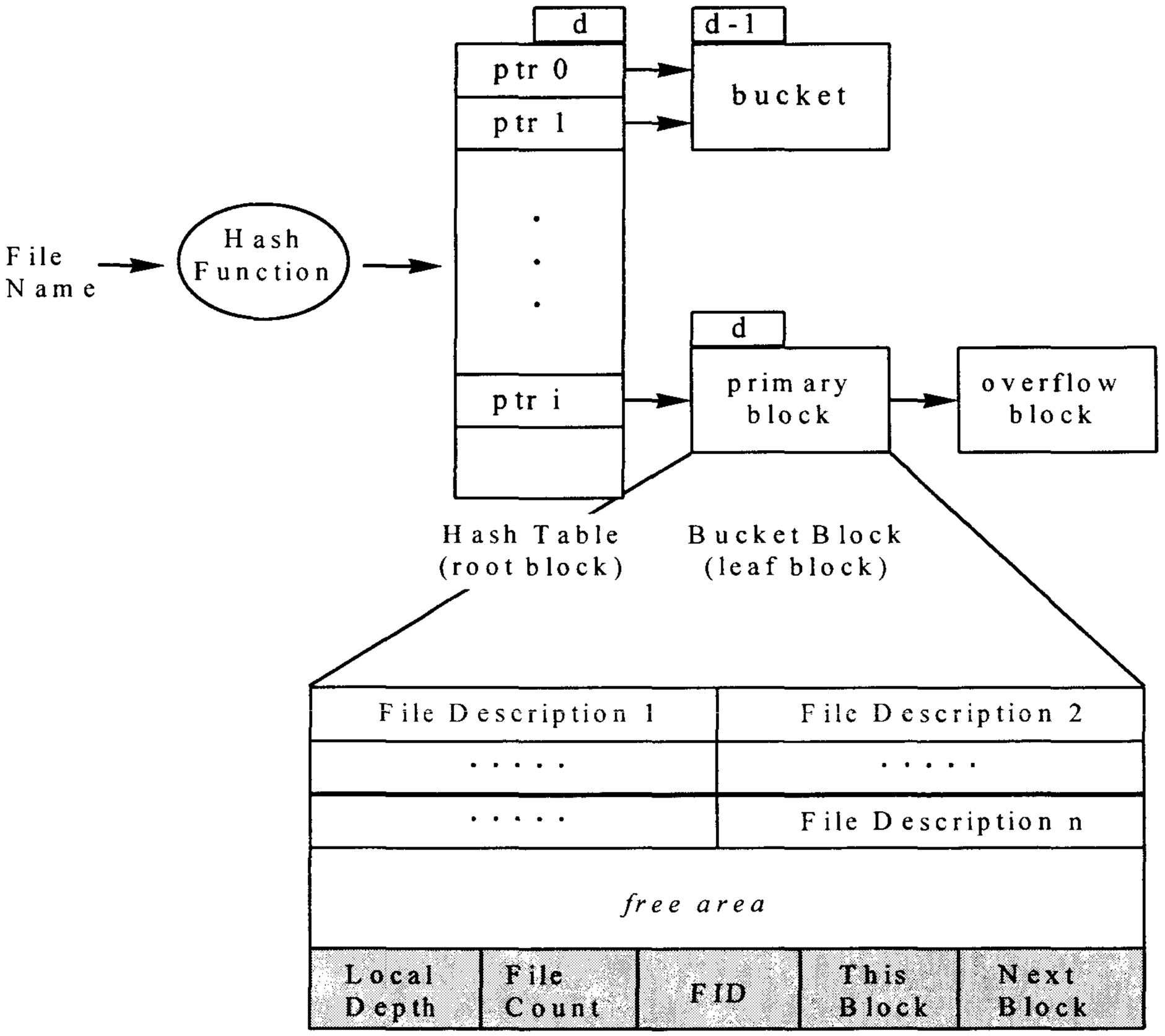
2.3 화일 디렉토리 관리기

화일 디렉토리 관리기는 블럭이나 클러스터를 화일에 할당하는 작업과 같은 기본적인 화일 관리 인터페이스를 상위 모듈에 제공한다. 입출력 관리기와 더불어 외부 저장 장치인 볼륨의 물리적인 특징을 상위 모듈로부터 분리하도록 구현하였으며, 하나의 화일에 대한 저장 공간을 가능한 범위 내에서 물리적으로 결집(clustering)하기 위한 블럭 관리 방식을 취하고 있다. 이러한 기능의 수행을 위하여 화일 디렉토리 및 화일에 관한 정보를 유지하는 화일 테이블, 그리고 각 화일을 위해 할당된 클러스터와 블럭들에 관한 정보를 유지하기 위한 클러스터 할당 테이블 및 블럭 할당 테이블 등의 자료구조를 관리하며, 화일의 생성, 소멸, 개폐에 관한 연산 및 화일의 접근 제어에 관한 연산 등을 지원한다.

2.3.1 화일 디렉토리 관리기의 설계

본 과제에서 개발한 저장 시스템은 생성된 모든 화일들의 관리를 위해 볼륨내에 화일 디렉토리(file directory)를 유지한다. 화일 디렉토리는 사용자 가 화일 구분을 위해 사용하는 화일 이름을 시스템 내부에서 관리되는 화일 정보와 연결한다. 화일 디렉토리에서 관리하는 내부 정보에는 화일 식별자(file ID), 화일의 첫번째 블럭 식별자, 화일의 마지막 블럭 식별자 등이 포함된다. 화일 디렉토리는 각각의 볼륨마다 하나씩 생성되며, 생성된 디렉토리의 시작 블럭에 대한 주소가 볼륨 헤더의 고정된 위치에 기록된다.

[그림 2.4]에 도시되어 있는 것처럼 화일 디렉토리는 확장성 해싱(extendible hashing)과 체인화된 버켓 해싱(chained bucket hash)을 결합함으로써 구현되었다. 대개의 경우 화일 디렉토리는 UNIX와 같은 운영체제에서 메모리에 저장할 수 없을 정도로 크기 때문에 특정한 화일을 참조하기 위해 여러 번의 볼륨 접근이 요구된다. 확장성 해싱 방법은 최소한의 볼륨 접근으로 원하는 화일을 참조할 수 있도록 한다. 즉, 해시 테이블을 탐색하기 위한 볼륨 접근과 리프 블럭을 참조하기 위한 볼륨 접근만이 필요하다.



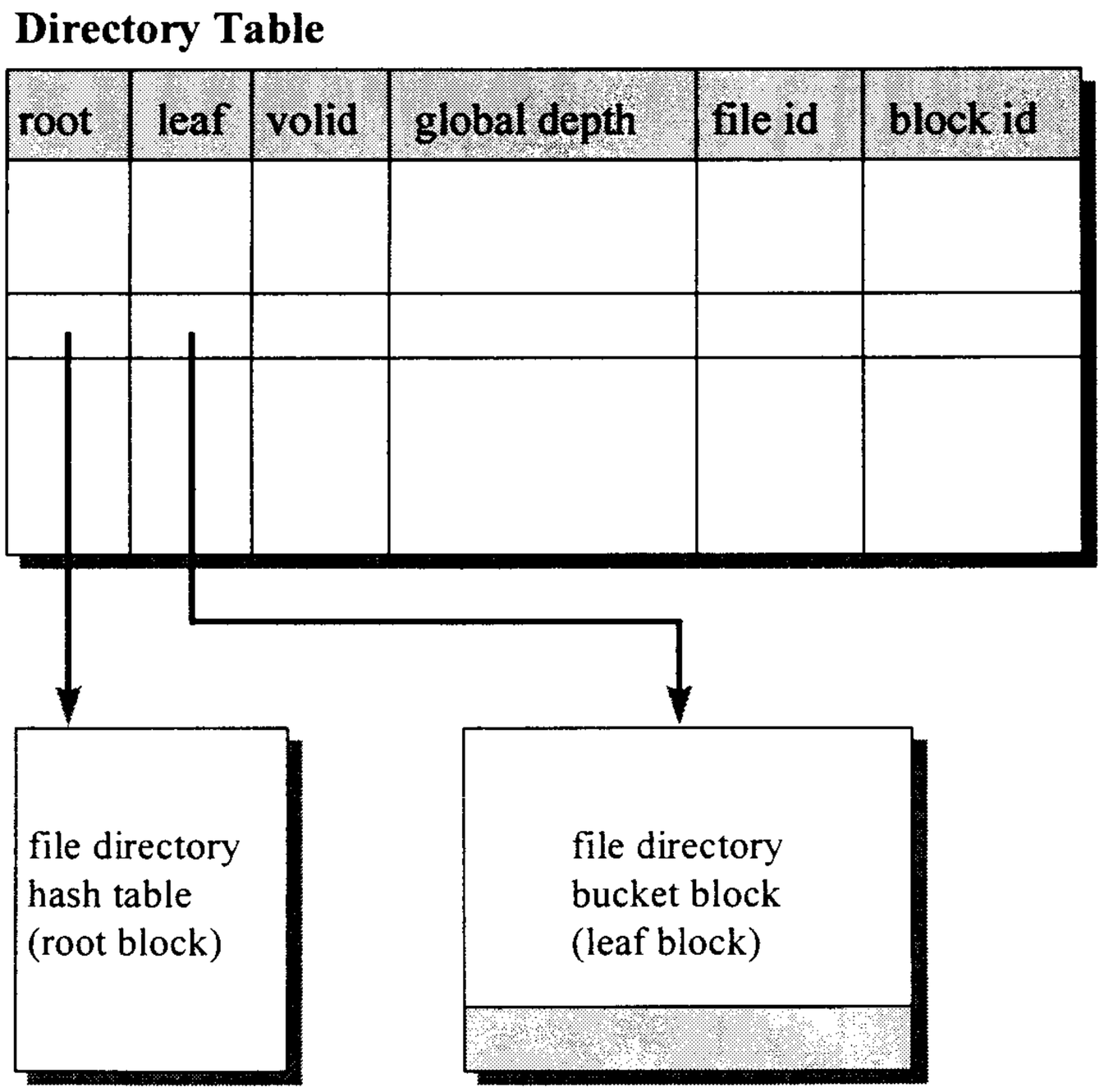
[그림 2.4] 화일 디렉토리의 구조

이 외에도 확장성 해싱을 이용함으로써 얻을 수 있는 장점은 리프 블럭들의 수를 최소화할 수 있다는 점이다. 초기의 화일 디렉토리는 하나의 해시 테이블과 하나의 리프 블럭으로 구성된다. 만일 새로운 화일들의 계속되는 생성으로 인해 리프 블럭의 영역이 가득 차면, 해시 테이블의 크기는 두배로 증가하며, 화일 디렉토리의 엔트리들은 디렉토리의 깊이(depth)를 표시하는 정수값 d에 따라 두개의 리프 블럭들로 분산된다. 본 시스템에서는 화일 디렉토리의 해시 테이블에 할당되는 주기억 장치의 양을 제한하기 위해 각 해시 테이블의 최대 크기를 한 블럭으로 제한하였다. 그러나 이것은 새로운 엔트리를 추가하려는 블럭상에 빈 공간이 더 이상 없고 해시 테이블이 이미 가득 찼을 때 문제를 발생시킬 수 있다. 이러한 경우 해시 테이블을 두개의 블럭으로 확장하는 대신 새로운 오버플로우 블럭을 생성하고 이를 오버플로우된 버켓 블럭에 연결하도록 구현하였다.

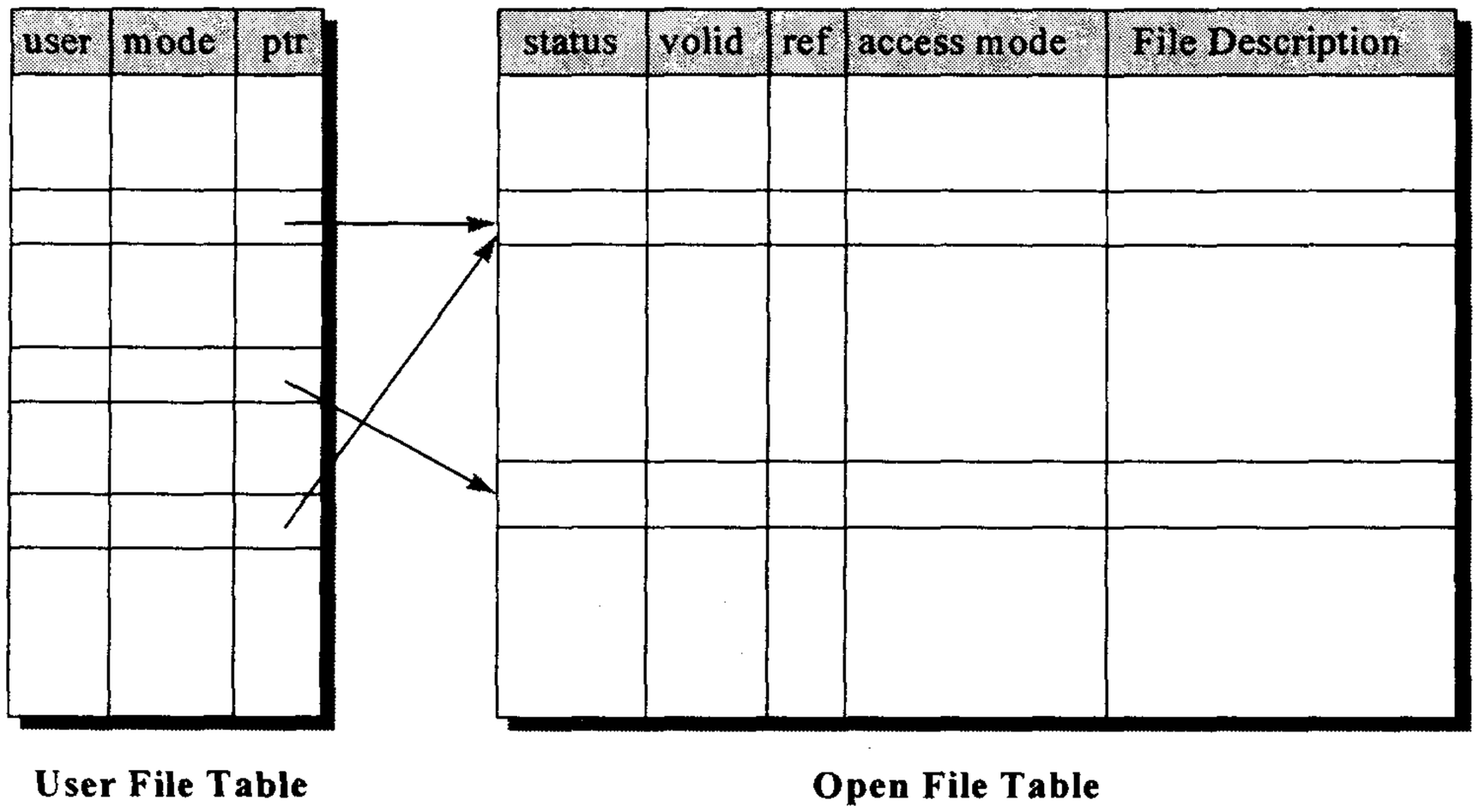
화일 디렉토리 관리기에서 해시 테이블은 메모리에 상주하도록 구현되었다. 그렇게 함으로써 응용 프로그램들은 단지 한번의 볼륨 접근으로 원하는 화일에 대한 정보를 참조할 수 있다. 화일 디렉토리에서 화일 정보의 관리를 위해 유지하는 화일 디스크립터는 다음과 같은 항목들로 구성된다.

- 화일이름
- 화일식별자
- 화일크기
- 화일소유자
- 화일의 접근 권한을 나타내는 플래그
- 화일타입
- 화일의 첫번째 블럭
- 화일의 마지막 블럭
- 화일에 있는 블럭들의 수
- 화일내에 있는 레코드의 수

화일 디렉토리 관리기는 효율적인 화일 관리를 위해 내부적으로 디렉토리 테이블(directory table), 화일 테이블(open file table), 사용자 화일 테이블(user file table)을 사용한다. 이 테이블들은 모든 마운트된 볼륨에 있는 모든 화일들의 정보를 유지하기 위한 메모리 캐쉬로 생각할 수 있다. 디렉토리 테이블은 화일 디렉토리의 메모리 테이블로서, 각각의 볼륨마다 하나의 엔트리를 유지하며, 각 화일 디렉토리의 루트 블럭과 현재 접근 중인 화일의 정보를 포함하고 있는 리프 블럭을 갖는다. [그림 2.5]는 디렉토리 테이블의 구조를 보여준다.



[그림 2.5] 디렉토리 테이블의 구조



[그림 2.6] 화일 테이블과 사용자 화일 테이블

화일 테이블은 사용자에 의해 개방된 모든 화일의 정보를 저장하는 테이블로서, 테이블의 각 엔트리는 다음의 항목들을 유지한다.

- 화일이 속해 있는 볼륨의 식별자
- 화일에 대한 참조의 수
- 화일 접근 모드(READ 또는 WRITE)
- 화일 디스크립터(화일 디렉토리의 리프 블럭에서 읽어 들인다)

이외에도 화일 디렉토리 관리기는 각각의 사용자들이 접근하고 있는 화일들을 관리하기 위해 사용자 화일 테이블을 유지한다. 시스템에서는 사용자가 개방시킨 화일의 접근 모드와 화일 테이블에 대한 포인터로 구성된 엔트리들을 저장한다. [그림 2.6]은 화일 테이블과 사용자 화일 테이블의 관계를 도시하고 있다.

2.3.2 화일 디렉토리 관리기의 구현

dir_initialize( )

화일 관리를 위한 디렉토리 테이블, 화일 테이블, 사용자 화일 테이블을 초기화한다.

가) 볼륨 화일 디렉토리의 관리를 위한 디렉토리 테이블을 할당하고 초기화한다.

나) 사용자에 의해 개방된 모든 화일들의 정보를 관리하기 위한 화일 테이블과 사용자 화일 테이블을 할당하고 초기화한다.

dir_final( )

화일 관리를 위해 할당된 디렉토리 테이블, 화일 테이블, 사용자 화일 테이블을 종료시킨다.

가) 볼륨의 화일 디렉토리 관리를 위한 디렉토리 테이블을 종료한다.

나) 모든 개방된 화일들의 정보를 관리하기 위한 화일 테이블을 종료한다.

다) 사용자에 의해 개방된 화일들의 정보를 관리하기 위한 사용자 화일 테이블을 종료한다.

dir_mount(volumename)
char *volumename;

volumename에 의해 지시되는 볼륨을 마운트한다.

가) 입출력 관리기의 io_mount()를 호출한다.

나) 볼륨의 화일 디렉토리를 주기억 장치의 디렉토리 테이블로 캐쉬한다.

dir_dismount(volumename)
char *volumename;

volumename에 의해 지시되는 볼륨의 사용을 종료한다.

가) 현재 개방되어 있는 화일들 중에서 volumename에 의해 지시되는 볼륨에 존재하는 모든 화일들을 닫는다.

나) 디렉토리 테이블에서 해당 화일 엔트리의 변경 내용을 볼륨의 화일 디렉토리로 복사한다.

다) 입출력 관리기의 io_dismount()를 호출한다.

dir_createfile(volid, filename, numblocks, clusterfillfactor, blockfillfactor)
int volid;
char filename;
int numblocks;
int clusterfillfactor;
int blockfillfactor;

volid에 의해 지시되는 볼륨에 화일을 생성한다. 여기서 filename은 생성하려고 하는 화일의 이름을 나타내고, numblocks는 화일에 할당될 블럭수를 나타낸다.

가) 입출력 관리기의 io_createfile()를 호출하여 볼륨에 새로운 화일을 생성하고, 그 화일에 numblocks 만큼의 블럭들을 할당한다.

나) 새로 생성된 화일에 대한 정보를 디렉토리 테이블에 기록한다.

dir_destroyfile(volid, filename)
int volid;
char *filename;

volid와 filename에 의해 지시되는 화일을 삭제한다.

가) 디렉토리 테이블에 접근하여 삭제하려는 화일에 대한 정보를 얻는다.

나) 사용자가 filename에 의해 지시되는 화일에 대해 소유권한과 접근권한을 갖고 있는지를 확인한다.

다) 만일 소유권한과 접근권한을 갖고 있다면, 화일 디렉토리로부터 화일에 대한 정보를 삭제하고,

라) 입출력 관리기의 io_destroyfile()을 호출함으로써 그 화일을 위해 할당된 모든 클러스터와 블럭들을 자유 영역으로 반환한다.

dir_openfile(volid, filename, accessmode)
int volid;
char filename;
int accessmode;

volid와 filename에 의해 지시되는 화일을 개방하고 화일 번호를 반환한다. 여기서 accessmode는 화일 접근 모드로서, READ 또는 WRITE의 값을 갖는다.

가) 새로운 사용자 화일 테이블 엔트리를 할당받는다.

나) 화일 테이블을 참조하여 이미 사용자가 개방하였는지를 확인한다.

다) 만일 이미 개방하였다면, 화일 접근 모드의 상충 여부를 확인한다. 그리고 참조 계수를 증가시키고, 접근 권한을 적절히 조정한다.

라) 만일 처음 개방하는 것이라면, 화일 테이블로부터 하나의 빈 엔트리를 할당받고, 화일에 대한 정보를 볼륨의 화일 디렉토리로부터 이 엔트리로 복사한다. 그리고 화일 테이블 엔트리의 기타 관련 정보(참조 계수, 접근 모드 등)를 설정한다.

마) (가) 단계에서 할당받은 사용자 화일 테이블 엔트리의 내용을 설정한다.

바) 개방 화일 번호를 반환한다.

dir_closefile(filenum)
int filenum;

화일 번호 filenum에 의해 지시되는 화일을 닫는다. 필요한 경우에 화일 테이브로부터 관련 엔트리를 제거하고 화일 디렉토리를 갱신한다.

가) filenum에 의해 지시되는 화일에 대한 참조 계수를 하나 감소시킨다.
나) 만일 화일에 대한 접근 모드가 WRITE이면, 화일과 관련된 모든 버퍼 블럭들의 내용을 볼륨에 기록한다.
다) 만일 화일의 내용이 변경되었다면, 화일 디렉토리의 해당 화일 디스크립터를 변경한다.
라) 관련된 사용자 화일 테이블 엔트리를 제거한다.

dir_volumesync(volumename)
char * volumename;

volumename에 의해 지시되는 볼륨의 변경된 내용을 갱신한다.

가) volumename에 의해 지시되는 볼륨의 모든 화일을 위해 할당된 버퍼 블럭들의 내용을 볼륨에 저장한다.
나) 화일 디렉토리의 내용을 갱신한다.
다) 볼륨 헤더의 내용을 갱신한다.

dir_rename(volid, newfilename, oldfilename)
int volid;
char *newfilename;
char *oldfilename;

oldfilename에 의해 지시되는 화일의 이름을 newfilename으로 변경한다.

가) newfilename의 이름을 갖는 화일이 이미 볼륨에 존재하는지를 확인하여 이미 존재하면 에러를 반환한다.

나) oldfilename에 의해 지시되는 화일에 대해 소유 권한과 접근 모드를 검사한다.

다) 변경하기 이전의 화일에 대한 디스크립터를 얻어온다.

라) 화일 디스크립터의 화일 이름 항목을 새로운 화일 이름으로 설정한다.

마) 변경된 화일 디스크립터를 다시 볼륨의 화일 디렉토리에 저장한다.

dir_syncfile(filenum)
int filenum;

filenum에 의해 지시되는 화일에 관련된 내용들을 볼륨에 저장한다.

가) 화일을 위해 할당된 버퍼 블럭들을 볼륨의 해당 블럭들로 복사한다.

나) 필요한 경우에 화일 디렉토리의 화일 디스크립터의 내용을 변경한다.

2.4 레코드 관리기

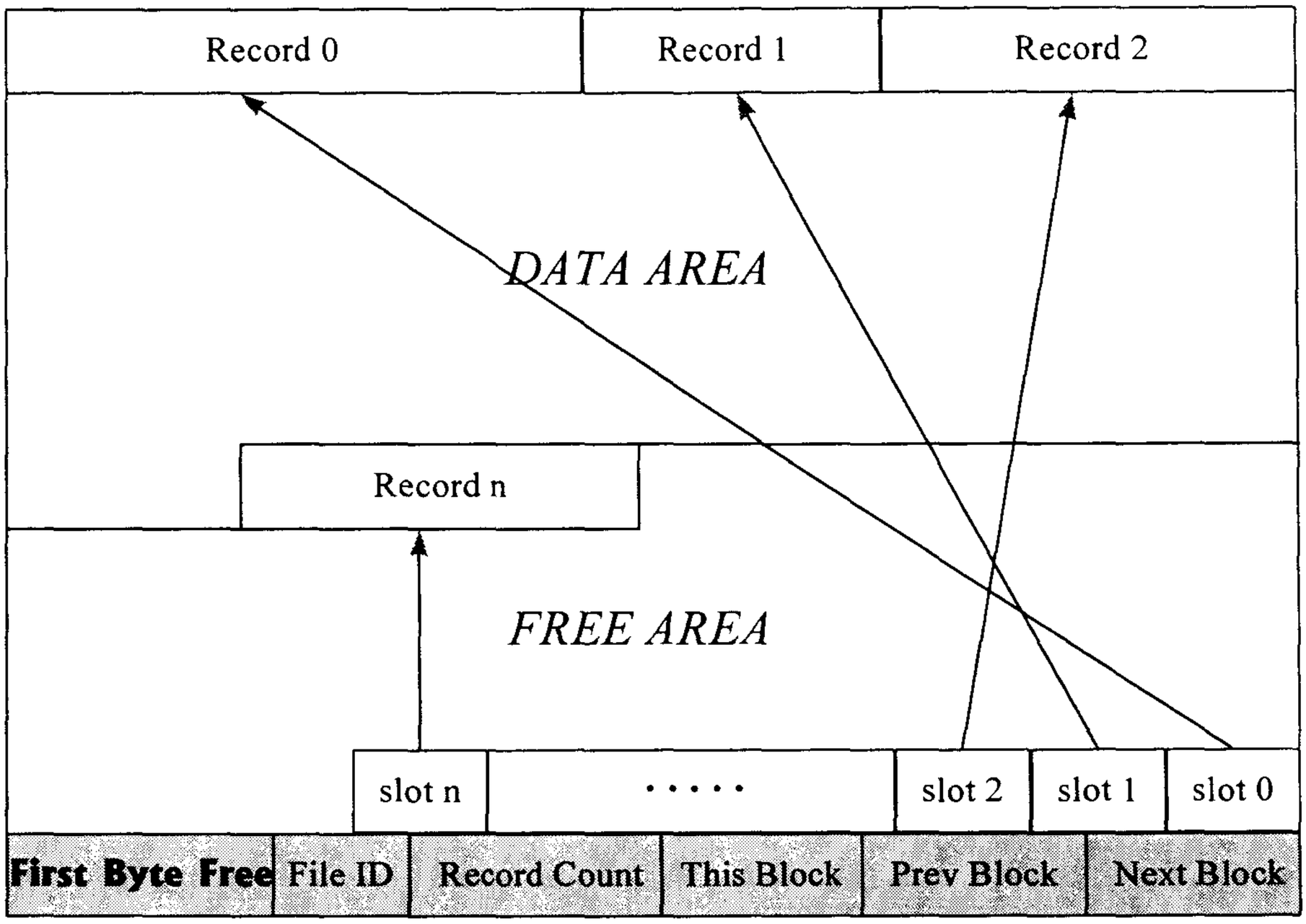
일반적으로 사용자의 관점에서 자료 접근은 문서와 같이 논리적인 저장 구조 단위로 이루어는데 반해, 앞에서 기술된 입출력 관리기와 버퍼 관리기는 블럭을 기본 단위로 하여 정보를 접근한다. 따라서 문서와 같은 저장 구조를 하위의 블럭 구조로 사상할 수 있는 방법이 필요하다. 본 절의 레코드 관리와 다음 절의 대용량 객체 관리기는 이를 위해 구현되었다. 이들 관리기는 사용자 관점에서 저장 구조를 일련의 바이트 열(byte stream)로 취급하여 하위의 블럭으로 사상한다. 그 이름에서 알 수 있듯이 레코드 관리기는 주로 작은 크기의 객체를 지원하기 위해 구현되었으며, 대용량 객체 관리기는 멀티미디어 데이터와 같은 대용량의 객체의 저장 관리를 위해 구현되었다.

2.4.1 레코드 관리기의 설계

본 절에서의 레코드 관리기는 주로 블럭보다 작은 크기의 레코드에 대한 관리를 지원한다. 이 레코드 관리기는 데이터베이스 관리 시스템의 저장 시스템에서와 같이 작은 크기의 정형화된 자료들을 저장하는데 이용될 수 있다. 그 뿐만 아니라 가변 길이의 레코드들을 관리하는데도 사용할 수 있도록 구현되었다. 본 과제에서 구현한 저장 시스템은 데이터베이스의 레코드를 식별하기 위해 각 레코드마다 식별자(Record Identifier, RID)를 부여한다. 레코드 식별자는 데이터베이스 전체에서 유일하도록 시스템에 의해 자동으로 부여되며, 볼륨 식별자, 블럭 식별자, 블럭 안에서의 슬롯 번호로 구성된다. 다음은 레코드 식별자의 구조체를 C 코드로 표현한 것이다.

```
typedef struct {
        TWO     volid;          // 볼륨 식별자
        FOUR    blockid;        // 블럭 식별자
        TWO     slotnum;        // 슬롯 번호
    } RID;
```

레코드 관리기는 순차 화일에 레코드를 추가하거나 혹은 저장된 레코드를 판독, 변경, 추출하기 위한 기능들을 수행하며, 이러한 기능들은 모두 레코드 식별자의 블럭 식별자를 이용하여 레코드가 속한 블럭을 버퍼 블럭으로 읽어들인 후 그 버퍼 블럭 내에서 수행된다. [그림 2.7]은 사용자의 논리적인 레코드들이 블럭으로 사상되는 모습을 보여준다.



[그림 2.7] 블럭에서의 레코드의 사상 구조

블럭은 볼륨에서 연속적으로 할당된 일련의 바이트들로, 같은 화일에서의 전방 블럭과 후방 블럭에 대한 블럭 식별자를 갖고 있다. 이것은 같은 화일에 속하는 블럭들의 이중 연결 리스트(doubly linked list)를 구현함으로써 그 화일 내에서 레코드들을 순차적으로 접근할 수 있도록 하기 위한 것이다. 화일 식별자와 블럭 식별자는 블럭 자신의 소속 정보를 기술함으로써 블럭의 적절성 확인을 용이하게 한다. 아울러 블럭의 제어 정보는 현재 블럭에서 유효한 슬롯들의 수(즉, 레코드들의 수)와 가용 영역의 크기를 저장하고 있다. 이 슬롯들의 수는 레코드의 삽입과 삭제 연산에 따라 증가하거나 감소한다. 또한 제어 정보의 슬롯은 블럭 내에 저장되어 있는 각 레코드의 시작 위치를 저장하고 있고, 실제 레코드의 내용은 블럭의 윗 부분에 저장된다. 따라서 새로운 레코드가 삽입될 때, 슬롯은 아래에서 위로 증가하며, 실제 레코드는 블럭의 위에서 아래로 저장된다.

레코드 관리기에서 데이터를 관리하기 위한 레코드의 형식은 다음과 같다. 레코드는 처음에 생성될 때 NOT MOVED의 레코드 타입을 갖는다. 만일 레코드가 내용 변경으로 인해 블럭의 원래 위치에 다시 저장할 수 없을 정도로 크기가 증가되면, 그 레코드는 새로운 블럭에 저장되고 NEW HOME의 레코드 타입을 갖는다. 그리고 원래의 위치에 있던 레코드는 MOVED의 타입을 가지며, 새로운 위치로 이동된 레코드의 식별자를 유지한다. 레코드의 종류는 NORMAL, SLICE, CRUMB, LDIDIR로 구분된다. NORMAL은 본 절에서 설명하는 소규모 크기의 레코드를 표시하며, 나머지들은 다음에 설명될 긴 자료 항목에서 사용된다.

Record Type	Record Kind	Record Length	Actual Data

2.4.2 레코드 관리기의 구현

```
rec_appendrecord( filenum, recaddr, reclen, newrid )
int       filenum;
char      *recaddr;
int       reclen ;
RID       *newrid ;
```

filenum에 의해 지시되는 화일에 하나의 레코드를 추가한다. 그리고 newrid를 통해 추가된 레코드의 식별자를 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.

나) 화일의 마지막 블럭에 레코드 삽입을 시도한다.

다) 위의 과정이 실패하면, 화일에 새로운 블럭을 할당하고, 할당된 블럭에 레코드를 추가한다.

라) 화일상태(cardinality, status 등)가 변경되었음을 설정하고, 함수를 종료한다.

rec_insertrecord(filenum, recaddr, reclen, nearrid, newrid)
int filenum;
char *recaddr;
int reclen;
RID *nearrid;
RID *newrid;

filenum에 의해 지시되는 화일에 레코드를 삽입하고, newrid를 통해 삽입된 레코드의 식별자를 반환한다. 만일 입력 인자 nearrid의 값이 NULL이면, 레코드를 화일의 마지막에 삽입한다. 그렇지 않고 nearrid의 값이 주어지면, nearrid에 의해 지시되는 레코드 근처에 새로운 레코드를 삽입한다.

가) 함수의 입력 인자들의 값을 확인한다.
나) 입력 인자 nearrid의 값이 NULL이면, 화일의 마지막 위치에 레코드를 삽입하고 함수를 종료한다.
다) 입력 인자 nearrid의 값이 NULL이 아니면, 다음을 수행한다.
라) 레코드 식별자인 nearrid를 통해 레코드를 삽입할 블럭을 찾는다. 그리고 그 블럭에 새로운 레코드를 삽입한다. 이것이 성공하면 함수를 종료하고, 실패하면 다음을 계속한다.
마) 화일에서 후방 블럭을 찾아 그 블럭에서 삽입 연산을 시도한다. 즉, 블럭을 버퍼 관리자를 통해 메모리로 읽어 들인 후 삽입을 시도한다. 이것이 성공하면 화일의 변경 사항들을 설정하고 함수를 종료한다. 그렇지 않고 실패하면 다음을 계속한다.
바) 새로운 블럭을 할당받아 그 블럭에 레코드를 삽입하고, 화일의 변경 사항들을 설정하고 함수를 종료한다.

rec_deleterecord(filenum, ridptr)
int filenum;
RID *ridptr;

filenum와 ridptr에 의해 지시되는 레코드를 삭제한다.

가) 함수의 입력 인자들의 적절성을 검사한다.
나) 삭제할 레코드를 포함하고 있는 블럭을 찾아 ridptr에 의해 지시되는 레코드 슬롯을 삭제한다.
다) 레코드의 삭제로 인해 생기는 블럭내의 빈 공간을 압축한다.
라) 블럭내에 아무런 레코드도 남지 않아 있지 않다면, 그 블럭을 자유 영역으로 반환한다.
마) 만일 삭제할 레코드의 타입이 MOVED라면(즉, 새로운 위치로 이동되었다면), 이동된 위치에서 레코드를 찾아 (나)~(라)를 반복 수행한다.
바) 화일의 변경 사항을 설정하고 함수를 종료한다.

rec_readrecord(filenum, ridptr, recaddr, reclen)
int filenum;
RID *ridptr ;
char *recaddr;
int reclen ;

filenum에 의해 지시되는 화일에서 ridptr에 의해 지시되는 레코드의 내용을 읽어 recaddr에 의해 지시되는 사용자의 메모리 영역으로 복사한다. 그리고 읽어들인 레코드의 길이를 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.
나) 읽어 들일 레코드를 포함하고 있는 블럭을 버퍼 블럭으로 캐쉬하고, 그 버퍼 블럭에서 해당 레코드의 위치를 찾는다.
다) 사용자의 메모리 영역으로 레코드의 내용을 복사한다.

라) 버퍼 블럭을 반환하고 함수를 종료한다.

rec_writerecord(filenum, ridptr, recaddr, reclen)
int filenum;
RID *ridptr ;
char *recaddr;
int reclen ;

recaddr에 의해 지시되는 레코드의 내용을 화일에 기록하고, 기록된 레코드의 크기를 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.
나) 블럭에서 레코드의 위치를 찾는다. 만일 레코드가 다른 위치로 이동되었다면(즉, 레코드 타입이 MOVED라면), 이동된 레코드의 위치를 찾는다.
다) 기록할 레코드를 위한 충분한 가용 영역이 있는지를 확인하여 자리를 마련한다. 그리고 레코드의 내용을 복사한다. 만일 영역이 충분하지 않으면, 다음을 계속한다.
라) 화일의 마지막 블럭에 레코드의 추가를 시도한다.
마) (라)의 작업이 실패하면, 새로운 블럭을 화일의 마지막에 추가하고 그 블럭에 레코드를 기록한다.
바) 레코드의 내용이 새로운 위치로 이동되었음을 표시한다. 즉, 레코드의 타입을 NEWHOME로 설정한다.
사) 이동하기 전의 레코드가 새로운 위치의 레코드를 지시하도록 설정한다.

rec_firstfile(filenum, firstrid)
int filenum;
RID *firstrid;

filenum에 의해 지시되는 화일에서 첫 레코드의 식별자를 firstrid를 통해 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.

나) 화일의 첫번째 블럭을 찾는다.

다) 럭의 첫 슬롯으로부터 시작하여 다음을 만족하는 레코드를 탐색한다.

(조건 1) 슬롯이 EMPTYSLOT이 아님

(조건 2) 레코드 타입이 NEWHOME이 아님

(조건 3) 레코드의 종류가 NORMAL임

라) 함수를 종료한다.

rec_nextfile(filenum, ridptr, nextrid)
int filenum;
RID *ridptr;
RID *nextrid;

ridptr에 의해 지시된 레코드의 뒤에 위치하는 레코드의 식별자를 nextrid를 통해 반환한다. 만일 ridptr이 NULL이면, 화일의 첫번째 레코드의 식별자를 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.

나) ridptr의 레코드 식별자로부터 블럭의 식별자와 슬롯 번호를 얻어낸다. 만일 ridptr의 값이 NULL이면, 첫번째 블럭의 식별자를 구하고, 슬롯 번호를 -1로 설정한다.

다) 블럭 식별자를 통해 블럭을 버퍼로 캐쉬하고, 그 블럭 내에서 슬롯 번호의 위치에 해당하는 슬롯부터 시작하여 다음의 조건을 만족하는 레코드를 찾는다.

(조건 1) 슬롯이 EMPTYSLOT이 아님

(조건 2) 레코드 타입이 NEWHOME이 아님

(조건 3) 레코드의 종류가 NORMAL임

라) 만일 위의 작업이 실패하면, 현재 블럭의 후방 블럭을 읽어 들이고 슬롯 번호를 -1로 설정하여 위의 작업을 반복 수행한다.

rec_prevfile(filenum, ridptr, prevrid)
int filenum;
RID *ridptr;
RID *prevrid;

ridptr에 의해 지시된 레코드의 이전 레코드에 대한 식별자를 prevrid를 통해 반환한다. ridptr이 NULL이면, 화일의 마지막 레코드의 식별자를 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.

나) ridptr의 레코드 식별자로부터 블럭의 식별자와 슬롯 번호를 얻어낸다. 만일 ridptr의 값이 NULL이면, 마지막 블럭의 식별자를 구하고, 슬롯 번호를 마지막 블럭의 최대 슬롯 번호로 설정한다.

다) 블럭 식별자를 통해 블럭을 버퍼로 캐쉬하고, 그 블럭 내에서 슬롯 번호의 위치에 해당하는 슬롯부터 시작하여 다음의 조건을 만족하는 레코드를 찾는다.

(조건 1) 슬롯이 EMPTYSLOT이 아님

(조건 2) 레코드 타입이 NEWHOME이 아님

(조건 3) 레코드의 종류가 NORMAL임

라) 만일 위의 작업이 실패하면, 현재 블럭의 전방 블럭을 읽어 들이고 슬롯 번호를 최대 슬롯 번호로 설정하여 위의 작업을 반복 수행한다.

rec_lastfile(filenum, lastrid)
int filenum;
RID *lastrid;

화일에서 마지막 레코드의 식별자를 반환한다.

가) 함수의 입력 인자들의 적절성을 확인한다.

나) 화일의 마지막 블럭을 찾는다.

다) 블럭의 마지막 슬롯부터 시작하여 다음을 만족하는 레코드를 탐색한다.

(조건 1) 슬롯이 EMPTYSLOT이 아님

(조건 2) 레코드 타입이 NEWHOME이 아님

(조건 3) 레코드의 종류가 NORMAL임

라) 함수를 종료한다.

rec_appendblock(filenum, block)
int filenum;
DATABLOCK *block;

filenum에 의해 지시되는 화일의 블럭 연결 리스트의 마지막에 block에 의해 지시되는 블럭을 추가한다.

가) 함수의 입력 인자들의 적절성을 확인한다.

나) filenum에 의해 지시되는 화일의 마지막 블럭의 식별자를 계산한다. 만일 화일내에 블럭이 존재하지 않으면, 추가할 블럭을 화일의 첫번째 블럭으로 설정한다.

다) 입출력 관리기의 io_allocblocks()를 이용하여 새로운 블럭을 할당받는다.

라) 할당 받은 블럭에 블럭 관리를 위한 제어 정보를 설정한다. 그리고 이 블럭을 블럭 연결 리스트에 추가한다.

마) 화일의 상태 변경를 기록한다.

2.4.3 긴 자료 항목 관리

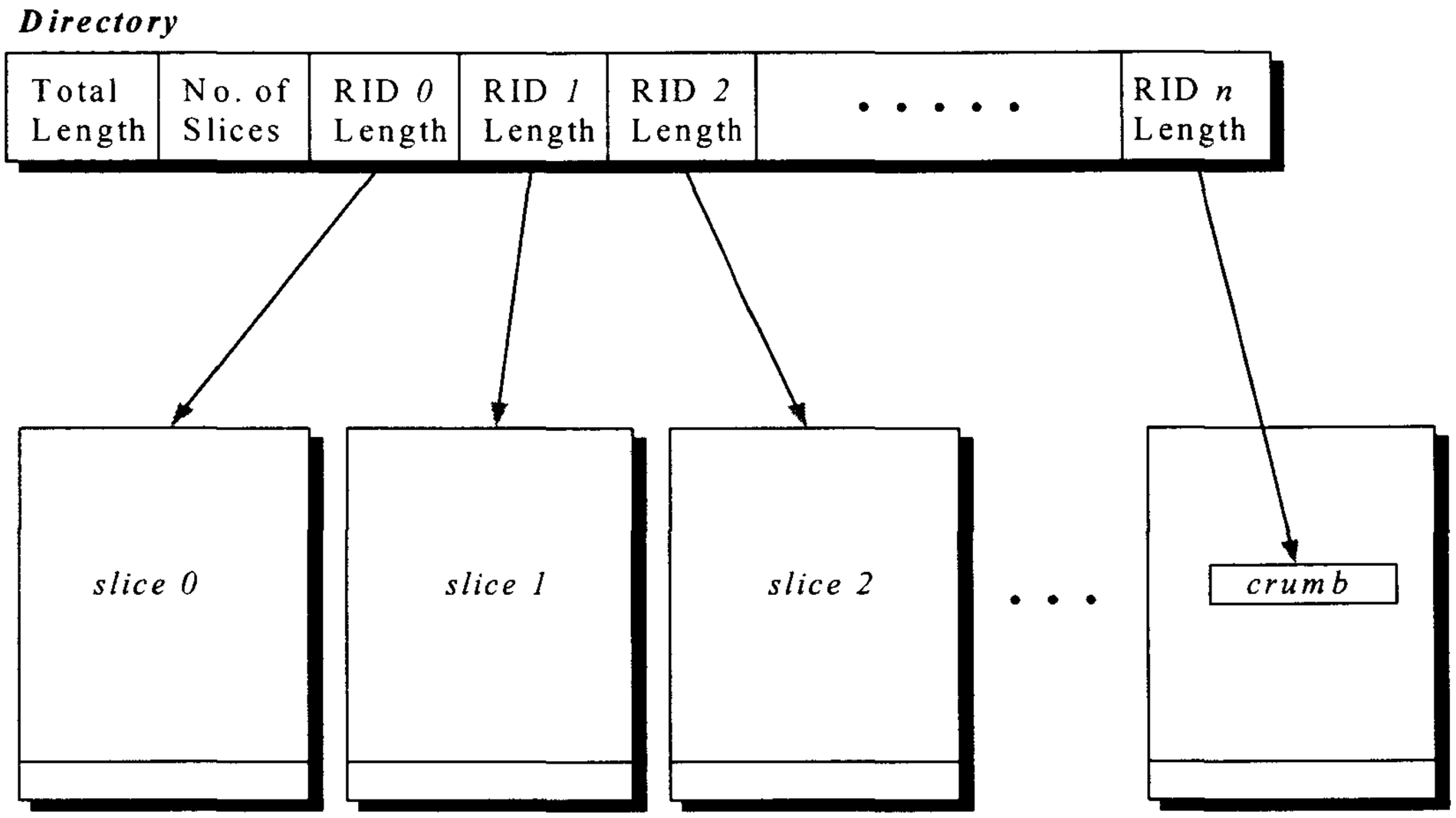
레코드 관리기의 레코드는 정형화된 정보와 가변 길이의 정보를 처리할 수 있도록 구현되었지만, 크기가 블럭의 크기에 의해 제한을 받는다. 그러나 일반적으로 정보 검색 시스템이 대상으로 하는 많은 정보들은 블럭보다 큰 문서들이 많기 때문에 레코드로써 이를 지원하는데는 한계가 있다. 본 절에서 설명하는 긴 자료 항목 관리 모듈은 이러한 문제를 보완하고자 구현되었다.

긴 자료 항목은 정보를 여러 조각들로 분할한 슬라이스(slice)들과 이들 슬

라이스를 관리하기 위한 디렉토리(directory)로 구성된다. [그림 2.8]은 긴 자료 항목을 관리하기 위해 설계된 내부 저장 구조의 모습을 보여주고 있다. 하나의 슬라이스는 블럭내에서 이용 가능한 모든 가용 영역을 할당함으로써 생성되므로 최대 하나의 블럭 크기까지 가능하며, 슬라이스가 생성될 때 블럭에는 단지 하나의 엔트리만이 있게 된다. 긴 자료 항목을 구성하는 슬라이스들 중에 마지막 슬라이스는 대부분의 경우 작게 되는데, 이와 같이 크기가 작은 슬라이스를 crumb라고 부른다. 따라서 crumb는 작은 크기의 레코드로써 구현되며, 다른 crumb들과 블럭을 공유하도록 함으로써 블럭의 저장 공간을 효율적으로 사용하도록 설계 구현되었다.

슬라이스들의 리스트를 유지하는 디렉토리는 레코드로서 구현되었으며, 각 슬라이스의 레코드 식별자와 길이를 저장한다. 디렉토리 구조를 이렇게 설계함으로써 긴 자료 항목의 임의의 위치에 효율적으로 접근할 수 있다. 블럭 크기를 4K 바이트로 정의하여 사용하고 있는 시스템에서 긴 자료 항목의 최대 크기를 계산하면 다음과 같다. 이러한 시스템에서 긴 자료 항목의 디렉토리는 대략 400개의 레코드 식별자와 길이 정보를 가지며, 슬라이스당 약 4K 바이트의 정보를 저장할 수 있으므로, 긴 자료 항목의 최대 크기는 거의 1.6M 바이트에 이른다.

본 모듈에서는 이러한 긴 자료 항목을 생성하거나 임의의 위치에서 내용을 판독, 기록, 삽입, 삭제하기 위한 연산을 제공한다. 아울러 긴 자료 항목의 일부분으로부터 자료를 삭제하였을 때 생기는 빈 공간들을 제거하기 위한 압축 인터페이스들도 지원하고 있다.



[그림 2.8] 긴 자료 항목의 내부 저장 구조

2.4.4 긴 자료 항목 관리의 세부 구현

ldi_createlong(filenum, ridptr)
int filenum;
RID *ridptr;

filenum에 의해 지시되는 화일에 아무런 정보도 저장하고 있지 않는 새로운 긴 자료 항목을 생성하고, ridptr를 통해 생성된 긴 자료 항목 디렉토리의 레코드 식별자를 반환한다.

가) filenum에 의해 지시되는 화일에 대한 접근 권한을 확인한다.

나) 긴 자료 항목의 관리를 위한 제어 정보를 설정한다. 즉, 긴 자료 항목의 total length와 slice count의 값을 모두 0으로 설정한다.

다) 긴 자료 항목의 디렉토리를 레코드 형식으로 초기화한다.

라) st_appendrecord() 함수를 호출하여 긴 자료 항목을 filenum에 의해 지시되는 화일에 추가한다.

ldi_destroylong(filenum, ridptr)
int filenum;
RID *ridptr;

filenum에 의해 지시되는 화일로부터 ridptr에 의해 지시되는 긴 자료 항목을 삭제한다.

가) filenum에 의해 지시되는 화일에 대한 접근 권한을 확인한다.
나) ridptr에 의해 지시되는 긴 자료 항목을 디렉토리를 버퍼로 캐쉬한다.
다) 디렉토리에 의해 지시되는 모든 슬라이스들을 자유 영역으로 반환한다.
라) 디렉토리를 포함하고 있는 버퍼 블럭을 반환한다.
마) st_deleterecord() 함수를 호출하여 디렉토리를 화일로부터 제거한다.

ldi_readframe(filenum, ridptr, offset, recaddr, length)
int filenum;
RID *ridptr;
int offset;
char *recaddr;
int length;

긴 자료 항목의 임의의 위치에서 일정량의 바이트 스트림을 사용자의 주소 공간으로 읽어 들인다. 그리고 읽어 들인 바이트 스트림의 길이를 반환한다. 여기에서 filenum은 화일을, ridptr은 긴 자료 항목의 식별자를 나타내고, offset과 length는 긴 자료 항목에서 읽어 들일 바이트 스트림의 위치와 길이를 표시한다.

가) filenum에 의해 지시되는 화일에 대한 접근 권한을 확인한다.
나) 긴 자료 항목의 디렉토리를 버퍼로 캐쉬한다.

다) offset을 통해 읽어 들일 바이트 스트림이 속해 있는 첫번째 슬라이스와 그 슬라이스내에서 바이트 스트림의 시작 위치를 찾는다.

라) 그 위치로부터 length 만큼의 정보를 recaddr에 의해 지시되는 사용자의 메모리 주소 공간으로 복사한다. 만일 현재의 슬라이스에서 정보를 모두 읽어 들이지 못하였다면, 반복적으로 다음의 슬라이스들을 찾아 나머지의 정보들을 읽어 들인다.

마) 디렉토리를 포함하고 있는 버퍼 블럭을 반환한다.

ldi_writeframe(filenum, ridptr, offset, recaddr, length)
int filenum;
RID *ridptr;
int offset;
char *recaddr;
int length;

recaddr에 의해 지시되는 사용자 메모리 주소 공간의 정보를 긴 자료 항목의 지정된 위치로 복사한다. 여기서 filenum과 ridptr은 각각 화일과 긴 자료 항목을 지시하며, offset과 length는 정보를 기록할 긴 자료 항목내에서의 위치와 길이를 표시한다.

가) filenum에 의해 지시되는 화일에 대한 접근 권한을 확인한다.

나) 긴 자료 항목의 디렉토리를 버퍼로 캐쉬한다.

다) recaddr에 의해 지시되는 사용자 주소 공간의 정보를 기록할 첫번째 슬라이스와 그 슬라이스에서의 시작 위치를 찾는다.

라) 사용자 영역의 정보를 그 시작 위치로 복사한다. 만일 현재의 슬라이스에 정보를 모두 기록할 수 없다면, 반복적으로 다음의 슬라이스들을 찾아 나머지의 정보들을 기록한다.

마) 긴 자료 항목의 디렉토리를 포함하고 있는 버퍼 블럭을 반환한다.

ldi_insertframe(filenum, ridptr, offset, recaddr, length)
int filenum;
RID *ridptr;
int offset;
char *recaddr;
int length;

recaddr에 의해 지시되는 사용자 메모리 주소 공간의 정보를 긴 자료 항목의 지정된 위치로 삽입한다. 여기서 filenum과 ridptr은 각각 화일과 긴 자료 항목을 지시하며, offset과 length는 정보를 삽입할 긴 자료 항목내에서의 위치와 길이를 표시한다.

가) filenum에 의해 지시되는 화일에 대한 접근 권한을 확인한다.
나) 긴 자료 항목의 디렉토리를 버퍼로 캐쉬한다.
다) 만일 긴 자료 항목이 새로 생성된 것이고, 삽입할 정보의 크기가 블럭보다 작은 경우, 정보를 crumb 레코드로 만들어 화일에 추가한 후, 긴 자료 항목의 디렉토리가 이 crumb를 지시하도록 한다. 그리고 기타 긴 자료 항목 관리를 위한 정보들을 변경하고, 함수를 종료한다.
라) 그 외의 경우 먼저 offset을 사용하여 정보를 삽입할 시작 위치를 찾는다.
마) 만일 시작 위치가 마지막 슬라이스이고, "삽입할 사용자의 데이타의 크기 + 마지막 슬라이스의 크기"가 MAXCRUMBSIZE보다 작다면, 사용자의 정보와 마지막 슬라이스의 내용을 결합하여 crumb를 만든다. 그리고 함수를 종료한다.
바) 시작 위치가 마지막 슬라이스가 아니고 삽입하려는 슬라이스가 crumb라면, 이 crumb를 슬라이스로 확장한다.
사) 정보를 삽입하기 위해 필요한 새로운 슬라이스 블럭들을 할당한다.
아) 사용자의 데이타를 삽입할 시작 위치를 슬라이스 i, 새로 할당된 슬라이스들 중에서 마지막 슬라이스를 j라고 할 때, 사용자의 정보를 i에서 j까지의 슬라이스로 복사한다.
자) 만일 마지막 슬라이스의 크기가 MAXCRUMBSIZE보다 작으면, 이를

crumb로 축소시킨다.

차) 긴 자료 항목을 관리하기 위한 제어 정보를 변경한다.

ldi_deleteframe(filenum, ridptr, offset, length)
int filenum;
RID *ridptr;
int offset;
int length;

filenum과 ridptr에 의해 지시되는 긴 자료 항목의 offset 위치로부터 length 만큼의 정보를 삭제한다.

가) filenum에 의해 지시되는 화일에 대한 접근 권한을 확인한다.

나) 긴 자료 항목의 디렉토리를 버퍼로 캐쉬한다.

다) offset을 통해 삭제할 정보의 시작 위치를 포함한 슬라이스를 찾는다. 이를 슬라이스 i라고 하자.

라) 만일 슬라이스 i가 긴 자료 항목의 마지막 슬라이스이고, 최대 crumb의 크기보다 작다면, 그 슬라이스를 crumb로 축소시키고, length 만큼의 정보를 삭제한다. 그리고 관련된 긴 자료 항목의 제어 정보를 변경한 후 함수를 종료한다.

바) 삭제할 정보의 마지막 위치를 포함한 슬라이스를 찾는다. 이를 슬라이스 j라고 하자.

마) i = j 인 경우, 슬라이스 내에서 뒷 부분의 정보를 이동시켜 앞 쪽의 정보와 연결한다. 삭제를 수행한 후에 슬라이스가 비게 되면, 그 슬라이스를 긴 자료 항목의 슬라이스 리스트로부터 제거한다.

바) i ≠ j 인 경우, 슬라이스 i와 슬라이스 j사이의 모든 슬라이스들을 제거한다. 그리고 가능한 경우 슬라이스 i와 슬라이스 j를 하나의 슬라이스로 결합한다.

사) 긴 자료 항목의 제어 정보의 내용을 변경하고 함수를 종료한다.

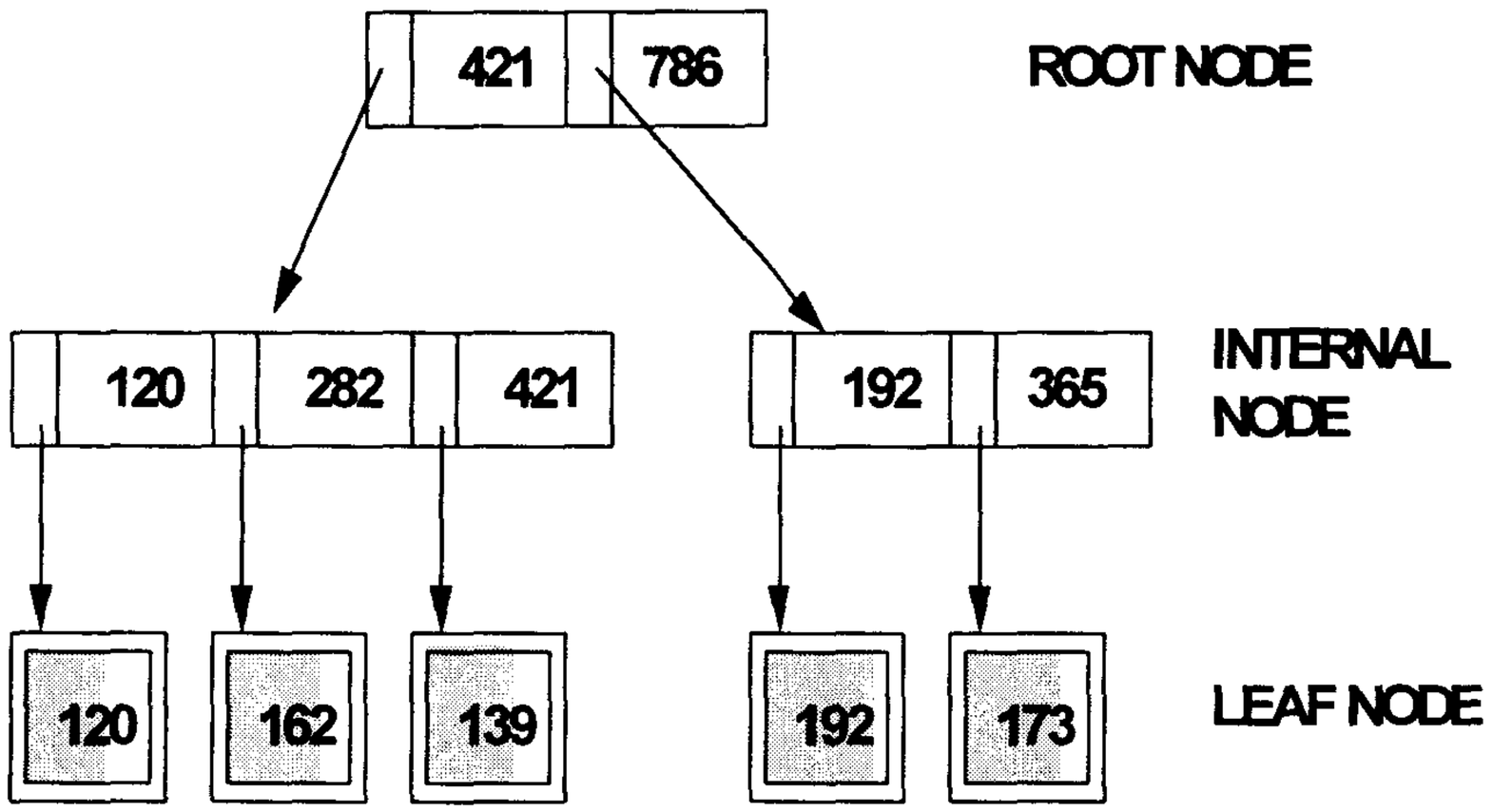
2.5 대용량 객체 관리기

정보 검색 시스템의 응용범위가 텍스트, 이미지, 그래픽, 오디오등과 같은 다양한 대용량 객체로 급속히 확대되고 있다. 이러한 다양한 형태의 대용량 객체를 효율적으로 저장하고 관리하기 위하여 다음과 같이 대용량 객체 관리기를 설계한다.

본 시스템에서는 블롭(BLOB : Binary Large Object)이라는 개념을 사용하여 대용량 객체를 지원한다. 블롭은 가변적인 크기를 갖는 자료 객체로서 이론적으로 보통 2^{31}바이트까지 가능하나, 실제로는 시스템의 물리적인 크기에 제한을 받는 대용량 객체를 말한다. 외부로부터 부분적인 갱신의 요구에 동적으로 대치할 수 있도록, EXODUS에서 사용한 객체관리 기법을 이용하여 블롭 데이타를 지원한다.

2.5.1 대용량 객체 관리기의 설계

블롭의 상태가 가변적인 데이타를 저장하고 처리하기 위해, EXODUS에서 사용한 방법을 이용하여 블롭 데이타를 지원한다. 블롭의 저장 방법은 [그림 2.9]와 같으며, 각 내부 노드는 (Size, BlobID) 의 순서적 배열로 구성되어 있는데, i번째 Size값은 첫번째부터 i번째까지의 자식 노드(BlobID)로부터 접근할 수 있는 데이타의 크기를 나타낸다. 루트 노드의 맨 오른쪽 Size는 전체 블롭 데이타의 크기를 나타낸다. 이와 같이 내부 노드를 구성함으로써 우리는 원하는 블롭 데이타의 특정 위치를 이진 검색을 사용하여 찾아갈 수 있다. 실제 블롭 데이타들은 리프 노드에 저장된다. 이들 내부 노드와 리프 노드들이 블롭 화일에 저장된다. 하나의 블롭 화일은 여러 개의 블롭 데이타들을 저장할 수 있다.



[그림 2.9] EXODUS를 기반으로 한 블롭의 구조

이와 같이 대용량 객체를 관리하는 방법은 구현하기는 비교적 어렵고 복잡하지만, 임의의 위치에서의 삽입, 삭제, 그리고 검색을 효율적으로 할 수 있다는 장점을 가지고 있다.

2.5.2 블롭 구현을 위한 자료 구조

본 설계에서는 INTERNALNODE와 LEAFNODE의 두 가지 새로운 레코드 타입과 자료 구조가 사용되며, 그 구조는 [그림 2.10]과 같다. INTERNALNODE와 LEAFNODE는 모두 레코드가 가질 수 있는 최대 크기를 가진다. 이 2개의 새로운 레코드 타입은 레코드 관리기에서 지원하는 레코드 구조에 변경을 가하지 않고, 몇가지 제어 정보를 레코드의 실제 데이타가 들어가는 곳에 첨가한 것이다.

```
typedef   struct   {
      int       type;              // type of record
      int       kind;              // kind of record
      int       size;              // size of record
      int       num;               // the number of the pairs in the data field
      char    data[NUM_OF_PAIR]; // array of pairs

} INTERNALNODE;
```

```
typedef   struct   {
      int       type;              // type of record
      int       kind;              // kind of record
      int       size;              // size of record
      int       length;            // length of actual BLOB
      char    data[MAX_DATA_SIZE];   // actual BLOB data
} LEAFNODE;
```

```
typedef   struct {
      int       size;        //  이 포인터와 이 포인터의 왼쪽에
                             // 있는 포인터들에 의해 가리켜지는
                             // 블롭 레코드에 있는 데이타의 크기
      RID     recordID; //  블롭 레코드의 RID
} PAIR;
```

[그림 2.10] 동적 블롭을 위한 자료 구조

2.5.3 대용량 객체 관리기의 구현

1) 블롭 생성

```
bl_make(Filenum, Root)
int     Filenum;        // 화일 번호
BID     *Root;          // 블롭의 루트 노드
```

Size = 0, ReftChild = NULL 인 Root 트리를 생성한 다음, 그 BID 를 복귀.
데이타 페이지 크기의 레코드를 하나 생성하고, 그 BID(Blob ID)를 반환하는 단순한 작업으로 블롭을 생성한다. 이후에 삽입이 계속되어 데이타 페이지의 크기를 넘어선다면, 그때 데이타 페이지에 분열이 일어나며, 루트 페이지와 내부 페이지가 생성되어 링크들이 형성된다.

2) 블롭 삭제

```
bl_remove(Filenum, Root)
int     Filenum;        // 화일 번호
BID     *Root;          // 블롭의 루트 노드
```

루트 노드에 딸려 있는 모든 내부 노드들을 읽어 들인 다음, 이 내부 노드들에 연결되어 있는 모든 리프 노드들을 삭제한 다음, 내부 노드들을 삭제하고, 최종적으로 루트 노드를 삭제함으로써, 블롭을 화일내에서 제거한다.

3) 블롭의 임의 위치에서의 데이타 읽기

```
bl_read(Filenum, Root, Offset, RecAddr, Len)
int       Filenum;          // 화일 번호
BID       *Root;            // 블롭의 루트 노드
int       Offset;           // 읽기가 시작되는 위치
char      *RecAddr;         // 읽어 들인 데이타를 저장할 버퍼
int       Len;              // 읽혀질 길이
```

블롭 내의 임의의 위치에서 부터 원하는 크기만큼, 데이타를 읽어 들이는 연산이다. 주어진 시작위치(Offset)를 포함하는 리프 노드를 접근한 다음, 원하는 데이타를 요구한 응용 프로그램 내의 버퍼로 이동시킨다. 첫번째 리프노드를 데이타 버퍼로 이동시킨 후, 다음 리프노드를 접근하기 위해 첫번째 리프노드를 찾아온 경로를 스택상에 보관해 놓아야 한다. 다음은 블롭 화일내에 있는 한 블롭 데이타의 일부분을 읽어들이는 알고리즘이다.

C[i] : i 번째 (Size, BlobID) 쌍의 Size 값

P[i] : i 번째 (Size, BlobID) 쌍의 BlobID 값

(1) c[i] = 0, start = offset ;

(2) 루트 페이지를 읽은 다음 그 페이지를 P라 한다.

(3) While (P가 리프 노드가 아니다) do

가) P의 BID 를 스택에 저장한다.

나) start <= C[i]인 가장 작은 C[i]를 찾기 위해 P를 이진 검색을 한다.

다) start = start - C[i-1]

라) P[i]가 가리키는 페이지를 읽어 들이고 , 그 페이지를 P라 부른다.

(4) 리프에 도달하면, 원하는 첫번째 바이트는 페이지 P에 위치한다.

(5) 나머지 N 바이트를 얻기 위하여, 2) 에서 유지되었던 포인터의 스택을 사용하여 트리를 순회하면 된다.

4) 블롭의 임의 위치에서의 데이타 치환

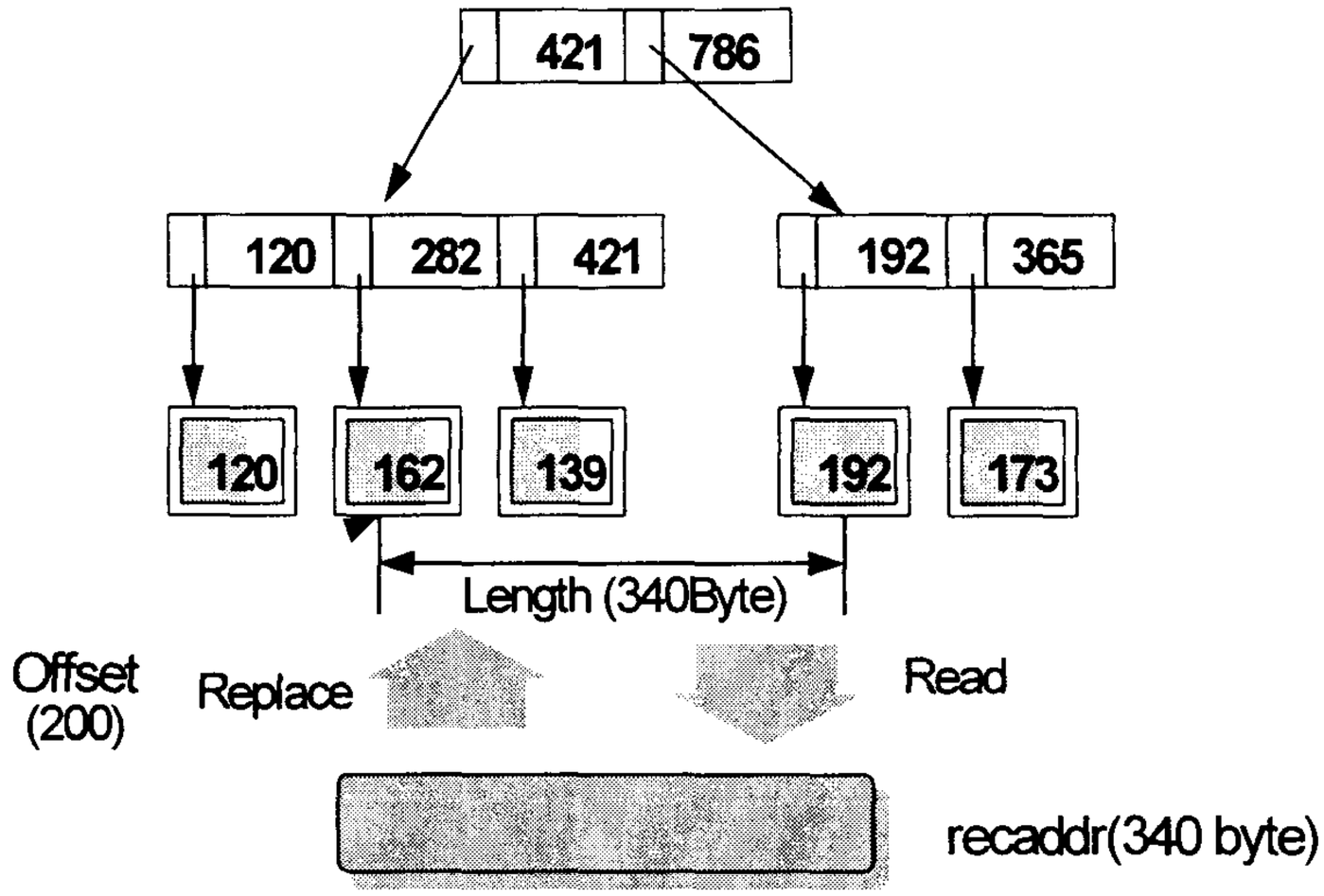
```
bl_write(Filenum, Root, Offset, RecAddr, Len)
int       Filenum;          // 화일 번호
RID       *Root;            // 블롭의 루트 노드
int       Offset;           // 쓰기가 시작되는 위치
char      *RecAddr;         // 쓰여질 데이타를 담고있는 버퍼
int       Len;              // 쓰여질 길이
```

동적 블롭 내의 임의의 위치에서 부터 원하는 데이타를 새로운 내용으로 치환하는 연산이다. 이 연산은 읽기 연산과 동일한 메카니즘을 사용하여 원하는 데이타에 접근한 다음, 그 내용을 새로운 내용으로 치환하고, 다시 스토리지로 저장하여 주어진 연산을 수행한다. 단지 기존의 데이타를 치환하는 것이므로, 루트 노드나 내부 노드의 내용은 변화 되지 않고, 리프 노드의 데이타만 변경되게 된다. 다음은 블롭 화일내의 한 블롭 데이타의 내용을 치환하는 알고리즘이다.

(1) 치환할 시작 지점을 갖고 있는 리프 노드에 도달할 때까지 블롭 트리를 순회하여 그 리프 노드를 찾고, 그 노드를 L이라 부르자.

(2) L에 도달했을 때, 치환을 시작할 위치에서 부터 RecAddr가 가리키고 있는 메모리에 있는 Len바이트를 L의 내용과 치환한다.

(3) 만약, L에 Len 바이트를 전부 치환할 수 없다면, 트리를 순회할 때, 저장된 정보를 사용하여 다음 리프 노드들로 이동하면서, 그곳들에 나머지 데이타를 치환한다.



[그림 2.11] 블롭 내의 데이타 읽기와 치환

5) 블롭에 데이타 삽입

```
int   bl_insert(Filenum, Root, Offset, RecAddr, Len)
int       Filenum;          // 화일 번호
RID       *Root;            // 블롭의 루트 노드
int       Offset;           // 삽입이 시작되는 위치
char      *RecAddr;         // 삽입될 데이타를 담고 있는 버퍼
int       Len;              // 삽입될 길이
```

이 연산은 블롭의 임의의 위치에서부터 원하는 크기의 데이타를 삽입시키는 연산이다. 삽입으로 인해 리프 노드의 분열이 일어날 수 있으며, 이 분열은 내부 노드와 루트 노드의 내용을 변화 시킬 것이다. [그림 2.12-a]는 삽입 연산이 일어나기 전의 블롭의 상태를 보여 주고 있다. 이 블롭에 120바이트의 데이타를 500바이트 위치에 삽입을 시도하면, 오버플로우가 발생하게 되고, 이에 따라 새로운 리프 노드가 생성되어야 한다. 생성된 리프노드와 형제 노드들에 데이타를 균등하게 분배한 다음, 내부 노드와 루트 노드를 변경 시켜줌

으로써, 삽입 연산을 수행하게 된다. 이 삽입 연산이 수행된 결과는 [그림 2.12-b]에서 보는 바와 같이, 새로운 리프 노드가 생성되어, 데이타의 재분배가 일어났음을 알 수 있으며, 그로 인해 변경된 사항이 내부 노드와 루트 노드의 내용을 변화시켰음을 보여 준다. 삽입 연산은 한 블롭 데이타의 위치 Offset에서부터 연속적으로 Len바이트 삽입을 지원한다.

(1) 삽입의 시작점을 갖고 있는 리프에 도달할 때까지 블롭 트리를 순회한다. 트리가 검색될 때 삽입될 바이트 수를 반영하기 위하여 그 노드에 있는 (Size, BlobID) 쌍의 Size값을 변경하고 스택에 검색 경로를 저장한다.

(2) 데이타가 삽입될 리프를 L이라 부르자. L에 도달했을 때, 그곳에 Len바이트 삽입을 시도한다. 만약 오버플로우가 일어나지 않으면, 삽입은 행해진 것이고, 내부 노드의 count는 (1)에서 처럼 갱신될 것이다.

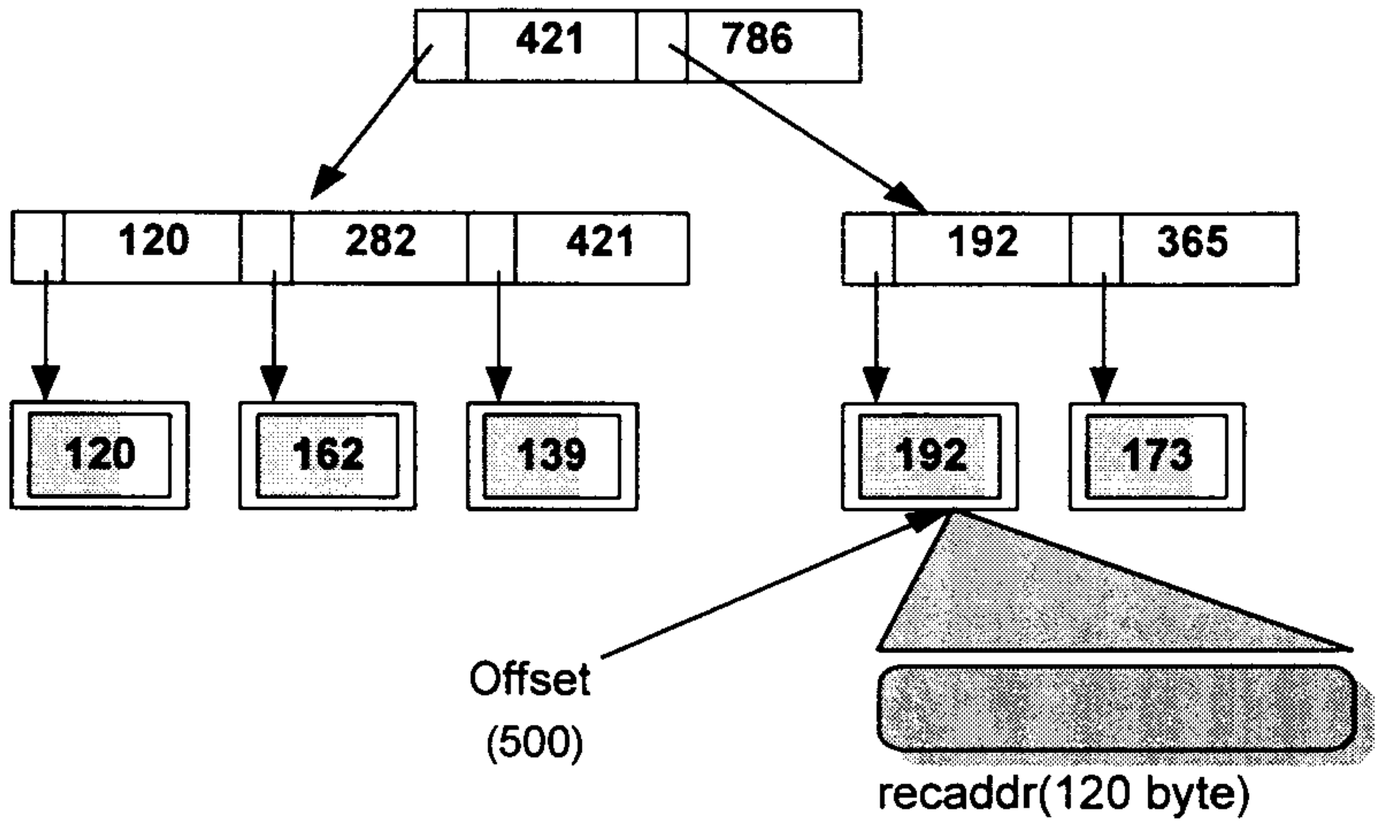
(3) 만약 오버플로우가 발생하면 다음과 같이 진행 된다.

가) L 의 부모 노드에 있는 (Size, BlobID) 쌍들의 Size정보를 조사하여 가장 많은 자유 공간을 가진 L의 왼쪽이나 오른쪽 이웃을 M이라 하고, B 를 리프 노드 L에 저장할 수 있는 바이트 수라 한다.

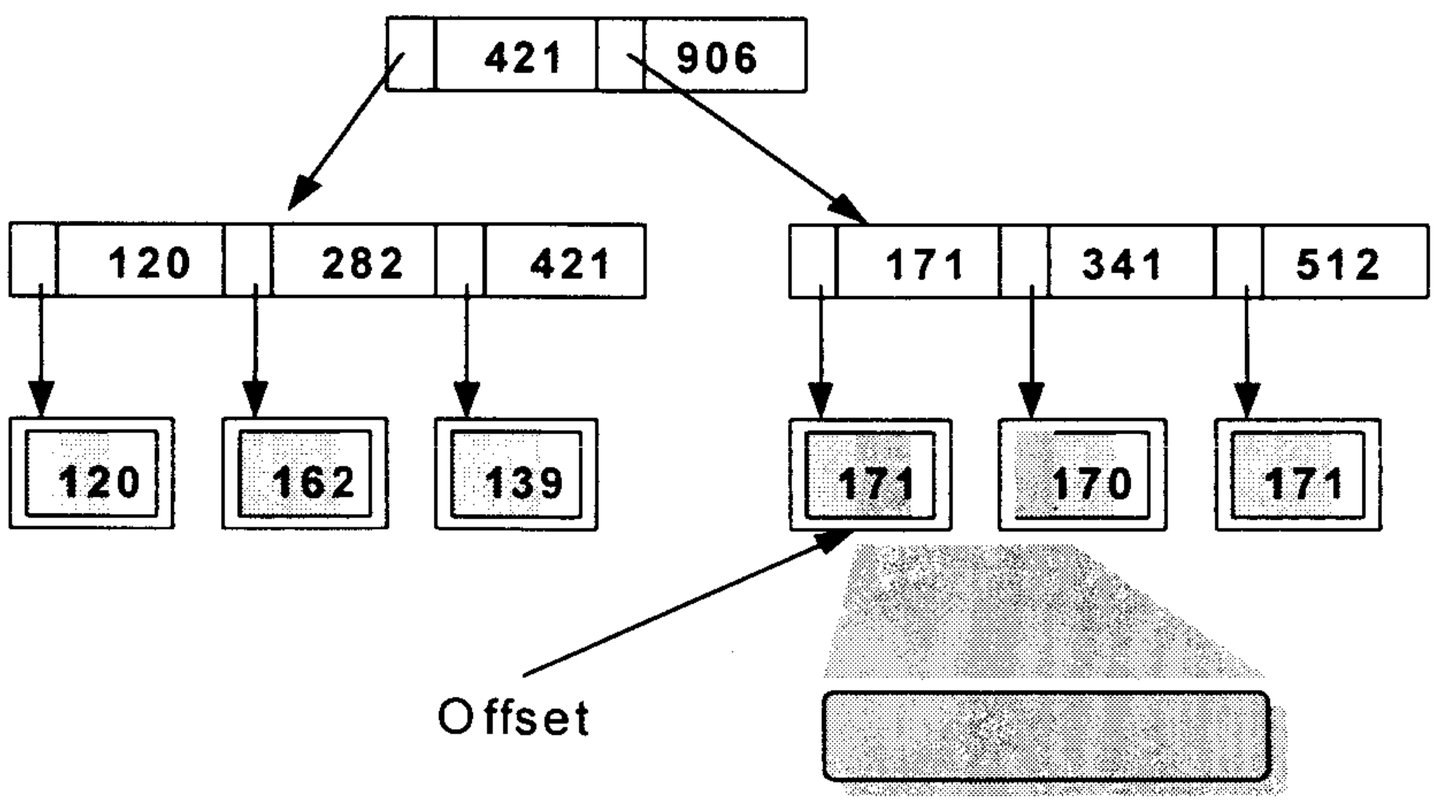
나) 만약, L 과 M 의 자유 공간을 합한 것이 (Len) modulo B 바이트의 데이타를 저장하기에 충분하다면, 이 2개의 노드 L 과 M, 그리고 [N/B] 개의 새로 할당되는 리프 노드들에 Len 바이트의 새로운 데이타와 L과 M의 이전 내용을 균등하게 배분한다.

다) 만약, L 과 M의 자유 공간을 합한 것이 (len) modulo B 바이트의 데이타를 저장하기에 충분치 못하다면, L 로 부터의 오버플로우를 저장하기에 충분할 만큼의 리프 노드를 구성한 다음, L에 저장된 이전 내용과 삽입될 Len바이트를 L과 새롭게 구성된 리프 노드들에 배분한다.

(4) 이 새롭게 할당된 노드들에 대한 (Size, BlobID) 정보를 트리의 위로 전파 한다. 만약, 내부 노드가 오버플로우 되면, 리프 노드가 오버플로우 되었을 때, 처리하는 방법과 동일한 방식으로 처리한다.



(a) 블롭의 임의의 위치에 데이타 삽입전



(b) 블롭의 임의의 위치에 데이타 삽입후

[그림 2.12] 블롭의 임의의 위치에 데이타 삽입

6) 블롭의 데이타 삭제

```
bl_delete(Filenum, Root. Offset. Len)
int     Filenum;        // 화일 번호
RID     *Root;          // 블롭의 루트 노드
int     Offset;         // 삭제가 시작되는 위치
int     Len;            // 삭제될 길이
```

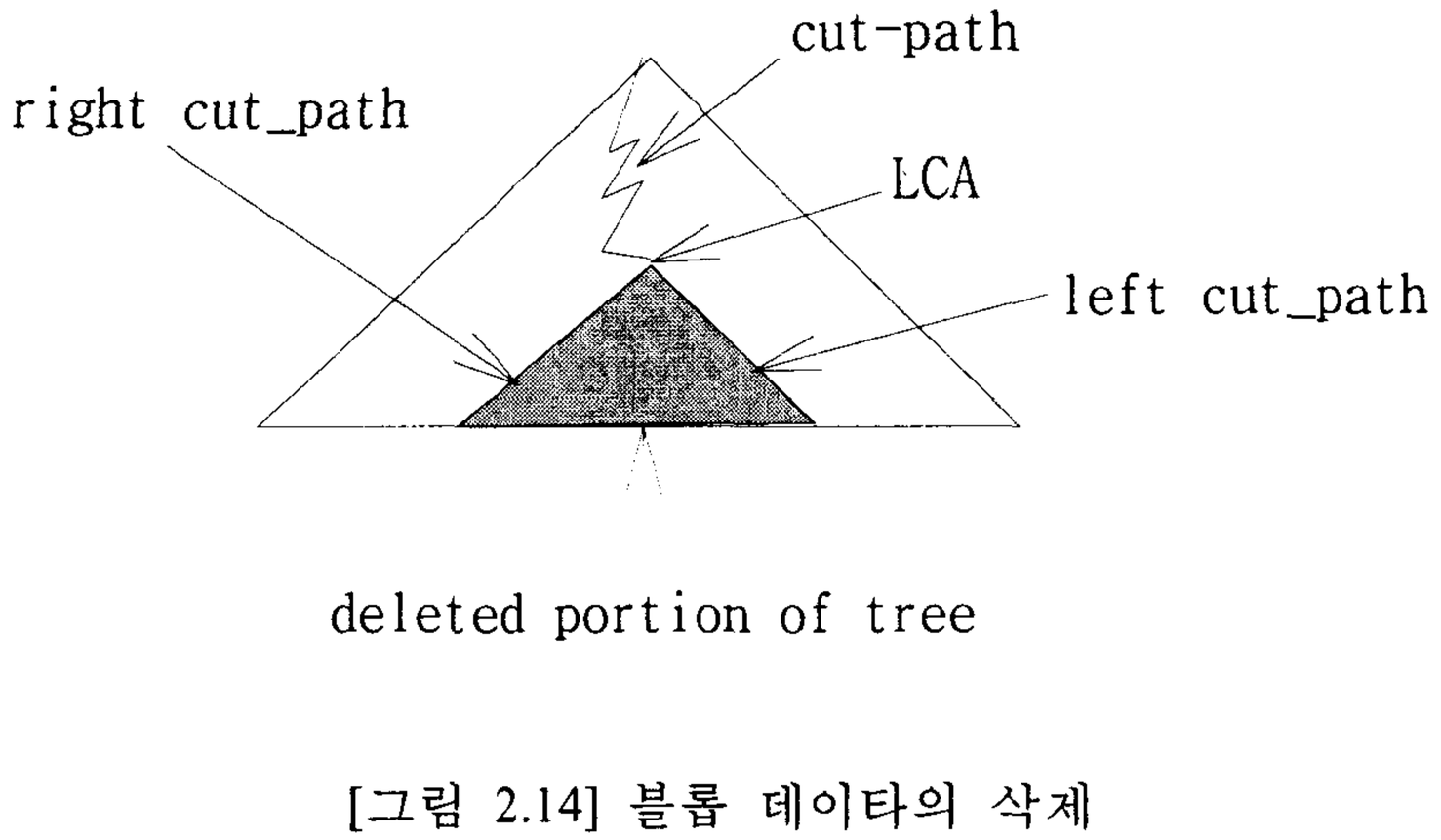
이 연산은 블롭의 임의의 위치에서, 원하는 크기의 데이타를 삭제하는 연산이다. 삭제 연산이 수행되면, 트리의 균형이 깨지게 되는데, 이 균형을 다시 유지 시켜주기 위해, 리프 노드들간의 병합이 발생하며, 이로 인해, 내부 노드들간의 병합이 발생하기도 한다. 새로이 구성된 트리에 대한 정보가 루트 노드와 내부 노드에 반영된다.

[그림 2.13-a]는 블롭에 삭제 연산이 적용되기 전의 상태를 보여 주고 있다. 블롭의 120 Byte 위치부터 150 Byte 만큼이 삭제될 것이다. 이로 인해 왼쪽 리프 노드들은 균형을 잃게 될 것이며, 삭제된 리프 노드들에 대한 합병이 발생하게 된다. 이 합병이 수행됨으로써, 내부 노드와 리프노드들에 대한 정보가 변경되고, 이 변경이 수행된 후의 블롭의 모습은 [그림 2.13-b]와 같다.

삭제 연산은 Offset 에서 시작하는 Len바이트의 삭제를 지원한다. 어떤 블롭 트리에서 중간에 임의의 바이트를 삭제하면 [그림 2.14]의 빗금친 같이 트리의 중간 부분에 빈 공간이 생긴다. 이 빈 공간으로 인해서 트리는 균형을 잃게 되는데 균형을 잃은 트리를 균형 잡힌 트리로 만들어 주는 것이 삭제 알고리즘에서 제일 중요하다.

원하는 바이트를 찾기 위해서 트리를 검색할 때, 검색 경로를 유지하게 된다. 이러한 검색 경로는 삭제시에 cut_path 를 형성하게 된다. 그림 22에서 cut_path 는 삭제할 데이타를 찾기 위한 왼쪽 cut_path 와 오른쪽 cut_path 로 구성되어 있으며, LCA (lowest common ancestor) 은 경로가 왼쪽과 오른쪽으로 나뉘어지는 곳이다.

삭제된 노드는 엔트리의 수가 적을 때, 그 노드가 완전히 비는 것을 방지하기 위해 주변의 다른 노드와 병합을 하거나 두 노드의 엔트리들을 재배열한다. 노드가 병합되거나 재배열되기 위한 조건은 노드가 어느 수준이하의 엔트리를 가지거나(underflow) 그럴 가능성이 있을 경우이다.



[그림 2.14] 블롭 데이타의 삭제

다음 두 조건은 언더플로우인 경우이다.

가) 리프 노드이고, 노드의 1/2 이하가 사용중일 때

나) 내부 노드이고, 노드의 엔트리 수가 Ne 보다 작을 때(루트 노드인 경우는 엔트리의 수가 2보다 작을 때). 여기서 Ne 는 루트가 아닌 내부 노드가 가질 수 있는 최소 엔트리의 수이다.

다음과 같은 조건이 만족되면, 노드는 언더플로우가 될 가능성이 있다.

가) 실제로 언더플로우일 때

나) 내부 노드이고, 엔트리의 수가 정확히 Ne (루트 노드의 경우는 엔트리의 수가 2 개) 이고, cut_path 상에서 하나의 자식 노드가 danger 일 때

다) LCA 노드이고 엔트리의 수가 정확히 Ne + 1 (루트 노드의 경우는 엔트리의 수가 정확히 3개) 이고, cut_path 상의 두개의 자식 노드 모드

가 danger 상태일 때) 실제로 언더플로우가 아니지만 자식 노드가 병합이 될 경우에 언더플로우가 될 수 있기 때문에 이러한 노드는 삭제 알고리즘에서 danger 로 지정한다. 그리고 후에 danger 상태의 노드는 형제 노드와 병합되거나 재배열 한다.

삭제 알고리즘은 원하는 바이트를 트리에서 삭제한 후에 위에서 정의한 danger 상태를 각 검색 경로를 따라 지정하는 삭제 단계와 루트 노드부터 시작해서 리프 노드까지 트리를 다시 균형화 시키는 균형 단계로 분류된다.

(1) 삭제 단계

가) 삭제의 left limit과 right limit을 찾기 위해 블롭 트리를 순회 한다.

나) 삭제의 left limit과 right limit에 대응하는 블롭 트리의 서브트리를 삭제하고, cut_path에 있는 모든 노드의 (Size, BlobID)쌍 정보가 삭제의 결과를 반영하도록 변경한다. 또한 cut_path에 있는 각각의 노드에 대해 (트리가 검색될 때) 그 노드의 BID와 그것이 갖고 있는 자식들의 수를 기록한 데이타 구조인 path를 메모리에 생성한다.

다) 상향식으로 path데이타 구조를 순회하며 danger인 노드를 합병한다.

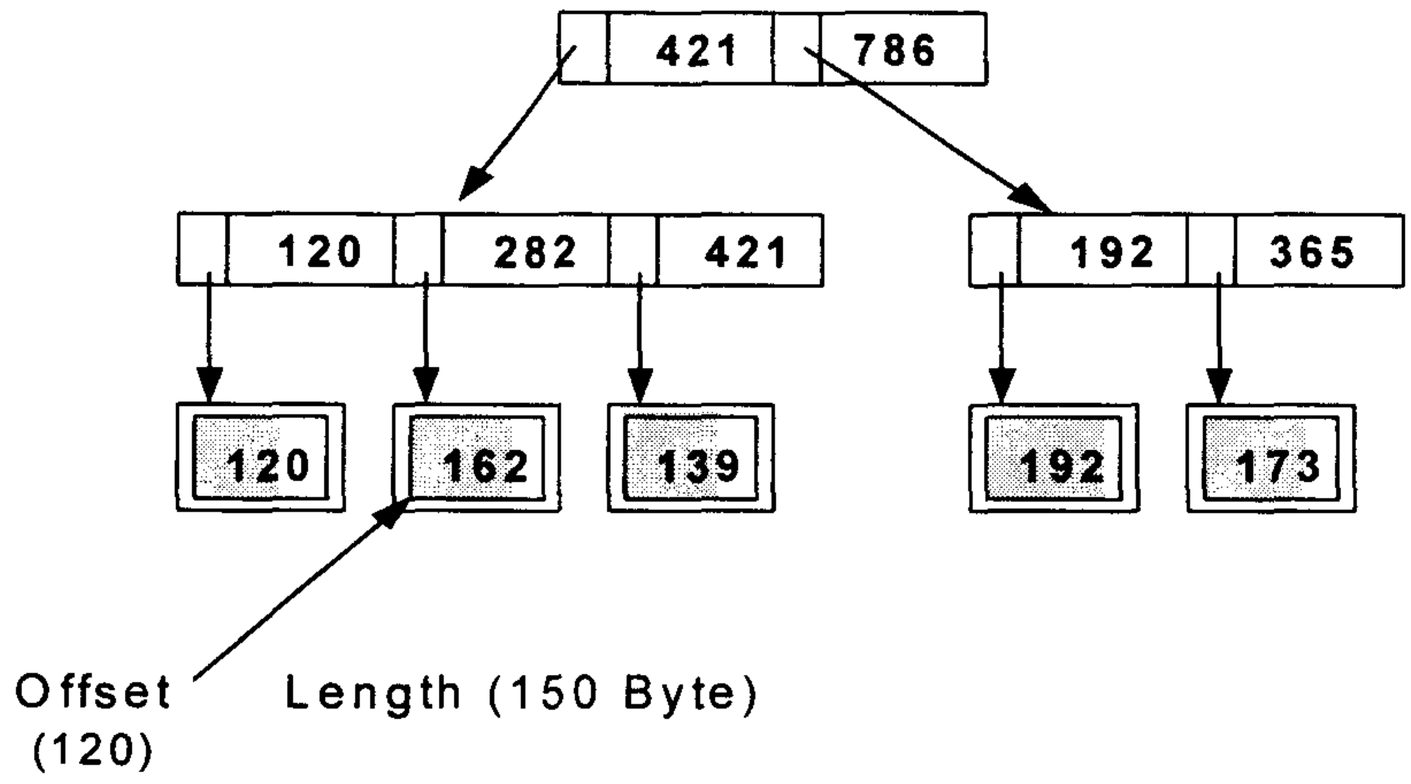
(2) 재균형 단계

가) 만약 루트가 danger가 아니면, 나)로 간다. 만약 루트가 오직 하나의 자식만 갖고 있다면, 이 자식을 새로운 루트로 하고 가)로 간다. 그렇지 않으면, danger인 루트의 자식들을 합병/재분배하고 가)로 간다.

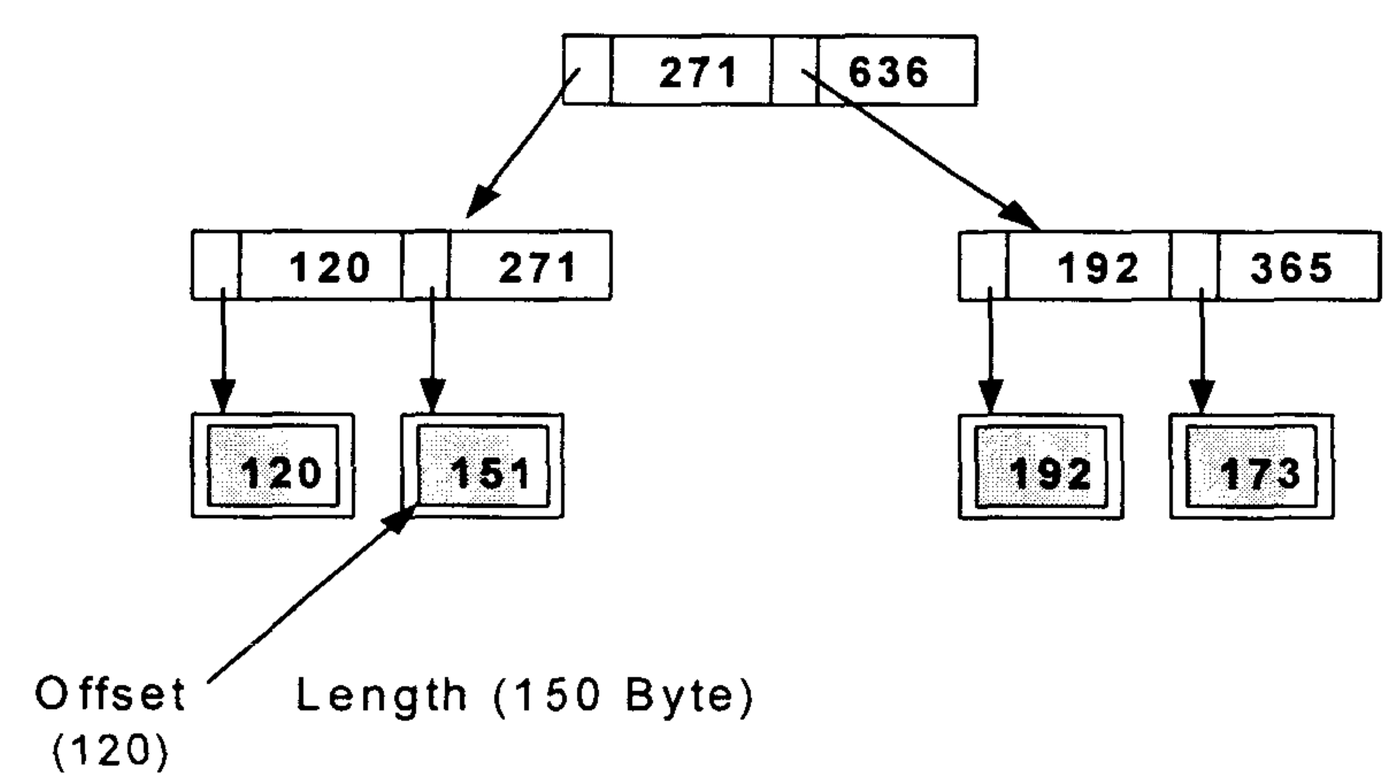
나) cut_path를 따라 다음 노드로 간다. 만약, 노드가 없으면, 그 트리는 재균형 상태가 된다.

다) 만약 현재 노드가 danger이면, 그것을 형제와 합병/재분배시킨다.

라) 나)로 가서 계속 반복한다.



(a) 블롭의 임의의 위치의 데이타 삭제전



(b) 블롭의 임의의 위치의 데이타 삭제후

[그림 2.15] 블롭의 임의의 위치의 데이타 삭제

제 3 장 정보 저장 관리기

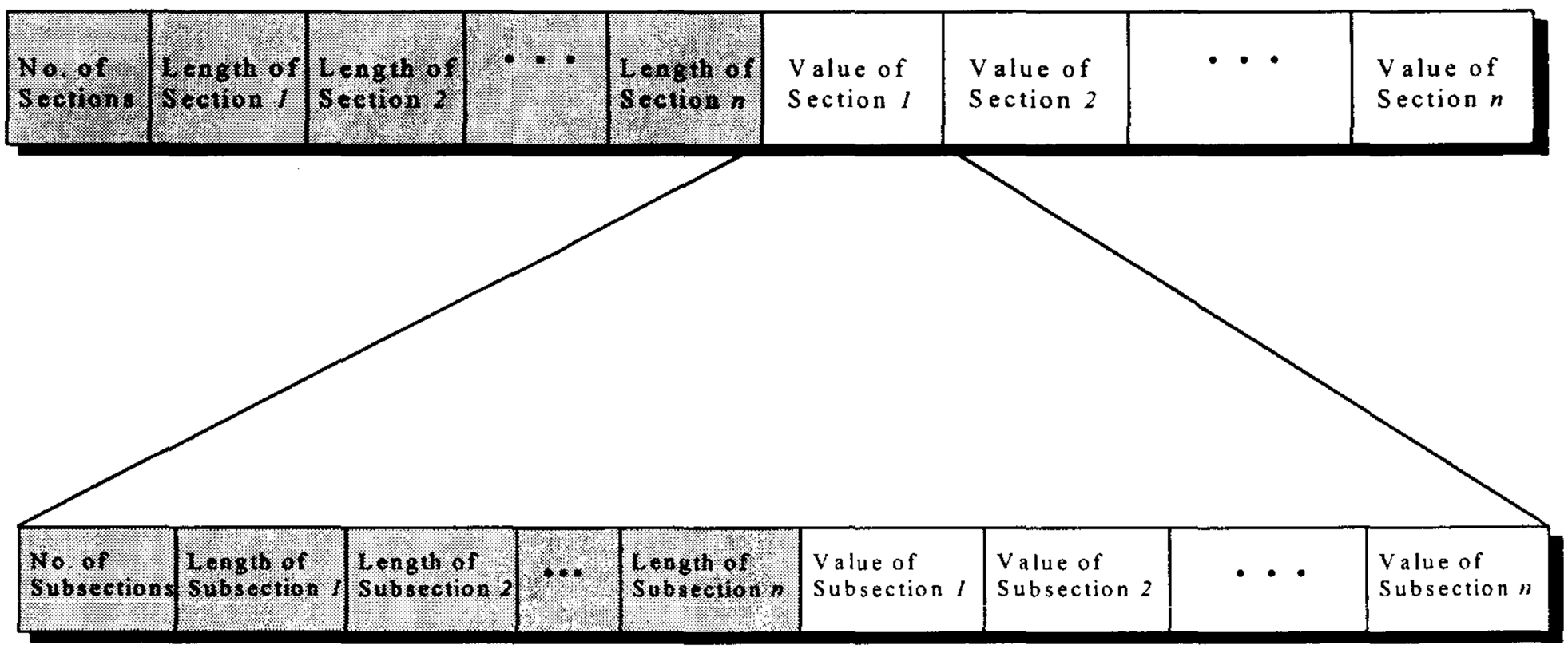
정보 저장 관리기는 하부의 정보 저장 지원기를 바탕으로 비정형화된 정보들을 저장 관리하기 위한 기능들을 지원한다. 그리고 이들 정보의 효율적인 검색을 지원하기 위해 역화일(inverted file) 구조의 색인어 접근 방식을 제공한다. 일반적으로 정보 검색 시스템은 사용자 또는 관리자로 하여금 데이터베이스를 보다 편리하게 접근하여 관리할 수 있도록 하기 위해서. 비정형화된 정보 모델과 데이터베이스에 저장된 문서들의 구조에 관한 정보들을 제공하여야 한다. 정보 저장 관리기의 카탈로그 관리기는 이러한 데이터베이스 목록 정보들의 관리를 목적으로 설계 구현되었다.

3.1 문서 관리기

3.1.1 문서 관리기의 설계

문서 관리기는 정보 검색에서 주된 대상이 되는 비정형화된 저장 구조의 문서를 저장 관리하기 위한 기능들을 지원한다. 본 과제의 1차년도 수행 기간 중에 우리는 과학 기술 정보 실 데이터의 표본 분석을 통해 문서 정보들의 특성을 분석한 바 있다. 그 분석 결과를 요약하면 대략 다음과 같다. 일반적으로 과학 기술 정보는 다양한 형태의 비정형화된 문서들로 구성되며, 데이터베이스의 문서들의 길이는 상당히 가변적이다. 그리고 각 문서는 여러 개의 섹션(section) 또는 필드를 지니며, 각각의 섹션 역시 여러 개의 서브 섹션을 포함할 수 있다는 특성을 지닌다. 예를 들어, 과학 기술 단행본 데이터베이스의 문서는 제목, 저자, 초록 등의 섹션들을 가지며, 특히 저자의 섹션은 저자가 여러 명인 경우 다중 값을 가질 수 있다.

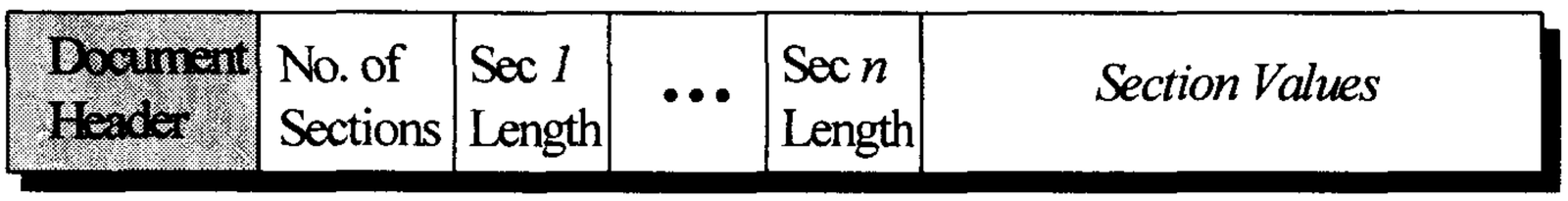
문서 관리기는 이러한 특성을 갖는 과학 기술 정보들을 관리하기 위한 메카니즘을 제공하며, 이를 위해 정보 저장 지원기의 레코드 관리기를 이용한다. 즉, 가변적인 길이의 문서들 뿐만 아니라 각 섹션이 다중의 서브 섹션 값을 갖는 문서들도 저장할 수 있도록 구현하였다. 문서 관리기는 이러한 문서들을 데이터베이스에 추가하거나 지정된 문서나 문서의 섹션 값을 판독하는 연산을 수행한다. 상위의 사용자들에게 보여지는 문서들의 저장 구조는 논리적으로 [그림 3.1]과 같다. 이 저장 구조의 헤더에 문서를 구성하는 모든 섹션들의 길이 정보가 저장되며, 실제 섹션의 내용은 섹션을 구성하는 서브섹션들의 길이 정보와 함께 뒤에 저장된다. 현재 이러한 저장 구조에서 문서는 최대 32767개의 섹션들을 가질 수 있으며, 섹션은 최대 32k 바이트의 값을 저장할 수 있도록 구현되었다.



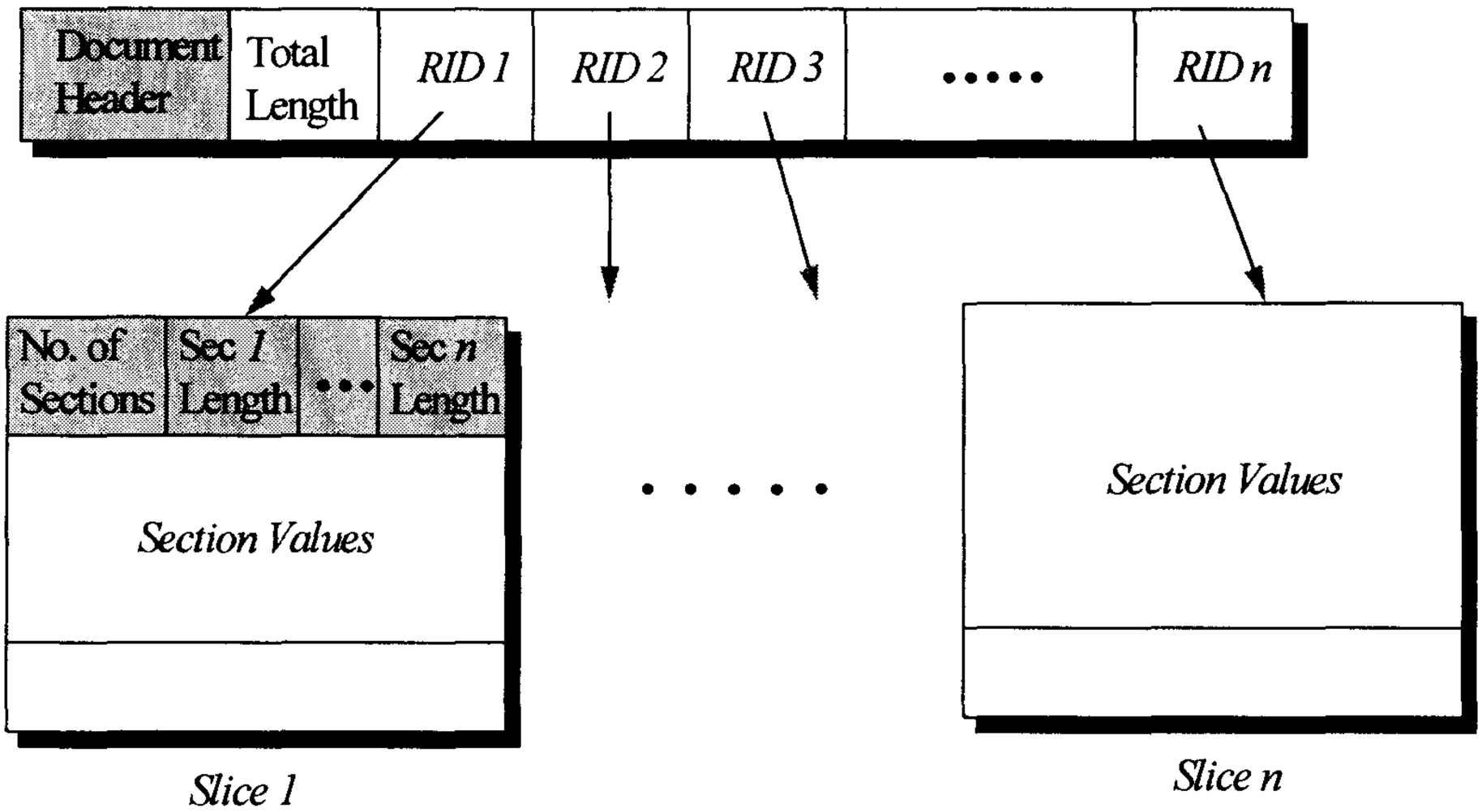
[그림 3.1] 문서의 논리적 저장 구조

[그림 3.1]의 논리적인 문서 저장 구조는 실제로 문서 관리기에서 내부적으로 [그림 3.2]와 같이 설계 구현되었다. 블럭의 크기보다 작은 문서들은 레코드 관리기를 이용하여 소규모의 레코드로 저장되며, 블럭의 크기보다 큰 문서들은 데이터베이스에 추가되는 과정에서 긴 자료 항목으로 변환되어 저장된

다. 전자의 경우 레코드의 종류는 NORMAL이 되며, 후자의 경우에는 긴 자료 항목임을 표시하는 LDIDIR이 된다. 문서 관리기는 이들 문서의 식별을 위한 문서 식별자로서 전자의 경우에는 레코드 식별자를 사용하며, 후자의 경우에는 긴 자료 항목의 디렉토리에 대한 레코드 식별자를 사용한다. 문서를 접근할 때, 문서 관리기는 이 문서 식별자를 통해 문서가 어떤 형식으로 저장되어 있는지를 알아냄으로써 각각의 형식에 맞는 레코드 관리 모듈이나 긴 자료 항목 관리 모듈을 호출한다.



(a) Small document (< BLOCKSIZE)



(b) Long document (> BLOCKSIZE)

[그림 3.2] 문서의 물리적 저장 구조

3.1.2 문서 관리기의 구현

doc_append(filenum, documentaddr, length, newdocrid)
TWO　　filenum;
char　　*documentaddr;
FOUR　　length;
RID　　*newdocrid;

filenum에 의해 지시되는 문서 화일에 새로운 문서를 추가하고, newdocrid를 통해 추가된 문서의 식별자를 반환한다. documentaddr는 추가할 문서의 메모리 주소를 지시하며, length는 추가할 문서의 길이를 나타낸다.

가) filenum에 의해 지시되는 문서 화일에 대한 접근 권한을 확인한다.
나) 추가할 문서의 크기를 검사한다.
다) 만일 문서의 크기가 블럭보다 작으면, 작은 크기의 레코드 형식으로 변환하여 문서 화일에 추가한다.
라) 만일 문서의 크기가 블럭보다 크면, 문서를 긴 자료 항목으로 변환하여 문서 화일에 추가한다. 즉, 먼저 DOC_createlong()를 호출하여 새로운 긴 자료 항목을 생성하고, 긴 자료 항목 관리 모듈의 ldi_insertframe() 함수를 호출하여 문서의 내용을 긴 자료 항목에 삽입한다.
마) filenum에 의해 지시되는 문서 화일의 상태가 변경되었음을 화일 디렉토리에 기록하고 함수를 종료한다.

doc_readdoc(filenum, ridptr, documentaddr, length)
TWO　　filenum;
RID　　*ridptr;
char　　*documentaddr;
FOUR　　length;

문서 화일로부터 ridptr에 의해 지시되는 문서를 사용자의 메모리 영역으로 읽어 들인다.

가) filenum에 의해 지시되는 문서 화일에 대한 접근 권한을 확인한다.

나) ridptr에 의해 지시되는 문서가 어떤 종류인지를 확인한다.

다) 만일 문서의 종류가 소규모의 레코드라면, 레코드 관리 모듈을 호출하여 문서를 사용자의 메모리 영역으로 읽어 들인다.

라) 만일 문서의 종류가 긴 자료 항목이면, 긴 자료 항목 관리 모듈의 ldi_readframe() 함수를 호출하여 긴 자료 항목의 시작 위치로부터 문서를 사용자의 메모리 영역으로 읽어 들인다.

doc_readsection(filenum, ridptr, secnum, secbuf, length)
TWO filenum;
RID *ridptr;
TWO secnum;
char *secbuf;
FOUR length;

문서 화일로부터 ridptr에 의해 지시되는 문서의 secnum번째 섹션을 읽어 사용자의 메모리 영역으로 복사한다.

가) filenum에 의해 지시되는 문서 화일에 대한 접근 권한을 확인한다.

나) 문서의 내용을 임시로 보관할 메모리 영역을 할당한다.

다) doc_readdoc() 함수를 통해 ridptr에 의해 지시되는 문서를 임시 영역으로 읽어 들인다.

라) 임시 영역에 저장된 문서의 섹션 관리 정보를 이용하여 secnum번째 섹션의 시작 위치를 계산한다.

바) 계산된 시작 위치로부터 섹션의 값을 secbuf에 의해 지시되는 사용자의 메모리 영역으로 복사한다.

사) 읽어 들인 섹션의 길이를 반환한다.

doc_firstdocument(filenum, firstdocrid)
TWO filenum;
RID *firstdocrid;

filenum에 의해 지시되는 문서 화일의 첫번째 문서에 대한 식별자를 firstdocrid를 통해 반환한다. 실제로 이 모듈은 docridptr의 값을 NULL로 하여 doc_nextdocument() 함수를 호출함으로써 수행된다.

가) 문서 화일의 첫번째 블럭을 버퍼로 캐쉬한다.
나) 캐쉬된 블럭으로부터 첫번째 문서의 위치를 찾아 그 문서의 식별자를 반환한다.

doc_nextdocument(filenum, docridptr, nextdocrid)
TWO filenum;
RID *docridptr;
RID *nextdocrid;

filenum과 docridptr에 의해 지시되는 문서의 뒤에 위치하는 문서를 찾아 그 문서의 식별자를 nextdocrid를 통해 반환한다. 만일 docridptr이 NULL이면, 첫번째 문서의 식별자를 반환한다.

가) filenum에 의해 지시되는 문서 화일에 대한 접근 권한을 확인한다.
나) docridptr가 NULL인지를 검사한다.
다) 만일 docridptr이 NULL이라면, 문서 화일의 첫번째 블럭을 버퍼로 캐쉬한다. 그리고 캐쉬된 블럭에서 첫번째 문서의 식별자를 구한다.
라) 만일 docridptr이 NULL이 아니라면, docridptr에 의해 지시되는 문서를 저장하고 있는 블럭을 버퍼로 캐쉬한다. 그리고 캐쉬된 블럭에서 다음 문서를 찾아 그 문서의 식별자를 계산한다. 만일 다음 문서가 캐쉬된 블럭에 존재하지 않으면, 반복해서 후방 블럭들을 읽어 들이면서 다음 문서를 찾아 그 문서의 식별자를 반환한다.

마) 캐쉬된 모든 버퍼 블럭들을 반환한다.

DOC_appendrec(filenum, docaddr, doclen, newrid)
TWO filenum;
char *docaddr;
FOUR *doclen;
RID *newrid;

docaddr에 의해 지시되는 작은 크기의 레코드를 문서 화일에 추가하고, 추가된 레코드의 식별자를 반환한다.

가) filenum에 의해 지시되는 문서 화일에 대한 접근 권한을 확인한다.
나) 문서 화일의 마지막 블럭을 캐쉬한 후, 그 블럭에 레코드를 추가할 만한 여분의 가용 영역이 있는지를 검사한다.
다) 만일 여분의 가용 영역이 있다면, 레코드를 그 블럭에 추가한다.
라) 만일 여분의 가용 영역이 없다면, 새로운 블럭을 할당 받는다. 그리고 할당받은 블럭에 레코드를 추가하고, 그 블럭을 화일의 블럭 리스트에 연결한다.
마) 화일의 상태가 변경되었음을 기록하고 함수를 종료한다.

DOC_createlong(filenum, ldiridptr)
TWO filenum;
RID *ldiridptr;

filenum에 의해 지시되는 문서 화일에 공백의 긴 자료 항목의 디렉토리를 생성하고, ldiridptr를 통해 생성된 디렉토리의 레코드 식별자를 반환한다.

가) filenum에 의해 지시되는 문서 화일에 대한 접근 권한을 확인한다.
나) 긴 자료 항목의 디렉토리를 위한 임시 영역을 할당 받는다.
다) 디렉토리를 초기화하고, 문서 화일에 추가한다.
라) ldiridptr를 통해 생성된 디렉토리의 식별자를 반환한다.

3.2 색인 관리기

3.2.1 색인 관리기의 설계

일반적으로 정보 검색 시스템에서의 색인은 시스템의 검색 속도를 높일 뿐만 아니라 검색 효과에도 큰 영향을 미치는 것으로 알려져 있다. 따라서 적절한 색인 방법의 지정은 시스템의 성능과 질을 크게 향상 시킬 수 있다. 본 과제에서 구현된 문서 색인 관리기는 문서의 색인 과정을 색인어 추출과 색인어 저장 방식의 두 가지 측면으로 구분하여 고려하였다.

3.2.1.1 색인어 추출 방식

본 과제에서 구현된 문서 색인 관리기는 문서로부터 색인어를 추출하기 위해 네 가지의 색인어 추출 방식을 지원한다. 데이터베이스 관리자는 문서의 섹션마다 이들 색인어 추출 방식 중의 하나를 선택하여 적용할 수 있다. 사실 색인어 추출에 대한 논의는 저장 시스템의 개발을 목표로 하는 본 과제의 범위를 벗어나다. 따라서 본 보고서에서는 현재 설계 구현된 색인어 추출 방식들에 대해서 간략하게 기술하기로 한다.

(1) EXACT_UNIQUE 와 EXACT_DUPLICATE

이들 색인어 추출 방식은 섹션의 내용 전체를 하나의 색인어로 추출한다. 그렇게 함으로써 그 섹션에 대한 완전일치(Exact matching)의 검색을 지원한다. EXACT_UNIQUE 는 관계형 데이터베이스의 기본키(Primary key)처럼 데이터베이스 전체에서 유일해야 하는 섹션에 적용할 수 있다. 예를 들어, 도서 데이터베이스의 '도서번호'와 같은 섹션에 적용가능하다. 반면에 EXACT_DUPLICATE 는 섹션의 값이 유일해야 한다는 제약성을 갖지 않으며, 전체 데이타베이스내에서 값의 중복을 허용하는 섹션에 사용된다.

(2) INCLUSIVE_NONE 과 INCLUSIVE_MA

이들 색인어 추출 방식은 텍스트 검색과 같이 내용 기반의 부분 일치 (Partial matching) 검색을 지원해야 하는 섹션에 적용 가능하다. 이들 색인 방식은 섹션의 내용 전체를 색인어로 추출하는 것이 아니라, 섹션내의 어절 또는 단어들 중에서 색인어를 선정한다. 여기에서 INCLUSIVE 는 'partial'의 의미를 갖는 것이다.

INCLUSIVE_NONE 은 섹션에서 불용어를 제외한 어절 또는 단어들을 색인어로 추출하는 초보적인 색인 방식으로, 별도의 후처리를 수행하지 않고 원문에 나타난 형태 그대로의 어절이나 단어를 색인어로 사용한다. 따라서 이 방식은 '사람이름'이나 '지명'과 같은 고유명사들을 주로 포함하고 있는 섹션이나 논문의 '키워드리스트'와 같이 별다른 후처리가 필요 없는 섹션들에 적합하다.

INCLUSIVE_MA 는 INCLUSIVE_NONE 의 색인 방식에서 한 단계 더 나아가 한글 텍스트의 색인을 위해 연구개발정보센터에서 개발한 한글 형태소 해석기(Morphological analyzer)를 이용한다. 즉, 한글 텍스트의 각 어절에 대해 형태소 해석을 수행함으로써 명사, 조사, 접미사, 동사, 형용사 등의 최소 형태소 단위를 구분한 후, 섹션의 내용을 대표할 수 있는 단순 명사(simple noun)를 색인어로 추출한다. 따라서 이 방식은 단순한 INCLUSIVE_NONE 의 어절 단위 색인보다 양질의 색인을 수행하므로 효과적인 정보 검색을 지원할 수 있다. INCLUSIVE_MA 는 예를 들면, '논문제목'이나 '초록'과 같이 주로 텍스트를 포함하는 섹션에 적용할 수 있다.

3.2.1.2 역화일 구조의 색인어 저장

본 과제에서 구현한 정보 저장 관리기는 문서의 빠른 검색을 지원하기 위

해 역화일(inverted file) 구조의 색인어 저장 방식을 제공한다. 역화일은 텍스트 검색과 부분 일치(partial match)의 검색을 지원하고자 하는 정보 검색 시스템에서 주로 사용되고 있는 대표적인 텍스트 화일 접근 방법으로, 기존의 대부분의 상용 정보 검색 시스템에서 많이 채택하고 있는 방식이다. 비록 대용량의 저장 공간을 필요로 하는 단점을 갖지만 고속의 접근 속도를 지원할 수 있는 장점 때문에 많이 이용되고 있다. 예를 들어, 잘 알려진 록히드사의 DIALOG, SDC의 ORBIT, 미국 국립 의학 도서관의 MEDLINE, IBM사의 STAIRS 등의 정보 검색 시스템들이 역화일을 기반으로 하고 있다.

역화일의 각 엔트리는 기본적으로 색인어와 색인어를 포함하고 있는 문서들의 식별 번호 또는 문서 화일내에서의 색인어의 위치들로 구성된다. 그 뿐만 아니라 색인어를 포함하고 있는 문서들의 총 수를 추가로 소장함으로써 검색 과정에서 이를 이용하기도 한다. 예를 들면, 데이터베이스에 존재하는 일부 문서에서 다음과 같은 색인어들이 추출되었다고 가정하자.

문서 D1 : term 1, term 2, term 4
문서 D2 : term 2, term 3, term 4
문서 D3 : term 1, term 3, term 4

이들에 대하여 생성되는 역화일은 다음과 같다.

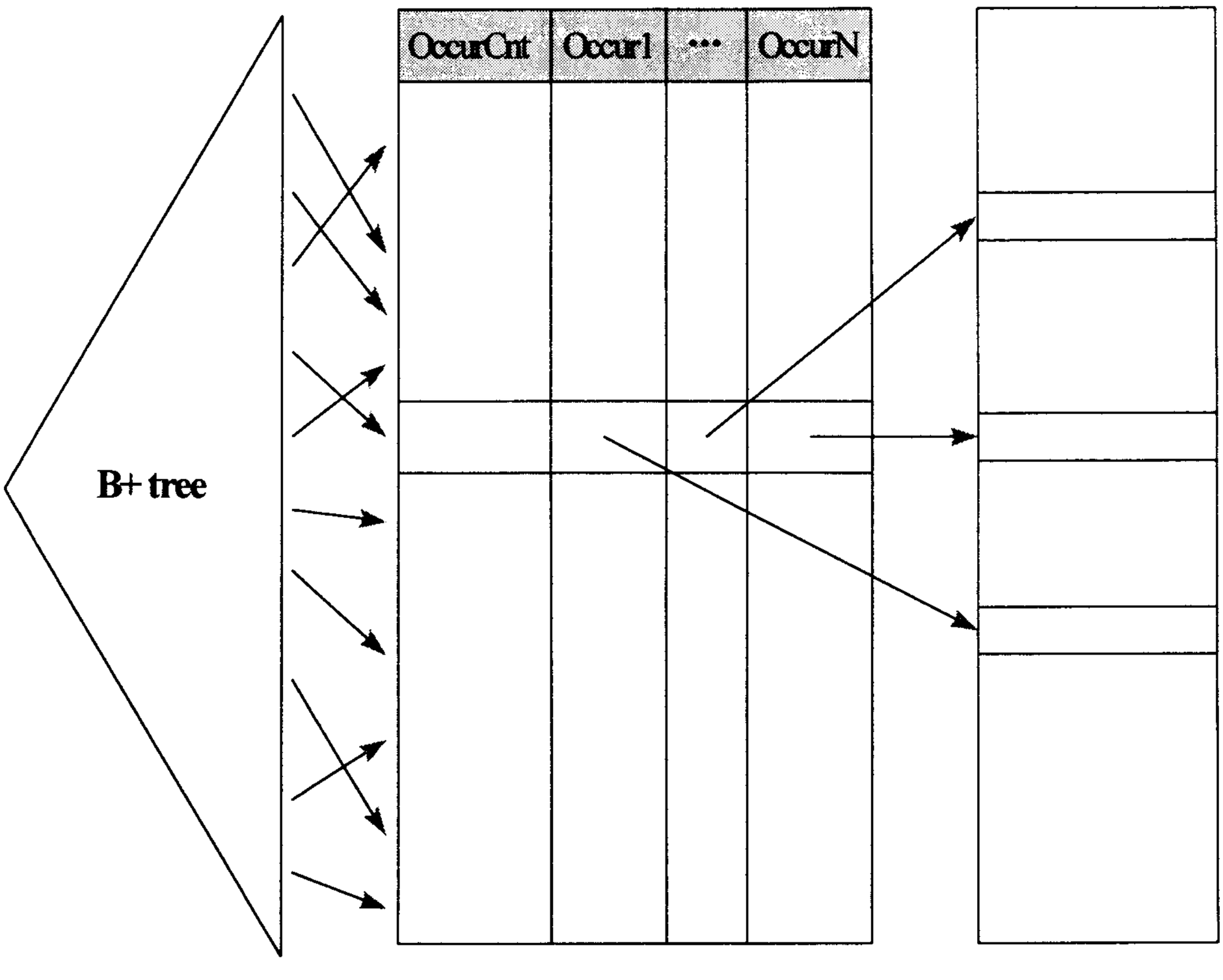
term 1 : D1, D3
term 2 : D1, D2
term 3 : D2, D3
term 4 : D1, D2, D3

그러나 대부분의 경우 정보 검색의 대상이 되는 텍스트들은 색인어에 따라 해당 문서들의 수가 각기 다르기 때문에 문서 번호 리스트의 길이에 큰 차이가 있을 수 있다. 따라서 실제로 역화일을 구현하는 과정에서 문서 식별 번호만을 별도로 소장하는 것이 편리한 경우도 있다. 이러한 경우에 일반적으로 역화일은 사전 화일과 문서 번호 화일로 구성된다. 사전 화일에는 색인어와

함께 색인어를 포함하고 있는 문서들의 수 및 문서 번호 화일의 엔트리에 대한 포인터가 저장된다. 그리고 문서 번호 화일에는 색인어에 대한 해당 문서들의 식별자 리스트가 저장된다. 그렇게 함으로써 색인어의 검색 연산을 수행할 때 문서 번호 화일에서 해당되는 문서들의 식별자를 먼저 찾은 후 문서 화일을 접근함으로써 원하는 문서를 검색할 수 있다.

역화일은 실제 문서 화일의 전체 항목이 아닌 색인어만으로 사용자의 검색 요구에 관련된 문서들을 검색하기 때문에 정보 항목으로의 접근을 빠르게 할 수 있다. 또한 색인어를 키의 값에 따라 순차적으로 배치할 수 있으므로 새로운 색인어를 추가할 때에 문서 화일내에서의 특별한 순서에 관계없이 사용자가 편리한 위치에 어느 곳에나 추가할 수 있다는 장점을 갖는다. 그러나 역화일은 문서 화일에 따라 크기가 커질 수 있으므로 많은 저장 영역을 필요로 한다는 단점을 갖기도 한다.

[그림 3.3]은 본 과제에서 설계 구현한 역화일의 구조를 도시하고 있다. 그림에서 보는 바와 같이 구현된 역화일은 색인어 접근 경로 화일(index path file), 포스팅 화일(posting file)로 구성된다. 색인어 접근 경로 화일은 문서 화일의 지정된 섹션에서 추출된 색인어들과 이들 색인어의 문서 내에서의 위치 정보 및 각 색인어가 포함되어 있는 문서들의 수로 이루어지며, 빠른 속도의 접근을 제공하기 위해 B+ 트리 구조와 비슷하게 구현되었다. 이 색인어 접근 경로 화일의 리프 노드는 색인어와 색인어를 포함하고 있는 문서들의 수를 저장하며, 포스팅 화일에 대한 포인터를 유지한다.



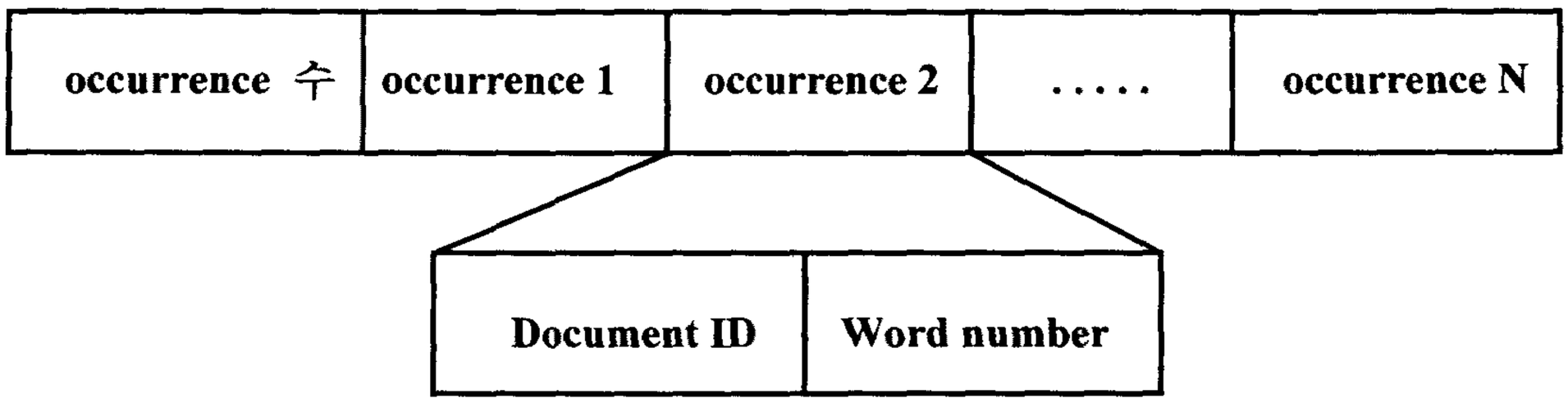
[그림 3.3] 역화일의 구조

다음은 색인어 접근 경로 화일의 리프 노드 엔트리의 구조를 보여준다.

Key(length, value)	Document Count	Posting file Pointer

포스팅 화일은 색인어가 데이터베이스 전체에 걸쳐 출현한 횟수, 그리고 그 색인어를 포함하고 있는 문서들의 식별자 및 문서 내에서의 색인어의 위치를

소장하고 있다. 이 화일은 색인어 접근 경로 화일에 있는 각각의 색인어에 대해 하나의 엔트리를 유지하는데, 각 엔트리는 오커런스(occurrence) 리스트와 오커런스의 수로 이루어진다. 여기서 오커런스는 색인어가 포함된 문서의 식별자와 색인어의 문서내에서의 위치를 표시하는 색인어 번호 또는 단어 번호로 구성된다.



포스팅 화일은 앞 절에서 설명한 문서 관리기를 사용하여 순차 화일의 구조로 구현되었다. 다시 말해, 포스팅 화일 엔트리가 블럭보다 작은 경우에는 작은 크기의 레코드 형식으로, 그리고 블럭의 크기보다 큰 경우에는 긴 자료 항목으로 변환하여 저장되도록 구현하였다.

3.2.2 색인 관리기의 구현

```
idx_createsectidx(volid,docfilename,sectnum,stopwordfilenames,indextype)
short     volid;
char      *docfilename;
int       sectnum;
char      stopwordfilenames[][];
int       indextype;
```

문서 화일의 지정된 섹션으로부터 색인어를 추출하고, 이에 대한 역화일 구조의 색인 화일을 형성한다. 여기서 volid는 볼륨 식별자, docfilename는 문서 화일, sectnum는 색인을 생성하고자 하는 섹션의 번호를 나타내며,

stopwordfilenames는 불용어 리스트를 포함하고 있는 유닉스 화일들의 이름을 나타낸다. 그리고 indextype은 섹션에 대한 색인어 추출 방식을 표시하는 것으로, EXACT_UNIQUE, EXACT_DUPLICATE, INCLUSIVE_NONE, INCLUSIVE_MA 중의 하나로 지정된다.

Idx_destroysectidx(volid, docfilename, sectnum)
short volid;
char *docfilename;
int sectnum;

문서 화일의 지정된 섹션에 대한 역화일(색인어 접근 경로 화일과 포스팅 화일)을 제거한다. 여기서 문서 화일은 volid와 docfilename에 의해 지시되며, sectnum은 삭제할 역화일의 섹션 번호이다.

가) docfilename과 sectnum을 통해 색인어 접근 경로 화일과 포스팅 화일의 이름을 구한다.
나) 화일 디렉토리 관리기를 호출하여 색인어 접근 경로 화일과 포스팅 화일을 제거한다.

idx_openpathfile(volid, docfilename, sectnum, accessmode)
short volid;
char *docfilename;
int sectnum;
int accessmode;

문서 화일의 지정된 섹션에 대한 색인어 접근 경로 화일을 개방한다. 여기서 volid와 docfilename은 문서 화일을 지시하고, sectnum은 문서 화일에서의 섹션 번호를 나타낸다. 그리고 accessmode는 색인어 접근 경로 화일에 대한 접근 모드를 표시한다.

가) docfilename과 sectnum을 통해 문서 화일의 지정된 섹션에 대한 색인어 접근 경로 화일의 이름을 구한다.

나) 화일 디렉토리 관리기를 호출하여 accessmoce에 의해 지시되는 화일 접근 모드로 색인어 접근 경로 화일을 개방하고, 화일 번호를 반환한다.

idx_openpostfile(volid, docfilename, sectnum, accessmode)
short volid;
char *docfilename;
int sectnum;
int accessmode;

문서 화일의 지정된 섹션에 대한 포스팅 화일을 개방한다. 여기서 문서 화일은 volid와 docfilename에 의해 지시되고, sectnum은 문서 화일에서의 섹션 번호를 나타낸다. 그리고 accessmode는 포스팅 화일에 대한 접근 모드를 표시한다.

가) docfilename과 sectnum을 통해 문서 화일의 지정된 섹션에 대한 포스팅 화일의 이름을 구한다.

나) 화일 디렉토리 관리기를 호출하여 accessmoce에 의해 지시된 화일 접근 모드로 포스팅 화일을 개방하고, 화일 번호를 반환한다.

idx_getoccurcnt(filenum, expression, occurcnt)
int filenum;
FIELDNUMEXP *expression;
short *occurcnt;

문서 화일의 지정된 섹션에서 사용자가 입력한 수식(expression)을 만족하는 모든 색인어들의 총 출현 횟수를 반환한다. 여기서 filenum은 문서 화일의 화일 번호를 나타내고, occurcnt는 이 함수에서 반환하는 색인어 출현 횟수의 값을 저장하기 위한 임시 변수이다. 그리고 사용자는 expression 구조체를 통해 섹션 번호와 찾고자 하는 색인어들의 표현식을 지정할 수 있다. 본 모듈에서 사용자는 표현식으로서 단순한 색인어뿐만 아니라 우측 절단 색인어와 중간

절단 색인어도 입력할 수 있도록 구현되었다. 예를 들면, 사용자는 "인공지능", "데이타베이스*", "객체지향*시스템"과 같은 표현식을 입력할 수 있다.

가) 문서 화일의 화일 번호 filenum을 통해 색인어 접근 경로 화일과 포스팅 화일을 개방한다.

나) 사용자가 입력한 수식(expression)을 만족하는 첫번째 색인 엔트리의 위치(리프 블럭과 슬롯번호)를 탐색한다.

다) 그 위치의 색인 엔트리로부터 시작하여 인접한 후위 색인 엔트리들을 반복적으로 탐색함으로써 수식(expression)을 만족하는 모든 엔트리들을 구한다. 그리고 포스팅 화일을 접근하여 이들 모든 엔트리의 색인어에 대한 오커런스의 수를 얻는다. expression을 만족하는 색인 엔트리들이 더 이상 없을 때까지 이 작업을 계속 수행한다.

라) 모든 오커런스들의 수를 totaloccurcnt에 저장한다.

마) 색인어 접근 경로 화일과 포스팅 화일을 닫고 함수를 종료한다.

```
idx_getoccurlist( filenum,expression,listbufsize,occurlist,occurcnt )
int                 filenum;
FIELDNUMEXP *expression;
short               listbufsize;
RIDWORDNUM occurlist[];
short               *occurcnt;
```

문서 화일의 한 섹션에서 사용자가 입력한 수식(expression)을 만족하는 모든 색인어들의 출현 정보 즉, 오커런스의 리스트를 반환한다. 여기서 오커런스는 색인어가 출현하는 문서의 식별자와 문서내에서 색인어의 위치를 나타내는 색인어 번호로 구성된다. filenum은 문서 화일의 개방 화일 번호를 나타내고, occurcnt는 이 함수에서 반환하는 색인어 출현 횟수의 값을 저장하기 위한 매개 변수이며, expression은 섹션 번호와 사용자가 찾고자 하는 색인어들의 표현식을 지정하기 위한 구조체다. 그리고 occurlist[]는 반환되는 오커런스 리스트

를 저장하기 위한 배열에 대한 포인터이며, listbufsize는 이 배열의 크기를 나타낸다.

가) 문서 화일의 화일 번호 filenum을 통해 색인어 접근 경로 화일과 포스팅 화일을 개방한다.

나) 사용자가 입력한 수식(expression)을 만족하는 첫번째 색인 엔트리의 위치(리프 블럭과 슬롯번호)를 탐색한다.

다) 그 위치의 색인 엔트리로부터 시작하여 인접한 후위 색인 엔트리를 탐색함으로써 수식(expression)을 만족하는 모든 엔트리들을 구한다. 그리고 이들 모든 엔트리의 색인어에 대한 오커런스 리스트를 포스팅 화일을 접근하여 배열 occurlist[]에 복사한다. 수식을 만족하는 색인 엔트리들이 더 이상 없을 때까지 이 작업을 계속 반복 수행한다.

라) 모든 오커런스들의 수를 totaloccurcnt에 저장한다.

마) 색인어 접근 경로 화일과 포스팅 화일을 닫고 함수를 종료한다.

3.3 카탈로그 관리기

지금까지 기술한 관리기들은 실세계 관점에서의 자료 관리 모형을 지원하지 않고 단지 하부 저장 시스템에서의 관점인 디스크 볼륨, 블럭, 화일, 레코드, 색인 등의 개념을 제공할 뿐이다. 따라서 상위 레벨의 사용자로 하여금 실세계의 데이터베이스를 보다 편리하게 표현하고 관리할 수 있도록 하는 방법이 요구된다. 예를 들어, 과학 기술 단행본 데이터베이스는 단행본 도서라는 개체들의 모임으로 표현되며, 각 개체는 제어 번호, 자료 형태, 서명, 저자명, 초록 등의 속성으로 이루어지고, 또한 속성과 속성을 결합하여 새로운 속성이 생성될 수 있다. 본 절의 카탈로그 관리기는 이러한 사용자나 관리자의 관점에서의 정보 관리를 위한 기능을 수행한다.

한편 기하 급수적으로 증가하는 데이터베이스는 더 이상 하나의 볼륨에 저장할 수 없는 상황을 초래하고 있다. 따라서 늘어나는 데이터베이스를 여러 개의 볼륨상에 분산하여 저장하는 다중 볼륨(multi-volume) 기능의 지원이 필수적인데, 본 과제에서는 이를 카탈로그 관리기에서 구현하도록 하고 있다. 즉, 하나의 데이터베이스를 여러 볼륨으로 사상시키는 기능을 지원한다.

3.3.1 카탈로그 관리기의 설계

정보 검색 시스템의 상위 검색 엔진이나 데이터베이스 관리기를 위하여 카탈로그 관리기는 데이터베이스 카탈로그(database catalog), 문서 화일 카탈로그(document file catalog), 섹션 카탈로그(section catalog), 결합 섹션 카탈로그(combined section catalog)의 시스템 화일들로 구성되며, 이들 시스템 화일을 통해 데이터베이스 목록 정보를 관리한다. 이들은 별도의 카탈로그 볼륨(대개의 경우 CATALOG.SYS)에 저장된다. 사용자는 새로운 데이터베이스를 생성하기에 앞서 카탈로그 볼륨 생성 유틸리티를 이용하여 카탈로그 볼륨을 생성하고 초기화한다. 그리고 상위의 검색 엔진은 문서를 검색하기에 앞서 이 카탈로그 볼륨을 마운트하고 모든 시스템 화일들을 개방한다. 데이터베이스 목록 정보를 위한 시스템 화일들은 모두 순차 화일의 간단한 구조로 구현되었으며, 순차적인 스캔을 통해 데이터베이스나 문서 섹션들에 대한 필요한 정보를 얻을 수 있다.

데이터베이스 카탈로그(database catalog)는 시스템 상에 존재하는 모든 데이터베이스들에 대한 다음과 같은 정보를 저장한다. 데이터베이스의 식별자는 관리자에 의해 부여되며, 섹션은 데이터베이스의 각 문서들을 구성하는 속성들을 의미한다.

- 데이터베이스 이름

- 데이터베이스 식별자
- 문서를 구성하는 섹션들의 수

본 과제에서 구현한 저장 시스템은 대용량의 데이터베이스에 대해 정보를 여러 볼륨의 문서 화일로 분산하여 저장한다. 문서 화일 카탈로그는 이러한 물리적인 구성 정보를 유지하기 위한 시스템 화일로서, 다중 볼륨의 기능을 지원한다. 즉, 여러 볼륨에 분산되어 있는 문서 화일들에 대한 정보를 포함한다. 문서 화일 카탈로그는 데이터베이스를 구성하는 각각의 문서 화일에 대해 하나의 엔트리를 저장하며, 각 엔트리에 포함된 내용은 다음과 같다.

- 데이터베이스 식별자
- 데이터베이스를 구성하는 문서 화일의 이름
- 문서 화일을 포함하고 있는 볼륨의 이름
- 문서 화일에 있는 블럭들의 수
- 문서 화일에 있는 문서들의 수

데이터베이스의 문서들을 구성하는 섹션들에 대한 정보는 섹션 카탈로그와 결합 섹션 카탈로그에 의해 관리된다. 섹션 카탈로그는 각 데이터베이스의 기본적인 섹션들의 형식을 기술하는 시스템 화일로서, 다음의 정보들을 저장한다.

- 데이터베이스 식별자
- 섹션의 이름
- 섹션의 별칭
- 섹션의 번호
- 색인을 위한 정보(색인 타입, 불용어 리스트 화일의 이름 등)

결합 섹션은 여러 개의 기본 섹션들을 결합한 가상의 섹션이다. 예를 들면, "author"라는 결합 섹션은 "personal name", "cooporate name", "conference name"

등의 기본 섹션들로 구성될 수 있다. 결합 섹션 카탈로그는 이러한 가상의 섹션들에 대한 정보를 유지하기 위해 사용되는 시스템 화일이다. 결합 섹션 카탈로그에서 관리되는 정보는 다음과 같으며, 결합 섹션을 구성하는 각각의 기본 섹션에 대해 하나의 엔트리를 갖는다.

- 데이터베이스 식별자
- 결합 섹션의 이름
- 결합 섹션의 별칭
- 결합 섹션을 구성하는 기본 섹션의 이름

3.3.2 카탈로그 관리기의 구현

CM_Initialize(catalogvolname, accessmode)
char *catalogvolname;
int accessmode;

카탈로그 관리기를 초기화시키는 모듈로서, catalogvolname에 의해 지시되는 카탈로그 볼륨을 마운트하고 모든 시스템 카탈로그 화일들을 accessmod에 의해 지시되는 화일 접근 모드로 개방한다.

가) 카탈로그 관리를 위한 모든 시스템 전역 변수들을 초기화한다.
나) 입출력 관리기, 버퍼 관리기, 화일 디렉토리 관리기, 레코드 관리기, 문서 관리기, 색인 관리기 등의 모든 하위 관리기들을 기동시킨다.
다) catalogvolname에 의해 지시되는 카탈로그 볼륨을 마운트한다.
라) 데이터베이스 카탈로그의 시스템 화일을 개방한다.
마) 문서 카탈로그의 시스템 화일을 개방한다.
바) 섹션 카탈로그의 시스템 화일을 개방한다.
사) 결합섹션 카탈로그의 시스템 화일을 개방한다.

CM_Terminate(catalogvolname)
char *catalogvolname;

카탈로그 관리기를 종료하는 함수로서, 모든 카탈로그 시스템 화일들을 닫고 catalogvolname에 의해 지시되는 카탈로그 볼륨의 사용을 중지시킨다.

가) 데이터베이스 카탈로그 화일을 닫는다.
나) 문서 카탈로그 화일을 닫는다.
다) 섹션 카탈로그 화일을 닫는다.
라) 결합섹션 카탈로그 화일을 닫는다.
마) 카탈로그 볼륨을 디스마운트한다.
바) 입출력 관리기, 버퍼 관리기, 화일 디렉토리 관리기, 레코드 관리기, 문서 관리기, 색인 관리기 등의 모든 하위 관리기들을 종료한다.

CM_GetDBCatRec(DBname, DBCatRec)
char *DBname;
DATABASECAT *DBCatRec;

DBname에 의해 지시되는 데이터베이스의 목록 정보를 데이터베이스 카탈로그 화일로부터 읽어 들여 DBCatRec를 통해 반환한다.

가) 데이터베이스 카탈로그 화일을 탐색하기 위한 부울식을 작성한다.
나) 작성된 부울식을 바탕으로 데이터베이스 카탈로그 화일에 대한 순차 스캔을 시작한다.
다) 부울식을 만족하는 첫 목록 정보의 위치를 찾아 DBCatRec에 복사한다.
라) 순차 스캔을 종료한다.

CM_OpenDocCatScan(DBID)
TWO DBID;

카탈로그 관리기는 대용량의 데이터베이스를 여러 볼륨의 문서 화일들로 분산 저장하는 다중 볼륨 기능을 지원한다. 본 모듈은 분산된 문서 화일들에 대한 정보를 얻기 위한 것으로서, 문서 카탈로그 화일에 대한 순차 스캔을 시작한다. 여기서 DBID는 데이터베이스의 식별자이다.

가) 문서 카탈로그로부터 DBID에 의해 지시되는 데이터베이스의 문서 화일들을 탐색하기 위한 부울식을 작성한다.

나) 문서 카탈로그 화일에 대한 순차 스캔을 시작한다.

CM_DocCatRecFirst(DocCatRec)
DOCUMENTCAT *DocCatRec;

데이터베이스를 구성하는 문서 화일들 중에서 첫번째 문서 화일에 대한 정보를 DocCatRec를 통해 반환한다. 이 모듈은 CM_OpenDocCatScan()을 수행한 후에 호출된다.

가) CM_OpenDocCatScan에서 작성된 부울식을 만족하는 첫번째 문서 카탈로그 정보의 위치를 찾는다.

나) 그 정보를 구조체 DocCatRec에 복사한다.

CM_DocCatRecNext(DocCatRec)
DOCUMENTCAT *DocCatRec;

데이타베이스를 구성하는 문서 화일들에 대한 목록 정보 중에서 다음 위치의 목록 정보를 DocCatRec를 통해 반환한다.

가) CM_DocCatRecFirst()의 스캔 커서 위치에서 부울식을 만족하는 다음 목록 정보의 위치로 스캔 커서를 이동시킨다.

나) 그 위치의 목록 정보를 DocCatRec로 복사한다.

CM_CloseDocCatScan( )

문서 카탈로그 화일에 대한 스캔을 종료한다.

CM_OpenSecCatScan(DBID)
TWO DBID;

데이터베이스를 구성하는 문서 섹션들의 정보를 얻기 위한 모듈로서, 섹션 카탈로그 화일에 대한 순차 스캔을 시작한다. 여기서 DBID는 데이터베이스의 식별자이다.

가) 섹션 카탈로그 화일에서 DBID에 의해 지시되는 데이터베이스의 문서 섹션 정보를 탐색하기 위한 부울식을 작성한다.

나) 문서 카탈로그 화일에 대한 순차 스캔을 시작한다.

CM_SecCatRecFirst(SecCatRec)
SECTIONCAT *SecCatRec;

데이터베이스를 구성하는 문서 섹션들 중에서 첫번째 섹션에 대한 정보를 SecCatRec를 통해 반환한다. 이 모듈은 CM_OpenSecCatScan()을 수행한 후에 호출된다.

가) CM_OpenSecCatScan()에서 작성된 부울식을 만족하는 첫번째 섹션 목록 정보의 위치를 찾는다.

나) 그 위치의 목록 정보를 SecCatRec에 복사한다.

CM_SecCatRecNext(SecCatRec)
SECTIONCAT *SecCatRec;

데이타베이스를 구성하는 문서 섹션들에 대한 목록 정보 중에서 다음 위치의 섹션 목록 정보를 SecCatRec를 통해 반환한다.

가) CM_SecCatRecFirst()의 스캔 커서 위치에서 부울식을 만족하는 다음 섹션 목록 정보의 위치로 스캔 커서를 이동시킨다.
나) 그 위치의 섹션 목록 정보를 SecCatRec로 복사한다.

CM_CloseSecCatScan( )

섹션 카탈로그 화일에 대한 스캔을 종료한다.

CM_GetSecCatRec(DBID, SecAliasName, SecCatRec)
TWO DBID;
char *SecAliasName;
SECTIONCAT *SecCatRec;

SecAliasName에 의해 지시되는 특정 섹션에 대한 목록 정보를 SecCatRec를 통해 반환한다. 여기서 DBID는 데이터베이스의 식별자이고, SecAliasName는 정보를 얻으려고 하는 섹션의 별명이다.

가) SecAliasName에 의해 지시되는 특정 섹션의 목록 정보를 탐색하기 위한 부울식을 작성한다.
나) 섹션 카탈로그 화일에 대한 순차 스캔을 시작한다.
다) 부울식을 만족하는 섹션 목록 정보의 위치를 찾는다.
라) 그 위치의 섹션 목록 정보를 SecCatRec으로 복사한다.
마) 섹션 카탈로그 화일에 대한 순차 스캔을 종료한다.

CM_OpenComSecCatScan(DBID, ComSecAliasName)
TWO DBID;
char *ComSecAliasName;

본 과제에서 구현하는 카탈로그 관리기는 데이타베이스의 기본 섹션들을 결합한 가상의 결합섹션을 지원한다. 본 모듈은 이러한 가상의 결합섹션을 구성하고 있는 기본 섹션들의 목록 정보를 얻기 위한 것으로서, 결합섹션 카탈로그 화일에 대한 순차 스캔을 시작한다. 여기서 DBID는 데이터베이스의 식별자이며, ComSecAliasName는 결합섹션의 별칭이다.

가) 결합섹션 카탈로그로부터 DBID와 ComSecAliasName에 의해 지시되는 데이터베이스 결합섹션에 대한 정보를 탐색하기 위한 부울식을 작성한다.
나) 결합섹션 카탈로그 화일에 대한 순차 스캔을 시작한다.

CM_ComSecCatRecFirst(ComSecCatRec)
COMSECCAT *ComSecCatRec;

결합섹션을 구성하는 기본 섹션들 중에서 첫번째 기본 섹션에 대한 정보를 ComSecCatRec를 통해 반환한다. 이 모듈은 CM_OpenComSecCatScan()을 수행한 후에 호출된다.

가) CM_OpenComSecCatScan()에서 작성된 부울식을 만족하는 첫번째 기본 섹션 목록 정보의 위치를 찾는다.
나) 그 위치의 목록 정보를 ComSecCatRec에 복사한다.

CM_ComSecCatRecNext(ComSecCatRec)
COMSECCAT *ComSecCatRec;

결합섹션을 구성하는 기본 섹션들에 대한 목록 정보 중에서 다음 위치의 기본 섹션 목록 정보를 ComSecCatRec를 통해 반환한다.

가) CM_ComSecCatRecFirst()의 스캔 커서 위치에서 부울식을 만족하는 다음 기본 섹션 목록 정보의 위치로 스캔 커서를 이동시킨다.

나) 그 위치의 목록 정보를 ComSecCatRec로 복사한다.

CM_CloseComSecScan( )

결합섹션 카탈로그 화일에 대한 스캔을 종료한다.

제 4 장 정보 압축 복원기

4.1 영상 정보의 압축 복원기

본 절에서는 이진 문서 영상 데이터베이스를 구축하기 위한 가장 적절한 압축 복원기를 선택하여 구현하고, 또한 그레이 및 칼라 영상의 국제 표준 압축 복원기인 JPEG(Joint Photographic Experts Group)[1]을 보다 개선시키기 위한 방법을 개발 구현한 내용을 기술한다.

먼저 문서 영상 데이터베이스를 구축하는 방법론을 분류 검토한 후, 현실적인 면에서 가장 타당성이 있는 단계적 접근 방법론을 제시하였다. 그리고 이 방법론에 있어서의 첫 접근 단계인 이진 문서 영상 데이터베이스를 구축하기 위한 조건 및 사용자 환경 변화에 대해 검토한 후, 이진 문서 영상의 스캐닝 해상도 및 압축 복원 방법에 대해 분석하였다. 스캐닝 해상도로는 600 dpi 가 여러 가지 측면에서 가장 적합하였으며, 압축 복원기로는 CCITT 의 표준안 JBIG(Joint Bi-level Image experts Group)[19]이 가장 적합하였다. 이에 따라 본 연구에서는 JBIG 을 기반으로 하는 압축 및 복원 프로그램을 구현하였고, 아울러 통신망에서의 송수신 시간 및 사용자 인터페이스의 편의를 고려하여 문서 영상의 단계적 처리 방법을 개발 구현하였다.

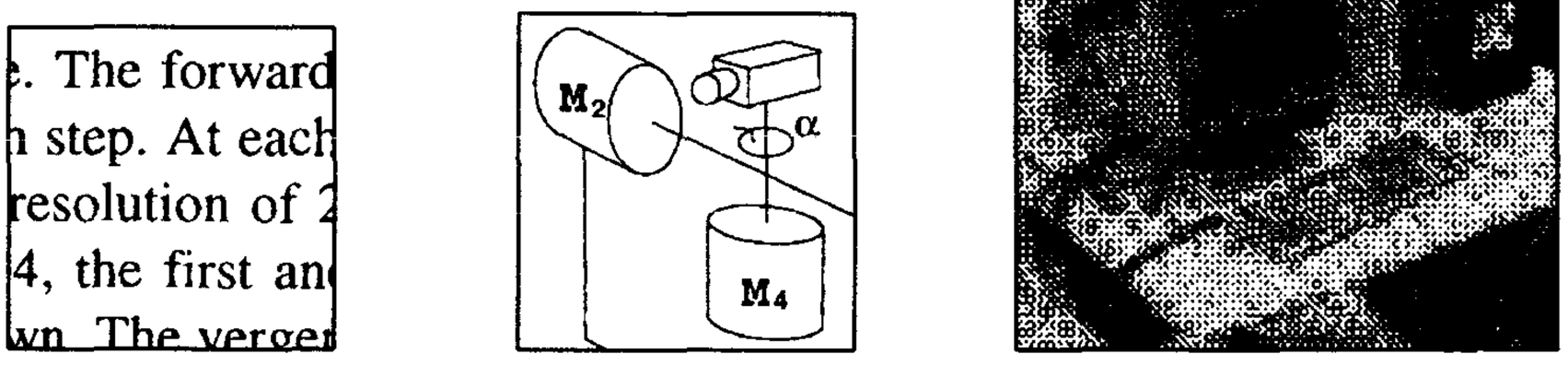
한편 칼라 및 그레이 영상을 압축 복원하는 방법인 국제 표준안 JPEG 을 보다 효과적으로 활용하는 방법으로서, 인간의 시각 특성을 고려한 압축 복원 시스템[38]에 대해 기술한다.

4.1.2 문서 영상 데이터베이스 구축 방법의 분류

4.1.2.1 문서영상의 영역별 분류 및 특성

문서 영상을 영역별로 분류하면 문자와 심볼로 구성된 텍스트 영역, 도표 및 선 도형으로 구성된 그래픽 영역, 그레이 및 칼라 그림이 이진화 되어 나타나는 해프토닝 그림 영역으로 구분할 수 있다.

텍스트 영역과 그래픽 영역은 저해상도로 스캐닝하여도 어느 정도 양질의 정보를 제공하며 이진 영상으로 서비스해도 무방하다. 또한 매우 높은 압축율로 저장이 가능하다. 반면에 해프토닝 그림 영역은 저해상도에서 질 저하 현상이 심각하며, 압축율도 좋지 않다. CCITT T.6[26]와 같은 압축 방법에서는 오히려 압축시킨 데이터 양이 본래의 데이터 양보다 많아진다. 이 영역은 그레이나 칼라 영상으로 서비스해야 사용자에게 양질로 느껴진다.



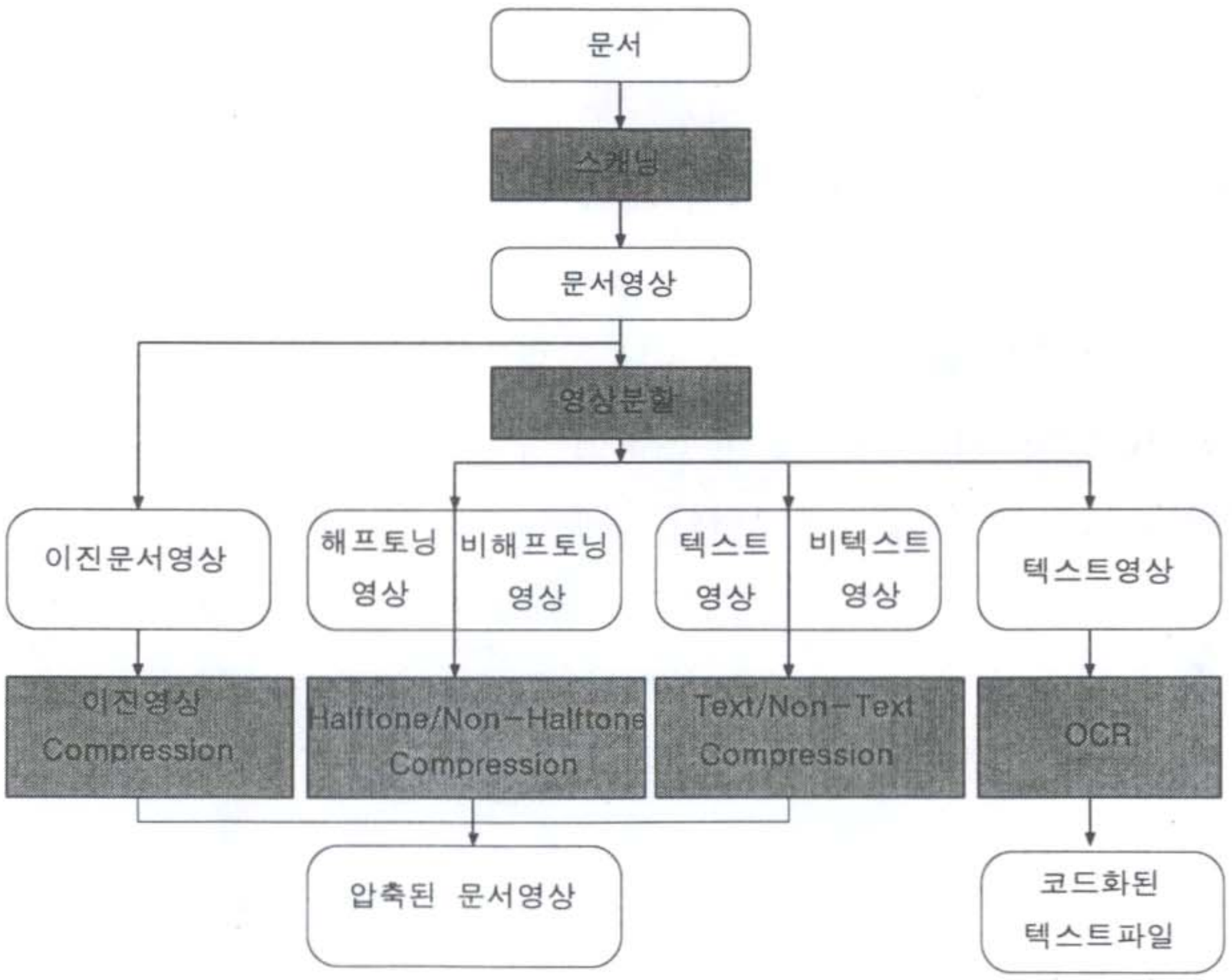
(a) 텍스트 영역 (b) 그래픽 영역 (c) 해프토닝 그림 영역

[그림 4.1] 문서 영상의 영역별 분류

이와 같이 서로 다른 압축 및 디스플레이 특성을 갖는 영역들은 따로 구분하여 처리해야 보다 효율적이다. 문서 영상에서의 영역 분할 방법은 문자 인식 연구 분야에서 많이 연구되어 있다[10, 11, 31].

4.1.2.2 문서영상 데이터베이스 구축 방법

압축 방법론에 따른 문서 영상 데이터베이스를 구축 방법은 [그림 4.2]와 같이 나타낼 수 있다. 여기에서는 문서 영상의 텍스트 영역에 대해 문자 인식 기법을 이용하여 코드화된 텍스트 파일로 저장하는 것도 하나의 구축 방법론으로 간주하였다.



[그림 4.2] 문서영상 데이터베이스 구축 방법론 분류

먼저 가장 저렴하게 문서 영상 데이터베이스를 구축하는 방법으로서 문서를 이진 영상으로 스캐닝하여 적절하게 압축 저장하는 방법이 있다. 이 방법

은 현재 가장 보편적으로 사용되고 있으나, 해프토닝 그림 영역에서의 질 저하 현상과 압축율 저하 문제가 심각하게 나타나고 있어, 이에 대한 해결 방법이 조속히 강구되어야 한다. 다음 절에서 사용자의 사용 장비 변화 및 양질의 서비스를 제공하기 위한 여러 가지 조건들을 고려하여 해결 방법을 제시할 것이다.

다음 방법으로는 해프토닝 그림 영역과 비해프토닝 그림 영역으로 구분하여 처리하는 방법이 있을 수 있다. 해프토닝 그림 영역은 이 영역의 특성을 잘 반영한 압축 방법[25]으로 압축하여 저장하고, 비해프토닝 그림 영역인 텍스트 영역과 그래픽 영역은 이진 영상으로 압축 처리한다. 해프토닝 그림 영역은 필요에 따라 그레이 영상으로 처리하든지 또는 이진 영상으로 처리가 가능하다.

한편 텍스트 영역과 그래픽 영역 사이에도 서로 다른 특성이 있다. 텍스트 영역을 구성하는 문자나 심볼은 정해진 일정한 형태를 가지므로, 이러한 특성을 이용하면 보다 효율적인 압축이 가능하다. 텍스트 영역만으로 구성된 영상인 경우, TIC (Textual Image Compression) 방법[17]을 이용하여 압축하면 매우 높은 압축율을 얻을 수 있다. 그러나 그래픽 영역과 해프토닝 그림 영역이 혼재하는 복합 문서 영상인 경우에는, 텍스트 영역, 그래픽 영역, 해프토닝 그림 영역 각각에 대해 서로 다른 압축 방법을 적용해야 하며, 또한 이들을 효과적으로 묶어서 관리할 수 있는 방법이 필요하다.

문서 영상은 본래의 정보를 왜곡시키지 않고 사용자에게 서비스할 수 있는 장점이 있는 반면에 정보 검색 면에서는 매우 불편하다. 즉, 주요어 색인이나 전문 검색 기법 등을 통한 정보 검색을 수행할 수 없다. 따라서, 문서 영상의 정보를 잘 나타내 줄 수 있는 일부분을 OCR(Optical Character Recognition)을 통해 코드화된 자료로 표현할 필요가 있다. 한편 문자 인식을 통한 텍스트 영상 정보의 코드화는 압축면에 있어서도 매우 좋은 결과를 보이며, 서비스 측

면에서도 사용자에게 다양한 기능을 제공해 줄 수 있어 매우 편리하다. 그러나 문자 인식 기술이 완전하지 못하면, 오히려 사용자에게 왜곡된 정보 또는 손실된 정보를 제공하게 되어 좋지 않다[16, 23].

이와 같은 네 가지 압축 처리 방법론들은 영상 분할이나 문자 인식 기법을 활용하는 방법일수록 다음과 같은 특성이 있다.

- 문서 영상 데이터베이스 구축 시간 및 비용의 증가
- 문서 영상의 압축율 증가
- 서비스 질의 향상

여기에서 알 수 있듯이 최종적으로는 문자 인식을 통한 문서 영상 데이터베이스를 구축할 필요가 있다. 그러나 현실적으로는 위의 방법론들을 단계적으로 확장시켜 구현할 수 있는 측면이 있다. 따라서 먼저 전체 문서 영상의 이진화에 의한 압축 방법을 이용하여 구축한 후, 차후에 단계적으로 확장해 나가는 것도 좋은 접근 방법이 될 수 있다. 이때 이진화에 따른 문제점 및 미흡한 기능을 보완하고, 문서 영상의 스캐닝 해상도 및 압축 방법은 차후의 확장성을 고려하여 결정해야 할 것이다.

4.1.3 이진 문서 영상 데이터베이스 구축 방안

4.1.3.1 이진 문서 영상 스캐닝 해상도 선정

4.1.3.1.1 사용자의 이용 장비 및 이용 환경 변화 분석

최근의 컴퓨터 통신망 기술의 발전은 사회 전반적으로 많은 변화를 가져오고 있다. 요즘은 컴퓨터 전문가 뿐만 아니라 일반인들도 인터넷을 통해 다양한 정보를 검색 활용하고 있다. 이러한 추세는 앞으로 더욱 확대될 것이다.

따라서 현재 뿐만 아니라 앞으로 정보 검색 시스템 사용자들은 대부분이 컴퓨터 단말기와 프린터를 이용할 것이다. 컴퓨터 단말기인 모니터는 보다 높은 해상도를 갖는 것이 활용될 것이며, 프린터로는 고해상도의 잉크젯이나 레이저 프린터가 주류를 이룰 것이다.

문서를 교환하는데 사용하고 있는 팩시밀리는 앞으로도 일시적이면서 긴급한 문서자료의 송수신에 계속 사용될 것이나, 검색 기능이 없으므로 정보 검색 시스템에서의 정보 송수신 장비로는 활용되지 않을 것이다. 따라서 문서 영상 데이터베이스 구축에 사용되던 압축 방법인 CCITT recommendation T.6 는 팩시밀리 기반의 방법이므로 지양되어야 한다. 아울러 팩시밀리의 가격 및 전송 속도와 관련하여 설정되었던 200 dpi 해상도의 문서 영상 스캐닝 방법도 지양되어야 한다. 왜냐하면 이 해상도의 문서 영상은 향후의 주요 출력 장비가 될 레이저 프린터로 출력하였을 경우에 저해상도로 인해 질이 저하되어 보이기 때문이다.

4.1.3.1.2 문서영상의 시각적인 질 평가

구축될 문서 영상의 질을 결정하는 것은 사용자이며, 사용자는 모니터나 레이저 프린터를 통해 나타나는 문서 영상을 보고 결정하므로, 모니터와 레이저 프린터의 특성을 고려하여야 한다. 최근의 개인용 컴퓨터에서는 대부분이 VGA 나 Super VGA 카드를 이용하여 모니터를 구동하고 있으므로, 모니터에서의 디스플레이 모드로는 1024 x 768, 800 x 600, 640 x 480 모드가 지원된다. 모니터는 대략 80-90 dpi 해상도를 갖는 장비이다. 1024 x 768 모드의 모니터에 줌 기능을 사용하지 않고 1:1 로 문서 영상을 디스플레이하였을 경우에, 문서 영상의 해상도에 따른 인간 시각에 의해 느껴지는 화질을 실험적으로 Good(O 표시)과 Bad(X 표시)의 두 단계로 평가하면 [표 4.1]과 같다. 이 실험에서는 해프토닝 그림 부분에 대해서는 평가하지 않았다. 이 실험 결과에서 알 수 있

듯이 모니터에서 양질의 문서 영상을 보려면 150 dpi 이상이 되어야 한다.

레이저 프린터에서 지원하는 기본 해상도는 300 dpi 이거나 600dpi 이다. 이 기본 해상도에 대해 2 의 배수로 해상도가 변환되지 않으면, 인쇄된 결과물이 다소 왜곡되어 보인다. 따라서 75, 150, 300, 600 dpi 등과 같은 해상도의 문서 영상은 제대로 잘 인쇄되지만, 다른 해상도의 문서 영상은 해상도 변환에 있어 부수 효과(side effect)가 발생하여 질이 저하되어 보인다.

한편 그레이 영상이 해프토닝 되어 프린트된 해프토닝 그림 영역은 스캐닝 할 때의 해상도에 따라 문서 영상의 질이 결정된다. 예를 들어, 200 dpi 로 해프토닝된 그림을 300 dpi 로 스캐닝 하거나 300 dpi 로 해프토닝된 그림을 200 dpi 또는 400 dpi 로 스캐닝할 경우에는 블럭 현상(blocking effect)과 같은 부수 효과(side effect)가 발생한다. 이러한 문제를 해결하기 위해서는 고해상도로 스캐닝하여 문서 영상을 구축하는 것이 바람직하다.

여러 해상도의 문서 영상을 인쇄하여 시각적으로 화질을 평가한 결과가 [표 4.1]에 있다. 여기에서 인쇄할 문서 영상은 적어도 300 이나 600 dpi 이 되어야 함을 알 수 있다.

dpi / 장비	75	100	150	200	300	400	600	1200	2400
모니터	X	X	X	O	O	O	O	O	O
레이저 프린터	하	하	하	하	중	중	상	상	상

[표 4.1] 스캐닝 해상도에 따른 모니터 및 프린터 출력의 문서 영상 질 평가

4.1.3.1.3 스캐닝 해상도 선정 요건

문서 영상 데이터베이스를 구축하는데 있어 문서 영상의 스캐닝 해상도를 결정하기 위해서는 다음과 같은 항목을 고려하여야 한다.

(1) 사용자가 양질의 문서 영상으로 인식할 수 있어야 한다.

(2) 레이저 프린터에서 양질의 출력을 얻기 위해 고해상도 문서영상이어야 한다.

(3) 차후에 문자 인식 기술을 이용하여 코드화된 문서를 작성할 수 있도록 충분한 해상도로 스캐닝하여야 한다.

(4) 문서 영상의 데이터 양은 가능하면 작아야 한다.

(5) 사용자가 구비한 장비들의 해상도에 알맞게 해상도 변환이 용이해야 한다.

(6) 문서 영상 데이터베이스 구축 비용을 가능하면 저렴하게 해야 한다

(7) 차후에 영역 분할 및 화질 개선 기법들을 적용할 수 있어야 한다.

(8) 미세한 문자도 판독이 가능해야 한다.

이러한 고려 항목들이 각각 요구하는 조건을 고해상도, 저해상도, 해상도 변환 기능 등으로 구분하여 정리하면 [표 4.2]와 같다. 항목(4)와 (6)은 구축 경비 및 관리 비용을 줄이기 위한 것으로서 저해상도를 선호하고 있지만, 나머지 항목들은 서비스의 질과 장래의 활용성을 고려한 것으로서 고해상도를 선호하고 있다. 따라서 시스템 구축 예산과 지향하는 목표에 따라 스캐닝 해상도가 최종적으로 결정 될 수 있다. 그러나 시스템의 확장성, 서비스 질 향상, 장래의 활용성 등을 고려하면, 고해상도로 문서 영상을 스캐닝할 필요가 있다.

이때 스캐닝 해상도는 [표 4.1]에서의 문서 영상의 질 평가 결과와 항목 (3)에서의 문자 인식을 위한 400 dpi 이상의 해상도, 그리고 해상도 변환기능을 고려하면 600 dpi 로 스캐닝하는 것이 적절함을 알 수 있다.

항목 / 요구조건	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
저해상도				O		O		
고해상도	O	O	O		O		O	O
해상도 변환 기능					O			

[표 4.2] 스캐닝 해상도 결정을 위한 고려 항목들의 요구 사항 분석

4.1.3.2 이진 문서 영상의 압축 방법 선정

문서 영상은 매우 많은 데이터 양으로 표현되므로 적절한 방법으로 압축하여 저장할 필요가 있다. 기존의 대표적인 이진 영상 압축 방법으로는 국제 표준안으로 제시된 CCITT recommendation T.4, CCITT recommendation T.6, CCITT recommendation T.82[26]와 dictionary-based 압축 방법인 LZW[4], 그리고 영상 분할에 의한 텍스트 압축 방법인 Textual Image Compression(TIC) [17]가 있다.

CCITT recommendation T.4 와 CCITT recommendation T.6 는 문서 전송 장치인 팩시밀리를 위한 압축 기법으로서, 이들은 각각 1980 년과 1984 년에 제정되었다. 이들은 텍스트 및 그래픽으로 구성된 문서의 경우에는 압축율이 좋은 반면에, 해프토닝 그림에 대해서는 오히려 원래의 데이터 양보다 압축된 데이터의 양이 더 많아진다. 이들 방법에서는 1 차원적 또는 2 차원적 Run-length Coding 방법과 Huffman Coding 방법을 이용하고 있다.

CCITT recommendation T.82 는 1993 년에 제정된 것으로, JBIG 으로 더 많이 알려져 있다. JBIG 은 기본적으로 이진 영상을 위한 압축 기법으로서, 그레이 영상에 대해서는 비트 평면(bit plane)으로 분할하여 압축할 수 있다. 그레이 영상에서 픽셀당 비트수가 5 이하인 경우에는 JBIG 압축 방법이 보다 우수한 압축 효율을 나타내고, 픽셀당 비트수가 6 이상인 경우에는 JPEG 압축 방법이 보다 우수한 압축 효율을 나타낸다. 이 압축 방법에서는 Adaptive 2D coding

모델과 Adaptive arithmetic coding 방법을 채택하고 있는데, 이 방법은 앞에서의 CCITT recommendation T.4 나 CCITT recommendation T.6 와 달리 해프토닝 영상에 대해서도 상대적으로 압축율이 매우 좋다. 앞으로의 문서에 해프토닝 그림이 보다 많이 사용될 것임을 고려하면 JBIG 을 정보 검색 시스템에서의 문서 영상 압축 방법으로 정하는 것이 바람직할 것이다.

JBIG 다음으로 압축 효율이 좋은 LZW 방법은 코드화된 텍스트를 사전 기반의 coding 방법으로 압축하는 것으로서 그레이 영상 및 이진 영상에도 적용시킬 수 있다. 이진 영상의 경우 LZW 압축 방법이 CCITT recommendation T.4 나 CCITT recommendation T.6 압축 방법보다 압축 효율이 좋음을 실험을 통해 알 수 있었다.

TIC 방법은 LZW 와 유사하게 dictionary-based 접근 방법을 이용하는 것으로서, 텍스트 영역에서의 새로운 문자열들을 dictionary 에 비트맵으로 저장해 놓고, 이것의 인덱스를 이용해 coding 하는 압축 방법이다. 이 방법은 소수의 문자체 및 문자 세트로 구성된 텍스트 영상의 경우에 효과적으로 적용할 수 있으며, 텍스트 위주의 영상인 경우는 압축 효율이 매우 높아 JBIG 보다 더 높은 압축 효율을 나타낸다. 그러나 그래픽이나 해프토닝 그림 영역에는 적용하기가 곤란하다. 또한 영어와 달리 한글의 경우에는 같은 자음이더라도 초성일 때와 종성일 때의 위치 및 모양이 다르기 때문에 dictionary 가 엄청나게 커지게 되는 문제가 있다. 앞으로의 문서에서 그래픽이나 해프토닝 그림이 자주 사용 될 것임을 고려한다면 보다 많은 기술이 개발되어야 이 방법을 사용할 수 있을 것이다.

영역 분할을 해야 하는 TIC 방법을 제외한 나머지 방법들에 대해 기존의 연구 결과와 실제의 실험 결과에 의한 압축율을 비교하면 다음과 같다.

JBIG > LZW > T.6 > T.4

JBIG 이 정보 검색 시스템에서의 이진 문서 영상의 압축 방법으로서의 좋은

점을 정리하면 다음과 같다.

(1) 국제 표준안이므로 자료의 공유성이 좋다.

(2) CCITT recommendation T.4 나 CCITT recommendation T.6 와는 달리 해프토닝 그림 영역에 대해서도 압축 효과가 있어 압축율이 매우 좋다.

(3) Resolution Reduction 기능이 있어 해상도 변환이 용이하다.

(4) 그레이 영상에도 적용시킬 수 있다.

(5) 해상도의 증가에 따른 압축 데이터의 크기는 선형적으로 증가한다.

(6) Prototype 의 source code 를 쉽게 구할 수 있다.

4.1.3.3 JBIG 을 이용한 이진 문서 영상의 압축

4.1.3.3.1 JBIG 표준안 소개

이진 영상의 비손실 압축 방법 중에서 압축 효율이 가장 좋은 JBIG 은 순차적 모드와 점진적 모드로 압축과 복원이 가능하다. 점진적 모드의 경우 다른 해상도를 가진 여러 출력장치에 효율적인 서비스가 가능하며, 저속 통신망을 통한 응용에서 CRT 화면을 통한 영상정보의 브로우징시 우수한 성능을 제공한다. 그러나 점진적 모드에서는 이전 상위 해상도 (high resolution)에 대한 정보를 보관하는 page buffer 가 있어야 한다.

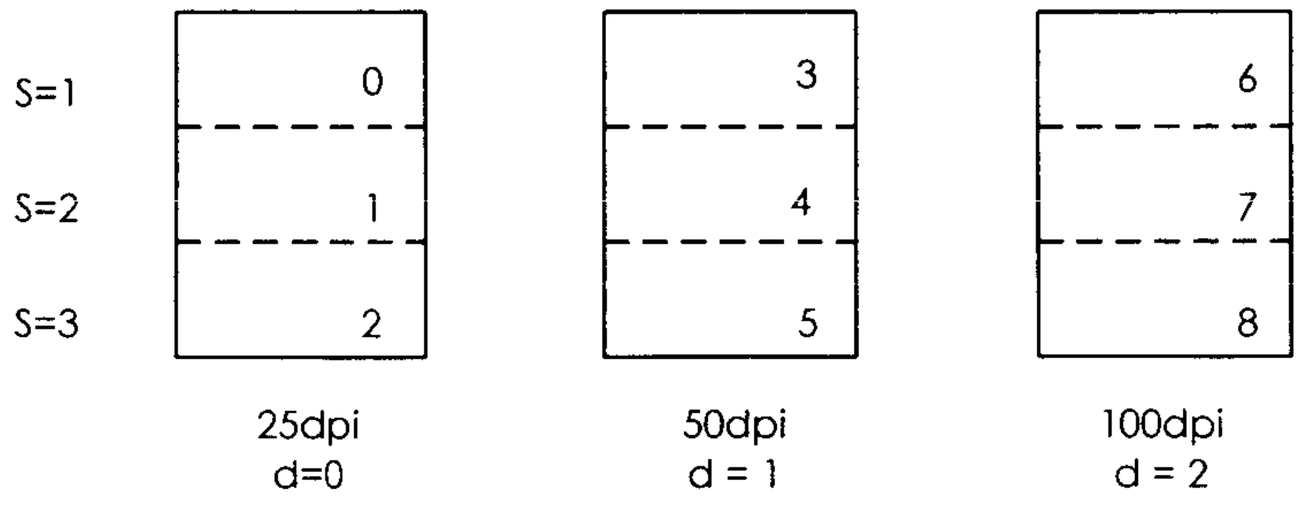
이진 영상의 압축 방법인 JBIG 은 비트평면(bit plane)이 5 이하인 경우에는 압축 효율이 높지만 6 이상인 경우에는 JPEG 압축 방법이 더 효율적이다.

4.1.3.3.1.1 스트립의 순서 및 JBIG 화일의 구성

점진적 모드에서 전체 영상에 대해 하나의 page buffer 를 가질 경우 큰 메모리 공간이 필요하게 되어 영상 복원시 어려움이 있다. 이런 문제를 해결하기 위하여 전체 영상을 적절한 크기로 분할하여 압축 및 복원하게 되는데, 이

때 각 해상도별로 수평 방향으로 분할된 영상을 스트립(strip)이라고 한다. 이진 영상의 경우에는 비트 평면이 하나이지만 그레이 또는 칼라 영상의 경우 비트 평면이 2 개 이상이므로, 그 다중 비트 평면에 대해서도 동일한 크기의 스트립으로 나눈다. 이렇게 분할된 영상정보의 단위를 압축 및 복원할 때, 그 순서를 지정하는 변수가 SEQ, HITOLO, ILEAVE, SMID 이다. 여기서 SEQ 변수는 각 해상도에 대해 순차적 또는 점진적인 것에 대한 우선순위를 지정하고, HITOLO 변수는 각 해상도에서 스트립의 순서가 높은 것 또는 낮은 것에 대한 우선 순위를 지정한다. 그리고 ILEAVE 및 SMID 변수는 다중 비트 평면에서 스트립의 순서를 지정하는 것이다.

HITOLO	SEQ	Example order
0	0	0,1,2,3,4,5,6,7,8
0	1	0,3,6,1,4,7,2,5,8
1	0	6,7,8,3,4,5,0,1,2
1	1	6,3,0,7,4,1,8,5,2



[그림 4.2] 하나의 비트 평면을 갖는 이진 영상에서 스트립의 순서

BIE	
BIH	BID
Varies	Varies

BIH											
D_L	D	P	-	X_D	Y_D	L_0	M_X	M_Y	Order	Options	DPTABLE
1	1	1	1	4	4	4	1	1	1	1	0 or 1728

BID						
Floating marker segment(s)	$SDE_{s,d,p}$	Floating marker segment(s)	$SDE_{s,d,p}$	···	Floating marker segment(s)	$SDE_{s,d,p}$
Varies	Varies	Varies	Varies	···	Varies	Varies

Order							
MSB	···						LSB
-	-	-	-	HITOLO	SEQ	ILEAVE	SMID
1/8	1/8	1/8	1/8	1/8	1/8	1/8	1/8

Options							
MSB	···						LSB
-	LRLTWO	VLENGTH	TPDON	TPBON	DPON	DPPRIV	DPLAST
1/8	1/8	1/8	1/8	1/8	1/8	1/8	1/8

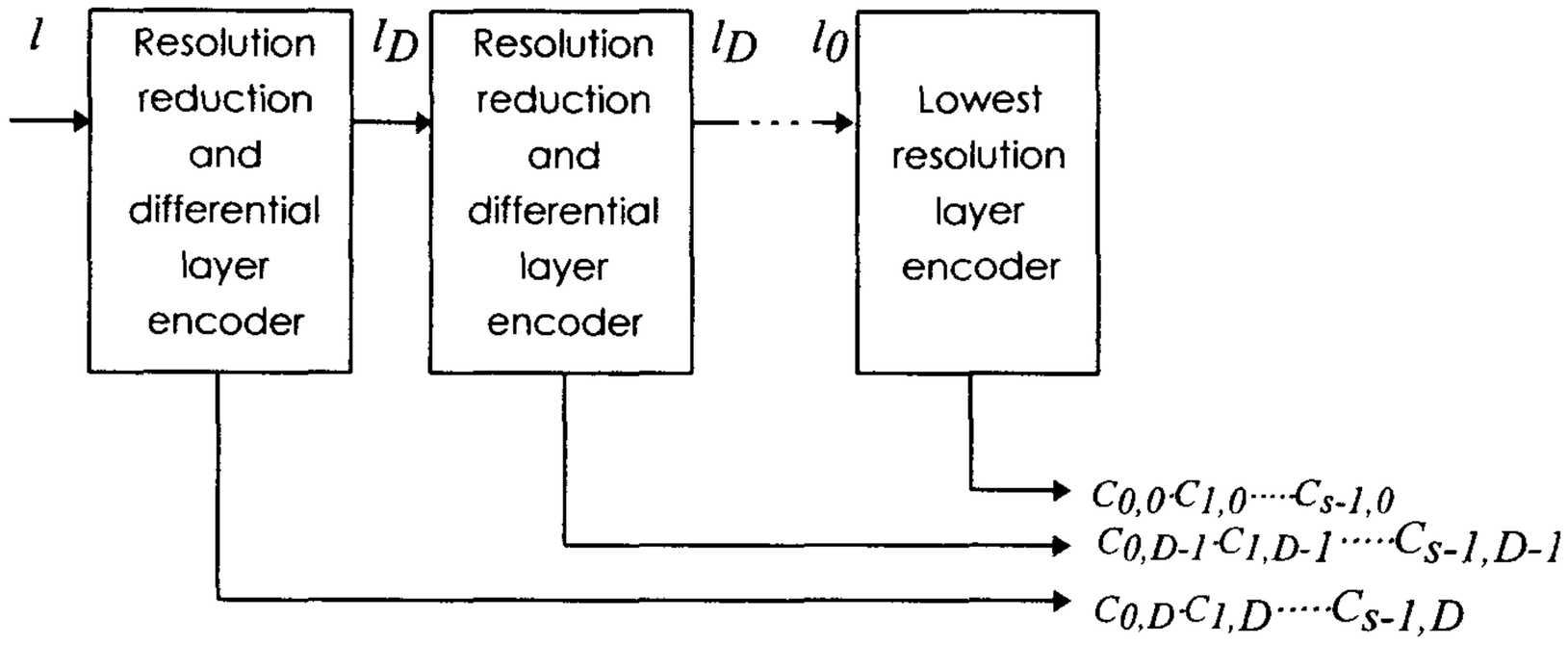
[그림 4.3] JBIG 화일 구성

4.1.3.3.2 JBIG 표준안에서의 압축 방법

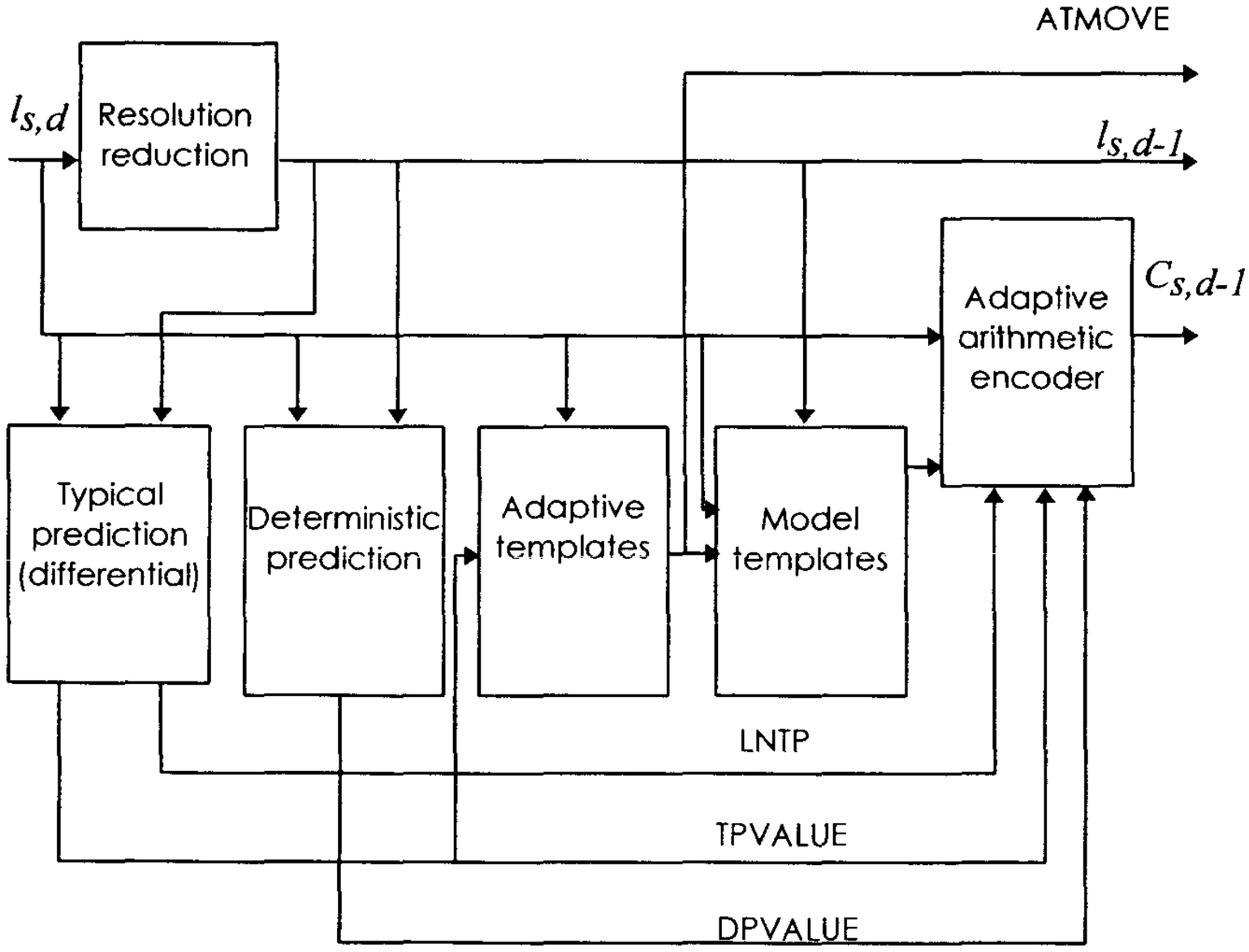
4.1.3.3.2.1 압축기

점진적 모드의 단계에 따라 압축기의 단계가 결정된다. 그러므로 순차적 모드에서는 최소 해상도 단계(lowest resolution layer)의 압축기만 사용된다. 최

소 해상도 단계에서는 결정적 예측을 실행하지 않는다. 여러 단계에 걸쳐서 해상도 감소가 이루어지더라도 물리적인 압축기는 하나만 구현하고 여러 단계에 걸친 압축 과정을 순환적으로 실행한다.



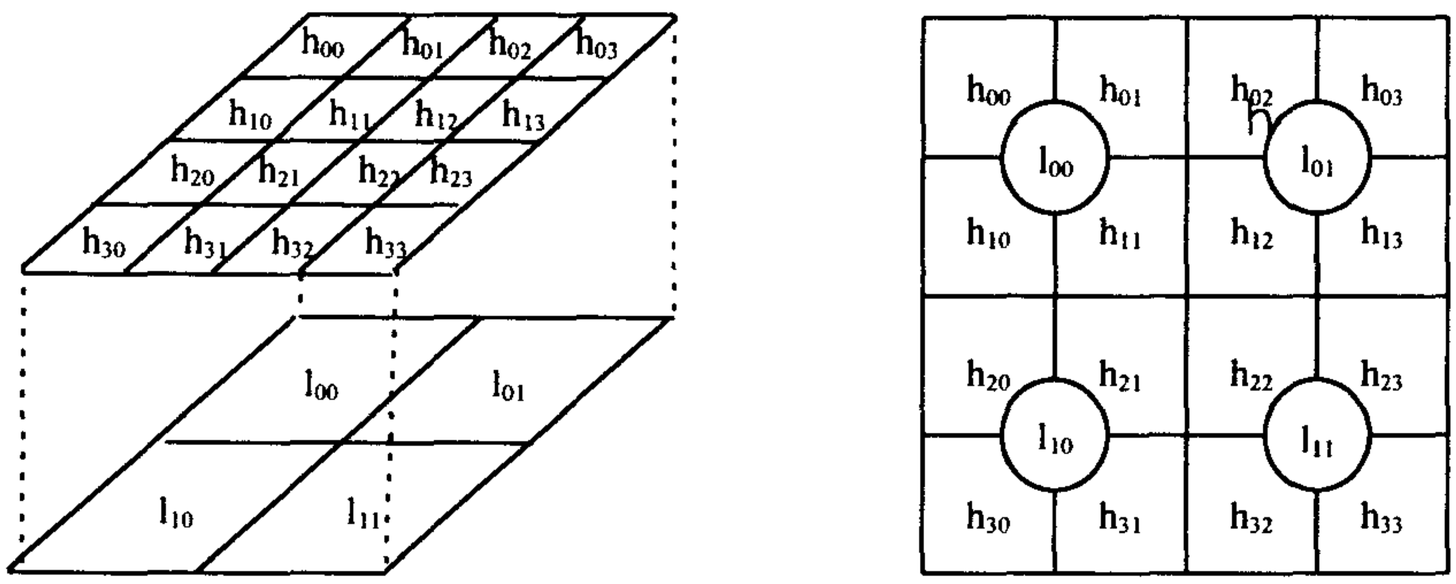
[그림 4.4] 압축기의 전체구성 (decomposition of encoder)



[그림 4.5] 한 단계에 대한 압축기의 세부 구성

(1) 해상도 감소 (Resolution Reduction)

일반적인 방법인 2 x 2 블럭에 대한 평균값으로 해상도를 감소시키는 방법이 이진 영상에서는 가능하지 않으며, subsampling 방법으로 해상도를 감소시킬 경우에는 감소 작업이 두 번 이상 반복되면 손실되는 정보의 양이 많아진다. 그러므로 보다 더 좋은 질의 문서 영상을 얻기 위해, 여러 가지의 예외 경우를 고려하여 하위 해상도(low resolution)에서의 픽셀값을 테이블 참조 방법으로 구한다.



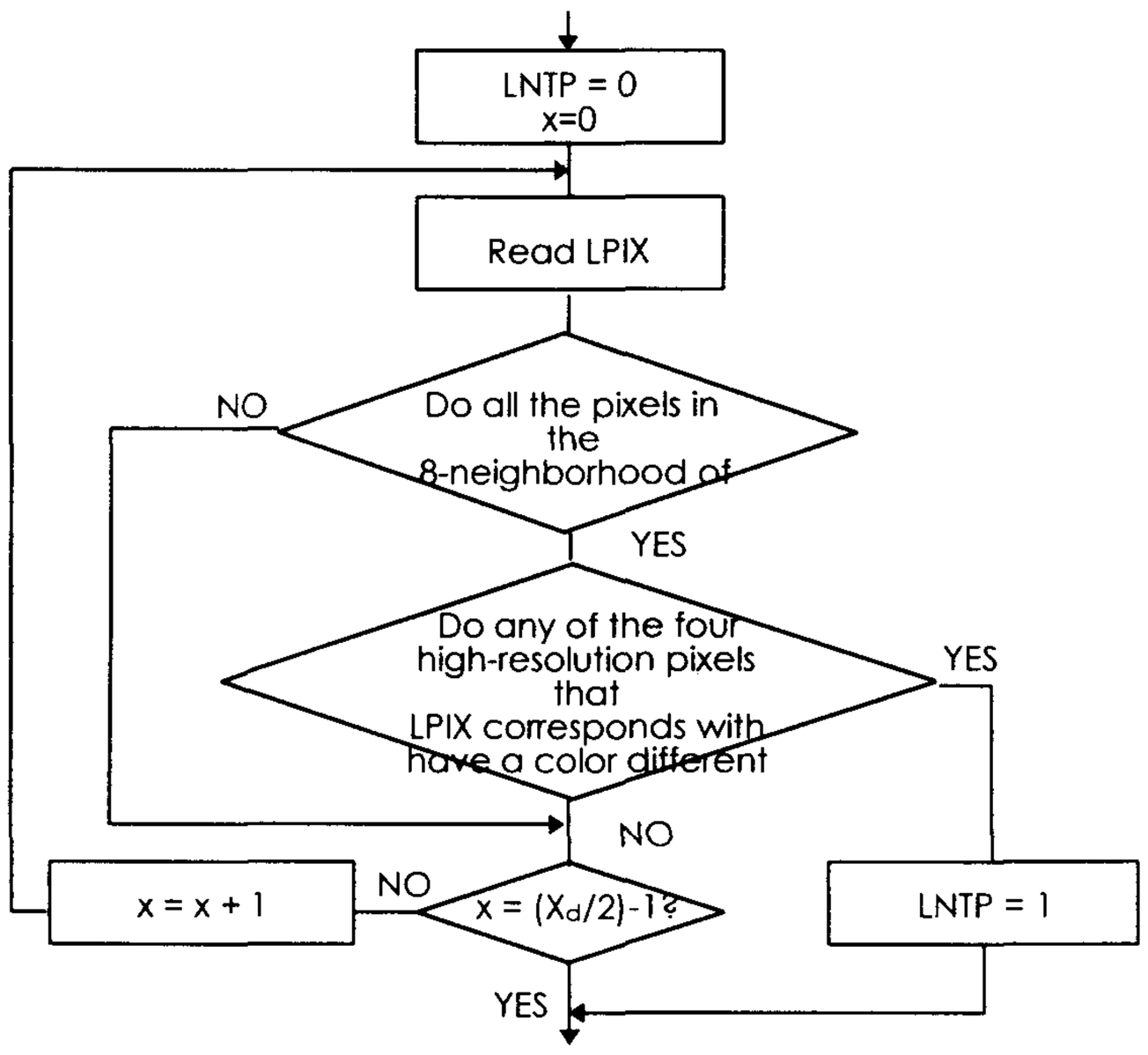
(a) 3 차원적 표현 (b) 2 차원적 표현

[그림 4.6] 상위 해상도와 하위 해상도 사이에서 픽셀의 상호관계

(2) 정형적 예측 (Typical Prediction)

하위 해상도에서 중심 픽셀을 기준으로 이웃하는 여덟 개의 픽셀이 동일한 값을 가지고 상위 해상도에서 관련되는 4 픽셀 모두가 동일한 값을 가지는 경우, 그 중심 픽셀을 정형적이라고 한다. 그리고 그 라인의 픽셀이 모두 정형적인 경우 상위 픽셀의 관련 라인도 동시에 정형적이므로 압축시 그 라인에 대한 데이터는 생략하고, 그 생략된 데이터의 상태 정보를 플래그 비트에 저

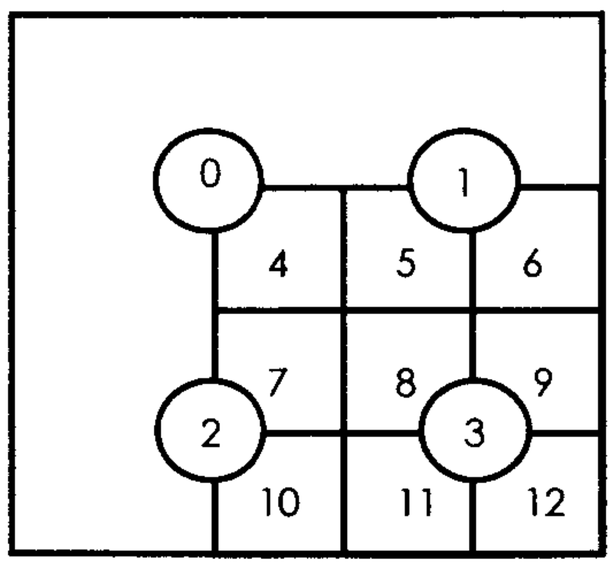
장한다. 이와 달리 하위 해상도에서 여덟 개의 이웃하는 픽셀이 동일한 값을 가지더라도 상위 해상도의 관련 4 픽셀중 하나 또는 그 이상이 다른 값을 가질 경우 하위 해상도의 그 픽셀은 비정형적이라고 한다. 또한, 그 픽셀과 관련있는 상위 해상도의 라인도 비정형적이므로 압축시 생략할 수 없다. 그러므로 정형적 예측을 이용함으로써 압축 효율을 높일 수 있다.



[그림 4.7] 정형적인 라인을 판단하는 과정

(3) 결정적 예측 (Deterministic Prediction)

결정적 예측도 압축 효율을 높이기 위한 방법으로서, 해상도 감소에서와 마찬가지로 관련있는 픽셀의 값에 따라 테이블 참조 방식으로 결정되어진다. 결정적 예측에 의한 DPVALUE는 0, 1, 2의 값을 가질 수 있는데, 0인 경우는 해당 픽셀이 배경(backgroun:0) 값을 가지고, 1인 경우는 대상물(foreground : 1) 값을 가지며, 2인 경우는 결정적 예측을 하지 않는다는 정보를 가지고 있다.



[그림 4.8] 결정적 예측에 의해 사용되는 픽셀의 번호

Phase	Target pixel	Reference pixels	Number of hits with default resolution reduction
0	8	0,1,2,3,4,5,6,7	20
1	9	0,1,2,3,4,5,6,7,8	108
2	11	0,1,2,3,4,5,6,7,8,9,10	526
3	12	0,1,2,3,4,5,6,7,8,9,10,11	1044

[표 4.3] 각 공간의 페이즈(phase)에 대한 결정적 예측의 픽셀 값

(4) 적응적 형판 (Adaptive Template)

산술 부호화기(arithmetic coder)에서 사용할 주변 정보(context)를 얻기 위해 모델 형판(model template)이 사용되는데, 이 주변 정보의 효용성을 높이기 위해 적응적 픽셀(Adaptive Pixel)을 하나 둔 것이 적응적 형판이다. 이 적응적 픽셀은 필요에 따라 위치를 변경할 수 있다. 적응적 형판은 주기성을 잘 반영할 수 있어 해프토닝 그림의 경우에 최대 80%의 코딩 이득(gain)을 얻을 수 있다. 적응적 픽셀의 위치가 변경되면, ATMOVE 제어 신호를 통해 변경 사항을 전달해 준다. 그러나 이러한 위치 변경은 매우 드물게 발생한다.

4.1.3.4 실험 및 토의

본 연구에서 제안한 이진 영상 정보의 압축 및 복원 방법인 JBIG 은 기존의 압축 방법들 중에서 가장 압축 효율이 높은 것으로 나타났으며, 해상도 감소 기법을 이용하여 적절한 해상도의 영상 정보를 제공하는 기능이 매우 우수한 것으로 나타났다.

모니터를 통해 사용자는 주로 문서를 검색하고 개략적인 문서 영상 확인 작업을 할 것이다. 따라서 고해상도의 문서 영상을 제공받기 위해 많은 시간과 디스플레이 시간의 지연 등을 감수할 필요가 없다. 150 dpi 의 문서 영상을 기본 모드로 제공하면, 사용자는 문서 영상에서의 문자를 쉽게 읽을 수 있다. 그러나 모니터의 한 화면에 문서 영상 전체가 출력되지 않으므로 스크롤 및 패닝 기능을 제공하여야 한다. 만약 사용자가 영상 정보 전체의 윤곽만을 파악하면서 빠른 탐색을 원하면, JBIG 의 해상도 감소 기능을 이용해 해상도를 75 dpi 로 낮추어 한 화면에 한 페이지 전체가 디스플레이 되도록 한다. 이와 같이 낮은 해상도의 영상정보를 이용해 검색을 하게 되면 통신 시간 및 자료 검색 시간을 절약 할 수 있다.

만약 영상 정보를 프린터로 출력하고자 할 경우, 사용자는 자신의 프린터 해상도를 고려하여 고해상도의 영상 정보를 요청하면, 고해상도의 영상정보를 정보 검색 시스템으로부터 제공받아 출력할 수 있도록 한다.

이러한 방식으로 동작하는 JBIG 압축 복원기를 개인용 컴퓨터에서 Win95 프로그래밍을 이용해 구현해 보았다. 아직 압축 및 복원 속도에 대한 고려가 충분히 되지 않아 전반적인 처리속도가 떨어지나 원하는 결과는 확인할 수 있었다. 압축 복원기의 처리속도는 고속 스캐너에서의 처리속도와 균형을 이루어야 한다. 그러나 시중에서 판매되고 있는 고속 스캐너에는 JBIG 으로 동작하는 전용 하드웨어 및 전용 소프트웨어가 마련되지 않아 현실적으로 문제

가 될 수 있다. 이 문제점은 당분간 기존에 사용되던 CCITT recommendation T.6 방식을 이용해 스캐닝을 행한 후, 스캐닝된 문서 영상에 대해 JBIG 으로 변환하는 프로그램을 개발하여 사용하면 해결될 수 있다.

4.1.4 칼라 및 그레이 영상의 시각 특성을 고려한 압축 방법

4.1.4.4.1 인간 시각 체계의 특성

손실 압축(lossy compression) 기법을 사용하여 이미지를 압축하는 경우, 가장 먼저 고려되어야 할 사항은 압축된 이미지와 원시 영상을 비교하였을 때, 압축 과정에서 발생하는 왜곡을 사람이 인식할 수 없을 정도로 압축하면서 압축 효율을 늘리는 것이다. 압축된 영상의 질을 평가하는 궁극적인 평가 기준은 인간에 의해 측정되어 진다고 할 수 있으므로 압축 기법에 인간의 시각 체계의 특성을 분석하고 반영함으로써 압축된 영상의 질을 향상시킴과 동시에 압축율을 증가시킬 수 있다면, 이것은 매우 중요한 일이다.

그러면 먼저 인간의 시각 체계의 특성을 살펴보도록 하겠다. 인간이 빛의 밝기에 대한 변화를 인지하는 정도는 웨버의 법칙[27]으로 설명되는 비선형적인 특성에 의해 표시되어진다. 그리고 특히 너무 밝거나 너무 어두운 곳에서는 밝기의 변화를 쉽게 인지할 수 없다는 것도 알려져 있으며, 이것을 휘도 마크킹이라 부른다[3,24].

영상의 질을 측정하는데 커다란 영향을 미치는 것으로는 공간적 마스킹(spatial masking)[2,3,9,24,28]이라고 불리는 특성이 있다. 이것은 영상의 복잡도가 높거나 대비 효과(contrast)가 큰 부분에서는 압축 과정에서 발생한 왜곡이 쉽게 눈에 인지되지 않는 특성을 말한다. 그리고 안구의 망막에 있는 빛을 감지하는 세포가 아주 미세한 부분까지도 구분해 낼 수 있을 정도로 높은 밀도

로 분포되어 있지 않기 때문에 망막을 통과하면서 저주파 필터링(low-pass filtering)되는 효과를 가져오며, 망막 세포에서의 상호 억제 작용에 의해서는 고주파 필터링 효과가 발생된다. 이와 같은 효과들의 상호작용으로 전체적으로는 밴드패스 필터링 매커니즘이 적용된다고 볼 수 있고, 이것을 공간적 필터링이라 부른다[9].

4.1.4.2 영상 왜곡의 종류와 분석

이미지를 블럭 단위로 변환하여 높은 압축율로 압축했을 때, 압축된 이미지를 직접 관찰하여 인지할 수 있는 왜곡은 변환의 종류에 따라서 차이가 있겠지만, 본 연구에서 사용하는 방법은 JPEG 이므로 이 압축 방법을 대상으로 하여 발생할 수 있는 왜곡 현상에 대하여 분석하였다.

압축된 영상에서 인간이 인지할 수 있는 왜곡의 종류는 압축된 이미지의, 블럭과 블럭 사이의 경계 부분에서 불연속성이 뚜렷이 나타나는 블럭 효과, 블럭내의 자세한 부분들이 많이 흐려져 보이는 번짐 효과, 비교적 평평한 배경을 가지는 물체의 경계 부근에서 원래의 이미지에는 없었던 잔물결 무늬가 복원된 영상에 나타나는 물결 효과 등이 있다. 그러면 이들 각각에 대해서 살펴보도록 하겠다.

4.1.4.2.1 블럭 효과

블럭 효과는 블럭들 사이의 경계 부근에서 불연속성이 나타나는 왜곡의 한 종류이다. 이것은 이미지를 블럭으로 나누어서 높은 압축율로 압축하는 경우에 나타나며, 블럭 효과가 특히 눈에 잘 띄는 곳은 사람의 얼굴 부분이나 하늘의 구름 부분과 같이 이미지의 복잡도가 매우 낮으면서 색이 천천히 변화하는 부분이다. 즉, DCT 블럭에서 블럭에서의 밝기의 변화를 나타내는 AC 성분의 에너지가 그다지 크지 않은 블럭들이다.

블럭 효과가 발생하는 기본적인 원인으로 변환된 각 블럭의 DC 성분을 들 수 있다. DC 성분은 해당 블럭의 평균적인 밝기를 나타낸다. 그러므로 이 DC 성분에 대한 양자화 손실이 많이 발생하게 되면 이웃한 블럭들 사이에서 밝기가 갑자기 변화화는 불연속성이 나타나게 되고 이것이 블럭 효과로 연결된다. 또한 DC 성분과 가까이 위치한 AC 성분에서의 손실일수록 블럭 효과에 영향을 많이 줄 수 있다. 그러나 블럭 효과가 발생한 블럭이라 할지라도 블럭의 밝기가 너무 밝거나 너무 어두운 경우, 휘도 마스킹의 영향을 받아 블럭 효과를 인지할 수 없게 된다.

4.1.4.2.2 번짐 효과

번짐 효과는 복잡한 내용을 많이 포함하고 있는, 즉 AC 성분이 많은 블럭에서 양자화 손실이 많이 발생하여 이미지의 상세한 정보가 손실되어 흐리게 보이는 현상을 말한다. 여기에서 모든 AC 성분에서의 양자화 손실이 같은 정도로 번짐에 영향을 미치는 것이 아니므로, 번짐 효과에 영향을 특히 많이 주는 AC 성분들을 찾아내고, 이 영역에 대한 에너지 분포 특성을 살펴볼 필요가 있다.

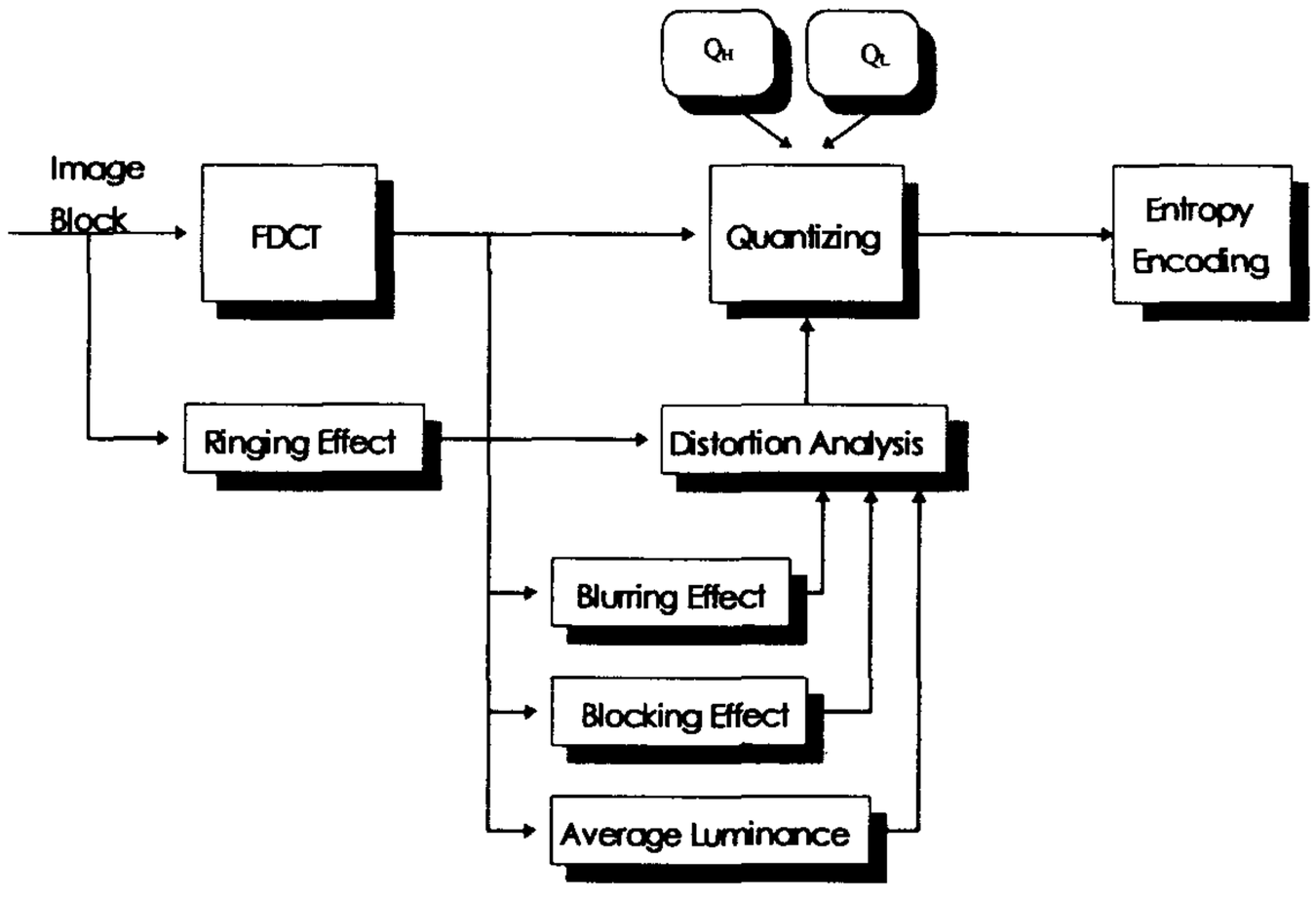
그러므로 주어진 블럭이 양자화 된 후에 번짐 효과가 나타날 것인지에 대한 판단은 저주파수 성분에서 중간 정도의 주파수 성분들에 대한 에너지의 양이 어느 정도 되는가를 계산하여 봄으로써 가능하다. 즉 중간 정도의 주파수 성분들에 대한 에너지가 아주 낮을 경우에는 블럭 자체가 번짐 효과를 발생시킬 만한 복잡한 정보를 많이 가지고 있지 않다는 말이 되고, 또 반대로 에너지가 많이 집중되어 있을 경우에는 공간적 마스킹 때문에 번짐 효과가 심하게 나타나지 않는다고 판단할 수 있다. 그리고 번짐 효과가 휘도 마스킹 때문에 실질적으로는 눈에 관찰되지 않는 블럭도 있다.

4.1.4.2.3 물결 효과

물결 효과란 밝기의 변화가 비교적 완만한 배경과 어떤 물체 사이에 뚜렷한 경계가 포함되어 있는 블럭을 높은 압축율로 압축하였을 때 나타날 수 있다. 즉, 압축된 이미지의 물체와 배경의 경계 근처에서 원래 이미지에는 없었던 잔물결 모양의 무늬가 육안으로 관찰될 정도로 어느 정도 뚜렷이 나타나는 효과를 말한다. 이 효과 역시 복원된 이미지의 질을 저하시키는 중요한 왜곡들 중의 하나이다. 이러한 블럭들은 배경과 물체 사이의 뚜렷한 경계 부분으로 인하여 비교적 낮은 주파수 영역에 에너지가 집중되어 있는 경우이다. 그러나 낮은 주파수 성분들의 기저 함수가 가지고 있는 파형 중에서 경계 부분과 상관 없는 파형을 상쇄시켜 줄 보다 높은 주파수 영역의 성분들이 양자화 되는 과정에서 없어져 버리기 때문에 물결 효과가 발생한다. 물결 효과 역시 휘도 마스킹의 적용을 받는다.

4.1.4.3 시각 특성을 고려한 JPEG 압축 효율의 개선 방법

인간 시각 체계의 특성을 분석하여 영상 압축 방법에 적용한 시각 적응적 영상 압축은 각 블럭 단위로 저손실 혹은 고손실로 압축할 지를 적응적으로 결정하는 방법이다. 이 방법은 전체 영상에 대해서 손실을 많이 허용하면서 높은 압축율로 압축하되, 인간이 보아서 인지할 수 있을 정도의 왜곡 현상이 발생할 소지가 있는 블럭을 미리 예측하여 저손실로 압축하는 방법이다. 전체적인 압축 과정은 [그림 4.9]와 같다.



[그림 4.9] 시각 적응적 이미지 압축 과정

압축 과정을 간단히 살펴보면, 각 블럭별로 인간이 인지할 수 있는 왜곡의 유무를 왜곡량 분석 과정을 통하여 조사한다. 그리고 분석된 결과를 이용하여 각 블럭에 저손실 또는 고손실 양자화 테이블 중의 어느 것을 사용할 것인지를 적용적으로 선택하여 양자화하게 된다. 왜곡 현상중 블럭 효과 및 번짐 효과에 관한 분석은 FDCT를 거친 후의 변환 계수에 대하여 적용되며, 물결 효과에 관한 분석은 FDCT를 거치기 전의 원시 블럭에 대하여 적용된다. 그리고 블럭의 평균 밝기를 나타내는 DC 성분을 이용함으로써 각 블럭이 휘도 마스킹에 적용받는 지의 여부를 조사하였다. 본 과제에서는 손실을 줄여서 양자화하기 위한 조건으로 각 블럭에 블럭 효과, 번짐 효과, 물결 효과 중에서 적어도 한가지 이상이 발생하는 경우를 선택하였다. 그러나 앞의 세가지 왜곡 중에서 한가지 이상이 발생하였다고 하더라도 휘도 마스킹이 적용되는 블럭의 경우에는 저손실로 보상하지 않고 그대로 고손실로 하였다.

4.1.4.3.1 휘도 마스킹을 위한 평균 휘도값의 범위

DC 성분은 한 블럭의 평균적인 밝기를 나타내고, 값의 범위는 일반적으로 -1000 ~ 1000 정도이다. 이때 평균적인 밝기가 너무 어둡거나 밝은 경우에는 이미지의 내용을 판별하기 어렵다고 볼 수 있으며, 이러한 블럭에서는 왜곡이 많이 발생하더라도 사람의 눈에 쉽게 인지되지 않는다. 그러므로 왜곡 분석의 결과에 관계없이 블럭이 너무 밝거나 어두운 블럭에 대해서는 손실을 많이 허용하여 압축함으로써 압축율의 향상을 기하였다.

4.1.4.3.2 블럭 효과 발생 블럭의 추출 방법

블럭 효과는 이미지상의 각 블럭과 블럭 사이에 경계선이 뚜렷하게 드러나는 현상을 말한다. 이러한 현상은, 아주 복잡한 블럭에서는 높은 압축율로 압축하더라도 눈에 쉽게 띄지 않는 특성을 가지고 있으며, 이미지가 그다지 복잡하지 않고 밝기의 변화 등이 서서히 이루어지는 사람의 얼굴 부분이나 하늘의 구름 부분과 같은 블럭에서 쉽게 관찰된다. 그러므로 주어진 블럭을 높은 압축율로 압축했을 때, 블럭 효과가 발생할 지의 여부를 판별하기 위해서는 블럭이 얼마나 평평한가를 계산하여 봄으로써 블럭 효과를 예측할 수 있다. 이를 계산하기 위해서 Slow 와 Smid 를 실험을 통하여 다음과 같이 정의하였다.

$$S_{low} = |D(1)| + |D(2)|$$
$$S_{mid} = \sum_{i=3..20} |D(i)| \quad \text{---------- (1)}$$

즉 D(1), D(2)는 AC 성분 중 주파수가 가장 낮은 저주파수 성분의 에너지양을 나타내는데 Slow 가 너무 작을 경우에는 밝기의 변화가 너무 없기 때문에 블럭 효과가 나타나지 않고, 또한 값이 너무 큰 경우에는 뚜렷한 경계 성분이

존재하게 되어 블럭 효과가 나타나지 못하므로 고손실로 압축할 수 있다. 또한 Smid 는 나머지 AC 성분들의 값의 합을 구하여 계산되어지는데, 밝기 변화의 복잡도를 나타낸다. Smid 가 너무 큰 경우는 블럭 내부에 복잡한 정보를 많이 포함하고 있는 것을 나타낸다. 이 경우에도 블럭 효과가 나타나지 않게 되어 해당 블럭을 고손실로 압축할 수 있게 된다.

4.1.4.3.3 번짐 효과 발생 블럭의 추출 방법

DCT 는 블럭 에너지의 대부분을 중간 정도 이내의 계수들로 집중시키는 특성을 가지고 있어, JPEG 표준안의 양자화 테이블로 양자화하더라도 중간 이후의 계수들은 보통 0 으로 양자화 된다. 그러므로 고주파수 성분은 각 블럭의 번짐 효과를 발생시키는데 별다른 영향을 미치지 못한다고 볼 수 있다. 이에 따라, AC 성분의 첫번째 값부터 어느 정도 중간값까지의 값만을 더함으로써 블럭 내부의 번짐 효과 발생 여부를 판단할 수 있다. 즉 블럭 내부의 복잡도를 아래와 같이 조사함으로써 번짐 효과의 발생 여부를 판별할 수 있다.

$$S_{blur} = \sum_{i=1..35} |D(i)| \quad \text{---------- (2)}$$

Sblur 값이 너무 작으면 블럭 자체가 복잡하지 않기 때문에 번짐 효과가 처음부터 발생할 수 없으며, 반면에 너무 클 경우에는 공간적 마스킹 때문에 번짐 효과는 눈에 잘 관찰되지 않는다.

4.1.4.3.4 물결 효과 발생 블럭의 추출 방법

물결 효과가 발생할 수 있는 블럭은 비교적 평평한 블럭에서 블럭의 가장자리나 모서리 쪽에서 뚜렷한 경계 부분이 있는 블럭이다. 이 뚜렷한 경계 부분이 블럭의 가운데 영역에 자리잡고 있을 경우에는 계산상으로는 물결 모양의 무늬가 발생할 조건이 되지만, 실질적으로 그 무늬가 시각적으로 관찰될

만큼 8x8 블럭내에서 형성될 공간이 없다. 물결 효과의 분석을 위해서는 이전의 왜곡 분석과는 달리 변환하기 전의 공간(spatial domain)에서 분석을 하였다. 왜냐하면 주파수 공간에서는 경계 부근이 가장자리에서 발생하였는지, 아니면 블럭의 중간 부분에서 발생하였는지에 대한 구분을 하기가 어렵기 때문이다. 물결 효과를 분석하기 위한 영역의 정의는 [그림 4.10]과 같다.

[그림 4.10] 물결 효과 분석을 위한 영역 정의

물결효과가 발생하기 위해서는 우선 경계 부분을 제외한 나머지 부분은 비교적 평평한 상태여야 한다.

$$\sigma_T^2 \le T_{\sigma^2} \quad \text{----------} \quad (3)$$

식 (3)의 조건을 만족하는 블럭에 대해 블럭의 가장자리에서의 경계 부분 존재 여부를 조사하여야 한다. 이를 위해 [그림 4.10]에서 정의한 9 개의 영역에 대해서 각 영역별로 픽셀들의 Y 성분에 대한 평균값들을 구한다. 그런 후 영역 R0 의 평균값과 나머지 8 개 영역의 평균값의 차이를 구하여 그 값들이 모두 일정치 이하이면 이 블럭은 물결 효과가 발생하지 않는 것으로 간주한다. 그런데 차이값이 일정치 이상인 영역들이 존재하는 경우라도 여러 방향으로 경계 성분이 존재하게 되어 물결 효과가 발생된 블럭이라고 판단하기는 어렵다. 그러므로, 주어진 블럭이 물결 효과가 발생하기 위해서는 마지막으로

다음 조건을 만족하는 블럭으로 판단한다. 즉, 앞의 차이값이 일정치 이하인 영역(Rm)에 대해서 이 영역과 직접적으로 인접한 두개의 영역을 제외한 나머지 다섯 영역의 평균값을 구하고 영역 Rm 의 평균값과 비교하여 어느 정도 이상 차이가 발생하면 물결 효과가 나타나는 것으로 간주한다. 여기에서 나머지 영역의 평균값을 계산할 때, 영역 Rm 과 인접한 두개의 영역을 제외시킨 이유는 Rm 의 픽셀값들이 이웃한 영역의 픽셀값들과 연관을 가지고 있을 확률이 높기 때문에 직접 이웃한 영역 두개는 제외하고 나머지 다섯 개의 영역에 대해서만 고려하였다.

4.1.4.4 실험 및 토의

본 과제에서 제안한 인간의 시각 체계의 특성을 분석하여 이미지 압축에 적용한 시각 적응적 이미지 압축 방법은 칼라 및 그레이 영상에 대한 국제 표준안인 JPEG 을 기반으로 시각 적응적인 압축 방법을 추가 보완하여 Sun Sparc 2 에서 C 와 motif 를 사용하여 구현하였다.

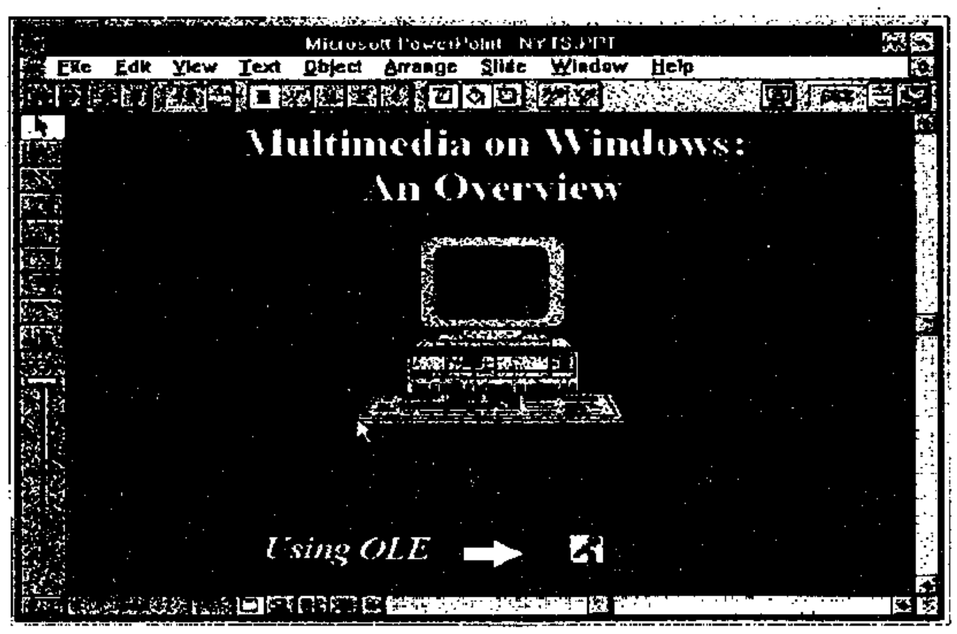
특히 본 과제의 실험에서 중요한 부분은 양자화 테이블의 선택이라고 할 수 있다. 본 과제에서는 두 종류의 양자화 테이블을 사용하였는데, JPEG 표준안에서 제공하는 양자화 테이블 Q 와 DC 성분을 제외한 나머지 계수들을 3 배하여 얻은 Q3 를 사용하였다. 양자화 테이블의 선택은 기본적으로, 손실이 적게 압축할 블럭에 대해서는 Q 를 선택하여 이미지의 질을 높이고, 그렇지 않은 블럭은 Q3 를 선택하여 양자화시켰다. 그리고 각 블럭에 대한 선택 결과를 1 비트로 코딩하여 압축 데이터에 포함시켜 전달하였다.

본 과제에서 사용한 실험 데이터는 일반 책의 그림을 포함하는 페이지를 150dpi 로 스캐닝하여 사용하였다. 이 정도의 해상도를 사용하더라도 화면상에서 텍스트와 그림의 내용을 파악하는데는 문제가 없었다. [그림 4.11]의 (a)는

책의 내용 중에서 그림을 포함하는 일부분을 이진 영상으로 스캐닝한 것이고, (b)는 그레이 영상으로 스캐닝한 영상이다. 그런데 [그림 4.11]의 (a)에서 보는 것처럼 그림을 포함하는 문서를 일괄적으로 이진 영상으로 스캐닝하게 되면 포함된 그림의 내용을 제대로 파악할 수 없는 문제가 발생하게 된다. 따라서 이진 영상이 아닌 색상이 있는 그림을 포함하는 경우에는 가능하면 그레이 영상이나 칼라 영상으로 스캐닝해야 한다.

[그림 4.12]는 [그림 4.11]의 (b)를 본 연구에서 제안한 시각 적응적 방법을 적용하였을 경우, 이미지를 압축하는 과정에서 각 블럭별로 왜곡 현상이 발생할 지의 여부를 예측하여 왜곡 현상이 발생한 블럭과 그렇지 않을 블럭으로 구분한 이미지이다. 이때 블럭의 왜곡을 측정하기 위해 사용한 값은 다음과 같다. 실험에서 사용한 값은 여러 이미지에 대해서 실험하여 경험적으로 설정하였다. 우선 휘도 마스킹을 적용하기 위한 값은 -640 에서 600 으로 정하였다. 이 범위에 속하는 블럭들의 경우에는 왜곡이 발생되게 되면 그 왜곡을 인식할 수 있게 된다. 그러나 이 범위에 속하지 않는 블럭들은 왜곡이 발생하더라도 블럭이 너무 밝거나 어둡기 때문에 왜곡을 인식할 수 없으므로 고손실로 압축하게 된다.

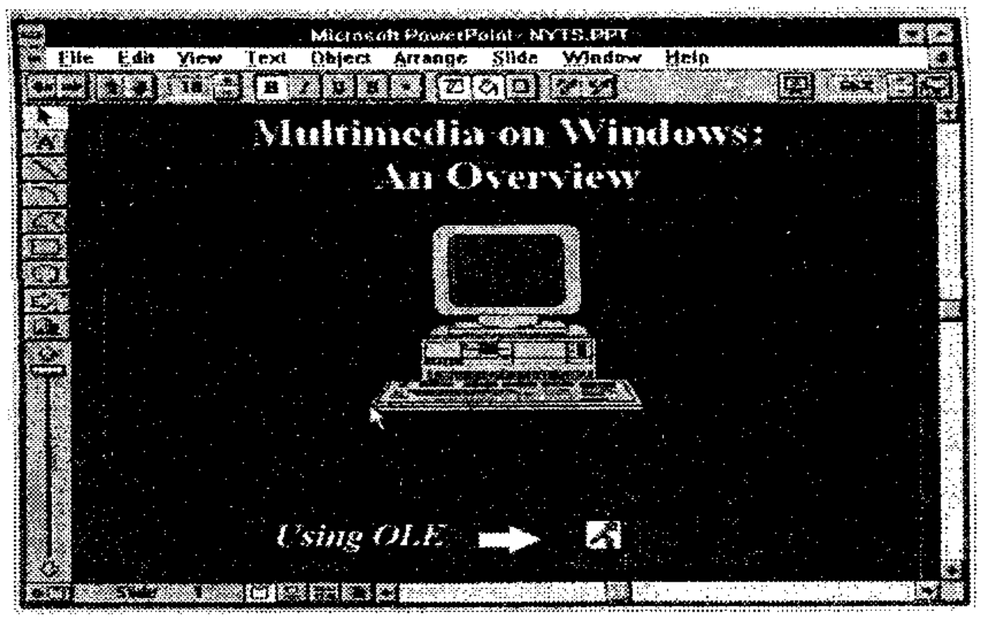
그리고 블럭 효과가 발생하는 블럭을 결정하기 위해서는 Slow 는 10 에서 50 사이의 값으로, Smid 는 210 이하의 값으로 정하였다. 이 두 조건을 만족하는 블럭은 블럭 효과가 나타날 수 있는 블럭으로 간주하여 저손실 압축을 한다. 번짐 효과에 대해서는 Sblur 를 310 에서 820 사의 값으로 정하여 이 조건을 만족하는 블럭은 번짐 효과가 발생할 수 있는 것으로 간주하였다. [그림 4.11]의 (b)를 표준 JPEG 으로 압축하였을 경우 압축률은 1.001940 bits/pixel 이였으며, 시각 적응적 방법을 사용하였을 경우에는 0.839199 bits/pixel 로 더 나은 압축률을 보였다. 물론, 복원된 영상의 경우, 사람이 두 가지 압축 방법의 차이는 구별할 수 없었다.



[그림 1.1] 마이크로소프트사 제품인 파워 포인트는 사운드나 비디오 같은 멀티미디어를 사용해서 설명자료를 만들 수 있게 해준다.

여러가지 미디어 요소들을 결합하고 객체지향(Object-Oriented) 환경을 사용함으로써 그 어느때보다 더욱 창조적인 방법으로 더 많은 정보를 가지고 작업할 수 있다. 멀티미디어는 PC가 책상위를 떠나, 새로운 소비자 용품으로 특징 지워지게 하는 수단

(a) 문서를 이진 영상으로 스캐닝한 영상

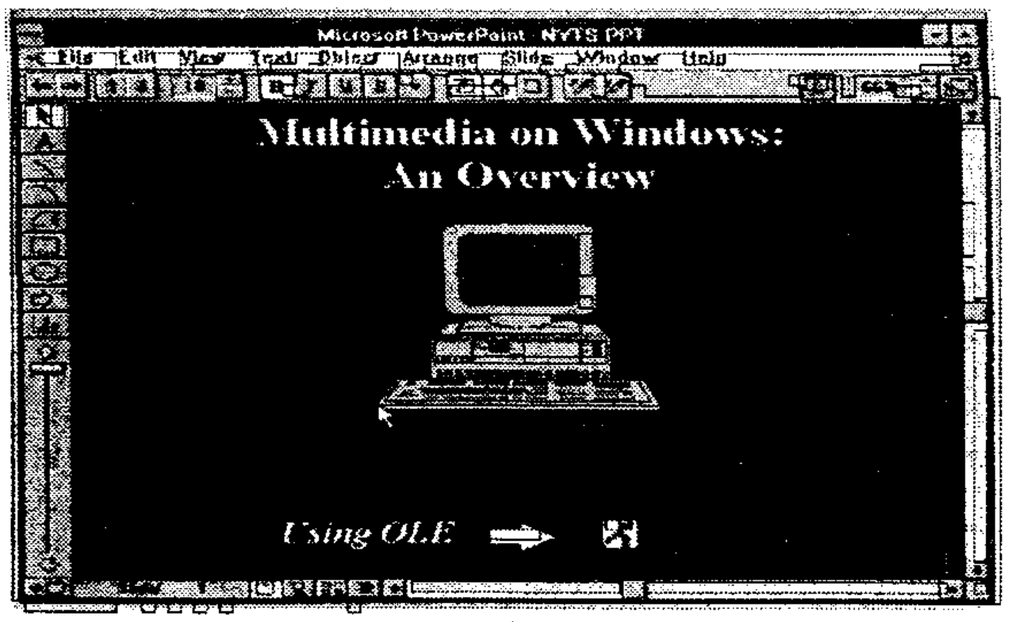


[그림 1.1] 마이크로소프트사 제품인 파워 포인트는 사운드나 비디오 같은 멀티미디어를 사용해서 설명자료를 만들 수 있게 해준다.

여러가지 미디어 요소들을 결합하고 객체지향(Object-Oriented) 환경을 사용함으로써 그 어느때보다 더욱 창조적인 방법으로 더 많은 정보를 가지고 작업할 수 있다. 멀티미디어는 PC가 책상위를 떠나, 새로운 소비자 용품으로 특징 지워지게 하는 수단

(b) 문서를 그레이 영상으로 스캐닝한 영상

[그림 4.11] 테스트 영상



[그림 1.1] 마이크로소프트사 제품인 파워 포인트는 하프드나 비디오 같은 멀티미디어를 사용해서 설명자료를 만들 수 있게 해준다.

여러가지 미디어 요소들을 통합하고 객체지향(Object-Oriented) 환경을 사용함으로써 그 이느때보다 더욱 창조적인 방법으로 더 많은 정보를 가지고 작업할 수 있다. 멀티미디어는 PC가 책상위를 떠나, 세계를 소비자 중심으로 혁명 지워지게 하는 수단

[그림 4.12] 시각 적응적으로 선택된 블럭

지금까지 여러 문서들에는 다양한 종류의 그림들이 포함되어있다. 그런데 문서가 이진 영상이 아닌 그레이 또는 칼라의 그림 영역을 포함하고 있을 경우에는 이 문서를 일괄적으로 이진 영상으로 스캐닝한다면 그림에 포함된 많은 정보가 손실될 수 있다. 따라서 그림을 포함하는 페이지는 그레이 혹은 칼라 영상으로 스캐닝되어야 한다. 하지만 이들 그림들은 페이지의 일부분에 지나지 않으므로 문서 전체를 이진 영상이 아닌 칼라 영상으로 스캐닝하는 것은 시간과 저장 공간의 낭비이다. 따라서 문서에 포함되어있는 그림은 칼라 영상으로 스캐닝하여 따로 관리하고, 문서에 포함된 문자들만 이진 영상으로 스캐닝해야 할 필요가 있다. 그림을 포함하는 페이지를 화면에 보여줄 필요가 있을 때에는 문서상에서의 문자와 그림의 위치를 찾아내고, 적절한 방법을 사용해 이들을 통합해 화면에 출력할 수 있으면 될 것이다. 이런 방법을 사용하면 저장 공간을 절약하면서 색상이 있는 그림 영역에 손상을 주지 않으면서 효율적으로 전체 페이지를 화면에 보여줄 수 있을 것이다. 혹은 다른 방법으로는 필요한 페이지를 화면에 출력하고자 할 때 여러 단계의 출력 모드를 선택할 수 있도록 하는 방법이다. 즉 기본적인 출력 모드에서는 텍스트 데이터

와 그림 데이터를 모두 이진 영상으로 화면에 출력하도록 하고, 만약 사용자가 필요한 그림을 좀더 자세히 보고자 할 때는 그림 영역을 마우스로 선택하여 양질의 데이터를 추가로 제공하는 방법이다. 이런 기능이 추가되면 필요한 이미지 데이터에 대해서만 양질의 이미지로 제공함으로써 사용자가 관심이 없는 데이터도 화면에 양질로 제공하는 것보다 상당한 시간의 절약을 기할 수 있을 것이다. 그리고 차후에는 문서의 텍스트 영역은 문자 인식을 통해 그림 데이터가 아닌 문자 데이터로 변환시킴으로써 저장 능력과 화면 출력 속도를 향상시킬 수 있는 시스템을 구현할 수 있을 것으로 기대된다.

4.2 문서 식별자의 압축 복원기

지금의 컴퓨터 기술은 이전까지 상상도 못할 정도의 대용량 정보의 효율적인 처리를 가능하게 해 주고 있다. 그런데 컴퓨터의 속도가 갈수록 빨라지고 그 저장 매체의 양이 증가됨에 따라 컴퓨터에 사용되는 데이타의 종류와 특성도 달라지고 있다. 이전의 컴퓨터용 데이타는 자료 저장의 효율화를 위하여 특별한 인코딩(encoding) 방법을 사용하였지만, 지금은 그러한 코딩에 드는 비용과 노력보다 저장 매체의 용량과 가격이 충분히 경쟁력을 가지게 됨에 따라, 최근에는 데이타의 이미지나 텍스트 자체가 바로 저장 매체에 입력으로 들어가게 되었다. 그런데 컴퓨터 기술 중에서 이전의 기술과 비교해서 가장 발전 속도가 느린 분야가 기계적인 부품을 사용하는 부분이다. 예를 들어, 컴퓨터 초기의 CPU 의 계산 속도는 이전에 비해서 수만 배나 빨라졌고, 반도체 소자를 이용한 내부 기억장치의 패치 속도도 수 백배 이상 빨라졌지만, 아직 외부의 보조기억장치인 하드 디스크나 자기 테이프 등과 같은 대용량 저장 매체의 속도는 10 배 이상 증가하지 못하고 있는 수준이다.

이것은 컴퓨터 시스템에서 속도 문제에 대한 일반적인 해결책은 다음의 사

항을 고려해야 한다는 것을 의미한다. 우리가 먼저 고려해야 할 문제를 CPU 상의 작업(CPU bounded job)과 입출력 상의 작업(I/O bounded job)으로 나누어서 생각해 본다. 먼저 CPU 상의 작업일 경우에는 CPU 자체의 속도를 증가시키는 방법과 주기억장치의 용량을 증가시키는 방법을 고려해 볼 수 있다. 특히 시분할 체제(time sharing system)와 같은 경우 주기억 장치의 크기를 증가시키는 것이 대단히 효율적인 방법이 될 수 있다. 이 외에 특별한 알고리즘을 위한 하드웨어 가속기(hardware accelerator)를 부착시킬 수 있다. 그러나 입출력상의 작업인 경우에는 문제가 달라진다. 이 경우에는 전체의 속도는 CPU 의 계산 속도에 있는 것이 아니라 대부분의 경우에 하드 디스크와 CPU 의 버퍼와의 통신 속도에 좌우되기 때문이다. 이 부분이 병목 현상이 일어나므로 이 과정에 대한 최적화가 준비되지 않은 한 다른 부분의 개선은 전체 시스템의 속도에 별 영향을 주지 못한다. 따라서 속도 개선을 위하여 디스크와 주기억 장치 사이에 전달해야 할 정보의 양을 압축하는 것이다. 압축된 정보는 보통의 정보보다 약 70% 이상 감소하므로 전송속도에 있어서 약 30 ~ 40% 정도의 시간적 이익을 볼 수 있다. 그런데 이 작업에는 부가적으로 주어진 자료를 압축하는 시간과 압축된 자료를 주기억 장치에서 다시 복원하는 시간이 필요하다. 따라서 이 부가적인 시간을 어떤 알고리즘으로 조절해서 디스크와 CPU 간의 시간을 줄여서 전체의 작업 시간의 이익을 꾀할 수 있는지를 다루어 보는 것이 중요한 문제이다.

본 과제에서 개발된 저장 시스템의 색인 화일은 역화일 구조의 접근 방식을 취하고 있다. 이러한 역화일 방식을 취할 때의 문제는 시스템이 색인어들에 대한 문서 식별자 리스트를 유지하기 위해 많은 저장 공간을 소모한다는 점이다. 따라서 본 절에서는 Record ID로서 표현되는 문서 식별자의 리스트를 효율적으로 압축 복원할 수 있는 방법을 모색하고자 한다. 먼저 기존의 압축 알고리즘의 일반에 관하여 기술하고, 기존의 압축 방법들의 성능을 비교한다. 그리고 RID 자체를 이용하여 검색할 때와 압축된 RID를 이용하여 검색할 때

의 시간적 관계를 실험적 방법을 통하여 분석하고, 앞의 실험 결과를 토대로 효율적인 RID 입출력을 위한 성능 분석을 수행한다. 그리고 마지막으로 현재의 압축 알고리즘과 하드웨어적인 기술을 고려해 볼 때, 새로운 방법이 제시되는 것이 타당함을 보인다.

4.2.1 텍스트 압축 알고리즘의 일반

일반적으로 압축이란 한정된 공간에 보다 많은 양의 정보를 저장할 수 있도록 정보의 부피를 줄이는 것을 일컫는다. 공학적인 의미에서 압축은 통신상의 정보 전송에 있어서 전송 시간을 줄이기 위한 목적으로 사용되며, 정보의 저장 측면에서는 정보를 저장하기 위한 공간을 보다 많이 확보하기 위한 목적으로 사용되어지는 방법을 의미한다.

압축은 크게 손실 압축과 비손실 압축으로 나누어진다. 손실 압축이란 말 그대로 압축 후 다시 원래의 데이타를 그 이전 상태 그대로 복원시키지 않아도 무방한 압축 기법을 말하는 것으로, 주로 멀티미디어의 영상 이미지를 압축하기 위해서 많이 사용되는 기법이다. 영상 이미지는 약간의 손실이 있더라도 전체적인 이미지의 형상에는 크게 영향을 미치지 않기 때문에 약간의 손실을 감수할 수 있다. 비손실 압축이란 텍스트 데이타의 압축에 주로 사용되는 방법으로, 압축 후 다시 원래의 데이타를 정확히 복원할 수 있는 압축 기법을 말한다. 본 연구에서 대상으로 삼는 것은 원래의 데이타를 그대로 복원할 수 있는 비손실 압축에 관한 것이다. 앞에서 압축은 공간과 시간을 줄이기 위한 방법이라고 언급하였다. 특히 본 연구에서 알아 보고자하는 대용량 데이터베이스에서의 RECORD id 에 대한 압축은 그러한 의미에서 상당히 중요한 의미를 갖는다. 즉, 압축 기법을 통해 RECORD id 를 압축시킴으로서 RECORD id 를 저장하기 위한 공간을 상당히 줄일 수 있고, RECORD id 처리를 위해 디스크상에서 메모리로 읽어 들이는데 필요한 입출력 시간의 오버헤드를 상당

히 감소시킬 수 있다.

현재의 압축 기법은 상당한 수준에 도달해 있다. 그리고 많은 압축에 관한 패키지들이 통용되고 있기도 하다. 하지만 어떤 하나의 압축 기법이나 패키지가 모든 데이타에 관하여 항상 같은 압축 시간과 압축률을 제공해 주는 것은 아니다. 압축 기법이나 압축 패키지는 제작 의도나 목적에 따라 적절한 데이타에 적용된다면 상당한 정도의 압축률과 압축 시간을 제공해 주지만, 그렇지 못한 데이타에 적용된다면 의도하지 않았던 결과를 나타낼 수도 있다. 그러므로 데이타에 얼마나 적합한 압축 기법 이나 압축 패키지를 사용하는가 하는 것이 대단히 중요하다.

앞으로 연구 대상이 될 데이타가 어떠한 압축 기법이나 압축 패키지에 적당한지를 정확히 알아낼 수가 있다면, 이를 이용하여 좀더 나은 결과를 얻을 수 있을 것이다. 그러므로 기존의 압축 기법들을 고찰하고 그 알고리즘을 바탕으로 만들어진 압축 패키지에 대한 성능을 비교 분석해 봄으로써 앞으로 연구할 과제에 적합한 알고리즘에 대한 이해를 향상시킬 수 있고, 나아가 기존의 압축 기법과 압축 패키지에 대한 분석을 통해 그 장단점을 파악하고 연구 목적에 맞는 효율적인 알고리즘을 개발하는데 도움을 얻을 수 있다.

4.2.2 기존의 압축 알고리즘

압축 기법은 약 40 년의 역사를 가지고 있으며, 그동안 수많은 알고리즘들이 속출하였다. 현재 가장 많이 쓰이는 압축 기법인 DBC(Dictionary Based Coding)가 개발된 것은 1977 년의 일이다. 압축 기법의 기원은 1949 년 벨 연구소의 C.E.Shannon 과 MIT 의 R.M.Fano 가 거의 동시에 개발한 Shannon-Fano 코딩에서 찾을 수 있다. Shannon-Fano 코딩은 빈도가 높은 문자에는 적은 비트를 낮은 문자에는 많은 비트를 할당하는 VLC(Variable Length Coding) 기법으로서 당시로는 파격적인 압축률을 얻을 수 있었다. 이 알고리즘이 발표된 직후

정보 이론 (Information Theory)이 탄생했다. 정보 이론은 데이타를 가공하는데 사용되는 여러 가지 기법을 연구하는 수학의 한 분야로 주된 연구 과제는 암호화와 압축이다. 그로부터 4 년 후 D.A.Huffman 은 논문을 통해 Shannon-Fano 코딩을 개량한 Huffman 코딩을 발표한다. Huffman 코딩은 기존의 알고리즘을 약간 변형한 것으로서 압축률이 Shannon-Fano 코딩과 거의 같거나 약간 높다. Huffman 코딩은 발표 후 약 30 년간 사용되었으며 그 당시 대부분이 Huffman 코딩을 최적화하기 위해 노력했다. 이 시기에 나온 대표적인 알고리즘이 Dynamic Huffman 코딩과 산술 코딩이다.

1970 년대 후반 이스라엘 공과대학의 연구원이던 Jacob Ziv 와 Abraham Lempel 이 발표한 두 논문은 DBC(Dictionary Based Coding)라는 전혀 새로운 압축 기법을 탄생시켰다. DBC 는 VLC 와는 다른 근본적인 차이점을 가지고 있었다. 기존의 VLC 계열이 한번에 하나의 문자만을 처리할 수 있었던 것에 비하여 DBC 는 한번에 둘 이상의 문자를 처리할 수 있다는 점이 다르다. 본격적인 압축 프로그램이 나오기 시작한 것도 DBC 가 등장한 이후의 일이다. LZ77 이 발표된 직후 COMPACT 가 선을 보였으며, LZ78 이 개발된 다음에는 COMPRESS 가 등장했다. COMPACT 나 COMPRESS 가 LZ77 이나 LZ78 을 그대로 구현한 것은 아니었다. LZ77 이나 LZ78 은 최적화된 알고리즘이 아니었기 때문이다. LZ77 의 경우는 LZSS, LZ78 의 경우는 LZW 라는 변형된 알고리즘이 주로 사용되고 있다. 이들 알고리즘에 대해 자세히 알아보도록 하겠다.

4.2.2.1 Run Length Coding (RLC)

RLC 는 초기의 압축 기법으로서 반복되는 부분을 최소화하는데 중점을 두고 있다. 이 압축 기법의 특징은 다음과 같다.

- 압축 기법이 간단하기 때문에 속도가 빠르다.
- 코딩하기에 편리한 압축 기법이다.

RLC 압축 기법의 보통의 표현 방법은 다음과 같다.

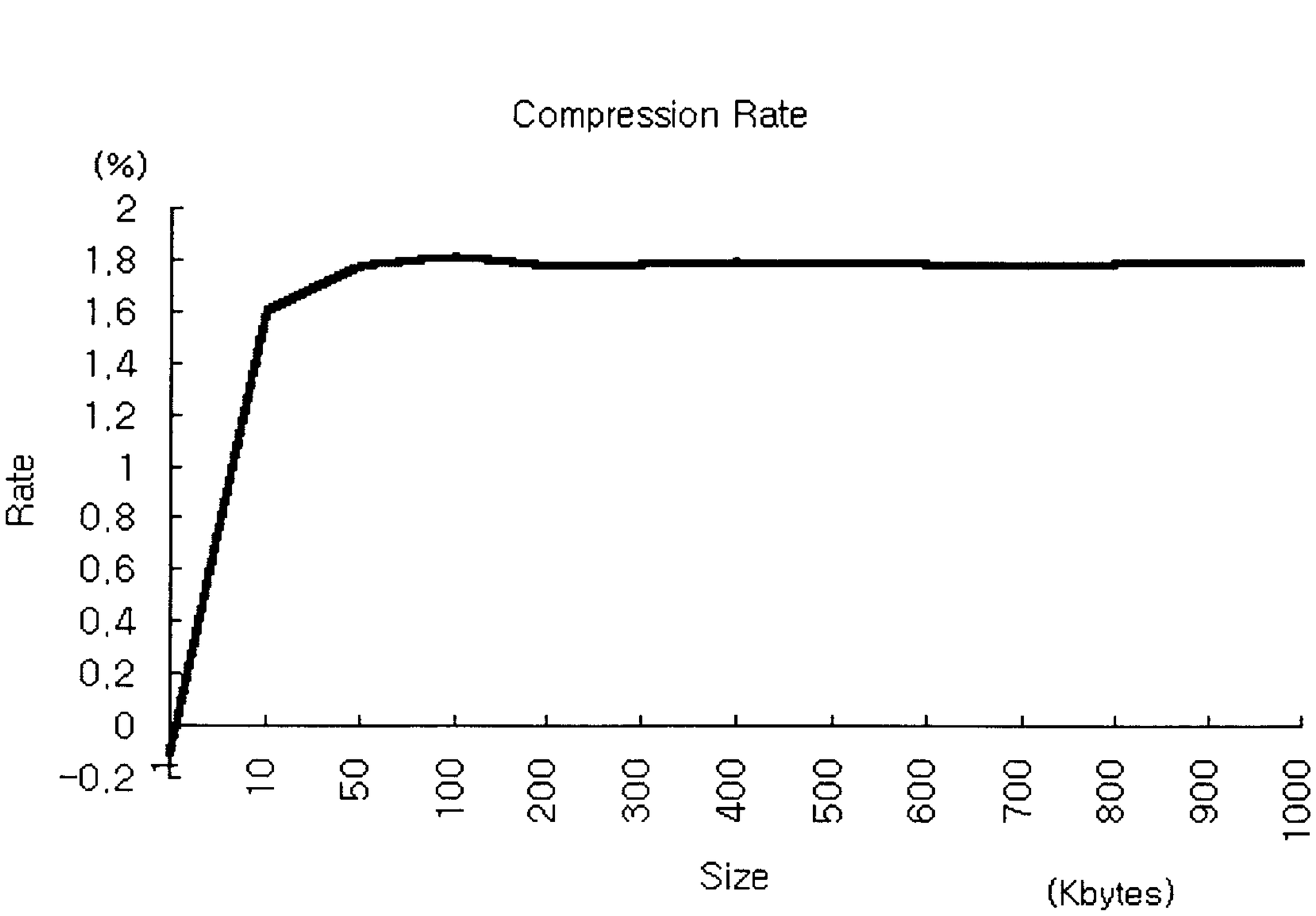
[문자, 반복횟수] 또는 [반복횟수, 문자]

RLC 압축 기법의 문제점은 압축률이 그렇게 높지 않다는 것이다. 반복되는 문자만을 압축할 수 있기 때문에 반복되는 문자가 자주 나타나는 데이터 파일의 경우에만 유용하게 사용될 수 있다. RLC 에서는 반복되는 문자를 문자와 반복 횟수로 표현하기 때문에 최소한 2 문자가 반복되어지는 문자만이 압축의 대상이 된다.

구현상의 문제점은 반복 횟수로 사용되는 문자가 실제 데이터 파일상의 압축 가능한 문자로 나타날 때 이를 처리하는 문제이다. 예를 들어, 4AAAA33333CC 를 RLC 로 표현하면 4A435C2 또는 44A532C 가 된다. 이와 같은 경우에 숫자와 반복 횟수를 구분한다는 것은 사실상 불가능하다. 반복 횟수와 문자를 구분하기 위한 일반적인 해결책으로서 특수 문자를 사용한다. 특수 문자는 이어지는 데이터가 문자와 반복 횟수로 구성된다는 것을 알려준다. 이때 앞의 예는 4QA4Q35QC2 또는 4QA4Q53Q2C 로 표현된다. 그러나 이 경우에도 계속해서 문제점이 발생한다. 특수 문자가 문자로 사용될 수 있기 때문이다.

결론적으로 말해 RLC 는 속도가 빠르고 Coding 에 편리한 장점을 가지고 있지만, 압축률이 그다지 높지 않고 구현상에 있어서 여러 가지 문제들을 고려하여야 한다. [그림 4.13]와 [그림 4.14]는 각각 1K, 10K, 50K, 100K, 200K, 300K, 400K, 500K, 600K, 700K, 800K, 900K, 1Mbytes 에 대한 압축 시간과 압축률을 도시하고 있다. 사용된 데이터는 보통의 텍스트 데이터이다.

[그림 4.13] Run Length Coding 의 압축 시간



[그림 4.14] Run Length Coding 에서의 압축률

4.2.2.2 Variable Length Coding(VLC)

이 압축 기법은 데이터 파일에서 자주 쓰이는 문자에는 적은 비트를 할당하고, 그렇지 않은 문자에는 많은 비트를 할당하는 형식의 알고리즘이다. VLC는 뒤에 언급할 DBC(Dictionary Based Coding)가 나오기까지 약 30년 동안 압축 분야에서 사용된 알고리즘으로서, 그 응용 분야는 모뎀, 테이프 드라이브, 비디오 보드 등 거의 모든 분야에 적용되었다. VLC의 원조인 Shannon-Fano Coding과 이를 변형한 Huffman Coding 그리고 Dynamic Huffman Coding이 VLC의 주류를 이루고 있다.

4.2.2.2.1 Shannon & Fano Coding

이 압축 기법은 1949년 벨 연구소의 Shannon과 MIT의 Fano가 거의 동시에 통계적 모델을 이용해 개발한 기법이다. 데이터내의 빈도가 알려진 문자를 표현하는데 필요한 비트수를 의미하는 용어로 정보 용량 (information content)이라는 말을 정의하였는데, 이의 표현은 다음과 같다.

$$\text{정보 용량} = -\log_2 N \quad \text{(확률로 나타낸 빈도)}$$

Shannon-Fano Coding의 원리는 다음과 같다.

1. 파일을 끝까지 읽어서 사용된 문자의 빈도를 구하고 이를 오름차순 또는 내림차순으로 정렬한다.
2. 빈도의 합이 대략 반으로 나누어지는 지점에서 한쪽에는 0 다른 한쪽에는 1을 할당한다.
3. 둘로 나누었을 경우 어느 한쪽이 두개 이상의 문자를 포함하고 있으면 이를 대상으로 같은 작업을 수행한다.
4. 각 문자가 구분될 때까지 계속해서 반복한다.

만일 파일내의 정보 용량이 다음과 같다면 그 결과로 얻어지는 이론치는 다음과 같이 나타난다.

빈 도	정 보 용 량 ($-\log_2$ 빈도)	이론상 차지하는 비트 수
A : 5 (5%)	$-\log_2 0.05 = 4.32$	4.32 * 5 = 21.6
B : 12 (12%)	$-\log_2 0.12 = 3.06$	3.06 * 12 = 36.72
C : 3 (3%)	$-\log_2 0.03 = 0.18$	0.18 * 3 = 0.54
D : 18 (18%)	$-\log_2 0.18 = 5.06$	5.06 * 18 = 91.08
E : 50 (50%)	$-\log_2 50 = 1$	1 * 50 = 50
F : 12 (12%)	$-\log_2 0.12 = 3.06$	3.06 * 12 = 36.72
압축전 : 100bytes 압축후 : 29.58bytes 압축률 : 70.42%		

Shannon-Fano Coding 의 이론치와 실제치는 약간의 차이를 보이는데, 사용된 문자가 적고 이들간의 빈도차가 작을 경우에만 적용 가능(이론과 실제간의 차이가 적다)하다. 즉, 문자가 많으면 많을수록 이진 트리의 노드수가 증가하므로 차지하는 비트가 늘어나 많은 차이를 보일 것이다. Shannon-Fano Coding 은 파일을 구성하는 문자가 적을수록 이들간의 빈도차는 작을수록 높은 압축률을 낼 수 있다.

Shannon-Fano Coding 은 압축할 때 Shannon-Fano Code 를 얻기 위해 소스 파일을 처음부터 끝까지 한번 읽어야 하기 때문에 그 만큼 속도가 떨어진다. 또한 후에 압축을 해제하기 위해 문자의 빈도를 압축 파일에 다시 저장해야 하기 때문에 크기가 작은 파일의 경우 압축 후의 크기가 더 커지는 경우를 종종 볼 수 있다.

4.2.2.2.2 Huffman Coding

Huffman Coding 은 Shannon-Fano Coding 을 개량한 것으로서, 이진 트리를 만드는 방법만 달리 취하고 있다. Huffman Coding 은 Shannon-Fano Coding 처럼 빈

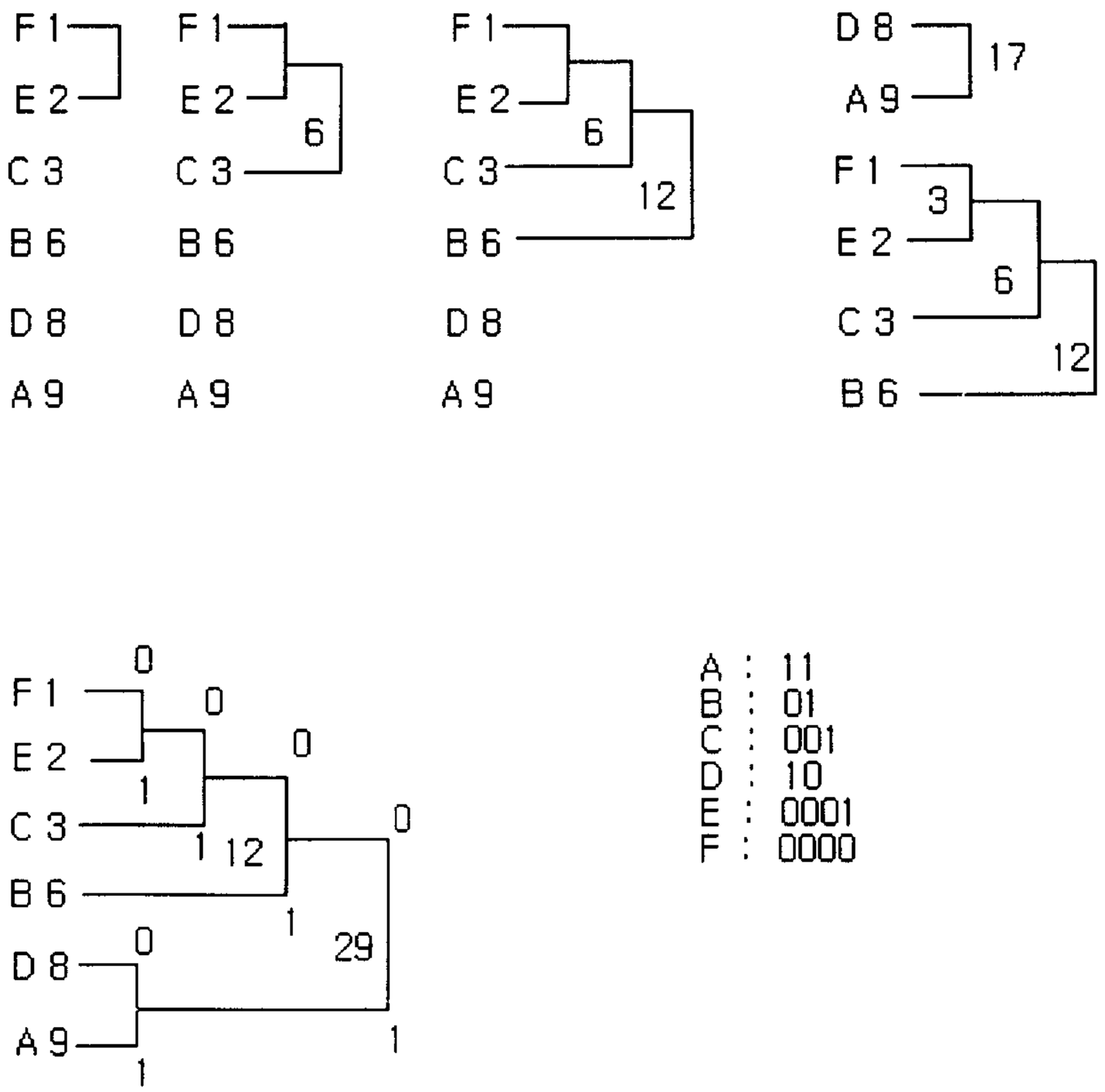
도의 합이 반으로 나뉘는 지점을 쪼개는 것이 아니라, 빈도가 가장 작은 두개의 노드들을 합쳐 나가는 형식으로 Huffman Tree 를 형성한다. 그 원리는 다음과 같다.

1. 파일내의 문자의 빈도가 가장 적은 두개의 노드를 취해 하나의 노드를 만든다. 한쪽에는 0 을 할당하고, 다른 한쪽에는 1 을 할당한다.
2. 파일내의 모든 문자에 대해 하나의 노드가 형성될 때까지 실행한다.

위의 Huffman Coding 의 처리 원칙에 따라 데이터는 다음과 같은 과정을 거쳐 처리된다. 예를 들어, 압축해야 할 문자가

ABDAEBCDABDFABDEABAAACDDDABCD

라고 가정하자. 이때 이를 Huffman 트리를 사용하여 표현하면 [그림 4.15]과 같다.

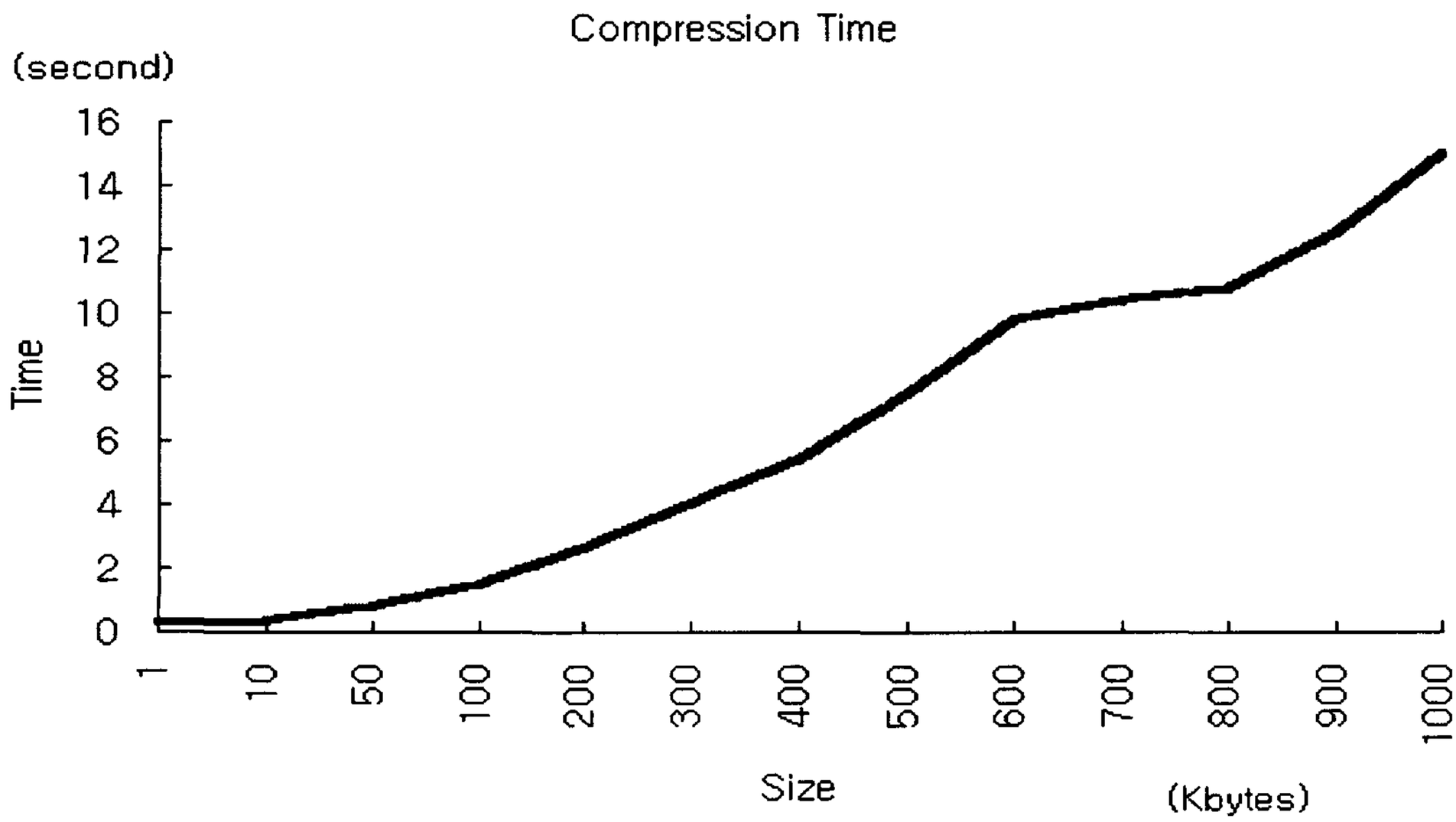


[그림 4.15] Huffman Coding 에서의 압축 과정

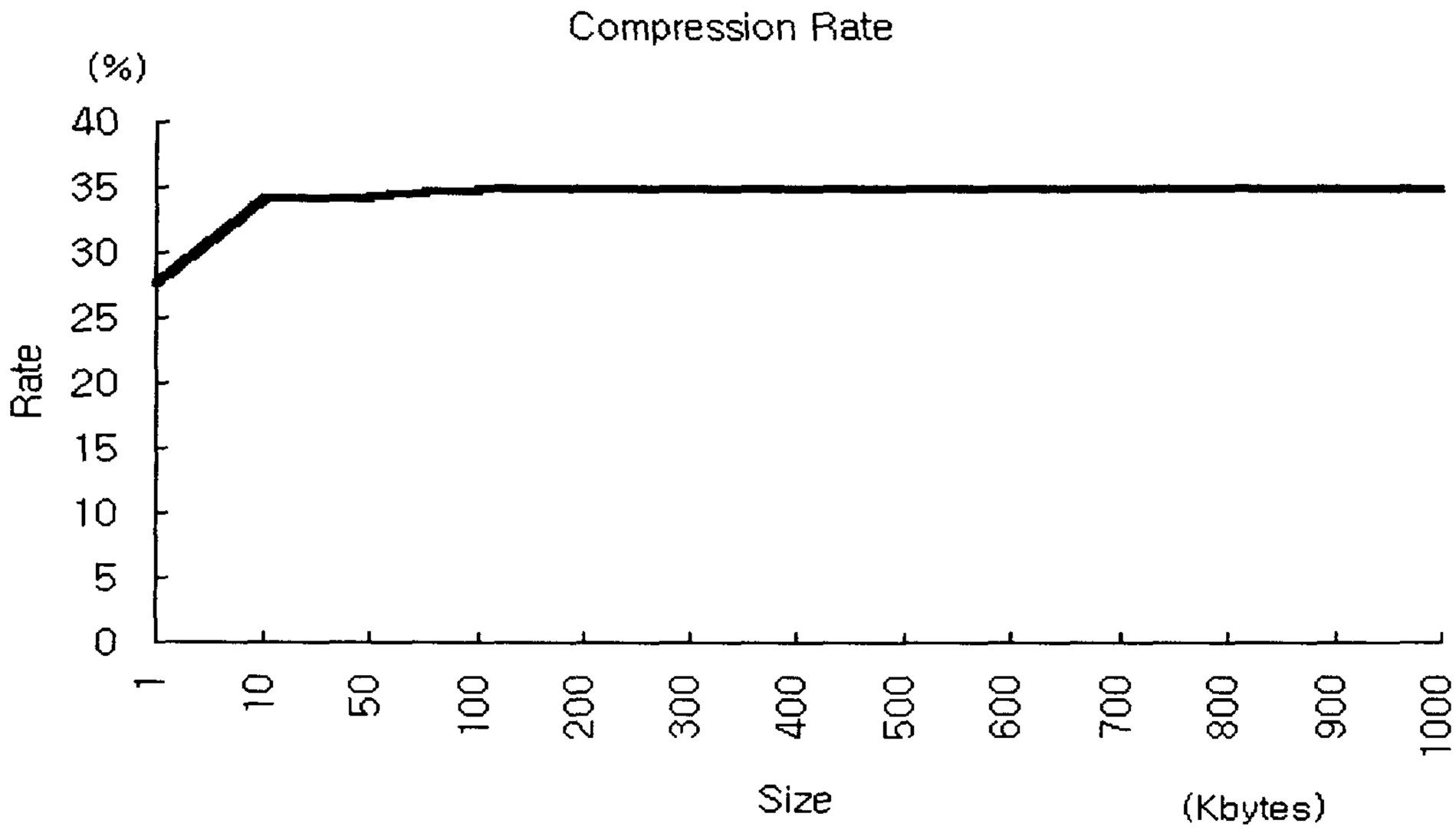
압축의 결과로 얻어지는 압축의 형태는 다음과 같다.

1101101100010100110110110000011011000011101111111001101010110100110

Huffman Coding 은 압축률이 Shannon-Fano Coding 보다 약간 더 높거나 같다. Huffman Coding 에서 염두에 두어야 할 것은 빈도가 작은 노드를 합한 이 후에는 이를 하나의 단위로 취급하여 처리하여야 한다는 것이다. [그림 4.16]과 [그림 4.17]은 RLC 와 동일한 데이타에 대한 Huffman Coding 에서의 압축 시간과 압축률을 보여준다.



[그림 4.16] Huffman Coding 에서의 압축 시간



[그림 4.17] Huffman Coding 에서의 압축률

4.2.2.2.3 Dynamic Huffman Coding

위에서 언급된 Huffman Coding 에는 압축률을 반감시키는 요소로서 Huffman Code 를 고정시키고, 압축 파일의 앞쪽에 빈도를 저장해야 하는 두 가지 요소가 있다. 이에 대한 자세한 내용은 다음과 같다.

1. Huffman Code 가 고정되어 있다.

   실행 파일의 경우 코드 부분에서 데이터 부분으로 넘어가면 문자의 빈도가 판이하게 바뀌는데, Huffman Coding 에서는 이러한 부분을 고려하지 않고 같이 취급하고 있다. 만일 코드 부분과 데이터 부분의 Huffman 트리를 따로 만든다면, 압축률을 향상시킬 수 있을 것이며, LHA 나 PKZIP 과 같은 program 에서는 이러한 방법을 채택하고 있다. 그 처리 과정은 어떤 특수한 문자를 정의해 놓고, 데이터를 블럭 단위로 처리하는 것이다. 만일 이와 같이 처리한다면, 다음과 같은 결과의 압축 파일이 생길 것이다.

헤더 1	압축된 내용 1	특수 문자	헤더 2	압축된 내용 2

특수 문자를 이용하면 헤더가 블럭 단위로 필요하기 때문에 더욱 많은 공간의 낭비를 초래할 수도 있으나, 블럭 단위 처리의 장점으로 인하여 얻어지는 이득을 고려한다면 충분히 상쇄할 수 있는 부분이다.

2. 각 문자의 빈도를 저장해야 한다

압축될 파일의 앞부분에 파일 내의 각 문자의 빈도를 저장해야 하기 때문에, 크기가 작은 파일을 압축하게 될 때는 헤더 부분으로 인해 압축된 파일의 크기가 더 커지는 경우도 발생한다. 즉, 다음과 같은 경우가 된다.

ABCDEFGH

위의 문자열을 압축한다면,

65 72 1 1 1 1 1 1 1 1 0 압축된 내용

과 같이 될 것이고, 원래의 크기보다 더욱 커진 것을 알 수 있다.

위의 (2)와 같은 경우 만일 헤더를 저장하지 않고 데이터를 압축할 수 있다면 더욱 좋은 결과를 얻을 수 있을 것이다. 이러한 이유로 만들어진 알고리즘이 Dynamic Huffman Coding 이다. DHC 의 대략적인 알고리즘을 살펴보면 다음과 같다.

1. 파일에서 한 문자를 읽어낸다.
2. 읽어 들인 문자를 처리한다.
3. 해당 문자의 빈도를 1 증가시킨 다음 Huffman tree 를 갱신한다.
4. 파일의 끝까지 위의 1-3 을 반복한다.

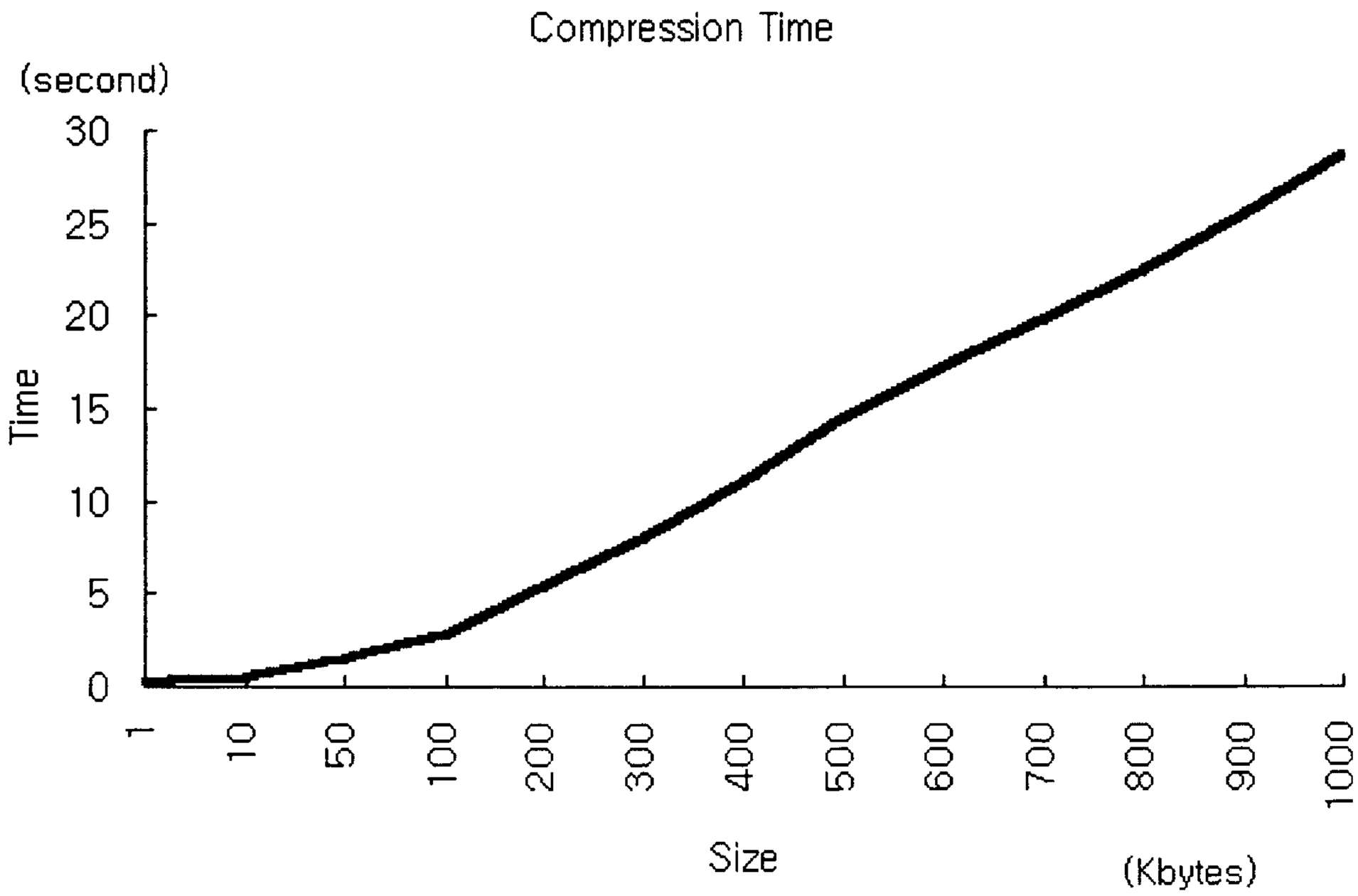
압축된 데이타를 원래대로 복구하는 경우는 다음과 같다.

1. 압축된 파일에서 한 문자를 읽어 들여 decode 한다.
2. 원래대로 복구된 문자를 파일에 출력한다.

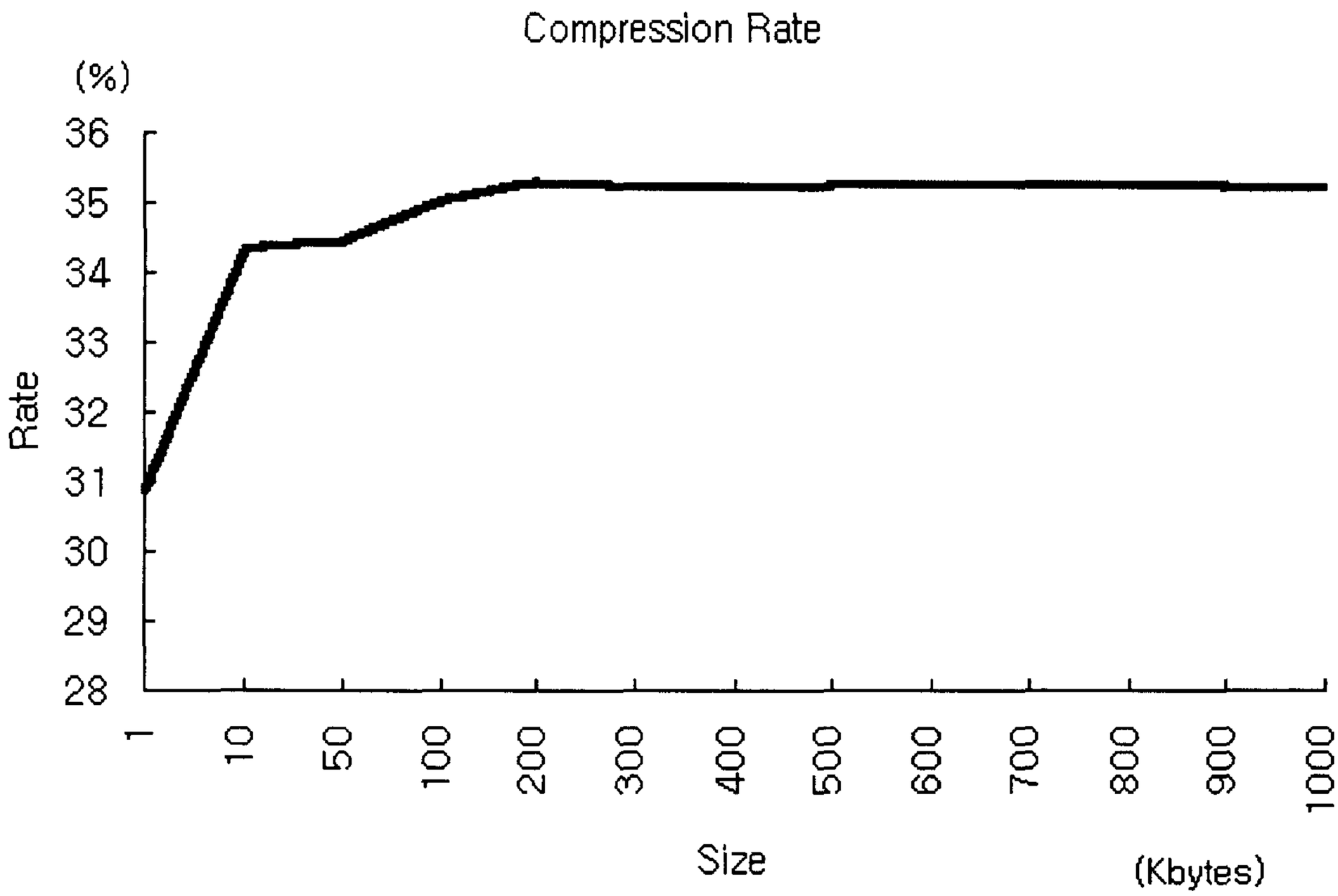
3. 해당 문자의 빈도를 1 증가시킨 후 Huffman tree 를 갱신한다.

4. 파일의 끝까지 위의 1-3 을 반복한다.

Dynamic Huffman Coding 에서는 같은 문자를 나타내는데 각기 다른 Huffman Code 를 사용할 수도 있다. Huffman tree 의 갱신으로 인해 위치가 계속해서 바뀌어 나가기 때문이다. [그림 4.18]과 [그림 4.19]는 RLC 의 데이터에 대한 Dynamic Huffman Coding 에서의 압축 시간과 압축률에 관한 그래프를 도시하고 있다.



[그림 4.18] Dynamic Huffman Coding 에서의 압축 시간



[그림 4.19] Dynamic Huffman Coding 에서의 압축률

4.2.2.3 Dictionary Based Coding(DBC)

문자 단위로 데이타를 처리하던 Huffman Coding 과는 달리 문자열을 하나의 token 으로 대치하는 압축 기법이다. DBC 알고리즘을 사용하면 기존의 Huffman Coding 에서 이론적으로 가능했던 최대 압축 효율이 87.5%인데 반해, 100%(엄밀하게 말하면 99.9999...%)까지 데이터를 압축할 수 있다. VLC 의 경우에는 문자의 빈도를 이용하여 데이터를 압축하기 때문에 데이터를 원래보다 적은 수의 비트로 변환하는 과정에서 압축이 발생한다. 이때 압축 효율은 빈도차에 따라 높아질 수도 있고 낮아질 수도 있다. 그러나 DBC 는 파일 내의 문자의 빈도와는 무관하게 압축을 수행한다. 그 뿐만이 아니라 VLC 처럼 데이터를 처음부터 끝까지 문자의 빈도를 알기 위해 미리 한번 읽을 필요도 없다. 그리고 VLC 처럼 하나의 문자에 대해서만 압축을 수행하는 것이 아니

라 여러 개의 문자에 대해 압축을 수행하게 되므로 훨씬 우수한 압축 효율을 획득하는 것이 가능하다. DBC 에서는 token 이라는 것을 사용하여 파일을 압축하게 되는데, token 은 다음과 같은 구성을 갖는다. DBC 의 압축 효율은 X 와 Y 의 크기에 의해 좌우된다.

문자/토큰 여부	사전으로서의 인덱스	일치하는 문자열의 길이
1 비트	X 비트	Y 비트

X : 사전의 크기

Y : 토큰의 최대 압축 효율

DBC 는 데이터를 압축할 때 사전을 참조한다. 이때 사전은 미리 사전에 만들어진 사전을 사용할 수도 있고, 압축할 때 동적으로 사전을 만들어 참조할 수도 있다. 이 두 가지 방법은 각각 나름대로의 장단점을 가지고 있기 때문에 데이터의 종류에 따라 어떤 형태를 사용할 것인지를 결정하는 것이 바람직하며, 보통 사용되는 형태의 사전은 압축할 때 만들어지는 동적인 형태를 많이 사용한다. 다음은 두 가지 형태의 사전을 비교한 것이다.

	정 적 사 전	동 적 사 전
만들어지는 시점	압축 수행 이전	압축 수행 중
장 점	데이터의 특성에 최적화	압축 파일에 사전 전달
단 점	압축 파일에 사전 전달	데이터의 특성에 최적화 안됨
사용 시기	파일의 크기가 큰 경우	항상 이용

정적 사전의 경우는 사전을 압축 파일에 함께 저장해야 하기 때문에, 같은 사전이 자주 쓰이는 경우라면 굳이 저장할 필요가 없겠지만, 파일의 크기가 작은 경우 너무 많은 공간을 차지하여 오히려 압축률의 저하를 가져올 수 있다.

DBC 알고리즘의 압축 효율은 일치하는 문자열의 길이와 직접적인 관련을 갖고 있다. token 하나에는 보통 "사전의 길이 + 일치하는 문자열의 길이" 만

큼의 데이터가 들어가게 되는데, 일치하는 문자열의 길이를 어떻게 정하느냐에 따라 압축 효율에 상당한 영향을 미친다고 볼 수 있다. 즉, 너무 긴 문자열을 선택할 경우 그 만큼의 길이의 일치하는 문자열이 자주 나타나지 않기 때문에, 늘어난 길이 만큼의 비트는 원하지 않는 공간을 차지하게 되어 쓸모없이 버려지게 되고 압축 효율을 떨어뜨리게 된다. 반대로 너무 작은 문자열의 길이를 일치하는 문자열로 선택하게 될 경우 일치하는 문자열이 더 긴 경우에도 이상을 수용하지 못하기 때문에 압축률을 저하시킨다. 그러므로 얼마나 적절히 이 값을 선택하느냐 하는 것이 대단히 심각한 문제가 될 수 있다.

4.2.2.3.1 LZ77

LZ77 은 Jacob Ziv 와 Abraham Lempel(보통 ZL 알고리즘이라고 부른다.)의 논문을 통해 발표된 알고리즘이다. LZ77 은 DBC 알고리즘으로서 사전을 사용하여 압축을 시도한다. LZ77 에서 사용하는 사전은 FIFO(First In First Out) 형태를 갖고 있다. 즉, 사전이 꽉 채워져 있을 경우 데이터를 추가하면, 사전의 가장 앞부분이 밀려나고 뒤쪽에 새로운 데이터가 추가된다. 이러한 현상이 마치 미끄러지듯이 앞으로 이동한다고 하여 Sliding Window Technique 이라고도 한다. LZ77 의 토큰은 다음과 같은 구조를 갖고 있다.

[Index, Length, Next char]

Index 는 사전에서 문자열을 찾아볼 수 있는 첫 위치를 나타낸다. 그리고, Length 는 Index 위치로부터 얼만큼의 위치까지의 문자열을 하나의 토큰으로 인식할 것인가를 지정해준다. Next Char 는 Index 와 Length 에 의해 결정되어진 문자열의 바로 다음에 올 문자를 나타낸다. 위의 토큰 형태에서 Length 의 값이 얼마인가에 따라 압축률이 좌우된다고 말할 수 있다. LZ77 에서 행하게 되는 압축 알고리즘을 간단하게 나타내면 다음과 같다.

1. 버퍼를 채운다.

2. 사전에서 일치하는 가장 긴 단어를 찾는다.
3. 문자열을 토큰으로 변환한 후 파일에 저장한다.
4. 토큰으로 바뀐 문자열을 버퍼에서 사전으로 옮기고 같은 작업을 계속 한다.

사전의 크기는 사전에 정해져 있어야 하는데, 보통 사용되는 사전의 크기는 4096 bytes 이며, 토큰은 25bits(char/token 1 , index 12 , length 4 , next char 8)의 길이를 갖는다. 토큰의 크기가 25bits 이기 때문에 그 길이가 1 또는 2 인 문자열을 토큰으로 변환하게 된다면 오히려 압축 파일의 크기를 증가시키게 된다. 이를 방지하기 위하여 char/token 여부를 결정하는 bit 를 하나만 설정하여 크기가 1 또는 2 인 경우는 원래의 문자를 그대로 압축 파일에 저장하게 된다. LZ77 의 속도는 크게 두 가지에 의해서 결정된다.

1. 일치하는 가장 긴 문자열을 찾는데 걸리는 시간. LZ77 은 압축 시 하나의 토큰 당 평균적으로 4096bytes 크기의 사전에 대해 약 2000 번의 비교를 수행해야 한다. 굉장히 시간 소모적인 일이 된다.
2. 사전을 갱신할 때 걸리는 시간. 새로운 문자열이 이미 가득 차 있는 사전에 들어 올 경우 이 문자열을 사전에 추가하기 위해서는 기존의 문자열을 밀어내고 문자열의 길이만큼 사전의 전체 문자열이 움직여야 하므로 상당한 시간을 소모하게 된다.

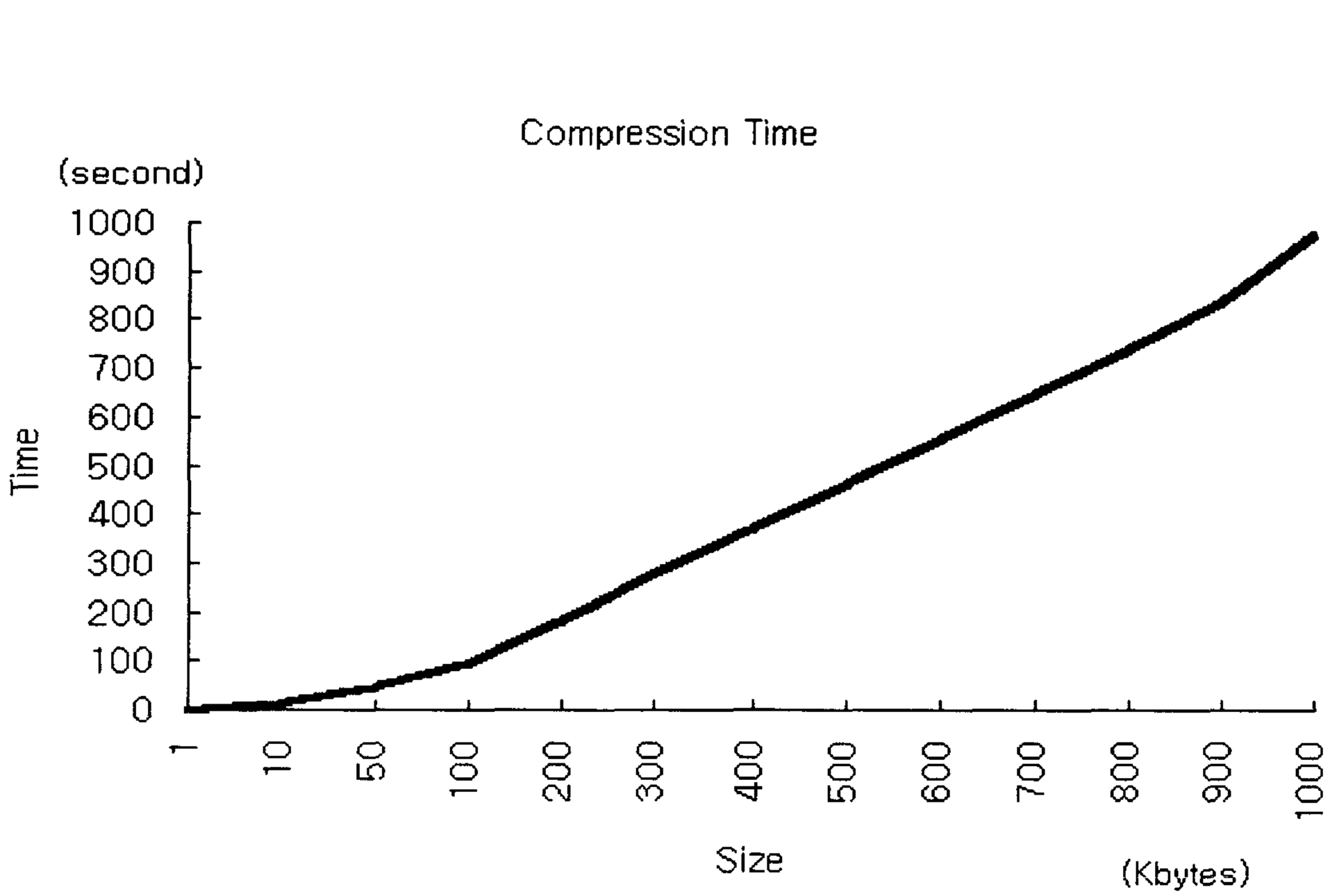
위의 1 번의 문제점을 극복하기 위해 주로 이진 트리를 사용하는데, 이진 트리를 사용할 경우 평균 2000 번의 비교 시간을 12 ~ 13 번으로 줄일 수 있어 시간의 절약을 꾀할 수 있으나, 사전을 갱신하는데 드는 시간이 더 많은 부분을 차지하기 때문에 그 속도는 상당히 느리다고 할 수 있다.

Huffman Coding 같은 경우는 압축을 해제 하기 위해 파일 내의 문자의 빈도를 압축 파일의 header 부분에 저장해 놓아야 하며, 이를 피하기 위해서는 DHC 와 같은 방법을 사용하여 동적으로 Huffman 트리를 관리하는 Routine 을

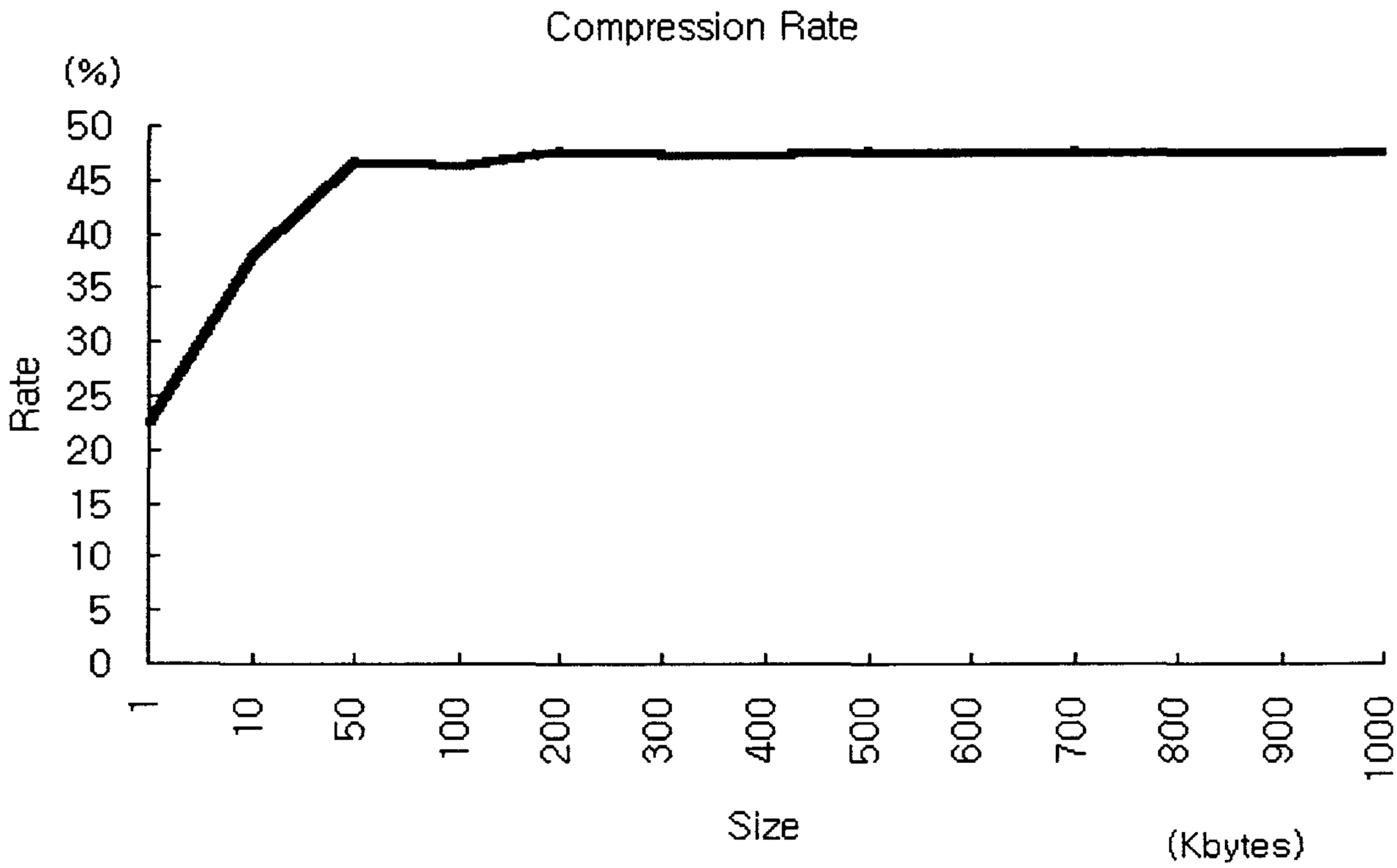
별도로 만들어 놓아야 한다. 이에 반해 LZ77 에서는 부수적인 정보를 일체 필요로 하지 않는다. 압축과 마찬가지로 파일을 처음부터 끝까지 읽으면서 사전만을 적당한 방법으로 관리하면 원래의 파일로 복구가 가능해진다. 또한 LZ77 은 복구시에는 사전을 탐색하지 않기 때문에 속도가 훨씬 빠르다. LZ77 의 복구 알고리즘은 대략 다음과 같다.

1. 한 bit 를 읽는다.
2. char 이면 문자를 읽고 이를 파일에 쓰고 , token 이면 index 와 다음 문자를 읽고 이를 파일에 추가한다.
3. 파일에 저장된 문자열을 사전에 추가한 다음 1 번으로 간다.

LZ77 은 복구시 index 를 이용해 사전을 탐색하는 시간을 절약할 수 있기 때문에 빠르게 복구하는 것이 가능하다. [그림 4.20]과 [그림 4.21]은 RLC 의 데이터에 대한 LZ77 에서의 압축 시간과 압축률에 관한 그래프이다.



[그림 4.20] LZ77 에서의 압축 시간



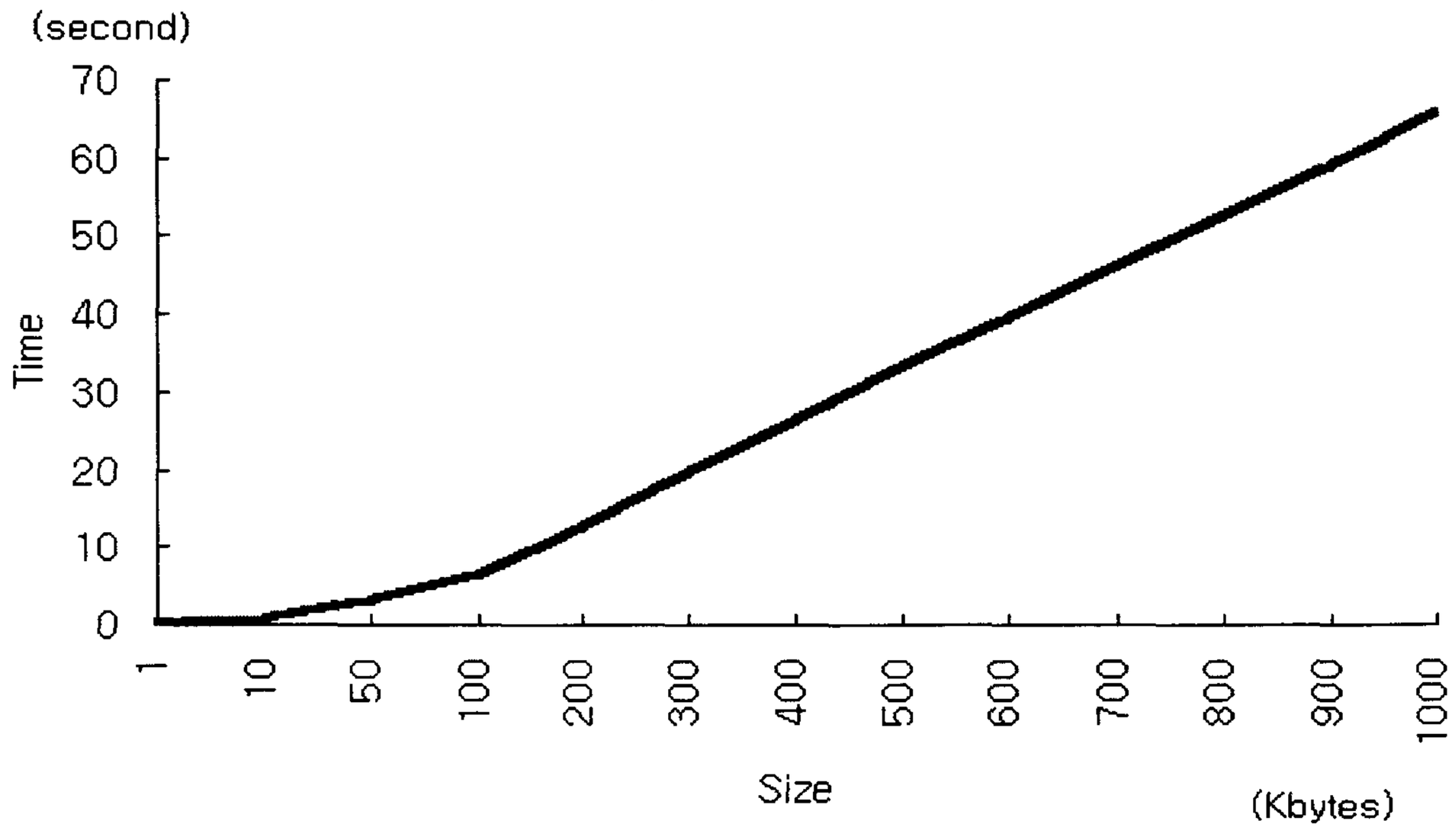
[그림 4.21] LZ77 에서의 압축률

4.2.2.3.2 LZSS

LZ77 알고리즘에서는 사전을 circularly queue 로서 관리하게 되면 사전에 새로운 문자열을 추가하는데 있어서 linear queue 를 사용하는 것보다 더 빠른 시간 안에 압축된 파일을 얻을 수 있다. 하지만 circularly queue 를 사용한다 하더라도 LZ77 의 속도는 그렇게 크게 향상되지 않는다. 사전에서 단어를 찾는 시간이 더 크게 비중을 차지하고 있기 때문이다. 즉 LZ77 에서 속도를 향상시키기 위해서는 짧은 시간에 index 를 계산할 수 있는 알고리즘이 필요하게 되는데, 이 부분에서의 속도를 향상시킨 것으로서 가장 많이 사용되는 알고리즘이 LZSS 이다. LZSS 는 LZ77 에서 두 가지를 수정한 알고리즘인데 그 주요 부분은 다음과 같다.

1. LZSS 에서는 토큰이 [index, 크기]로 구성되며, 다음 문자가 없다.
2. LZSS 에서는 사전을 queue 로서가 아니라 이진 트리 형태로 관리한다.

LZSS 는 위의 1 과 같은 특징을 갖기 때문에 토큰이 LZ77 보다 더 작은 길이로 존재할 수 있으며, 이는 곧 LZ77 에서는 압축을 포기하고 낱개의 문자로 처리해야 되는 경우에도 LZSS 에서는 하나의 토큰으로 처리하는 것이 가능하며, 그로 인해 발생하는 압축률이 크기가 큰 파일일 수록 더 좋은 결과를 나타낼 수 있다. 또한 LZSS 는 위의 2 번과 같은 특성을 갖기 때문에 속도면에서 LZ77 에 비해 압도적인 우위를 유지할 수 있다. 보통의 queue 에서 탐색을 하기 위해서는 하나씩 탐색을 함으로 인하여 평균적으로 전체 사전 크기의 대략 절반 정도의 탐색 횟수를 고려하여야 하지만, 이진 트리(binary tree)를 사용하는 경우는 $\log_2 N$ 만큼의 탐색 횟수 만으로도 LZ77 과 같은 결과를 얻을 수 있기 때문에 그만큼의 시간상의 이득을 볼 수 있다. 결론적으로 말해 LZSS 알고리즘은 LZ77 을 더욱 개선한 알고리즘으로서, LZ77 보다 속도나 압축률에서 더욱 좋은 효과를 보여주는 알고리즘이라고 말할 수 있다. [그림 4.22]와 [그림 4.23]은 RLC 의 데이터에 대한 LZSS 에서의 압축 시간과 압축률을 보여주는 그래프이다.



[그림 4.22] LZSS 에서의 압축 시간

[그림 4.23] LZSS 에서의 압축률

4.2.2.3.3 LZ78

LZ78 은 Ziv 와 Lempel 이 LZ77 을 선보인지 1 년 후 개량형으로서 발표한 알고리즘이다. LZ77 은 사전이 꽉찬 상태에서 새로운 단어를 추가할 경우 가장 앞쪽에 있는 단어를 삭제한 후에 한 칸씩 밀어내고 추가해야만 한다. 하지만 방금 삭제한 단어가 다시 나타날 경우 굉장한 손해를 볼 수 밖에 없게 된다. 이런 문제를 해결하기 위해서는 사전의 크기를 늘이는 방법을 생각해 볼 수 있는데, 만일 사전의 크기를 늘인다면 어떠한 문제가 발생할까를 생각해 보아야 할 것이다.

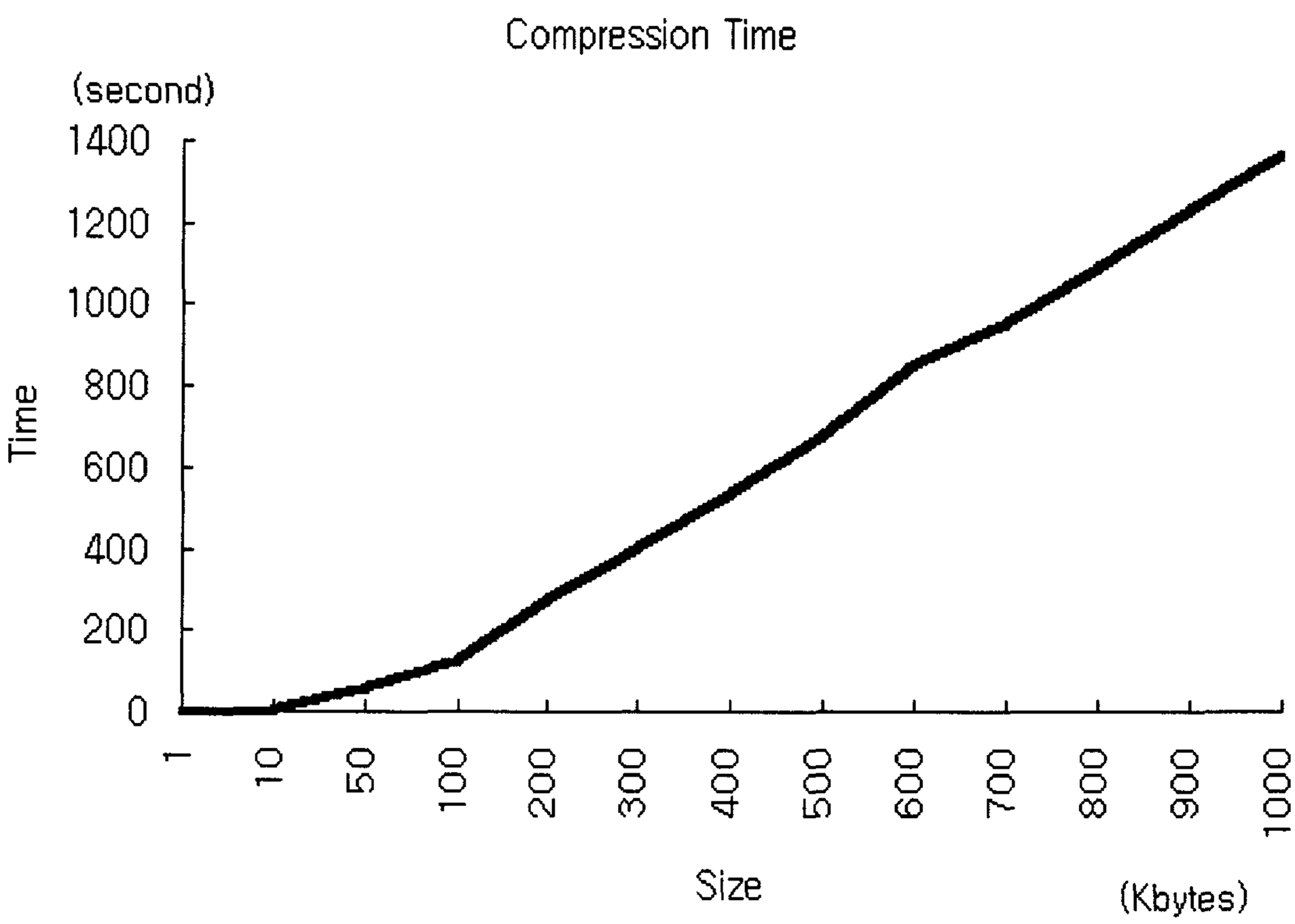
(1) 토큰의 크기가 늘어난다. 토큰의 크기가 늘어나게 되면 그만큼 압축할 수 있는 문자의 크기도 늘어나기 때문에 좋은 압축률을 기대하기는 힘들게 될 것이다.

(2) 사전의 크기가 늘어나게 된다면 사전을 관리하는데 드는 시간도 물론 더욱 많이 들 수 밖에 없기 때문에 좋은 속도를 기대하기는 힘들게 된다.

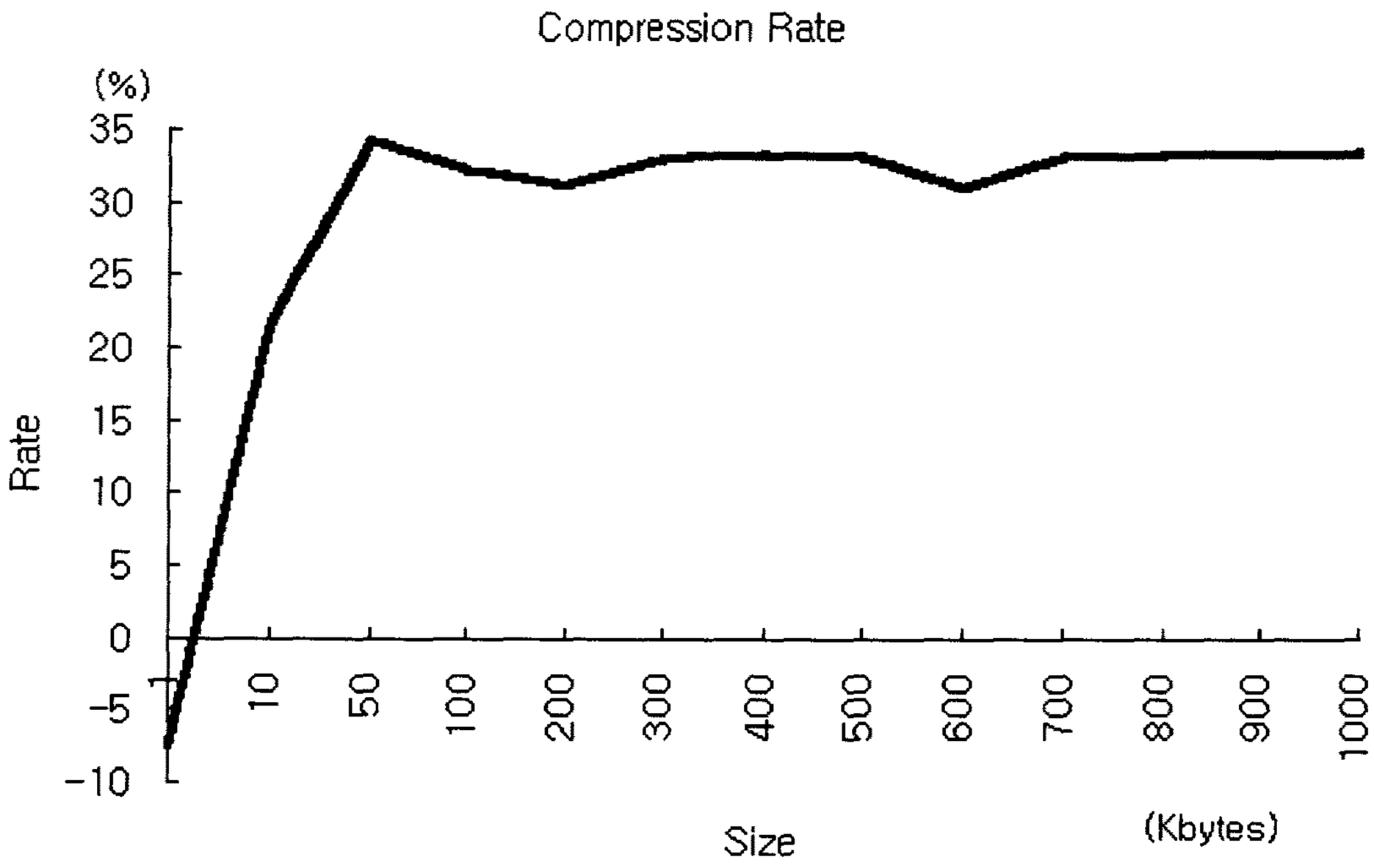
사전의 크기를 늘릴 경우 위와 같은 문제점이 발생하기 때문에 LZ78 에서는 사전의 크기를 늘리는 것이 아니라 사전에 이미 속해 있는 단어는 다시 사전에 추가하지 않는 것으로서 문제를 해결하고 있다. 이와 같은 해결책을 가지기 위해 LZ78 에서는 다중 트리(multi-branch tree)를 사용하여 이미 사전에 속해 있는 단어가 추가될 경우에는 사전에 속해 있는 단어는 그대로 두고 새로 추가되는 하나의 문자만을 다시 그 단어 밑에 추가함으로써 효율적으로 사전을 관리하게 된다.

LZ77 은 압축될 수 있는 문자열의 길이가 미리 정해진 길이를 초과 할 수 없기 때문에 더 긴 문자열이 존재할지라도 압축할 수 없는 경우가 발생하게 된다. 하지만 LZ78 에서는 토큰이 [index, next char]로 구성되기 때문에 긴 문자

열이라도 사전의 크기 한도 내에서는 표현이 가능하게 되며, 이로 인해 더 좋은 압축률을 나타낼 수 있다. [그림 4.24]와 [그림 4.25]는 RLC 의 데이터에 대한 LZ78 에서의 압축 시간과 압축률을 보여주는 그래프이다.



[그림 4.24] LZ78 에서의 압축 시간



[그림 4.25] LZ78 에서의 압축률

4.2.2.3.4 LZW

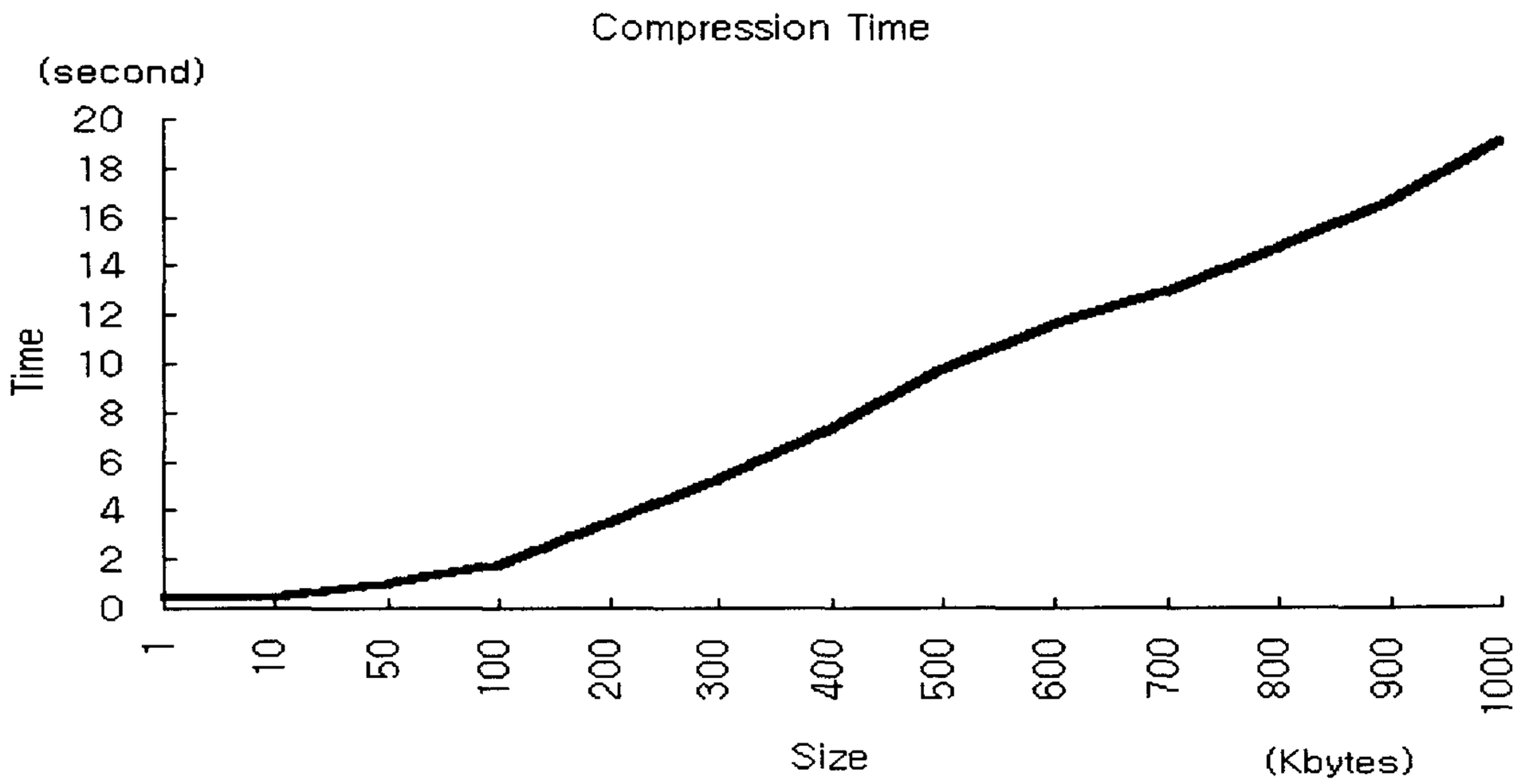
비록 LZ78 이 LZ77 의 단점을 어느 정도 해소하고 있다 할지라도 LZ78 에도 나름대로의 약점은 존재한다. 그 약점은 다음과 같다.

1. index 가 0 인 경우 발생하는 오버헤드를 없애거나 최소화해야 한다. 즉, LZ78 은 모든 문자를 토큰으로 변화시키기 때문에 최초로 사전에 들어가게 되는 문자는 갖지 않아도 될 낭비를 갖는다. 토큰의 크기가 문자 1 개의 크기보다 크기 때문에 발생하는 문제이다.
2. LZSS 처럼 토큰에서 다음 문자를 없앨 수만 있다면 더욱 좋은 효율을 나타낼 수 있을 것이다.

위와 같은 사실에 주지하여 나타난 알고리즘이 LZW 이다. LZW 에서는 처음 시작 전에 미리 사전에 256 개의 ASCII 문자를 추가시킨 후 시작하기 때문에

위의 1 과 같은 공간의 낭비를 어느 정도 해결한다. 그리고 다음 문자를 토큰에서 삭제하기 때문에 압축 효율을 좀더 높일 수 있도록 할 수 있다.

결론적으로 LZW 는 LZ78 에 비해 속도는 비슷한 수준을 유지하면서 압축률을 좀 더 높게 할 수 있는 특징을 갖는 알고리즘이라고 할 수 있다. [그림 4.26]과 [그림 4.27]은 RLC 의 데이터에 대한 LZW 에서의 압축 시간과 압축률을 보여주는 그래프이다.



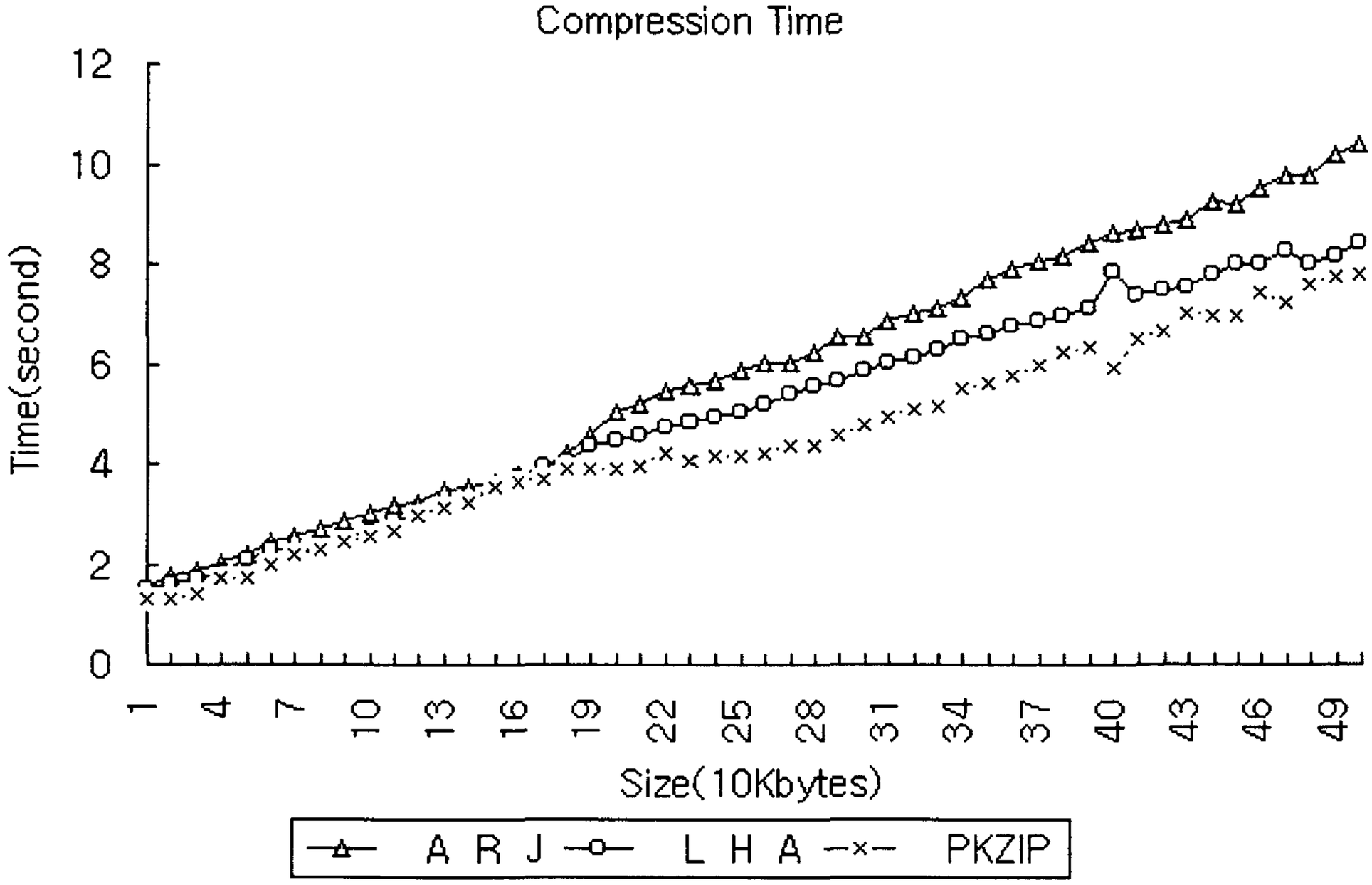
[그림 4.26] LZW 에서의 압축 시간

[그림 4.27] LZW 에서의 압축률

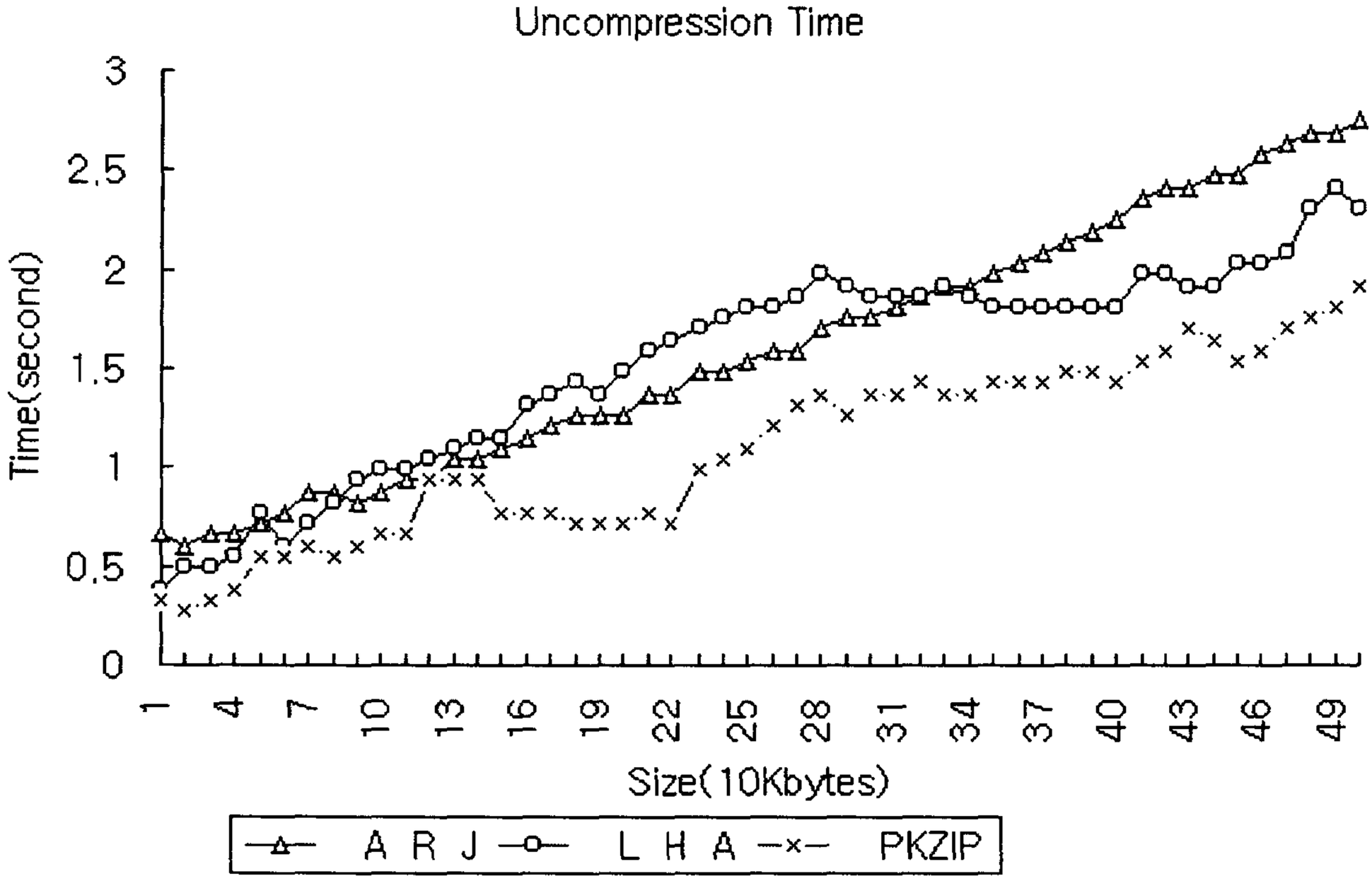
4.2.3 압축 알고리즘의 성능 실험

압축 알고리즘은 두 가지의 측면 즉, 원시 자료에 비해 얼마나 작은 크기로 줄어드는가를 측정하는 공간적 측면과 압축할 때 또는 복구하는데 얼마나 빠른 시간에 처리하는가 하는 시간적 측면에서 비교된다. 이미지 자료나 같은 정보가 반복적으로 나타나는 자료에 아주 유용한 RLC(Run Length Coding)가 있고, 전체 자료에 대해 나타난 심볼의 빈도에 따라 코드의 길이를 가변적으로 적용시키는 VLC(Variable Length Coding, 가변길이코딩) 방법인 허프만 코딩 방법이 발표되었다[8, 20]. 그 후 자콥 지브(Jacob Ziv)와 아브람 렘펠(Abraham Lempel)에 의해 사전식 압축 알고리즘인 DBC(Dictionary Based Coding) 방법 또는 LZ 계열의 알고리즘이 소개 되면서 압축 알고리즘 연구가 활발해졌다[13, 21, 30]. 이를 계기로 상용화되거나 하나의 유틸리티로 일반 사용자에 소개된

프로그램들은 대부분 허프만 코딩과 DBC 방법을 혼용하여 채택하고 있다. 이런 이유로 압축 알고리즘들은 현재 압축율에서는 자료의 종류에 매우 민감하며 같은 종류의 자료에 대해서는 거의 비슷한 압축율이나 처리 시간을 갖고 있다. 여기에서는 압축 유틸리티를 중심으로 각각의 성능 실험 분석을 하고자 한다. 압축 유틸리티로는 ARJ, LHA, PKZIP 이다. [그림 4.28]과 [그림 4.29]에서 나타난 것 처럼 압축 시간과 복구 시간은 PKZIP 과 LHA 가 조금 나은 결과를 얻을 수 있었다.



[그림 4.28] 압축 시간 비교



[그림 4.29] 복구 시간 비교

4.2.4 RECORDid 를 위한 입출력 시간의 비교 분석

4.2.4.1 RECORDid 의 구성

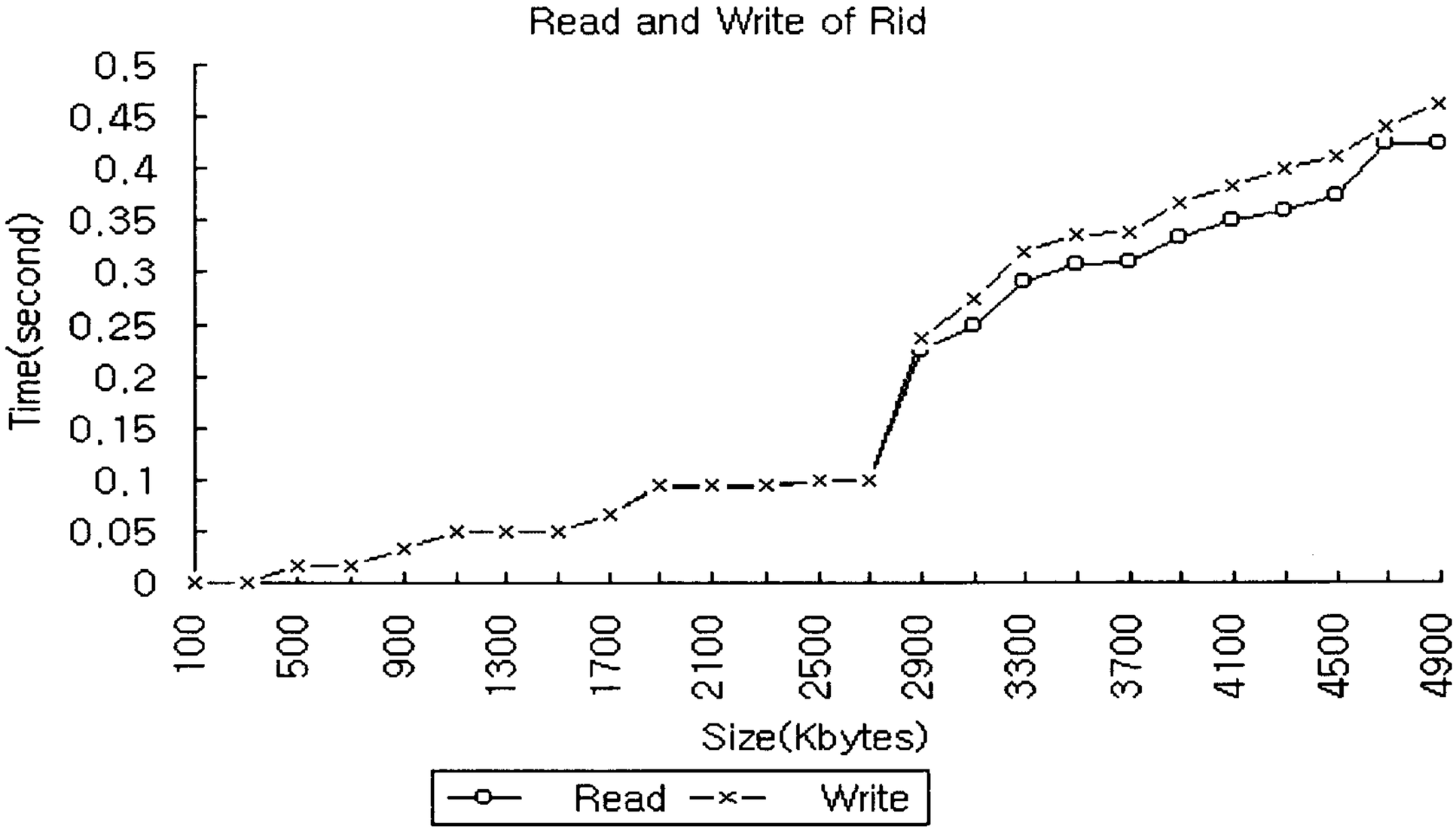
정보 검색 시스템은 대용량의 각종 정보 보관을 위해 분산된 위치에 데이터베이스를 유지해야 한다. 정보의 양이 증가함에 따라 키워드나 단어에 연관된 자료의 위치와 관련된 정보가 갱신되고 길어질 것이다. 이러한 정보의 동적인 변화가 RID 에 등록되고 이 정보에 의해 검색하고자 하는 영역을 제한시킴으로써 보다 빠른 검색이 가능하다. 그러나 RID 의 크기가 커진다면 이를 읽고 처리하는 시간이 소모될 것을 예상할 수 있다. 그리고 현재 조사된 바에

의하면 하나의 RID 크기는 수십 바이트에서 크게는 수만 바이트에 불과한 것으로 나타나 있다[7]. 약 8 만 여개의 RID 실험 데이터를 조사한 바에 의하면 0 byte 가 50 % 정도 나타나고 있는 것 외에는 크게 특징적인 것은 없었다.

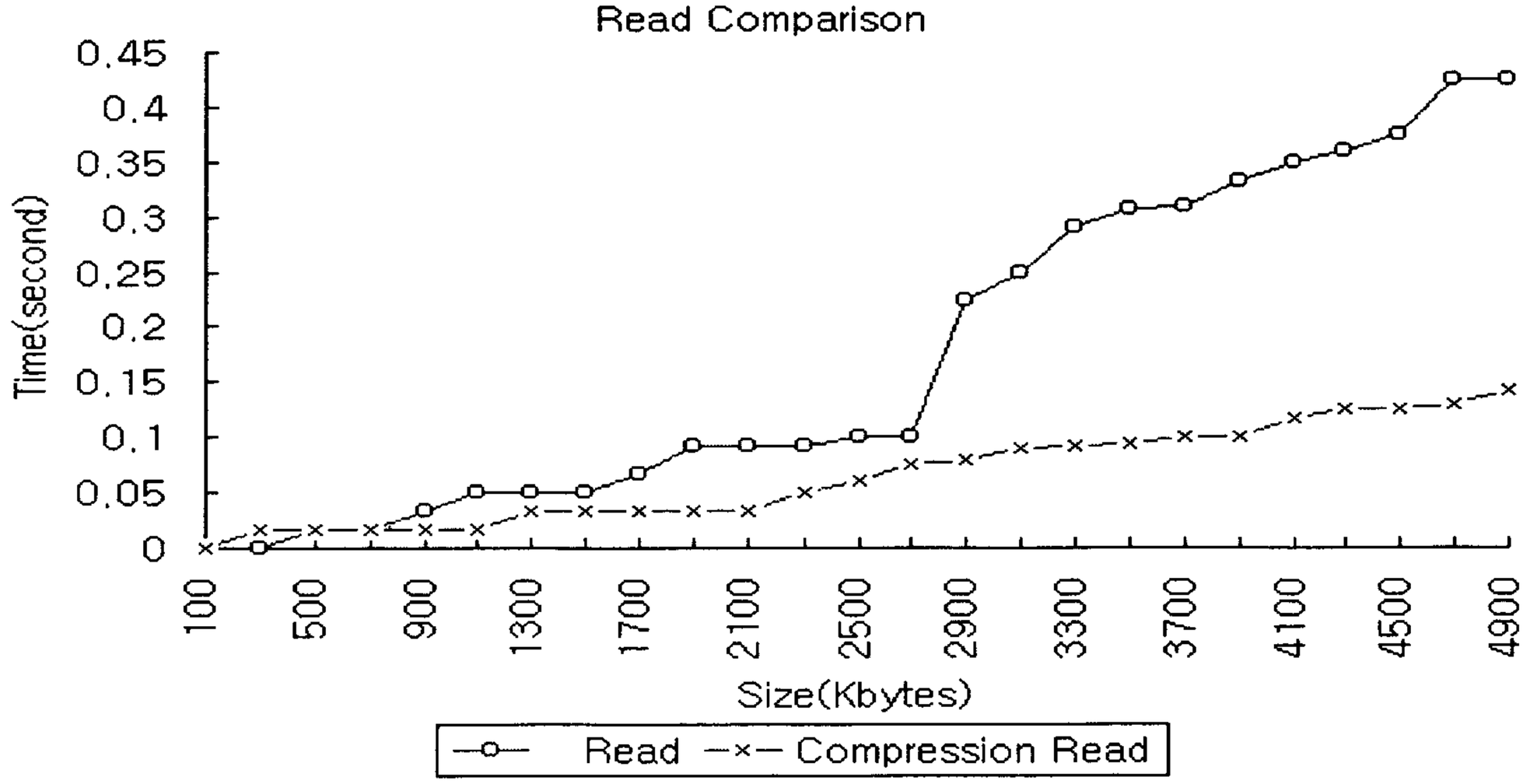
4.2.4.2 RECORDid 입출력 시간 분석

앞 절에서 살펴본 바와 같이 RID 의 크기는 매우 불규칙하다. 또한 크기가 작은 레코드들도 매우 많다. 일반적으로 효율적인 공간 활용을 위하여 압축 기법을 사용하게 되는데, RID 를 위한 압축을 했을 때의 경제성을 고려하지 않을 수 없다. 왜냐하면 파일의 크기에 따라 입출력 및 압축, 해제를 위해 소모되는 시간들간에 불균형으로 도리어 비효율적인 결과를 초래하기 때문이다. 이에 대한 레코드의 크기와 이에 대응하는 압축 파일의 크기, 또한 이를 원시 레코드로 만들기 위한 시간들을 실험을 통하여 살펴본다.

[그림 4.30]은 일반적인 RID 을 입출력하기 위해 소요되는 시간을 측정한 것이다. 이 실험에 사용된 시간은 Sparc 20 UNIX machine 에서 실행한 결과이다. 이때 RID 의 크기가 400 Kbyte 이하일 경우 시간이 똑같이 소요됨을 알 수 있었다. 이것은 시스템이 입출력을 위해 요구되는 최소한의 시간임을 알 수 있었다. 이는 RID 입출력을 위해서는 최소한 400 Kbyte 이상이 되어야 효율적임을 알 수 있었다. 또한 파일의 크기가 커짐에 따라 입출력 시간이 선형적으로 증가됨을 보이고 있다.



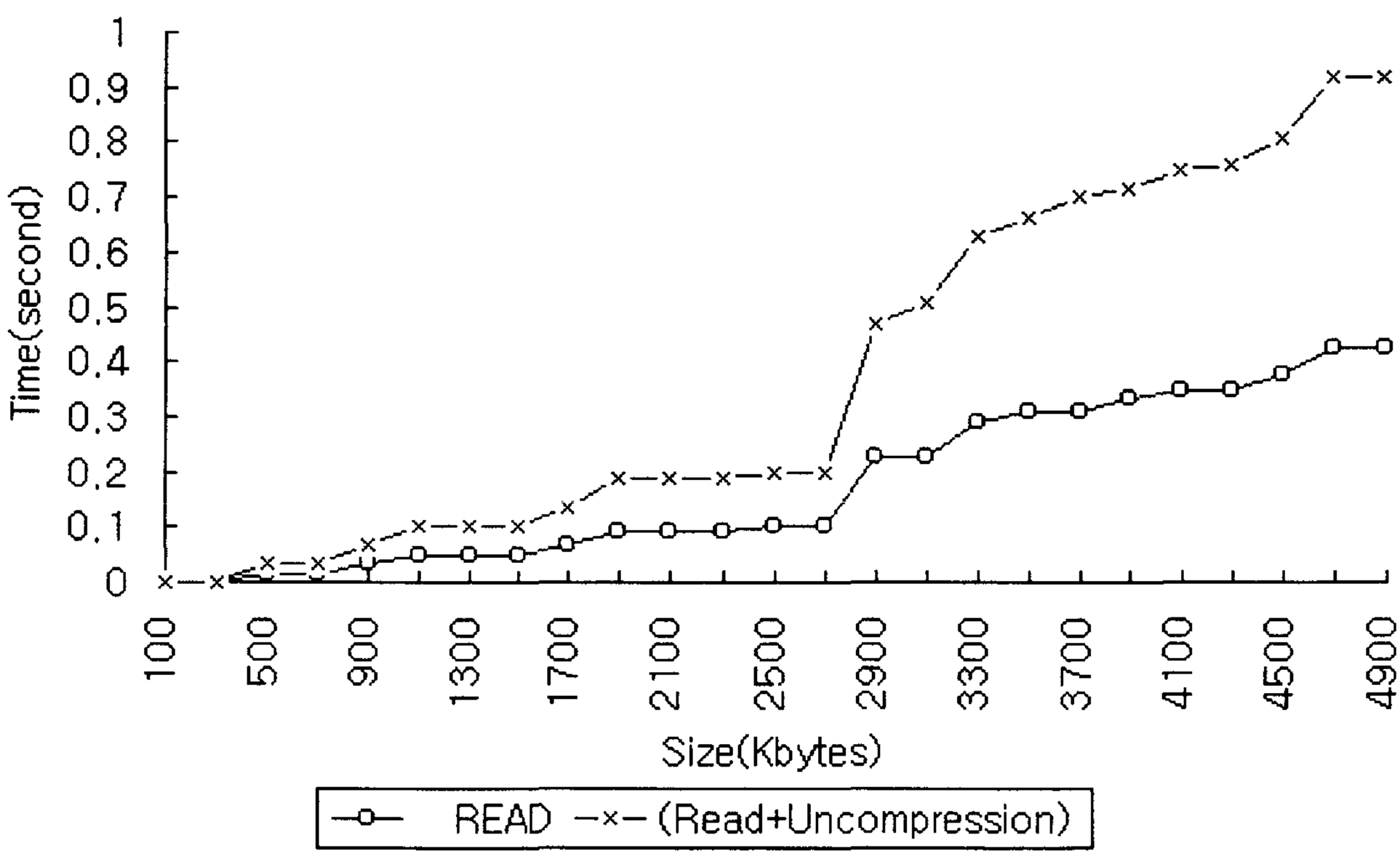
[그림 4.30] RID 크기에 따른 입출력 소요 시간



[그림 4.31] RID 와 압축된 RID 의 입출력 관계

[그림 4.31]은 RID 의 크기에 따른 입출력 시간을 나타내고, 같은 파일을 압축했을 때의 입출력 시간을 나타낸 것이다. 이 그림에서 나타나 있듯이 결국 파일의 크기가 압축에 의해 작아지므로 입출력 시간은 당연히 감소하게 된다.

앞의 실험에 의해 우리는 [그림 4.32]에 나타난 것처럼 RID 원시 파일의 입출력과 그 파일에 상응하는 압축된 파일의 출력 시간과 압축 해제 시간의 합을 나타낸 그래프를 비교하고 있다. 이 실험에서 우리는 RID 원시 파일의 입출력 시간과 압축된 RID 파일의 출력 시간과 해제 시간의 합이 서로 교차하여 만나는 시간의 분기점을 찾고자 하였다. 그러나 그림에서 나타난 것처럼 압축 파일의 복구 시간이 차지하는 시간 때문에 압축 알고리즘을 사용하는 것이 비능률적임을 보이고 있다.



[그림 4.32] RID 입출력 시간과 압축된 RID 의 입출력과 복구 시간과의 관계

4.2.5 성능 분석

4.2.5.1 성능 분석을 위한 변수

이 장에서는 RID 압축 시스템을 사용한 정보처리 시스템의 성능을 평가하고자 한다. 먼저 시스템의 성능을 평가하기 위해서 이를 결정하는 다양한 환경 변수를 미리 확정해야 한다. 본 논문에서 고려한 정보 검색 시스템 (Information Retrieval System) 성능은 다음의 변수로 평가하고자 한다.

1. System S 의 전체 디스크 공간 : D_S
2. System S 의 I/O 버퍼의 크기 : B_S
3. System S 의 메인메모리 크기 : M_S
4. Double Buffering 의 여부: boolean variable DB_S = *true*, *false*
5. CPU 의 속도, 하나의 기본 동작(basic operation)을 처리하는데 걸리는 시간 : $t_{cpu} = t$ sec/opr.
6. Disk Reading transfer Rate, 단위 시간당 디스크에서 메모리 버퍼로 읽히는 자료의 양, DR_S(r Mega Byte/sec). 그리고 단위 크기 M인 파일을 메모리로 읽어 들이는데 걸리는 시간은 t_{read} 로 표시한다.
7. 프로그램에서 디스크에 read 를 요구하는 파일의 평균 크기, 여기서는 보통 요구되는 RID 를 저장하고 있는 subfile 의 평균 크기: RID_{avg} 또는 좀 더 정확하게 분석하려면 RID 의 평균 크기와 그 확률 분포 함수 $F_{RID}()$가 필요하다.

위에서 입출력 버퍼 크기가 고려되어야 하는 이유는 그 크기 이하의 단위로 입출력을 한다 하더라도 실제 디스크 입출력 상의 이득은 없다. 왜냐하면 항상 디스크와 메모리 버퍼상으로 움직이는 단위는 B_S 단위로 이동이 이루어지므로 시간상으로 절약되는 것이 없다.

일반적으로 시스템의 성능을 평가하는 항목에는 여러 가지 변수가 있다. 예를 들면, 하나의 작업 요구가 받아 들여진 뒤 그 결과가 나올 때까지 사용자가 기다리는 시간인 turn around time, 또는 단위 시간당 처리된 트랜잭션의 수인 throughput, 각 트랜잭션 당 사용한 CPU 시간, 전체 시스템의 공간 활용 정도 등이다. 평가하고자 하는 항목을 식으로 나타내면 다음과 같다.

1. Turn around time TR(S,M); 시스템 S 에서 보통의 사용자가 크기 M 인 RID 를 요청하여 그 결과를 받아 볼 수 있을 때까지 걸리는 시간
2. 시스템 S 에서 파일을 읽어들이는 시간, 크기가 K byte 인 파일을 디스크에서 읽어들이는 시간 *Read(S,K)*
3. Disk utilization: 시스템 S 의 디스크에서 단위 공간당 저장되어 있는 레코드의 수, $U(S) = n \bullet RID_{avg}/D_S$, 여기서 n 은 시스템에 저장되어 있는 RID 의 갯수

본 보고서에서는 정보 검색 시스템에서 압축 시스템을 추가로 활용했을 때 어떤 이득이 있는지를 살펴보는 것이다. 따라서 압축 시스템의 성능을 변수화 할 필요가 있다.

1. Compression Rate : 파일 크기 M 인 보통의 RID 를 압축했을 경우 그 크기가 축소되는 비율, CompRate(M) = size of compressed file / original file size
2. Compress Time : 파일크기 M 인 RID 를 특정한 압축 프로그램으로 압축했을 경우에 걸리는 시간, CompTime(M)
3. Decompression Time : 압축된 파일을 복구할 때 걸리는 시간, DeCompTime(M).

본 보고서에서 행한 실험에 의하여 위의 압축에 관련된 세 함수를 결정해 보기로 하자. 먼저 압축율은 대략 1000 byte 이하의 RID 에 대해서는 거의 이득이 없음을 볼 수 있었다. 그리고 그 이상인 경우에는 대략 전체가 40%까지

압축이 가능하므로 우리는 이 압축율 함수를 파일 크기에 관계없이 0.4 로 결정했다. 즉, CompRate(M)은 다음과 같은 상수 함수로 나타낸다.

$$CompRate(M) = 0.4$$

그 다음 CompTime(M)은 실험에 의하면 거의 파일의 크기가 일정 정도 이상인 경우에 그 시간은 선형으로 증가됨을 볼 수 있었다.

$$\text{CompRate(M)} = t_{comp} \bullet \text{M} = 5.0 \bullet \text{M(MegaByte/sec)}$$

이와 유사하게 복구 시간도 거의 선형적으로 증가됨을 볼 수 있었다. 따라서

$$\text{DeCompRate(M)} = t_{decomp} \bullet \text{M} = 1.0 \bullet \text{M(MegaByte/sec)}$$

여기에서 사용된 t_{comp} 와 t_{decomp} 수는 Sparc20 UNIX machine 에서 실험적으로 얻어진 것이다.

4.2.5.2 성능 비교

이 절에서는 압축 시스템을 부가적으로 사용했을 때와 그렇지 않을 경우에 어떤 이득이 있는지 살펴본다. 먼저 S_n 은 보통의 시스템이라고 하고, 압축기를 첨가한 경우의 시스템을 S_c 라고 한다. 이 S_c 시스템은 디스크와 메모리 사이에 존재하여 디스크 입출력시에 압축과 복구를 담당한다. 먼저 크기 M 인 파일을 S_n 에서 사용할 때 걸리는 시간은 다음과 같다.

$$\text{Read}(S_n, \text{M}) = t_{read} \bullet \text{M(MegaByte/sec)}$$

이에 비해서 S_c 에서는 다시 압축된 파일을 복구하는 부가적인 시간이 필요하므로 다음과 같이 계산된다.

$$\text{Read}(S_c, \text{M}) = t_{read} \bullet \text{CompRate(M)} + \text{DeCompTime(CompRate(M))}$$

여기에서 $\text{Read}(S_c, \text{M})$이 $\text{Read}(S_c, \text{M})$ 보다 빠르기 위해서는 다음 조건이 성립해야 한다.

$$\text{Read}(S_c, \text{M}) < \text{Read}(S_n, \text{N})$$

$$t_{read}\,(\,M - CompRate(\,M\,)\,) - DeCompTime(\,CompRate(\,M\,)\,) > 0$$

위 조건이 만족되면 실제 시간상으로 이익을 볼 수 있다. 실제 본 실험에서 사용된 측정치를 이용해서 이 조건을 계산하면 다음과 같다.

$$Profit = 1.0\ sec/5m.b.(M - 0.4\ M\,) - 1.0\ sec/1m.b. \bullet (0.4M)$$

$$= 0.2\ (0.6\ M) - 0.4\ M = 0.12 - 0.4 = -0.28 < 0$$

위의 식으로 보아 초당 1 메가바이트 정도로 압축해서는 시간상으로 큰 이득이 없다고 볼 수 있다. 따라서 CPU 시간상으로 이득을 가지려면 위의 조건을 만족해야 하는데, 일반적으로 RID 의 압축률이 0.4 정도가 거의 하한선이라고 볼 때 이득이 있을 정도의 CPU 속도가 되려면 초당 5 메가 바이트를 전송할 수 있는 디스크 시스템에서 초당 약 4 메가바이트를 복구할 수 있는 정도의 속도가 보장되어야 한다. 식으로 표현하자면 다음과 같다.

$$t_{read} \bullet 0.4\ M + t_{decomp} \bullet 0.4\ M < t_{read}\ M$$

$$t_{decomp} / t_{read} < 1.5$$

이것은 단위 크기당 그것을 복구하는 시간이 디스크 입출력 보다 1.5 배 이하이면 이득을 가질 수 있다는 뜻이 된다. 즉 본 실험에서 사용된 결과를 가지고 말한다면, 초당 5 메가 바이트를 전송하는 시스템에서 압축을 해서 이득을 얻을 수 있으려면 복구 시간이 가장 빠른 LZ 계열의 알고리즘을 사용해서 대략 초당 3.5 메가 정도는 복구를 해야 한다. 하지만 현재 실험에 사용 중인 Sparc 20 계열의 CPU 성능을 고려해서 본다면 대략 초당 1 메가 정도이므로 시스템 내에서의 user time 면에서는 이득이 없다고 보겠다.

다음에는 디스크의 사용 효율면을 비교해 보자. 디스크 공간의 효율은 압축율 만큼이나 좋아진다. 따라서 공간 효율은 1/CompRate(M) 만큼 높아진다. 실험에 사용된대로 RID 를 압축한다면 1/0.4 = 2.5 배 만큼의 공간 효율을 극대화시킬 수 있다.

$$U(S_c) = 1/CompRate \bullet U(S_n) \approx 2.5$$

4.2.6 Bit Vector Algorithm

지금까지 제시된 알고리즘들의 압축률은 대체적으로 일반적인 데이터에 대하여 수용할만한 수준의 압축률을 제공해 준다. 하지만 압축률과 시간은 일반적으로 반비례 관계를 가지고 있기 때문에, 시간이 중요한 요소로 작용하는 데이터들에 관하여서는 위에 제시한 알고리즘들이 적당하다고 말할 수 없다. RID 에 있어서도 위에 제시된 알고리즘들이 압축률에서는 크게 떨어지지 않으나 압축된 데이터를 풀기 위해 많은 시간을 할당하여야 하기 때문에 적당한 방법이 되지 못한다.

지금까지 조사한 바에 의하면 RID 가 가지는 특징은 거의 절반에 가까운 ASC(0)을 가지고 있다는 것이다. 하지만 ASC(0)이 집중적으로 모여있지 않고 랜덤하게 흩어져 있기 때문에 비록 RLC 가 시간면에서는 다른 알고리즘에 비해 유리하지만 RLC 를 사용해서는 좋은 압축률을 획득하기가 힘들다.

따라서 이에 대한 대응 방안으로서 다음과 같은 알고리즘에 대한 제안을 가질 수 있을 것이다.

조 건 :

전체 데이터 중의 상당한 정도가 같은 문자에 치중되어진다. (현재 RID 의 경우가 이에 해당한다.)

알고리즘 :

데이타중 가장 높은 비율의 문자를 X 라 하자. X 가 전체 데이터에서 차지하는 비율을 R%라 하자. X 문자를 bit(0)으로 이외의 문자를 bit(1)로 나타낸다면

```
in = read (inbuffer);
if (in = X)
```

```
                outbit = bit(0);
else
                outbuffer = in;
                outbit = bit(1);
}
```

압축된 데이타의 크기 comp_size(D)는 다음과 같다. 즉, R 이 높을수록 더욱 높은 압축 효율을 나타낸다.

$$comp\_size(D) = \frac{1}{8}(size(D)) + size(D) \times \frac{(100 - R)}{100}$$

다음은 Bit Vector Algorithm 을 사용하여 압축한 예이다. 아래에서 a 가 bit 0 으로, 나머지 문자는 bit 1 로 압축되어진다.

Source Data = {a,b,c,a,f,a,a,r,g,a,a,k,a,l,o,a,p,s,a,e,a} - 21bytes

Compressed Data = {b,c,f,r,g,k,l,o,p,s,e} - 11bytes

Bit Vector = [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,,0,1,1,0,1,0] - 21bits

문자 a 가 전체 21bytes 중 10bytes 를 차지하기 때문에, 압축률은 50%에 미치지 못하지만 시간상의 이득을 얻을 수 있다.

위의 알고리즘으로 알 수 있듯이 실제 데이터를 읽는 시간과 한번의 비교만으로 압축과 관련된 작업이 끝나기 때문에 기존의 알고리즘에 비해 압축률이 다소 떨어진다 하더라도, 시간과 관련되어서는 상당히 높은 효율을 가진다고 할 수 있다. 압축을 풀 때의 알고리즘도 역시 한번의 비교만이 추가되므로 시간이 압축률에 비해 중요한 비중을 차지하는 데이터의 경우 고려할 수 있는 알고리즘이다.

4.2.7 실험 결과

우리는 본 실험을 통하여 알 수 다음 사실들을 있었다.

- 현재 정도의 randomness 를 가진 RID 구조라고 할 때, 그 압축율을 40% 이상 올리기는 어렵다. 따라서 압축 시스템이 첨가된 정보 검색 시스템에서 전체적인 시간상의 이득을 얻기 위해서는 CPU 의 처리속도가 디스크의 전송율(transfer rate)에 가까울 정도로 개선되어야 한다.

- 현재 보통의 SCSI 디스크에서 얻을 수 있는 자료 이동에 관한 성능이 초당 5 ~ 10 메가 바이트라고 할 때 압축 파일의 복구를 위한 시스템의 성능은 초당 4.8 메가 바이트 이상씩을 회복시킬 수 있어야 한다.

- 그러나 앞의 장에서 계산한 시간은 CPU 시간과 시스템 시간상의 계산이므로 실제 사용자의 turn-around time 을 고려한다면 위에서 제시된 모델은 훨씬 복잡해지므로 실제 동작 중인 시스템에서의 다중 사용자, 다중 프로그래밍 시스템에서 그 실제적인 상황을 만들어서 구체적인 압축 시스템의 성능을 측정할 필요가 있다. 특히 병렬 처리 장치가 있는 시스템에서의 압축이 지금보다 빨라진다면 의미를 찾을 수 있다.

이상의 사실을 종합해 볼 때 지브-렘펠(Ziv-Lempel) 계열의 압축 알고리즘은 텍스트 종류에 상관없이 상당히 좋은 성능을 보여주지만 본 정보 검색 시스템에서 사용하기에는 상대적인 압축 속도가 떨어짐을 알 수 있었다. 현재의 모든 LZ77, LZW 계열은 이미 압축된 자료에 대한 정보를 그 압축 파일 자체에 가지고 있다. 이 때문에 사용에 편리함은 있지만 그 복구 속도는 압축의 과정을 반대로 거슬러 올라가야 함으로 시간이 걸릴 수 밖에 없다. 따라서 LZW 계열의 알고리즘은 더 이상 고려할 가치가 없다고 보여진다. 따라서 압축기가 달린 정보 검색 시스템이 의미를 가지기 위해서는 디스크 입출력 속

도 정도의 빠른 압축 속도를 보장하는 다른 종류의 알고리즘이 제시되어야 할 것으로 믿어진다. 최근에 소개된 "Adaptive Encoding for Numerical Data Compression"은 수치 자료 도메인에서 좋은 성능을 보여주므로 이러한 domain-oriented compression technique 이 새로이 개발되어야 할 것이다. 또는 이미 전체 RID 자료에 관련된 정보 사전(code book)을 갖고 있다면 그 속도를 획기적으로 증가시킬 수 있을 것이므로 이러한 방향으로 연구가 진행되어야 할 것이다.

현재 실험 결과로 볼 때 압축된 자료를 한 번 접근하는 속도는 디스크 입출력 속도보다 훨씬 빠르므로 대략 2 ~ 3 번 정도의 code book 을 살펴보는 정도에서 압축을 풀 수 있다면 충분히 가치가 있을 것이다. 앞 장에서 보여준 Bit Vector Algorithm 은 이러한 면에서 볼 때 시간과 공간상의 이점을 가지는 알고리즘이라고 할 수 있을 것이다. 단지 같은 문자가 일정량 이상 반복되어지는 경우에만 수용할 만큼의 압축률을 획득할 수 있다는 단점을 극복하는 방안이 요구되어진다.

일반적으로 사용되어지는 압축 알고리즘은 변화량이 많은 실제 시스템에서의 사용이 부적합하기 때문에, 실제 사용되어지는 시스템의 성능(job scheduling, I/O channel 수, cpu 수, 평균적인 system load 등)을 측정하여 이를 반영한 압축 방법을 사용하여야만 좋은 효과를 얻을 수 있을 것이다.

제 5 장 결 론

과학 기술 정보 유통의 핵심 중의 하나는 복잡, 다양해지고 기하급수적으로 증가하는 정보들을 효율적으로 저장하고, 사용자에게 신속히 제공해 줄 수 있는 시스템의 연구 및 개발이다. 지금까지 국내에서는 DBMS 용 저장 시스템에 대한 연구 및 시제품 개발은 진행되어 왔으나, 비정형 데이터의 저장, 근접도 연산 등을 지원해야 하는 정보 검색용 저장 시스템의 연구 개발은 미비한 실정이다. 일반적으로 정보 검색용 저장 시스템은 사용 목적과 환경에 따라 그 기능을 확장 또는 축소하는 것이 바람직하다. 그러나 외국의 시스템을 도입하여 사용할 경우, 별도로 원시 코드가 제공되지 않는다면 기능의 확장 또는 축소를 위한 시스템의 변경이 근본적으로 불가능하다. 따라서 이러한 문제점을 궁극적으로 해결하기 위해서는 효율적인 정보 검색용 저장 시스템의 개발이 시급하다.

본 연구에서는 "정보검색을 위한 효율적인 저장시스템 개발" 과제의 최종 목표인 정보 검색용 저장 시스템의 개발을 완료하였다. 개발된 저장 시스템은 입출력 관리, 버퍼 관리, 화일 디렉토리 관리, 레코드 관리, 대용량 객체 관리를 담당하는 정보 저장 지원기, 문서 관리, 색인 관리, 카타로그 관리를 담당하는 정보 저장 관리기, 그리고 이진 영상 및 컬러/그레이 영상을 압축 복원하고 색인화일의 문서 식별자 리스트를 압축 복원하는 정보 압축 복원기로 구성된다.

본 연구에서 개발된 저장 시스템의 성능평가를 위하여 연구개발정보센터가 보유하고 있는 과학 기술 정보들을 대상으로 실정보 적용 실험을 수행하였으며, 이를 통하여 활용 가능성을 확인하였다. 개발된 저장 시스템은 연구개발

정보센터에서 개발 중인 정보 검색 시스템 KRISTAL-II 의 하부 저장 시스템으로 사용 가능하다. 또한 정보 시스템을 구축하고자 하는 모든 분야에 보급함으로써 다양한 정보 시스템의 하부 저장 시스템으로 활용될 수 있을 것으로 기대된다.

참고 문헌

[1] "JPEG CD 1098-1 Digital compression coding continuous-tone still image part I", 1991

[2] A. B. Watson, "DCT quantization matrices visually optimized for individual images", Proc. of SPIE, Vol. 1913, pp.202-216, 1993

[3] A. N. Netravali and B. G. Haskell, "Digital Pictures", Plenum Publishing Co., 1988

[4] C. A. Lindley, "Practical Image Processing In C", New York: John Wiley & Sons, Inc., 1991

[5] Carey J. Michael, David J. Dewitt, Joel, E. Richardson, and J. Shekita, "Object and File Management in the EXODUS Extensible Database System", Proc. of the 12th Int'l on VLDB, April 1986

[6] Computer Sciences Department University of Wisconsin Madison, "Design and Implementation of the Wisconsin Storage System", July 1984

[7] D. A. Huffman, "A method for the construction of minimum-redundancy codes", Proc. IRE Vol. 40, No. 9, pp.1098-1101, Sep. 1952

[8] D. A. Lelewer and D. H. Hirschberg, "Data Compression", ACM Computing Surveys, Vol. 19, No. 3, pp.261-296, Sep. 1987

[9] D. R. Fuhrmann, J. A. Baro, and Jerome R. Cox, "Experimental evaluation of psychophysical distortion for JPEG-encoded images", Proc. of SPIE, Vol. 1913, pp.179-190, 1993

[10] D. Wang, and S. N. Srihari, "Classification of Newspaper Image Blocks Using Texture Analysis", CVGIP-47, pp.327-352, 1989

[11] F. M. Wahl, K. Y. Wong, and R. G. Casey, "Block Segmentation and Text Extration in Mixed Text/Image Documents", CVGIP-20, pp.375-390, 1982

[12] Gregory K. Wallace, "The JPEG Still Picture Compression Standard", Communications of the ACM, Vol. 34, No. 4, pp.30-43, 1991

[13] H. Yokoo, "Adaptive Encoding for Numerical Data Compression", Information Processing and Management. pp.863-873, vol. 30, No. 6, 1994

[14] H.T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug, "Design and Implementation of the Wisconsin Storage System", Software Practice and Experience, Vol. 15(10), pp.943-962, Oct. 1985

[15] Hudson, G. P., Yasuda. H., and Sebestyen. I, "The international standardization of a still picture compression technique", Proc. of the IEEE Global Telecommunication Conference, IEEE Communication Society, pp.1016-1021, 1988

[16] I. H. Witten, A. Moffat, and T. C. Bell, "Managing Gigabytes", New York: Van Nostrand Reinhold, 1994

[17] I. H. Witten, T. C. Bell, H. Emberson, S. Inglis, and A.Moffat. "Textual Image Compression: Two-Stage Lossy/Lossless Encoding of Textual Images", Processing of IEEE, 82(6), pp.878-888

[18] ISO/IEC JTCI Committee Draft 10918-1, "Digital Compression and Coding of Continuous, Part 1, Requirements and Guidelines", 1991

[19] International Telecommunication Union, "Terminal Equipment and Protocol Telematic Services T.82", 1993

[20] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Trans. on Info. Theory, Vol. 23, No. 3, pp.337-343, May 1997

[21] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding", IEEE Trans. on Info. Theory, Vol. 24, No. 5, pp.530-536, Sep. 1978

[22] Leger. A, Mitchell. M, and Yamazaki. Y, "Still picture compression algorithm evaluated for international standardization", Proc. of the IEEE Global Telecommunication Conference, IEEE Communication Society, pp.1028-1032, 1998

[23] N. J. Muller, "Computerized Document Imaging Systems", Boston: Artech House, Inc., 1993

[24] N. Jayant, J. Johnson, and R. Safranet, "Perceptual coding of images", Proc. of

SPIE, Vol. 1913, pp.168-178, 1993

[25] R. A. V. Kam, and P. W. Wong, "Customized JPEG compression for grayscale printing", Proc. IEEE Data Compression Conference, Snowbird, Utah. Los Alamitos, Calif: IEEE Society Press, pp.156-165, 1994

[26] R. B. Arps, and T. K. Truong. "Comparison of International Standards for Lossless Still Image Compression", Proc. of IEEE., 82(6), pp.889-899, 1994

[27] Rafael C. Gonzalez and Richard E. Woods, "Digital Image Processing", Addison Wesley, 1992

[28] S. A. Karunasekera and N. G. Kingbury, "A Distortion Measure for Blocking Artifacts in Images Based on Human Visual Sensitivity", Proc. of SPIE, Visual Communications and Image Processing 93, Vol. 2094, pp.474-486, 1993

[29] S. A. Karunasekera and N. G. Kingsbury, "A Distortion Measure for Image Artifacts Based on Human Visual Sensitiry", ICASSP-94, Vol. V, pp.117-120, April 1994

[30] T. A. Welch, "A Technique for High-Performance Data Compression", IEEE Computer, Vol. 17, No. 6, pp.8-19, June 1984

[31] T. Pavlidis, and J. Zhou, "Page Segmentation and Classification", CVGIP-54, pp.484-496, 1992

[32] Tim Shetler, "Birth of BLOB", BYTE, Feb. 1990

[33] 변혜원, 원희선, 박찬용, 한옥영, 황규영, "JPEG 영상 압축 시스템의 순서적, 계층적, 점진적 모드 및 신호 파라메터의 구현", 학술발표논문집, 제 19 권 2 호, 한국정보과학회, 1992

[34] 연구개발정보센터, "멀티미디어 정보 저장 및 검색을 위한 하부 구조 연구", 1994

[35] 연구개발정보센터, "정보 검색을 위한 효율적인 저장 시스템 개발 (I)", 1995

[36] 연구개발정보센터, "정보 검색을 위한 효율적인 저장 시스템에 관한 연구", 1993

[37] 최윤수, "멀티미디어 정보 검색을 위한 효율적인 저장 구조 구현 및 성능 평가", 충남대학교 석사학위 논문, 1995. 2

[38] 허봉식 김민환, "블럭 DCT 기반의 시각적응적 이미지 압축에 관한 연구", 정보과학회논문지, 22(10), pp.1405-1415, 1995