韓國컴퓨터情報學會 論文誌
第14卷 第6號, 2009. 6.

2009-14-6-1-1

대용량 플래시 메모리를 위한 임베디드 텍스트 인덱스 시스템

윤 상 훈*, 조 행 래**

An Embedded Text Index System for Mass Flash Memory

Sanghun Yun *, Haengrae Cho **

요 약

플래시 메모리는 비휘발성이고 저전력으로 동작하며 가볍고 내구성이 강하다. 이러한 특성으로 휴대용 멀티미디어 재생기(PMP)와 같은 모바일 컴퓨팅 환경에서의 저장 장치로 많이 사용되고 있다. 대용량의 플래시 메모리를 저장 장치로 가진 모바일 기기들은 비디오/오디오/사진등과 같은 다양한 종류의 멀티미디어 데이터를 저장하고 재생한다. 모바일 컴퓨팅 장치를 위한 기존의 인덱스 시스템은 노래 가사와 같은 텍스트 형태의 정보 검색에 비효율적이다. 본 논문에서는 대용량 플래시 메모리 기반 임베디드 텍스트 인덱스(Embedded Text Index: EMTEX) 시스템을 제안한다. EMTEX는 먼저 임베디드 시스템을 고려한 압축 알고리즘을 사용하며, 텍스트 인덱스가 구성된 필드에 삽입 및 삭제시 인덱스에 즉시 반영된다. 뿐만 아니라, 플래시 메모리의 특성을 고려한 효율적인 삽입, 삭제, 재구성 기능을 수행하며, DBMS의 상위 계층에서 독립적으로 동작한다는 장점을 갖는다. 제안한 시스템의 성능 평가를 위해 다양한 환경에서 실험을 수행하였다. 그 결과 EMTEX는 임베디드 환경에서 Oracle Text나 FT3와 같은 기존의 인덱스 시스템보다 더 좋은 성능을 보여주었다.

Abstract

Flash memory has the advantages of nonvolatile, low power consumption, light weight and high endurance. This enables the flash memory to be utilized as a storage of mobile computing device such as PMP(Portable Multimedia Player). Potable device with a mass flash memory can store various multimedia data such as video, audio, or image. Typical index systems for mobile computer are inefficient to search a form of text like lyric or title. In this paper, we propose a new text index system, named EMTEX(Embedded Text Index). EMTEX has the following salient features. First, it uses a compression algorithm for embedded system. Second, if a new insert or delete operation is executed on the base table, EMTEX updates the text index immediately. Third, EMTEX considers the

• 제1저자 :윤상훈 교신저자 : 조행래
• 투고일 : 2009. 03. 19, 심사일 : 2009. 05. 25, 게재확정일 : 2009. 06. 07.
* 영남대학교 컴퓨터공학과 박사과정 ** 영남대학교 전자정보공학부 교수
※ 본 논문은 한국과학재단의 국제협력연구 지원사업(F011-2007-000-10062-0)의 연구결과로 수행되었습니다.

characteristics of flash memory to design insert, delete, and rebuild operations on the text index. Finally, EMTEX is executed as an upper layer of DBMS. Therefore, it is independent of the underlying DBMS. We evaluate the performance of EMTEX. The Experiment results show that EMTEX can outperform th conventional index systems such as Oracle Text and FT3.

▸ Keyword : 텍스트 인덱스(Text Index), 플래시 메모리(Flash Memory), 전문 검색(Full Text Search), 데이터베이스 관리 시스템(DBMS)

Ⅰ. 서 론

플래시 메모리는 소비 전력이 적고, 전원의 공급이 없어도 저장된 정보가 사라지지 않는 비휘발성의 특성을 가진다. 또한 하드 디스크와 같은 기존의 디스크에 비해 데이터 접근 성능이 우수하며 물리적인 충격에 대한 내구성이 강하다. 플래시 메모리는 최근 노트북, 휴대용 멀티미디어 재생기(PMP), MP3 플레이어, 휴대 전화기, 디지털 카메라 등의 휴대용 멀티미디어 정보 기기들의 저장 장치로 광범위하게 사용되고 있다[1,2]. 플래시 메모리의 사용 분야가 많아짐과 동시에 저장 용량도 급속도로 대용량화 되어가고 있다. 최근 기가바이트급의 플래시 메모리가 개발되었고, 이러한 플래시 메모리의 대용량화는 휴대용 멀티미디어 정보 기기의 저장 장치 또한 대용량으로 변화할 것으로 예상된다.

디지털 음성/영상 데이터의 압축 기술과 전송 기술의 발전으로 오디오, 비디오, 사진등과 같은 멀티미디어 컨텐츠의 활용이 늘어나고 있다. 이러한 이유로 많은 컨텐츠들이 정리되어 있지 않는 상태에서 사용자가 요구하는 컨텐츠를 효과적으로 검색 및 저장하기 위한 표준이 필요하게 되었다. 멀티미디어 컨텐츠를 효과적으로 묘사하기 위한 연구된 표준의 대표적인 예로 MPEG-7이 있으며, 이는 텍스트로 이루어진 XML을 이용하여 컨텐츠의 메타정보를 기술한다[3,4,5].

대용량의 플래시 메모리를 사용하여 많은 컨텐츠를 저장하고 있는 휴대용 멀티미디어 장치에서 사용자가 요구하는 컨텐츠를 찾아내는 일은 쉽지 않다. 검색을 위해 사용하던 기존 인덱스들은 컨텐츠의 가사, 제목, 자막, 설명 등과 같이 텍스트 형태의 메타 데이터에서 요구하는 컨텐츠의 검색이 불편하다. 또한 텍스트 인덱스의 사용이 가능한 Oracle Text나 FT3같은 텍스트 인덱스는 엔터프라이즈급의 대용량 데이터에 적합한 알고리즘을 사용하므로 플래시 메모리를 사용하는 임베디드 시스템에서는 부적합하다[6,7].

본 논문에서는 플래시 메모리에서 효율적으로 동작하는 임베디드 텍스트 인덱스(Embedded Text Index: EMTEX) 시스템을 개발한다. EMTEX의 특징은 다음과 같다.

- EMTEX는 임베디드 시스템을 고려하여 처리 성능이 낮은 컴퓨팅 환경에서도 효율적으로 동작하는 압축 알고리즘을 사용한다.
- EMTEX는 인덱스가 구성된 텍스트의 삽입과 삭제시 추가 연산 없이 인덱스에 즉시 반영된다.
- EMTEX는 플래시 메모리의 특성을 고려한 효율적인 데이터의 삽입 및 삭제 그리고 인덱스의 재구성 연산을 수행한다.
- EMTEX는 DBMS의 상위 계층에서 동작하므로, 특정 DBMS에 종속되지 않는다.

본 논문의 구성은 다음과 같다. 2절에서는 배경지식을 소개하고, 3절에서는 EMTEX의 관련 연구 및 동향들을 소개한다. 4절에서는 본 논문에서 제안하는 텍스트 인덱스의 전체 구조와 데이터 구조, 제공되는 API를 살펴보고, 5절에서는 실험 환경과 성능 평가를 분석한다. 마지막으로 6절에서 결론을 맺는다.

Ⅱ. 관련 연구

텍스트 데이터베이스는 빈번히 엑세스되는 전자 문서들의 모음이다. 이러한 텍스트 데이터베이스의 예로 웹과 전자 도서관이 있다. 텍스트 데이터베이스가 다른 데이터베이스와 다른 점은 데이터의 접근 방법이다. 일반적인 데이터베이스는 데이터 검색을 위해 키워드로 사용된 값과 데이터베이스에 저장된 값의 비교 연산이 일반적인 것에 비해, 텍스트 데이터베이스는 검색 키워드를 포함하는 문서를 찾기 위해 데이터베이스에 저장된 모든 텍스트를 순차적으로 검색하는 것이 일반적이다. 이 방법은 연산이 복잡하고, 연산에 소비되는 시간이 매우 길다. 이를 빠르고 편리하게 수행하기 위하여 단어를 인덱싱하는 텍스트 인덱스(text index)가 제안되었다[8].

텍스트 인덱스는 역-리스트(inverted list)를 이용하여 구성하는 것이 일반적이다[8,9,10,11,12,13]. 역-리스트는 인덱싱되는 단어인 색인어(indexer)와 그 색인어를 포함하고 있는 문서 식별자의 리스트 쌍으로 구성된다. 역 리스트를 이

용하여 수행하는 검색은 역-리스트의 색인어와 검색하는 키워드를 비교하여 만족하는 문서 식별자의 리스트를 반환하여 수행한다. 이러한 역-리스트를 이용하는 인덱스를 역-인덱스(inverted index)라 한다.

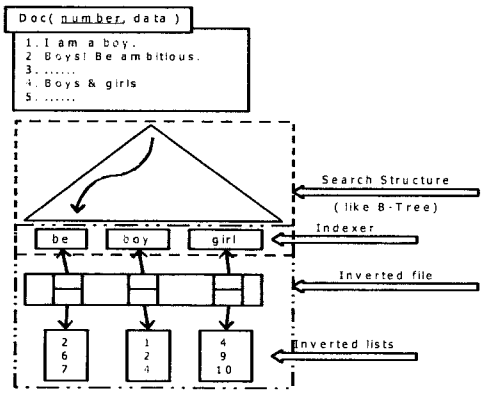

그림 1. 역-인덱스(inverted index)의 구성
Fig 1. Structure of Inverted Index

〈그림 1〉과 같이 역-인덱스는 크게 검색 구조(search structure)와 역-파일(inverted file)로 구성된다. 검색 구조는 역-리스트에 구성된 색인어를 빨리 찾기 위해 사용한다. 역-파일은 역-리스트와 색인어의 쌍으로 구성된다. 색인어는 색인된 단어를 말하며, 역-리스트는 색인어가 포함된 문서 식별자의 나열이다. 역-리스트는 저장 공간의 효율적 사용 및 검색 속도 향상을 위해 압축하여 저장하며, DBMS 혹은 파일 시스템을 이용하여 저장할 수 있다.

역-리스트를 저장하는 방법으로는 Unary, Gamma, Delta, Golomb, Variable byte 코딩등 다양한 압축 방법을 사용한다[8,12]. 일반적으로 많이 쓰며 높은 압축 효율을 보이는 것은 Golomb 코딩 방식이다. Golomb 코딩 방식은 특정 숫자를 기준으로 수학적 연산을 수행하여 압축/해독하는 방법이다. 이는 다른 방법에 비교하여 압축 용량 면에서 높은 효율을 가지지만 복잡한 수학적 연산을 사용하므로 낮은 데이터 처리 능력을 가진 임베디드 환경에선 압축/해독에 많은 시간이 소비되어 적절하지 않다. Variable byte 코딩 방식은 간단한 비트 연산으로 데이터의 압축을 수행한다. 압축된 각 바이트의 첫 비트는 압축된 숫자의 마지막 바이트인지 나타내고, 7비트는 데이터의 값을 이진법으로 표현한다. 이 방법은 단순한 비트 연산만으로 압축/해독이 가능하기 때문에 임베디드와 같은 낮은 처리 능력을 가진 컴퓨팅 환경에서 적절하다. 그러므로 본 논문에서는 임베디드 시스템을 고려하여 역-리스트의 압축 방법으로 variable byte 코딩 알고리즘을 이용한다.

Oracle Text는 Oracle 제품에 포함된 텍스트 인덱스이다[7,14,15]. Oracle Text는 역-리스트를 이용하여 텍스트 인덱스를 구성하고 있으며, 텍스트 인덱스를 위한 정보를 DBMS의 테이블로 구현하여 관리한다. 인덱스를 구성하는 테이블은 색인어와 문서 식별자 리스트, 이외 검색에 필요한 기타 정보들을 기록하는 $I-테이블, 삭제된 문서 식별자를 기록하기 위한 $N-테이블, 검색 결과로 반환된 문서 식별자를 이용하여 전체 텍스트를 읽어올 때 사용하는 $K-테이블과 $R-테이블 그리고 의미 없는 단어들의 색인을 예방하는 STOPLIST 테이블이 있다. 검색 구조는 DBMS가 제공하는 인덱스 기능을 이용한다. 사용자는 확장된 SQL문을 이용하여 텍스트 인덱스를 사용한다. Oracle Text는 대용량 데이터베이스를 목표로 만든 텍스트 인덱스로 많은 자원을 이용하여 대용량의 데이터를 색인하는 것에 최적화된 알고리즘을 사용한다. 그러므로 자원이 부족하고 성능이 낮은 임베디드 시스템에서는 사용할 수 없다. 또한 문서의 삽입 후 동기화를 수행하기 전에는 인덱스에 삽입된 문서의 정보가 반영되지 않는 단점이 있다.

FT3는 BSD 라이센스를 따르는 오픈 프로젝트에서 구현한 검색 엔진으로 Sqlite3 데이터베이스에 저장된 텍스트 데이터를 인덱싱한다[6]. 이 검색 엔진 또한 역-리스트 구조를 사용하여 인덱스를 구성한다. 인덱스 정보는 정보의 신뢰성을 위하여 DBMS를 이용하여 테이블형태로 저장하고 있으며, 인덱스 구성 및 검색 동작은 DBMS 상위 계층에서 API로 제공한다. 그러므로 특정 DBMS에 종속되지 않아 이식성이 뛰어나다. FT3는 기반 DBMS로 Sqlite3[16]를 사용하므로 임베디드 환경에서 사용가능하다. 그러나 FT3는 텍스트 데이터의 검색 시간이 느리고, 텍스트 데이터의 삭제 및 수정시 인덱스에 변화된 정보를 반영하는 기능이 없다.

본 논문에서 제안하는 텍스트 인덱스는 기존의 인덱스들과 같이 역-리스트 구조를 이용하여 인덱스를 구성하였고, 인덱싱 정보의 신뢰성을 위하여 인덱싱 정보를 DBMS에 테이블 형태로 저장하였다. 그러나 이전 인덱스들과는 다르게 역-리스트의 압축을 임베디드 시스템을 고려하여 압축/해제에 필요한 연산이 적은 방법을 사용하였고, 데이터베이스에 텍스트 정보의 추가, 삭제, 수정과 동시에 텍스트 인덱스에 반영된다. 그리고 DBMS에 종속적이지 않은 알고리즘으로 좋은 이식성을 보장한다.

III. 임베디드 텍스트 인덱스(Embedded Text Index: EMTEX)

3.1 시스템 구조

EMTEX는 DBMS에 독립적인 동작을 위해 사용자 프로그램과 DBMS 사이의 별개 계층으로 존재한다. 〈그림 2〉는 EMTEX의 전체 구조이다.

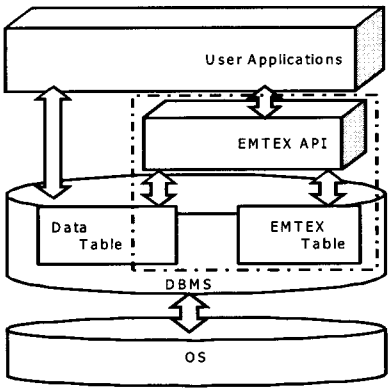

그림 2. EMTEX의 전체 구조
Fig 2. Architecture of EMTEX

EMTEX는 텍스트 인덱스에서 일반적으로 사용하는 역-리스트 자료구조를 이용하며, 데이터의 일관성 및 원자성을 보장하기 위해 인덱싱 정보를 DBMS의 테이블 형태로 구성하여 저장한다. 〈그림 3〉은 EMTEX를 구성하는 테이블의 모습이다.

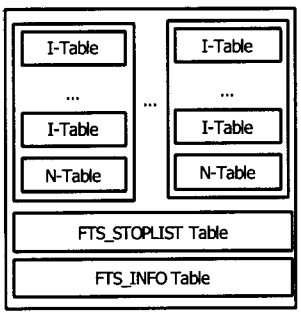

그림 3. EMTEX 구성 테이블
Fig 3. Table Schema of EMTEX

EMTEX를 구성하는 테이블은 총 네 가지가 있다. 각 테이블의 스키마는 〈부록 1〉에 나타난다.

- INFO 테이블 : 텍스트 인덱스가 구성된 테이블명과 필드명을 이용하여 인덱스 이름을 확인할 때 사용하는 테이블이다. 인덱스가 생성된 모든 (테이블, 필드) 쌍은 인덱스의 이름과 함께 이 테이블에 등록되어야 한다. 시스템에 하나만 생성된다.
- STOPLIST 테이블 : 이 테이블은 색인어로 의미가 없는 단어들을 저장하는 테이블이다. 테이블에 저장된 단어들은 인덱스의 구성 및 검색시 참조되어 연산에서 제외된다. 필요시 사용자가 추가적으로 등록할 수 있다. 시스템에 하나만 생성된다.
- I-테이블 : 이 테이블은 검색하는 단어, 즉 색인어(indexer)와 그 색인어의 역-리스트, 단어를 포함하는 문서의 수 등을 기록한다. 이후 인덱스를 이용한 검색에 사용된다. 이외에도 색인어를 포함하는 문서의 수, 색인어가 나타난 마지막 문서 식별자, 역-리스트의 마지막 오프셋 등의 추가 정보를 기록한다. 이는 각 인덱스마다 하나씩 생성된다.
- N-테이블 : 이 테이블은 테이블에서 삭제된 데이터의 식별자를 기록한다. 검색 수행시 이 테이블을 참조하여 저장된 문서 식별자들은 검색 결과에서 제거한다. 인덱스가 생성된 각 테이블마다 하나씩 생성된다.

EMTEX의 I-테이블에 사용되는 역-리스트는 데이터의 저장 공간과 검색 속도의 향상을 위하여 압축을 사용한다. 본 논문에서 사용한 역-리스트 압축 방식은 VARINT(Variable Integer)이다. VARINT는 Variable byte 코딩 방법을 임베디드 시스템에 적합하게 개선한 방식이다. Variable byte 코딩은 압축된 각 바이트의 첫 비트는 압축된 숫자의 마지막 바이트인지 나타내고(1 : 연속 바이트, 0 : 마지막 바이트), 나머지 7비트는 데이터의 값을 이진법으로 표현한다. 예를 들어 {5, 225, 300}을 압축하면 {00000110, 10000001 01100001, 10000010 00101100}로 표현된다. VARINT는 일반적인 숫자에 대한 압축 효율은 다소 떨어지나 압축이 빠르고 저장 공간을 적게 사용하는 장점이 있다. 이 방법은 역-리스트에 저장되는 식별자들이 연속적으로 증가하는 특징을 이용하여 저장되는 숫자들의 차이 값을 압축하여 압축을 더욱 빠르게 수행하고 저장 공간을 줄였다. 즉, {5, 225, 300}를 VARINT로 압축하면 먼저 두 수의 차를 계산하고({5, 220, 75}), 이를 압축하면 {00000110, 00000001 01011100, 01001011}이 된다.

3.2 EMTEX API

사용자는 API를 이용하여 EMTEX를 사용한다. 제공하는 API는 생성, 삽입, 삭제, 검색, 재구성이 있다. 각 API의 알고리즘 및 특성은 다음과 같다.

생성 API : EMTEX를 사용하기 위한 준비 과정으로 텍스트 인덱스 사용에 필요한 테이블들을 생성하는 과정을 수행한다. 생성 API의 알고리즘은 〈그림 4〉와 같다.

```
생성 알고리즘
입력: TN := Table Name, FN := Field Name
A. If ( TN is not exist or FN is not exist )
   1) 오류 반환
B. If ( INFO-table is not exist and STOPLIST-table is not
exist )
   1) INFO-table과 STOPLIST-table 생성
C. If ( IN(= Index Name) is not exist in INFO-table )
   1) INFO-table에 (IN, TN, FN) 삽입
   2) I-table 생성
   3) If ( N-table is not exist )
      a. N-table 생성
```

그림 4. 생성 알고리즘
Fig 4. Create Algorithm

• 삽입 API : 데이터 테이블에서 인덱스가 생성된 열을 추출하고, 추출된 텍스트를 일련의 과정을 거쳐 인덱스에 추가하는 과정을 수행한다. 삽입 API의 알고리즘은 〈그림 5〉와 같다.

삽입 알고리즘은 I-테이블에 인덱싱 정보를 삽입시 메모리에 삽입 항목을 먼저 구성하여 색인어의 중복된 삽입을 제거하였다. 또한 단어의 분리 작업에서 임베디드를 고려하여 간단한 분리 방법을 사용하여 빠른 반응 속도를 보여준다.

```
삽입 알고리즘
입력: TN := Table Name,    FN := Field Name,
   Sseq := Start Sequence, Eseq := End Sequence
A. SEQ := Sseq
B. STOPLIST-table로부터 STOPLIST 획득
C. TN, FN을 이용하여 INFO-table에서 IN(=Index Name) 획득
D. While ( SEQ <= Eseq )
   1) TN, FN에서 주키가 SEQ와 동일한 레코드의 TEXT 획득
   2) TEXT를 형태소 분석하여 TOKEN 하나 획득
   3) While ( TOKEN is exist )
      a. If ( TOKEN ∉ STOPLIST )
         - I-table에서 TOKEN을 이용하여 RECORD 획득
         - RECORD의 LIST에 SEQ를 압축하여 추가하고, 정보 수정
         - 데이터베이스의 RECORD 업데이트
      b. 다음 TOKEN 획득
   4) SEQ := SEQ + 1
```

그림 5. 삽입 알고리즘
Fig 5. Insert Algorithm

• 삭제 API : 데이터 테이블에서 삭제된 문서 식별자를 EMTEX에 알려주기 위한 기능이다. 삭제된 문서 식별자는 이후 검색 결과에서 제거된다. 삭제 API의 알고리즘은 〈그림 6〉과 같다.

삭제 알고리즘은 플래시 메모리의 블록 삭제 횟수가 제한적이라는 특성을 고려하여 삭제 과정 중 I-테이블의 역-리스트에서 문서 식별자를 제거하지 않고 N-테이블에 삭제된 문서 식별자만을 등록함으로써 플래시 메모리의 수명을 늘이고, 삭제시 필요한 연산을 줄여 반응 속도를 높였다. 지연된 삭제 연산은 인덱스의 재구성시 반영된다.

```
삭제 알고리즘
입력: TN := Table Name, PID := Primary Key
A. If ( TN is not Exist )
   1) 오류 반환
B. PK를 이용하여 TN에서 레코드 삭제
C. N-table에 PK 삽입
```

그림 6. 삭제 알고리즘
Fig 6. Delete Algorithm

• 검색 API : 인덱싱된 정보를 이용하여 사용자가 요구하는 키워드를 포함하는 문서를 찾아내는 기능을 수행한다. 검색 API의 알고리즘은 〈그림 7〉과 같다.

검색 알고리즘은 AND/OR/SUB 연산을 수행 할 수 있으며, 중위 표현법의 복잡한 연산식 구성할 수 있다. 연산식은 내부에서 후위 표현법으로 변환/처리한다. 그리고 연산식의 검색 결과는 N-테이블에 등록된 삭제 문서 식별자를 이용하여 보정한다.

```
검색 알고리즘
입력: TN := Table Name,  FN := Field Name,
   OP := Option,     WORD := Keyword
A. TN, FN을 이용하여 INFO-table에서 IN(=Index Name)을 획득
B. WORD를 infix 표현에서 postfix 표현으로 변경
C. IDL(=ID List) := null
D. WORD에서 첫 번째 TOKEN 획득
E. If ( TOKEN is exist )
   1) TOKEN을 이용하여 I-table에서 ID-List 획득하여 IDL에 저장
   2) 다음 TOKEN 획득
F. else
   1) null 반환
G. while ( TOKEN is exist )
   1) if ( TOKEN is operation )
      a. IDL과 TEMP 사이 연산자에 맞는 연산 수행하여 IDL에 저장
   2) else
      a. TOKEN을 이용하여 I-table에서 ID-List 획득하여 TEMP에 저장
   3) 다음 TOKEN 획득
H. N-table에서 D-IDL(=Deleted ID List) 획득
I. IDL에서 D-IDL에 포함된 ID 제거
J. IL 반환
```

그림 7. 검색 알고리즘
Fig 7. Search Algorithm

• 재구성 API : 인덱스의 반복적인 삽입과 삭제에 의해 효율성이 떨어졌을 때, 이를 최적화하는 기능을 수행한다. 재구성 API의 알고리즘은 〈그림 8〉과 같다.

재구성 알고리즘은 각 토큰의 역-리스트에 기록된 문서 식별자에 삭제된 문서 식별자가 있으면 이를 제거하고 재기록하는 과정을 반복한다. 이 과정에서 역-리스트 값의 변화가 없는 인덱스 정보는 재기록하지 않는다. 이는 플래시 메모리의 중첩쓰기가 지원되지 않는다는 특성을 고려한 방법이다. 플래시 메모리의 재기록은 삭제 후 쓰기 연산을 수행하는 것으로 이루어진다. 인덱스 정보의 재기록을 억제하는 것은 삭제 연산을 줄이는 것이고, 이는 플래시 메모리의 수명을 늘인다. 또한 줄어든 쓰기 연산으로 인덱스를 빠르게 재구성 할 수 있다.

```
재구성 알고리즘
입력: TN := Table Name
A. If ( TN is not Exist )
   1) 오류 반환
B. TN을 이용하여 INFO-table에서 IN(=Index Name) 하나 획득
C. While ( IN is exist )
   1) N-table에서 D-IDL(=Deleted ID List) 획득
   2) I-table에서 하나의 RECORD 획득
   3) While( RECORD is Exist )
      a. RECORD의 ID-List에서 D-IDL에 포함된 ID를 제거하고 정보 수정
      b. If ( RECORD.COUNT == 0 )
         - 데이터베이스의 RECORD 제거
      c. else If ( RECORD is changed )
         - 데이터베이스의 RECORD 수정
      d. 다음 RECORD 획득
   4) 다음 IN 획득
D. N-table의 모든 레코드 삭제
```

그림 8. 재구성 알고리즘
Fig 8. Rebuild Algorithm

IV. 실험 및 결과 분석

4.1 실험 환경

제안한 EMTEX의 성능을 평가하기 위하여 실험을 수행한 환경은 다음과 같다. 타블렛 혹은 UMPC 등과 같은 모바일 컴퓨터의 성능과 유사하며, Oracle 10g XE의 동작이 가능한 2.4GHz CPU와 1GB RAM을 사용하였다. 플래시 메모리를 저장 장치로 사용하는 환경을 구축하기 위해 MTRON 32GB SSD를 사용하였다. Linux Kernel 2.6 버전의 운영체제를 사용하여 성능평가를 수행하였다. EMTEX는 MobileLite[17]를 기반 데이터베이스로 사용하는 C 라이브러리 형태로 작성되었고, 이를 이용하여 C 언어로 벤치마킹 프로그램을 개발하였다. EMTEX의 비교대상으로는 Oracle Text와 FT3를 사용하였다. Oracle Text는 Oracle 10g XE 버전에 포함된 텍스트 인덱스로 Oracle 10g를 기반 데이터베이스로 동작하며 대용량 데이터베이스를 위하여 작성된 텍스트 인덱스이다. FT3는 Sqlite3를 기반 데이터베이스로 동작하고, 임베디드 시스템을 위해 작성된 텍스트 인덱스이다. 실험에 사용된 문서 데이터는 The 20 newsgroups data set[18]에서 임의로 1000개의 문서를 선별하였으며, Oracle에서 사용 가능한 데이터 타입인 VARCHAR의 최대 길이가 2KB이므로 삽입할 텍스트의 길이를 2KB 미만으로 제한하였다. 데이터베이스에 삽입된 데이터는 문서 식별자와 문서의 내용인 텍스트만 존재한다. 실험을 위한 환경 변수들을 요약하면 〈표 1〉과 같다.

표 1. 실험 환경 변수
Table 1. System Environment

항목	매개변수 값
CPU	2.4Ghz
RAM	1GB
저장장치	MTRON 32GB SSD
운영 체제	Linux Kernel 2.6
샘플 데이터	20 NewsGroups 문서
삽입 문서 수	1000개

실험에서 사용된 주요 성능 지표는 인덱스의 구성 시간과 인덱스를 사용한 검색의 향상된 성능, 삭제 동작의 효율성, 인덱스 재구성의 효율성, 그리고 인덱스 구성에 소비되는 저장 공간의 효율성이다.

4.2 결과 분석

EMTEX의 초기 구성 성능을 알아보기 위하여 데이터를 삽입하고, 텍스트 인덱스 구성 시간을 측정하였다. 인덱스 구성에는 데이터베이스에 문서를 삽입하는 시간과 삽입된 문서를 텍스트 인덱스에 반영하는 시간이 포함된다. 문서를 100개부터 1000개까지 증가하면서 삽입하였다.

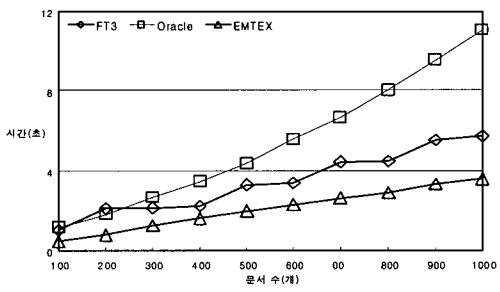

그림 9. 텍스트 인덱스의 초기 구성 시간
Fig 9. Build Time of Text Index

〈그림 9〉는 문서 수에 따른 텍스트 인덱스의 초기 구성 시간의 변화 추이를 보여주는 그래프이다. 문서 수가 증가함에 따라 인덱스 구성 시간이 증가하고 있으나, EMTEX의 구성 시간이 가장 낮고, Oracle Text가 가장 오래 걸렸다. Oracle Text는 대용량 데이터베이스를 위해 작성된 텍스트 인덱스이므로 저사양의 컴퓨터에서는 다른 두 시스템보다 구성 시간이 많이 소비되었다. FT3는 임베디드 환경을 고려하여 작성된 텍스트 인덱스이나 역-인덱스의 구조가 복잡하고, 문장에서 단어를 분리하는 과정에 소비되는 시간이 길어 본 논문에서 제안하는 EMTEX보다 낮은 성능을 보였다.

다음으로 인덱스를 사용하지 않을 때와 EMTEX를 사용할 때의 검색 성능을 비교하였다. 1000개의 문서가 이미 데이터베이스에 삽입되어 EMTEX가 생성되었다고 가정한다. 실험을 위해 "love", "like"를 포함하는 문서를 각각 검색하였고, 이 두 단어를 모두 포함한(AND 연산) 문서와 이 두 단어를 어느 하나라도 포함한(OR 연산) 문서들을 검색하였다. 실험을 위해 EMTEX는 검색 API를 이용하여 문서 식별자를 획득하였고, MobileLite는 SQL문의 LIKE 연산자를 이용하여 텍스트에서 검색 키워드가 포함된 문서의 식별자를 획득하였다.

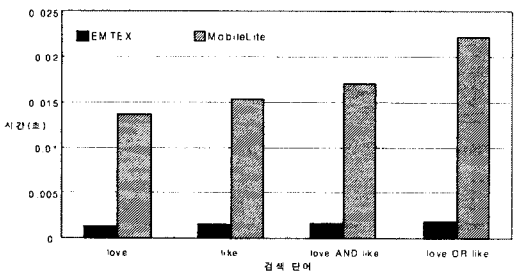

그림 10. 텍스트 인덱스의 사용 유무에 따른 검색 시간
Fig 10. Compare Search Time with using EMTEX

〈그림 10〉은 EMTEX를 사용했을 때와 그렇지 않았을 때의 검색 성능을 보여주는 그래프이다. 검색 결과로 획득된 문서 식별자는 인덱스를 사용한 경우와 그렇지 않은 경우 모두 동일함을 확인하였다. 실험 결과는 EMTEX를 사용하는 검색의 성능이 그렇지 않은 경우에 비해 평균 10배의 검색 성능을 보여주었다. "love"와 "like" 단어를 포함하는 문서 검색 시간이 다른 이유는 각 단어를 포함하는 문서의 수가 다르기 때문이다. AND와 OR 연산을 살펴보면 EMTEX는 두 연산에서도 효율적으로 동작함을 알 수 있다. EMTEX는 검색 결과의 수가 많을수록 보다 좋은 성능을 보여주었다.

다음 실험은 EMTEX와 기존 텍스트 인덱스의 검색 성능을 비교하는 실험이다. 검색을 위해 EMTEX와 FT3는 API를 사용하였고, Oracle Text는 확장된 SQL을 이용하였다.

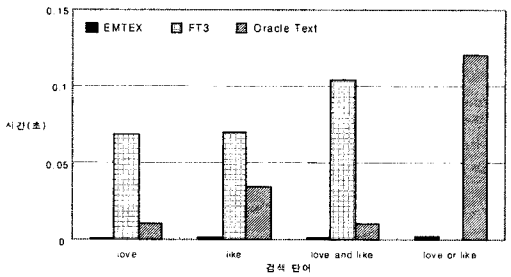

그림 11. 텍스트 인덱스의 검색 시간
Fig 11. Search Time of Text Index

〈그림 11〉은 단일 키워드 검색 및 검색 연산자를 사용한 검색시 각 텍스트 인덱스의 검색 성능을 보여주는 그래프이다. EMTEX는 "love"나 "loke"와 같은 단일 키워드 검색에서 FT3와 비교하여 약 50배의 성능을 보였고, Oracle Text에 비교하여 10~20배의 성능을 보였다. EMTEX나 FT3는 검색 결과의 수에 따른 검색 속도의 차이가 크지 않았으나 Oracle Text는 검색 결과의 수에 변화하여 검색 속도의 차이의 편차가 심했다. AND 연산에서는 EMTEX와 FT3는 단일 검색과 비교하여 약 25% 늘어난 검색 속도를 보였고, Oracle Text는 오히려 검색 속도가 줄었다. 이는 검색 엔진 내부에서 AND 연산을 수행하는 것과 외부에서 수행하는 특성에 따른 것으로 보인다. 즉, Oracle Text의 AND 검색은 내부에서 효율적으로 수행되는 것으로 보인다. OR 연산의 경우 EMTEX는 단일 검색보다 약 30%, AND보다 약 10%의 검색 시간이 늘어났지만, Oracle Text는 검색 시간이 급격히 증가되었다. FT3는 OR 연산이 구현되어 있지 않아 성능 평가를 하지 못하였다.

다음으로 기존 텍스트 인덱스와 EMTEX의 삭제 연산의 성능을 비교하였다. 이 실험은 문서의 삭제 후 인덱스에 반영되는 시간을 측정하였다. EMTEX는 삭제 API를 사용하였고, Oracle Text는 문서의 삭제 SQL을 수행 후, 인덱스 동기화 SQL을 수행하여 인덱스에 이를 반영하였다. FT3는 삭제된 문서를 인덱스에 반영하는 기능이 없었다.

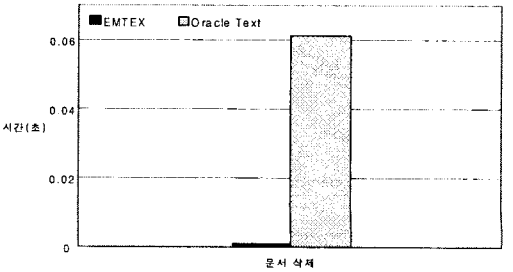

그림 12. 문서 삭제 후 인덱스 반영 시간
Fig 12. Delete Time of Text Index

〈그림 12〉는 각 텍스트 인덱스의 문서 삭제 후 인덱스 반영 시간을 보여주는 그래프이다. EMTEX는 문서의 삭제를 문서 식별자를 N-테이블에 등록하는 것으로 모든 삭제 과정이 완료되므로 수행 시간이 매우 짧다. 그리고 검색에서는 삭제된 문서가 등록된 N-테이블을 참조하므로 검색 결과에는 삭제된 문서를 포함하지 않음을 확인하였다. Oracle Text는 삭제된 문서를 $N-테이블에 등록하여 동작을 수행하지만 동기화 연산을 수행하기 전에는 이를 적용하지 않는다. 즉, 동기화 연산을 수행하여야 인덱스에 문서 삭제가 반영된다. 두 번의 연산을 수행하므로 수행 시간이 길어지게 된다. 뿐만 아니라, 하나의 SQL문을 수행하는 시간이 EMTEX보다 길어 더욱 오랜 시간이 걸린다.

다음으로 인덱스의 최적화를 위한 인덱스 재구성 시간을 측정하였다. 실험을 위하여 일정량의 문서를 삭제 후 재구성 연산을 수행하였다. FT3는 재구성 기능이 없으며, Oracle Text의 재구성은 ALTER 구문으로 수행하나 내부적으로는 인덱스를 DROP 후 새로이 인덱스를 생성하는 방법으로 수행한다. 그러므로 문서의 삭제 수와는 상관없이 동일한 수행 시간을 보인다. Oracle Text의 경우 1000개의 문서에 대하여 인덱스를 재구성하였을 때 약 2.2초의 재구성 시간을 보여주었다. 추가적으로 EMTEX에서 인덱스를 DROP 후 재생성하는 시간을 측정하여 EMTEX의 재구성 기능의 효율을 살펴보았다.

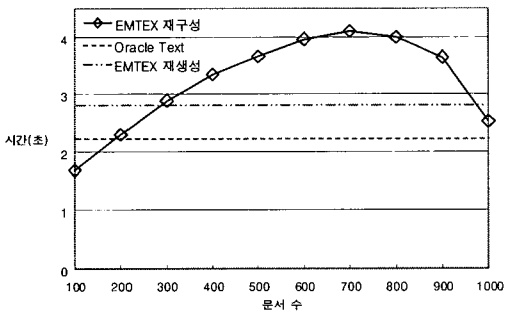

그림 13. EMTEX의 인덱스 재구성 시간
(Oracle Text 재구성 시간: 약 2.2초(1000개 문서 기준)
EMTEX 재생성 시간: 약 2.8초(1000개 문서 기준))
Fig 13. Rebuild Time of EMTEX
(Oracle Text Rebuild Time: about 2.2sec(1000 doc.)
EMTEX Recreate Time: about 2.8sec(1000 doc.))

〈그림 13〉은 EMTEX의 인덱스 재구성 시간을 나타낸 그래프이다. 위의 그래프에서 알 수 있듯이 EMTEX의 인덱스 재구성 시간은 문서 삭제 수가 늘어남에 따라 증가하였다. 이는 삭제된 문서 수가 많아지면 재구성을 수행할 때 각 색인어의 문서 리스트와 비교할 대상이 늘어나기 때문이다. 그러나 700개 이상의 문서가 삭제된 후 재구성을 수행할 때에는 재구성 시간이 감소하였는데, 이는 색인어의 문서 리스트와 비교할 대상은 늘어나지만 연산 결과 다시 디스크에 기록될 색인어의 수가 줄기 때문이다.

인덱스 재구성의 효율을 알아보기 위해 인덱스를 제거하고 다시 인덱스를 구성한 시간과 비교하였다. EMTEX에서 약 300개미만의 문서를 삭제 후 재구성을 수행하면 인덱스를 제거하고 다시 생성하는 시간보다 짧음을 보여준다. 즉, 인덱스 재구성의 수행 시점을 문서의 30%가 삭제되기 전에 수행하면 효과적이라는 것을 알 수 있다. 그리고 약 200개미만의 문서를 삭제 후 인덱스의 재구성을 수행하면 Oracle Text보다도 효과적으로 재구성할 수 있음을 〈그림 13〉에서 확인 할 수 있다.

마지막으로 텍스트 인덱스 구성에 소비되는 저장 공간을 측정하였다. 이 실험은 각 데이터베이스마다 텍스트 문서를 테이블로 구성하였을 때 소비된 저장 공간과 저장된 테이블에 텍스트 인덱스를 구성하였을 때의 소비된 저장 공간을 측정하였다. 두 측정값을 기반으로 인덱스만을 위한 저장 공간을 계산하고, 이를 이용하여 소비된 저장 공간에서 인덱스의 저장 공간 비율을 계산하였다. 〈그림 14〉는 각 데이터베이스가 텍스트 문서를 테이블로 구성하였을 때 소비된 저장 공간을 보여주고, 〈그림 15〉는 저장된 테이블의 텍스트 영역에 인덱스를 생성하였을 때 소비된 저장 공간을 보여준다. 〈그림 16〉은 〈그림 14〉와 〈그림 15〉의 값을 이용하여 데이터베이스에 테이블과 인덱스 구성에 소비된 저장 공간에 대한 텍스트 인덱스 구성에 소비된 저장 공간의 비율을 계산한 것이다.

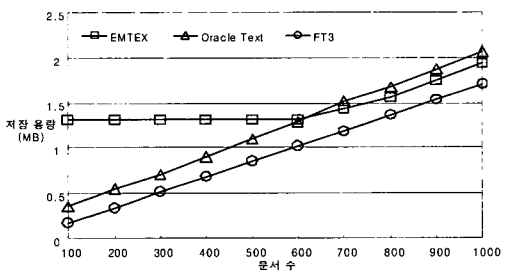

그림 14. 문서 저장에 사용된 저장 공간
Fig 14. Storage Usage for Document

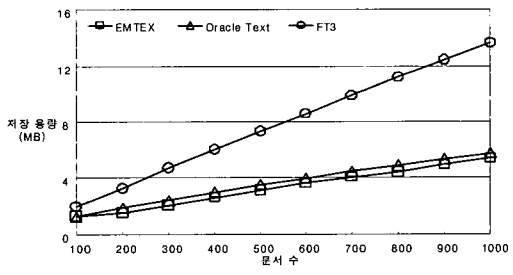

그림 15. 테이블 및 텍스트 인덱스 구성에 사용된 저장 공간
Fig 15. Storage Usage for Table and Text Index

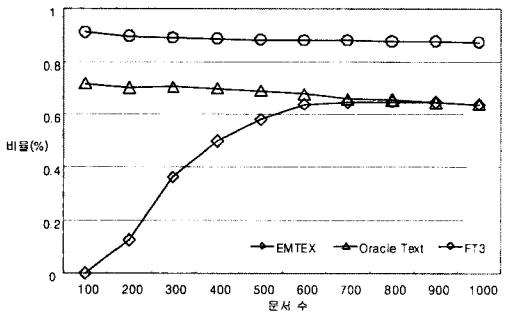

그림 16. 텍스트 인덱스의 추가적인 저장 공간 오버헤드
Fig 16. Additional Storage Overhead of Text Index

〈그림 14〉에서 데이터베이스에 문서를 저장할 때 소비되는 저장 공간을 살펴보면, Oracle Text와 FT3는 문서 수에 따라 저장 공간이 선형적으로 증가하는 것을 알 수 있다. EMTEX는 100~600개의 문서까지의 저장 공간의 변화가 없는데, 이는 EMTEX의 특징으로 초기 데이터베이스를 생성할 때, 최소의 저장 공간으로 약 1.3MB의 저장 공간을 소비한다. 내부적으로 데이터의 저장 공간은 더 작으나 이를 측정할 방법이 없어 파일 시스템에 요구되는 크기를 기록한 것이다. 하지만, 600개 이상의 문서에서 알 수 있듯이 문서의 증가에 따라 저장 공간이 선형적으로 증가함을 알 수 있다. 각 데이터베이스는 내부적으로 데이터의 저장 방법이 달라 저장 공간의 소비량이 다르나 많은 차이를 보이지 않았다.

〈그림 15〉에서 각 데이터베이스에 저장된 테이블에 텍스트 인덱스를 구성한 결과를 살펴보면, FT3는 다른 두 방법에 비교하여 상대적으로 많은 저장 공간을 소비하여 텍스트 인덱스를 구성하였으며, Oracle Text와 EMTEX는 비슷한 저장 공간을 소비하며 텍스트 인덱스를 구성하였다. 테이블의 저장 공간과 인덱스의 구성 공간의 합이 1.3MB보다 작은 구간(약 100개 이하의 문서)에서는 EMTEX가 Oracle Text보다 많은 저장 공간을 소비하나 그 이상에서는 EMTEX가 Oracle Text보다 작은 저장 공간을 소비하였다.

〈그림 16〉에서 테이블 및 텍스트 인덱스의 구성에 소비된 저장 공간에 대한 인덱스 저장 공간의 비율을 살펴보면, FT3는 약 90%의 저장 공간이 인덱스를 위하여 사용되며, 이는 테이블이 소비하는 공간의 의 약 9배이다. Oracle Text는 60~70%의 저장 공간을 인덱스를 위하여 사용하며, 이는 테이블이 소비하는 공간의 약 2배이다. EMTEX는 100~600개 문서까지는 테이블의 저장 공간 측정 오류로 인해 인덱스 구성 비율도 오류를 보인다. 테이블 저장 공간 측정의 오류가 없는 600개 이상의 문서에서는 약 60%의 저장 공간을 텍스트 인덱스가 소비하고 있으며, 이는 Oracle Text와 거의 동일한 결과이다. 즉, EMTEX는 텍스트 인덱스 구성의 저장 공간 효율성에서 FT3보다 약 3-4배의 효율성을 보였고, Oracle Text와는 동등한 수준의 저장 공간 효율성을 보였다.

V. 결 론

본 논문에서는 플래시 메모리를 저장 장치로 사용하는 임베디드 시스템에 적합한 텍스트 인덱스 시스템인 EMTEX를 개발하였다. EMTEX는 임베디드 시스템과 같이 처리 능력이 낮고 메모리가 적은 시스템에서도 효율적으로 동작하는 역-리스트 압축 방법을 사용하고, 플래시 메모리에 적합한 텍스트 인덱스의 테이블 구조를 사용한다. 또한 플래시 메모리의 특성을 고려한 삽입, 삭제, 재구성 알고리즘을 사용하여 효율적으로 동작하며 문서의 삽입과 삭제시 별도의 연산 없이 즉시 인덱스에 반영된다. 뿐만 아니라, 제안하는 시스템은 DBMS에 독립적으로 설계되어 좋은 이식성을 가진다. EMTEX의 성능 평가를 위하여 기존 텍스트 인덱스와 비교 실험하였다. 그 결과 EMTEX를 사용하여 검색하면 그렇지 않은 경우와 비교하여 평균 10배 이상의 성능을 보였다. 그리고 Oracle Text나 FT3와 같은 기존 텍스트 인덱스와 비교하여 인덱스의 구성 시간, 인덱스를 이용한 검색 시간, 문서 삭제 후 인덱스 반영 시간, 인덱스의 재구성에서 우수한 성능을 나타내었다.

참고문헌

[1] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," In First Symposium on Operating Systems Design and Implementation, pages 25 - 37, Monterey, California, November 1994

[2] New Samsung Notebook Replaces Hard Drive With Flash. *http://www.extremetech.com*

[3] SF Chang, T Sikora, A Purl, "Overview of the MPEG-7 Standard," IEEE Transactions Circuits and Systems for Video Technology, Vol. 11, pp. 988-695, Jun 2001

[4] MPEG-7 Overview, *http://www.chiariglione.org/mpeg /standards/mpeg-7/mpeg-7.htm*

[5] Nack, F., Lindsay, A.T., "Everything you wanted to know about MPEG-7. 1," Multimedia, IEEE , vol.6, no.3, pp.65-77, Jul-Sep 1999

[6] FT3: a full text indexer and search engine, *http://ft3.sourceforge.net*

[7] Oracle Press, "Oracle® Text Reference 10g Release 2 (10.2)," June, 2005
[8] E Bertino, KL Tan, BC Ooi, R Sacks-Davis, J Zobel, "Indexing Techniques for Advanced Database Systems," Kluwer Academic Publishers Norwell, MA, USA, 1997
[9] V. N. Anh, and A. Moffat, "Inverted index Compression using word-aligned binary codes" Information Retrieval, Vol 8, pages 151-166, 2005.
[10] P. Boldi and S.Vigna, "Compressed Perfect Embedded Skiplists for Quick Inverted-Index Lookups," String Processing and Information Retrieval 2005, page25-28, Oct.2005.
[11] A. Moffat and J. Zobel, "Self-Indexing Inverted Files for Fast Text Retrieval," ACM Transactions on Information Systems, Vol. 14, Issue 4, pages 349-379, Oct. 1996.
[12] A. Trotman and V. Subramanya, "Sigma Encoded Inverted Files," Proceedings of the 16th ACM Conference on Information and Knowledge Management, pages 983-986, Nov. 2007.
[13] J. Zobel, A. Moffat, and R. Sacks-Davis, "An Efficient Indexing Technique for Full-Text Database Systems, Proceedings of the 18th VLDB Conferencce Vancouver, British Columbia, Canada 1992.
[14] Oracle Technology Network, "How interMedia Processes Text DML," March, 2000
[15] Oralcle, "The World's Largest Enterprise Software Company," *http://www.oracle.com*
[16] SQLite Home Page, *http://www.sqlite.org*
[17] MobileLite - Inervit Product, http://inervit.com/main /product/02_01.jsp
[18] Home Page for 20 Newsgroups Data Set, *http://people.csail.mit.edu/jrennie/20Newsgroups/*

부 록

1. EMTEX 테이블 스키마

INFO 테이블 스키마

```
CREATE TABLE FTS_INFO (
  IDX_NAME varchar(32) primary key, — 인덱스 명
  TABLE_NAME varchar(32), — 인덱스가 가르키는 테이블명
  FIELD_NAME varchar(32)  — 인덱스가 가르키는 필드명
);
```

STOPLIST 테이블 스키마

```
CREATE TABLE FTS_STOPLIST (
  word varchar(32) primary key — 색인어에서 제외되는 단어
);
```

I-테이블 스키마

```
CREATE TABLE FTS_$tablename_$fieldname_I (
TOKEN_TEXT varchar(64) primary key,      — 색인어
TOKEN_LAST int,              — 색인어가 나타난 컨텐츠의 마지막번호
TOKEN_LAST_OFFSET int, — TOKEN_INFO의 마지막 오프셋
TOKEN_COUNT int, — 색인어가 나타난 문서의 개수
TOKEN_INFO varbyte(8000) — 바이너리로 압축된 역 리스트
);
```

N-테이블 스키마

```
CREATE TABLE FTS_$tablename_N (
  DOCID int primary key — 삭제된 컨텐츠의 번호
);
```

저 자 소 개

윤 상 훈(Sanghun Yun)

2006년 : 영남대학교 컴퓨터공학과(공학사)
2008년 : 영남대학교 컴퓨터공학과 (공학석사)
2008년~현재 : 영남대학교 컴퓨터공학과(박사과정)
관심분야 : 무선 센서 네트워크, 분산 데이타베이스

조 행 래(Haengrae Cho)

1988년 : 서울대학교 컴퓨터공학과 (공학사)
1990년 : 한국과학기술원 전산학과 (공학석사)
1995년 : 한국과학기술원전산학과 (공학박사)
1995년~현재 : 영남대학교 전자정보 공학부 교수
관심분야: 분산/병렬 데이타베이스, 트랜잭션 처리, DBMS 개발