KIPS Tr. Software and Data Eng.
Vol.5, No.4 pp.171~180 pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2016.5.4.171

An Update-Efficient, Disk-Based Inverted Index Structure for Keyword Search on Data Streams

Eun Ju Park† · Ki Yong Lee††

ABSTRACT

As social networking services such as twitter become increasingly popular, data streams are widely prevalent these days. In order to search data accumulated from data streams efficiently, the use of an index structure is essential. In this paper, we propose an update-efficient, disk-based inverted index structure for efficient keyword search on data streams. When new data arrive at the data stream, the index needs to be updated to incorporate the new data. The traditional inverted index is very inefficient to update in terms of disk I/O, because all index data stored in the disk need to be read and written to the disk each time the index is updated. To solve this problem, we divide the whole inverted index into a sequence of inverted indices with exponentially increasing size. When new data arrives, it is first inserted into the smallest index and, later, the small indices are merged with the larger indices, which leads to a small amortize update cost for each new data. Furthermore, when indices stored in the disk are merged with each other, we minimize the disk I/O cost incurred for the merge operation, resulting in an even smaller update cost. Through various experiments, we compare the update efficiency of the proposed index structure with the previous one, and show the performance advantage of the proposed structure in terms of the update cost.

Keywords : Inverted Index, Data Streams, Index Update, Keyword Search

데이터 스트림에 대한 키워드 검색을 위한, 효율적인 갱신이 가능한 디스크 기반 역색인 구조

박 은 주† · 이 기 용††

요 약

트위터와 같은 소셜 네트워킹 서비스(social networking service)의 확산으로 스트림 형태의 데이터가 크게 증가하고 있다. 스트림 형태로 들어와 누적되는 데이터를 효율적으로 검색하기 위해서는 색인이 반드시 필요하다. 본 논문에서는 스트림 형태로 들어와 계속 누적되는 데이터에 대한 키워드 검색을 효율적으로 할 수 있게 해주는, 효율적인 갱신이 가능한 디스크 기반 역색인(inverted index) 구조를 제안한다. 데이터 스트림을 검색하기 위해서는 데이터의 유입에 따라 역색인을 계속해서 갱신해 주어야 한다. 전통적인 역색인을 사용하는 경우, 역색인을 갱신하기 위해서는 매번 디스크에 저장된 모든 색인 데이터를 읽고 다시 써야 하므로 디스크 I/O 측면에서 매우 비효율적이다. 이러한 문제를 해결하기 위해 본 논문에서는 역색인을 크기가 지수적으로 증가하는 여러 역색인들로 나누어 저장한다. 새로운 데이터가 들어오면 우선 가장 작은 크기의 역색인에 삽입하고, 작은 크기의 역색인들을 더 큰 크기를 가진 역색인들과 나중에 병합함으로써 평균적으로 역색인을 갱신하는 비용을 크게 낮춘다. 또한 디스크에 저장된 역색인들을 병합할 때 발생하는 디스크 I/O 비용을 최소화함으로써 역색인의 갱신 비용을 더욱 낮춘다. 다양한 실험을 통해 기존 방법과 제안 방법의 효율성을 비교하고, 제안 방법이 갱신 비용에 있어 기존 방법에 비해 훨씬 효율적임을 보인다.

키워드 : 역색인, 데이터 스트림, 색인 갱신, 키워드 검색

1. 서 론

디지털 경제의 확산으로 많은 정보와 데이터가 생산되고 있다. 특히 소셜 네트워킹 서비스(social networking service)의 확산으로 스트림 형태의 데이터들이 크게 증가하고 있다. 데이터 스트림이란 시간의 흐름에 따라 끊임없이 연속적으로 생산되는 데이터를 말한다[1]. 데이터 스트림의 예로서 트위터의 경우 하루에 수백만 개의 트윗(tweet)이 다수 사용자에 의해 끊임없이 생성되고 있으며, 센서 네트워크의 경우 온도, 기압 등과 같은 물리현상에 대한 측정 데이터가 하루에 수 GB 이상 연속적으로 수집된다[2].

※ 이 논문은 2015년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No.NRF-2015R1C1A1A02037071).
† 준 회 원 : CJ올리브네트웍스 사원
†† 정 회 원 : 숙명여자대학교 컴퓨터과학부 부교수
Manuscript Received : March 7, 2016
Accepted : March 15, 2016
* Corresponding Author : Ki Yong Lee(kiyonglee@sookmyung.ac.kr)

이렇게 데이터 스트림 형태로 들어온 데이터는 다양한 검색 시스템의 검색 대상이 될 수 있다. 예를 들어 대표적인 소셜 네트워킹 서비스인 페이스북이나 인스타그램은 해시태그 또는 검색 기능을 사용하여 원하는 키워드가 포함된 글을 검색할 수 있다.

본 논문에서는 트윗과 같은 단문(short message)들이 데이터 스트림의 형태로 계속해서 유입되는 환경에서, 사용자가 지정한 단어들이 모두 포함된 k개의 가장 최근 단문을 찾는 문제를 다룬다($k \geq 1$). 데이터 스트림 형태로 계속해서 유입되는 데이터를 검색하기 위해서는, 지금까지 유입된 데이터를 빠르게 검색하기 위한 도구가 마련되어야 한다. 그 역할을 하는 것이 바로 색인(index)이다. 본 논문에서는 주어진 키워드를 포함하고 있는 단문을 검색하기 위해 역색인(inverted index)[3]을 사용하며, 역색인은 디스크에 저장되어 있다고 가정한다. 역색인은 주어진 키워드를 포함하고 있는 문서 검색에 가장 많이 사용되는 색인 구조이다.

데이터 스트림 환경에서는 데이터가 끊임없이 유입되므로 끊임없이 유입되는 데이터를 반영하기 위해 색인 정보가 매우 효율적으로 갱신될 수 있어야 한다. 기존의 전통적인 역색인은 데이터 스트림 환경과 같이 잦은 갱신이 발생되는 경우 매우 비효율적이다. 새로운 데이터가 유입되었을 때 이를 반영하기 위해 디스크에 저장된 전통적인 역색인을 갱신한다고 하자. 이를 위해서는 디스크에 저장된 역색인 파일을 모두 읽어 새로운 데이터를 추가하고, 갱신된 내용을 다시 디스크로 써야 한다. 따라서 새로운 데이터를 반영할 때마다 역색인 전체를 모두 읽어 다시 쓰는 일을 반복해야 한다. 특히 데이터가 증가함에 따라 역색인의 크기도 증가하기 때문에 이 방법은 매우 비효율적이다.

이러한 문제를 해결하기 위해, 본 논문에서는 매우 효율적으로 갱신될 수 있는 디스크 기반의 역색인 구조를 제안한다. 보다 구체적으로, 제안 방법은 역색인을 갱신할 때 발생하는 디스크 I/O 비용을 줄이는 것을 목표로 한다. 이를 위해 제안하는 색인 구조는 역색인 정보를 단일 역색인이 아닌 다수의 역색인 I_1, I_2, ... , I_m ($m \geq 1$)에 나누어 저장한다. 각 역색인 I_i가 저장할 수 있는 역색인 정보의 양은 제한되어 있다. 어떤 역색인 I_i가 최대로 저장할 수 있는 단문 ID의 수를 T_i라 하자. 이 때, $T_{i+1} = 2T_i$ 이다. 즉, I_{i+1}의 최대 가능 크기는 I_i의 최대 가능 크기의 2배가 된다. 새로운 단문이 유입되면 해당 단문의 ID는 메모리에 유지되고 있는 역색인 I_0에 우선 추가된다. 단문이 계속 유입되어 I_0가 그의 최대 가능 크기를 넘어가면, I_0의 내용을 디스크에 있는 I_1에 병합한다. 이 때, $T_1 = 2\,T_0$이다. 병합이 완료되면 I_0는 비워지게 되고 새로운 데이터를 받을 수 있게 된다. 같은 방식으로 I_1가 그의 최대 가능 크기를 넘어가면 I_1의 내용은 I_2에 병합된 후 I_1은 비워진다. 마찬가지 방법으로 반복적으로 병합을 하게 된다.

따라서 제안 방법은 I_0가 꽉 찰 때마다 디스크에 저장된 기존의 역색인을 모두 읽을 필요없이, 대부분의 경우 작은 역색인들만 접근하면 된다. 그에 따라 기존 방법에 비해 갱신에 필요한 디스크 I/O가 크게 줄어든다. 물론 역색인들 간의 병합으로 인해 디스크 I/O가 일시적으로 증가할 수 있지만, 평균적으로는 작은 역색인들만 주로 갱신에 참여하게 되므로 평균 디스크 I/O는 크게 감소한다. 더욱이 역색인들 간의 병합이 연쇄적으로 발생하는 경우에도, 제안 구조는 각 역색인의 이름만을 변경함으로써(예: I_i를 I_{i+1}로 변경) 병합에 발생되는 디스크 I/O 비용을 최소화한다. 결론적으로 매 갱신 때마다 디스크에 저장된 모든 역색인 내용에 접근해야 하는 기존 방법에 비해 제안 방법은 대부분 작은 역색인들만을 접근하여 갱신이 가능하므로, 기존 구조보다 훨씬 빠르게 갱신이 가능하다. 특히 제안 방법은 역색인 간 병합이 일어날 때에도 디스크 I/O 비용이 최소화되는 방식으로 병합 연산을 수행함으로써 갱신 비용을 더욱 줄인다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구들을 소개한다. 3장에서는 본 논문에서 고려하는 질의를 정의하고, 기존 방법의 문제점을 기술한다. 4장에서는 제안 방법을 설명하고, 5장에서는 그의 분석 결과를 제시한다. 6장에서는 실험 결과를 통해 제안 방법이 기존 방법에 비해 효율적임을 보인다. 7장에서는 결론을 맺는다.

2. 관련 연구

본 장에서는 문서 검색에 널리 사용되는 색인인 역색인 구조를 설명하고, 데이터 스트림 형태로 유입되는 데이터의 색인에 관한 기존의 관련 연구를 소개한다.

2.1 역색인(Inverted Index)

역색인은 주어진 단어가 어떤 문서에 나타나는지에 대한 정보를 저장하는 데이터 구조이다. 역색인의 목적은 특정 단어가 포함된 문서를 빠르게 검색하는 것이다. 역색인은 문서 검색 시스템에 쓰이는 가장 대중적인 데이터 구조로써, 검색엔진과 같은 대규모 데이터 검색에 널리 사용된다[3, 6]. 역색인을 이용한 빠른 검색에 대해서는 이미 많은 연구들이 이루어졌다[7-10].

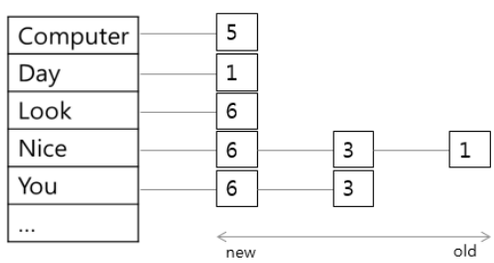

Fig. 1. Example of an inverted index

Fig. 1은 역색인의 구조를 나타내는 예이다. 역색인은 단어들을 저장하고 있는 하나의 해시 테이블과 여러 개의 포스팅 리스트(posting list)로 구성되어 있다. 각 포스팅 리스

트는 특정 단어가 나타나는 문서의 ID들을 나타내는 리스트이며, 해시 테이블은 특정 단어와 그의 포스팅 리스트를 연결해 주는 역할을 한다. 이 역색인 정보를 이용하여 특정 단어가 담긴 문서를 찾으려면, 먼저 해시 테이블을 통해 해당 단어에 대한 포스팅 리스트를 찾는다. 그리고 해당 포스팅 리스트를 읽어가며 해당 단어가 나타나는 문서의 ID들을 얻는다. 본 논문에서는 포스팅 리스트 내의 문서 ID들이 시간의 역순으로 저장되어 있다고 가정한다. 즉, 가장 최근에 유입된 문서의 ID가 가장 먼저 나오고, 가장 오래전에 유입된 문서의 ID가 가장 뒤에 나온다.

2.2 Tweet Index

Tweet Index(TI)는 트위터와 같은 소셜 네트워크 데이터 검색에 대한 연구로서, 메모리에 저장된 역색인의 갱신 비용을 줄이는 방법을 제안하였다[4]. T1은 자주 검색되는 키워드와 자주 검색되지 않는 키워드를 구분하여 자주 검색되는 키워드에 한해서만 역색인을 구축한다[11-13]. 즉, 키워드가 들어왔을 경우 자주 검색되는 키워드인 경우에만 역색인 정보에 추가한다. 따라서 전체 데이터 대신 일부 데이터에 대해서만 역색인을 구축하게 되므로, 역색인의 크기가 작아져서 갱신 효율이 증가되는 효과를 얻는다. 단, 키워드 검색을 할 때는 역색인에 포함된 키워드에 대해서만 검색을 할 수 있다. 따라서 T1은 두 가지 큰 단점이 존재한다.

첫째, 질의 로그 관련 연구[14]에 따르면 자주 발생하지 않는 질의는 전체 질의의 약 70%에 달한다. 따라서 자주 발생하지 않는 질의를 던진 70%에 해당하는 사용자는 만족하지 않은 결과를 얻게 될 확률이 매우 높다. 둘째, 관련 연구에 따르면 실시간으로 처리되는 데이터 스트림의 경우 자주 발생하는 키워드와 자주 발생하지 않는 키워드 간의 변동이 몇 분 사이에도 자주 발생한다[15]. 따라서 TI의 경우 이러한 데이터 스트림 환경을 제대로 반영하지 못해서 불완전한 결과를 낼 수 있다.

따라서 TI은 역색인을 사용하긴 하지만 일부 키워드에 대해서만 검색이 가능하며, 메모리가 아닌 디스크에 저장된 역색인의 갱신 방법에 대해서는 다루고 있지 않다.

2.3 EarlyBird

EarlyBird는 TI과 마찬가지로 소셜 네트워크 데이터의 검색을 위해, 메모리에 저장된 역색인의 갱신 비용을 줄이려는 연구 중 하나이다[2]. EarlyBird는 TI의 불완전함을 보완하여 모든 경우의 소셜 네트워크 문서를 검색할 수 있도록 하였다.

EarlyBird의 역색인 구조는 최근 문서 일수록 문서 ID가 포스팅 리스트의 앞쪽에 위치한다. 기존의 역색인에서는 최근에 들어온 문서의 ID가 포스팅 리스트 뒤쪽에 위치하였다. 그러나 포스팅 리스트의 배열 순서를 바꿔줌으로써 검색 성능을 향상시켰다. EarlyBird는 이러한 구조를 가진 하나의 역색인을 이용하여 데이터 스트림에 대한 검색을 한다.

하지만 본 논문에서 고려하는 디스크에 저장된 역색인을 갱신하는 경우, 3.2절에서 설명할 바와 같이 하나의 역색인만을 두는 것은 매우 비효율적이다. 즉, EarlyBird는 메모리에 저장된 역색인의 갱신 방법만을 논의하고 있으며, T1의 단점을 보완하였지만 디스크 기반의 역색인에 대한 갱신 방법에 대해서는 다루고 있지 않다.

2.4 The Log-Structured Merge Tree

The Log-Structured Merge Tree(LSM-Tree)[16]는 본 논문에서 고려하는 문제와 유사하게 디스크에 저장된 색인의 효율적인 갱신 방법에 대한 연구이다. 전통적인 디스크 기반 색인인 B-tree는 데이터 스트림과 같이 잦은 갱신이 일어나는 환경에서는 디스크 I/O가 매우 많이 발생한다는 단점이 있다. 따라서 LSM-Tree는 데이터가 계속 유입되는 환경에서 B-tree 형태의 색인을 적은 비용으로 유지하는 것을 목표로 한다.

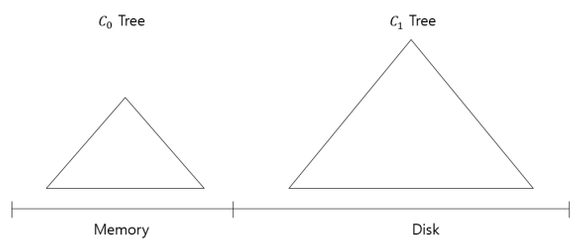

Fig. 2. Example of an LSM-tree with two trees

Fig. 2와 같이 LSM-Tree는 처음 메모리에 존재하는 C_0 트리와 디스크에 존재하는 C_1 트리가 있다. 새로운 데이터가 도착하면 이들은 우선 C_0 트리에 추가된다. C_0 트리는 크기 제한이 존재하며, 새로운 데이터가 삽입되어 C_0 트리가 할당된 최대 가능 크기에 도달하면 C_0 트리의 엔트리들을 삭제하여 디스크에 존재하는 C_1 트리에 병합한다. 마찬가지 방법으로 C_1 트리 또한 최대 가능 크기에 도달하면 새로운 C_2 트리를 생성하여 C_1 트리의 일부 항목들을 C_2 트리에 병합하게 된다. 같은 방식으로 병합이 진행되면 Fig. 3과 같은 형태가 된다. 여기에서도 C_{i+1} 트리의 최대 가능 크기는 C_i 트리의 최대 가능 크기의 2배이다.

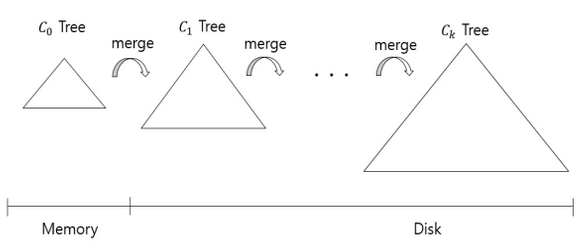

Fig. 3. LSM-Tree with (k + 1) trees

LSM-Tree는 새로 유입되는 데이터를 C_0에 모았다가 C_0가 꽉 차면 이를 일괄적으로 디스크에 반영한다. 또한 C_0가 꽉

찰 때마다 B-tree 전체를 접근하는 것이 아니라, 대부분의 경우 작은 트리들만 접근하거나 병합하여 갱신을 완료한다. 따라서 기존의 전통적인 B-tree에 비해서 갱신에 발생하는 평균 디스크 I/O의 양을 크게 줄일 수 있으므로 갱신 성능이 향상된다.

LSM-Tree는 제안 방법과 유사한 전략을 사용하지만 B-tree만을 다루고 있으며, 본 논문에서 고려하는 역색인에 대한 갱신 방법은 다루고 있지 않다.

2.5 Log-Structured Inverted Indices

Log-Structured Inverted Indices(LSII) 또한 소셜 네트워크 데이터에 대한 실시간 검색을 위한 역색인 갱신에 관한 연구이다[5]. LSII는 2.4절에서 설명한 LSM-Tree의 개념을 역색인에 적용한 연구이다. 하지만 LSII는 TI이나 EarlyBird와 마찬가지로 메모리에 저장된 역색인만을 고려한다.

LSII는 역색인의 내용을 다수의 역색인 I_1, I_2, ..., I_m ($m \geq 1$) 에 나누어 저장한다. 모든 역색인 I_1, I_2, ... , I_m은 메모리에 저장된다. I_{i+1}의 최대 가능 크기는 I_i의 최대 가능 크기의 2배이다. 데이터 스트림에 새로운 데이터가 도착하면 해당 데이터는 우선 I_0에 추가된다. I_0이 그의 최대 가능 크기를 넘어가면 I_0의 모든 엔트리는 I_1에 병합된다. 마찬가지로 I_1이 그의 최대 가능 크기를 넘어가면 I_1의 모든 엔트리는 I_2에 병합된다. 따라서 LSM-Tree와 마찬가지로 연쇄적인 병합이 발생할 수 있다.

본 논문에서는 LSII와 기본 전략은 유사하지만, 메모리가 아닌 디스크에 최적화된 역색인 구조를 제안한다. 제안 방법은 디스크에 최적화된 병합 연산 메커니즘을 제공한다. 제안 방법은 역색인들 간 병합에 발생하는 디스크 I/O 양을 최소화하는 한편, 디스크 I/O 성능을 크게 저하시키는 임의 쓰기(random writes)를 피하고 디스크 성능에 최적인 순차 쓰기(sequential writes)만을 발생시킨다. 다음 장에서는 제안 방법을 설명하기에 앞서 예비지식을 기술한다.

3. 예비 지식

본 장에서는 본 논문에서 다루는 문제를 정의하고, 기존 역색인 구조의 문제점을 기술한다.

3.1 문제 정의

본 논문에서는 10-20여개의 단어로 이루어진 짧은 문서, 즉 트윗(tweet)과 같은 단문(short message)들이 계속해서 유입되는 데이터 스트림 환경을 가정한다. 이러한 환경의 예로는 대표적으로 트위터(twitter), 페이스북(facebook), 텀블러(tumblr), 구글+(Google+)와 같은 마이크로블로그 사이트들이 있다. 본 논문에서는 이들 단문들에 대해 다음 형태의 검색 질의 q가 요청된다고 가정한다.

$$q = (W, k)$$

여기서 $W = \{w_1, w_2, \ldots, w_n\}$은 사용자가 지정한 단어들의 집합을 나타내며, k는 찾고자 하는 단문의 개수를 의미한다. 질의 $q = (W, k)$가 주어지면 시스템은 지금까지 유입된 단문들 중 W의 모든 단어가 포함된 가장 최근 k개의 단문을 찾아 반환한다. 예를 들어, q = {{*"computer", "database", "university"*}, 10}는 "computer", "database", "university"가 모두 들어간 최근 10개의 단문을 찾는 질의이다.

이러한 검색 질의의 처리를 위해 본 논문에서는 2.1절에서 설명한 역색인이 사용되며, 역색인은 디스크에 저장되어 있다고 가정한다. 이러한 역색인은 계속해서 유입되는 단문들을 반영하기 위해 주기적으로 갱신되어야 한다. 본 논문에서는 이러한 환경에서 적은 비용으로 갱신될 수 있는 역색인 구조를 제안한다. 본 논문에서 갱신 비용은 갱신 시 발생하는 디스크 I/O 양으로 나타낸다.

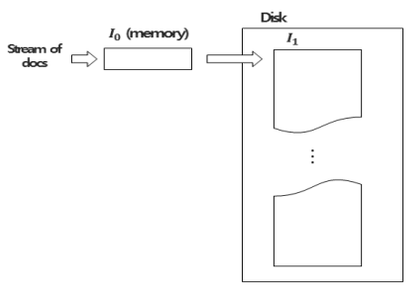

Fig. 4. The traditional inverted index used for keyword search on a data stream

3.2 기존 역색인 구조의 문제점

Fig. 4는 데이터 스트림 환경에서 기존의 전통적 역색인이 사용되는 모습을 나타내는 그림이다. 아직까지는 데이터 스트림 환경에서 잦은 갱신에 효율적인 디스크 기반의 역색인에 대해서는 많은 연구가 이루어지지 않았기 때문에, 본 논문에서는 Fig. 4와 같이 기존의 역색인을 사용하는 것을 기존 방법으로 가정한다. I_0는 메모리에 있는, 버퍼 역할을 하는 작은 크기의 역색인이며, I_1는 디스크에 저장된 전통적인 구조의 역색인이다. 매 단문이 들어올 때마다 디스크에 저장된 역색인을 갱신하는 것은 불가능한 일이므로, 새로 도착한 단문은 우선 메모리에 있는 역색인 I_0에 삽입된다. I_0는 메모리에 있고 크기도 작기 때문에 갱신 비용이 그리 크지 않다. I_0가 저장할 수 있는 엔트리, 즉 단문 ID의 최대 개수를 T_0라 하자. 단문이 계속 도착하여 I_0의 크기가 T_0를 넘어가면 I_0가 저장하고 있는 내용을 한꺼번에 디스크에 저장된 역색인인 I_0에 반영하여 I_1을 갱신한다. 따라서 I_1은 I_0의 크기가 T_0가 될 때마다 한 번씩 갱신된다.

I_0의 내용을 I_1에 반영하는 작업을 구체적으로 설명하면 다음과 같다. I_1의 각 포스팅 리스트에 대해, 동일한 단어에 대한 포스팅 리스트가 I_0에 있는지 확인한다. 만약 I_0에 동일한 단어에 대한 포스팅 리스트가 있으면 두 포스팅 리스트를 병합하여 병합된 포스팅 리스트를 디스크에 새로 쓴다.

만약 I_0에 동일한 단어에 대한 포스팅 리스트가 없으면 I_1의 포스팅 리스트를 그대로 디스크에 쓴다. I_1의 모든 포스팅 리스트에 대해 이와 같은 작업이 종료되면, 기존의 역색인을 제거하고 새로 생성된 역색인을 I_1로 만든다. 따라서 I_0에 저장된 단문 ID의 개수를 T_0라 하고, I_1에 저장된 단문 ID의 개수를 N이라 할 때, 이 방법은 디스크로부터 N개의 단문 ID를 읽고 디스크에 $(N+ T_0)$ 개의 단문 id를 쓰게 된다.

따라서 이 방법은 디스크에 저장된 I_1의 크기가 커지면 커질수록 I_1을 갱신하는 비용이 I_1의 크기에 비례하여 증가한다. n개의 단문이 들어왔을 때, 이 방법의 총 갱신 비용은 다음과 같다. 첫 T_0개의 단문이 도착하여 I_0가 꽉 차면, I_0에 저장된 T_0개의 단문 ID를 I_1에 쓰게 되며 이 때의 디스크 I/O 비용은 T_0에 비례한다. 다음 T_0개의 단문이 도착하여 I_0가 다시 꽉 차면, I_1에서 T_0개의 단문 ID를 읽고 I_0에 저장된 T_0개의 단문 ID들을 합해 총 $2T_0$개의 단문 ID를 디스크에 쓰게 된다. 따라서 이 때의 디스크 I/O 비용은 $T_0+2T_0=3T_0$에 비례한다. 다시 T_0개의 단문이 도착하여 I_0가 다시 꽉 차면, 디스크에서 $2T_0$개의 단문 ID를 읽고 I_0에 저장된 T_0개의 단문 ID들을 합해 총 $3T_0$개의 단문 ID를 디스크에 쓰게 된다. 따라서 이 때의 디스크 I/O 비용은 $2T_0+3T_0= 5T_0$에 비례한다. 따라서 만약 총 n개의 단문이 들어왔다면 총 갱신 비용은 $T_0 + 3T_0 + 5T_0 + \ldots + (2\lfloor n/T_0 \rfloor -1)T_0 = O(n^2)$ 이 된다. 따라서 단문이 누적됨에 따라 갱신 비용이 크게 증가하게 된다.

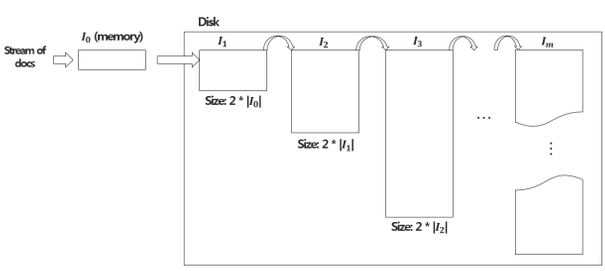

Fig. 5. The proposed disk-based inverted index structure

4. 제안 방법

본 장에서는 계속해서 유입되는 데이터를 주기적으로 반영하기 위해 적은 비용으로 갱신이 가능한 디스크 기반 역색인 구조를 제안한다.

4.1 제안 구조 및 갱신 알고리즘

제안 방법은 3.2절의 기존 방법과 마찬가지로 메모리에 버퍼 역할을 하는 작은 크기의 역색인 I_0를 둔다. 만약 단문들이 계속 도착하여 I_0가 꽉 차면, 기존 방법과 마찬가지로 I_0의 내용을 디스크에 저장된 역색인에 반영한다. 하지만 제안 방법은 기존 방법과 달리 디스크의 역색인 정보를 하나의 역색인에 모두 저장하는 것이 아니라, Fig. 5와 같이 다수의 역색인 I_1, I_2, ... , I_m $(m \geq 1)$에 나누어 저장한다. 각 I_i가 최대로 저장할 수 있는 단문 ID의 수, 즉 I_i의 최대 크기는 제한되어 있다. I_i의 최대 크기를 T_i라 할 때 $T_i = 2T_{i-1}$의 관계가 있다. 즉, I_i의 최대 크기는 I_{i-1}의 최대 크기의 두 배이다.

메모리에 있는 I_0가 꽉 찬 후 새로운 단문이 들어오면, I_0의 내용을 디스크에 있는 I_1에 병합하고 I_0를 비우게 된다. 만약 I_1이 병합으로 인해 최대 가능 크기를 넘어서면 I_1의 내용을 디스크에 있는 I_2에 병합하고 I_1을 비우게 된다. 일반적으로, 만약 I_i의 크기가 T_i를 넘어가게 되면, I_i의 내용을 I_{i+1}에 병합하고 I_i를 비운다. 만약 최대 크기를 가진 I_m의 크기가 T_m을 넘어가게 되면 새로운 역색인 I_{m+1}를 만들고 I_m의 내용을 I_{m+1}로 옮긴다. 이러한 구조는 다음과 같은 이유로 기존 방법에 비해 갱신에 효율적이다. 기존 방법은 I_0가 꽉 찰 때마다 디스크에 저장된 모든 역색인 정보를 접근하여 갱신 작업을 수행한다. 반면에 제안 방법은 I_0가 꽉 차면 디스크에 저장된 모든 역색인 정보에 접근하는 대신, 대부분의 경우 작은 크기의 역색인들만을 접근하여 갱신 작업을 완료한다. 물론 여러 역색인들 간의 병합이 연쇄적으로 발생할 수도 있지만, 평균적으로 훨씬 적은 적은 디스크 I/O 양만큼을 발생시킨다. 이것은 5.1절의 이론적 성능 분석에서 자세히 설명한다.

제안 방법은 앞서 설명한 구조를 디스크에 좀 더 최적화시키기 위해 다음과 같은 기법들을 추가로 사용한다. 첫 번째는 I_i의 내용을 I_{i+1}로 병합할 때, I_{i+1}이 비어있거나 존재하지 않는 경우 I_i의 내용을 I_{i+1}에 쓰는 대신 단지 I_i의 이름만을 I_{i+1}로 변경하는 것이다. 따라서 이 경우 I_i의 내용을 I_{i+1}에 옮겨 쓰는 디스크 비용이 사라진다. 예를 들어 I_3의 모든 내용이 I_4로 병합되어 I_3가 비어 있고, I_2가 꽉 찬 상황을 고려하자. I_2가 꽉 차서 그의 내용을 I_3로 병합해야 할 때, I_3는 현재 비어 있는 상황이므로 I_2의 내용을 I_3로 옮겨 쓰는 대신 I_2의 이름만을 I_3로 변경한다. 그러면 I_2의 내용을 I_3로 옮겨 쓰지 않고도 I_3를 얻을 수 있다. 따라서 병합에 필요한 디스크 I/O가 크게 줄어든다.

두 번째는 역색인들 간의 연쇄적인 병합이 발생할 때 이를 한 번의 병합연산만으로 처리하는 것이다. 예를 들어, I_i, I_{i+1}, ..., I_{i+j}이 꽉 차있고, I_{i-1}의 내용을 I_i로 병합해야 한다고 하자. 이 경우 I_i의 내용을 I_{i+1}로, I_{i+1}의 내용을 I_{i+2}로, ..., I_{i+j}의 내용을 I_{i+j+1}로 옮기는 연쇄적인 병합이 발생한다. 이 경우 제안 방법은 I_i의 내용을 I_{i+1}로, I_{i+1}의 내용을 I_{i+2}로, ..., I_{i+j}의 내용을 I_{i+j+1}로 연쇄적으로 병합하는 대신, I_{i+j}의 내용만을 I_{i+j+1}로 병합하고 I_i, I_{i+1}, ..., I_{i+j-1}의 이름을 각각 I_{i+1}, I_{i+2}, ..., I_{i+j}로 바꾼다. 이렇게 하면 추가적인 병합연산 없이 단 한 번의 병합연산만으로 모든 연쇄적인 병합을 완료할 수 있다.

세 번째는 I_i와 I_{i+1}을 병합할 때, 디스크에 최적화된 디스크 I/O 패턴만을 발생시키는 것이다. 이를 위해 제안 방법은 I_1, I_2, ..., I_m ($m \geq 1$)의 해시 테이블의 엔트리들을 동일한 기준에 따라 정렬된 상태로 유지한다(예: 알파벳순). I_i과 I_{i+1}의 해시 테이블 엔트리들이 동일한 기준으로 정렬되어 있다고 하자. 제안 방법은 병합 정렬(merge sort) 방식으로 I_i와 I_{i+1}를 병합한다. 즉, I_i와 I_{i+1}의 내용을 각각 디스크에서 한 블록씩 읽은 뒤, 양쪽의 해시 테이블 엔트리를 병합정렬 방식으로 비교하며 진행한다. 만약 엔트리가 나타내는 단어가 동일한 경우 해당 단어에 대한 포스팅 리스트를 병합하여 병합된 포스팅 리스트를 디스크에 새로 쓴다. 그 결과로 만들어진 병합된 역색인도 역시 해시 테이블 엔트리들이 동일한 기준으로 정렬된다. 이렇게 병합을 수행하면 I_i과 I_{i+1}을 각각 블록 단위로 한 번씩만 순차적으로 읽어 병합이 완료되는 한편, I_i과 I_{i+1}의 병합 결과도 임의쓰기가 전혀 없이 오직 순차쓰기만으로 생성이 완료된다. 이러한 디스크 I/O 패턴은 디스크 I/O 비용을 최소화하는 패턴이며, 따라서 병합을 위해 발생하는 디스크 I/O 비용이 최소화된다.

```
processDocument(document d) {
  if (I0 is full)
    merge(I0, I1);
  insert d to I0;
}

merge(Ii, Ii+1) {
  if (Ii+1 is full)
    merge(Ii+1, Ii+2);
  if (Ii+1 is empty or does not exist)
    rename Ii as Ii+1;
  else {
    merge Ii and Ii+1 to produce updated Ii+1;
    empty Ii;
  }
}
```

Fig. 6. Index update algorithm

Fig. 6은 지금까지 설명한 역색인 갱신 알고리즘을 나타내는 의사 코드이다. 새로운 단문 d가 도착하면 processDocument 함수가 호출된다. processDocument는 먼저 I_0의 공간을 보고, 만약 I_0의 공간이 꽉 찼다면 merge(I_0, I_1)을 호출하여 I_0과 I_1을 병합한 후 I_0을 비운 뒤 d를 I_0에 삽입한다. I_0과 I_1이 병합하는 과정에서 I_1 또한 공간이 꽉 찼다면 merge() 함수를 재귀적으로 호출하여 I_1과 I_2를 병합한다. I_i과 I_{i+1}을 병합하는 과정에서 만약 I_{i+1}이 비어있으면, I_i의 이름만을 I_{i+1}로 변경함으로써 불필요한 디스크 I/O를 제거한다.

4.2 검색 알고리즘

3.1절에서 정의한 바와 같이 질의 q = (W, k)는 W에 속한 단어가 모두 포함된 가장 최근 k개의 단문을 찾는 질의이다. 본 절에서는 제안한 역색인 구조를 사용하여 질의 q를 처리하는 알고리즘을 설명한다.

제안하는 역색인 구조를 사용하여 질의 q를 처리하기 위해서는 다음과 같은 알고리즘을 수행한다. 각 $w \in W$에 대해 I_1, I_2, ..., I_m 순으로 검색하여 w를 포함하고 있는 가장 최근 $c{\cdot}k$개의 단문 ID를 얻어낸다. c는 1보다 같거나 큰 양수로서 사용자 정의 상수이다. $C_{k,1}(w)$를 이렇게 얻어낸, w를 포함하고 있는 $c{\cdot}k$개 단문 ID의 집합이라 하자. 모든 $w \in W = \{w_1, w_2, \ldots, w_n\}$에 대해 $C_{k,1}(w)$를 각각 구한 뒤 $C_{k,1} = C_{k,1}(w_1) \bigcap C_{k,1}(w_2) \bigcap \ldots \bigcap C_{k,1}(w_n)$을 구한다. $C_{k,1}$는 $w_1, w_2, \ldots, w_n$을 모두 포함하는 단문 ID의 집합을 의미한다. 만약 $|C_{k,1}| < k$이면, 각 $w \in W$에 대해 w를 포함하고 있는 그 다음 최근 $c{\cdot}k$개 단문 ID의 집합 $C_{k,2}(W)$를 얻어낸다. 그리고 $C_{k,2} = C_{k,2}(w_1) \bigcap C_{k,2}(w_2) \bigcap \ldots \bigcap C_{k,2}(w_n)$을 구한다. 이러한 작업을 $|\bigcup_{i=1} C_{k,i}| \geq k$가 될 때까지 반복하여 최종적으로 k개의 단문을 얻는다. Fig. 7은 지금까지 설명한 검색 알고리즘을 나타내는 의사 코드이다.

```
// a query is given as q = (W, k),
// where W = {w1, w2, ..., wn} and
// k is the number of documents to be retrieved
processQuery(W, k) {
  c = 2; // a user-specified constant
  for each w in W do
    Ck(w) = the most recent c·k documents
            containing w;
  endFor
  Ck = Ck(w1) ∩ Ck(w2) ∩ ... ∩ Ck(wn);

  while do
    if |Ck| ≥ k then
      return the most recent k documents
      among those in Ck;
    for each w in W do
      Ck(w) = the next most recent c·k documents
              containing w;
    endFor
    Ck = Ck ∪ (Ck(w1) ∩ Ck(w2) ∩ ... ∩ Ck(wn));
  endWhile
}
```

Fig. 7. Keyword search algorithm

기존의 역색인을 사용하는 경우에는 검색 질의를 처리하기 위해 I_0와 I_1만 검색하면 된다. 반면에 제안 방법은 I_0, I_1, I_2, ..., I_m 순으로 여러 개의 역색인을 검색해야 하는 경우가 생긴다. 이것은 역색인 정보가 I_0, I_1, I_2, ..., I_m에 나누어 저장되기 때문이다. 따라서 제안하는 역색인 구조에서의 검색은 기존 역색인 구조에서의 검색에 비해 일반적으로 성능이 떨어지게 된다. 하지만 I_0, I_1, I_2, ..., I_m 순으로 최신 단문에 대한 역색인 정보를 가지고 있으므로, 대부분의 경우에는 I_0, I_1, I_2, ... 순으로 몇 개만 검색하고도 질의 q

에 대한 결과를 금방 얻을 수 있다. 5.2절에서는 제안하는 역색인 구조의 검색 성능이 기존 역색인 구조에의 검색 성능에 비해 다소 떨어지더라도 실제로 아주 큰 차이가 나지는 않는다는 것을 이론적으로 분석한다.

5. 성능 분석

본 장에서는 기존 방법과 제안 방법의 갱신 비용과 검색 비용을 이론적으로 비교한다.

5.1 갱신 비용 비교

본 논문에서는 디스크 상의 역색인을 갱신하는 비용은 역색인을 갱신할 때 디스크에서 읽고 쓰는 단문 ID의 개수에 비례한다고 가정한다. 메모리에 존재하는 역색인 I_0가 최대로 저장할 수 있는 단문 ID의 개수를 T_0라 하고 지금까지 들어온 단문의 총 개수를 N이라 하자. 분석의 용이를 위해 $N = nT_0$이라 하자. 즉, n은 I_0가 꽉 차서 I_0의 내용을 디스크에 저장된 역색인에 반영해야 하는 횟수가 된다.

1) 기존 방법의 갱신 비용

3.2절에서 설명한 바와 같이 기존 방법은 I_0가 꽉 찰 때마다 디스크에 저장된 역색인 I_1을 모두 읽고, 이를 I_0와 병합한 결과를 디스크에 모두 새로 쓰게 된다. 현재 I_1에 저장된 단문 ID의 개수를 T라 하고, I_0가 꽉 차 I_0의 내용을 I_1에 병합해야 한다고 하자. 이 경우 T개의 단문 ID를 디스크에서 읽고, I_0에 포함된 T_0개의 단문 ID를 더하여 총 $(T + T_0)$개의 단문 ID를 디스크에 써야 한다. 따라서 총 $(2T + T_0)$개의 단문 ID를 읽고 쓰게 된다. 따라서 지금까지 들어온 단문의 총 개수를 N이라 할 때, 역색인의 갱신을 위해 디스크에서 읽고 쓰는 총 단문 ID의 개수 M_{old}는 다음과 같다.

$$M_{old} = T_0 + 3T_0 + 5T_0 + \ldots + (2n-1)T_0$$
$$= T_0 \sum_{i=1}^{n} (2i - 1) = n^2 T_0$$

여기서 T_0는 상수이므로 $M_{old} = O(n^2)$이다.

2) 제안 방법의 갱신 비용

제안 방법은 디스크에 여러 개의 역색인을 가지고 있기 때문에 기존 방법과는 다른 양상을 보인다. I_0의 내용을 I_1에 병합해야 할 때, I_1의 상태에 따라 다음 세 가지 경우가 발생한다. 첫 번째는 I_1이 비어있을 때이며, 이 경우는 I_0의 내용을 그대로 I_1에 써주기만 하면 되므로 T_0의 디스크 I/O 비용이 발생한다. 두 번째는 I_1가 반 만 차있을 때이며, 이 경우는 I_1을 디스크로부터 읽은 후 I_0와 I_1을 합친 결과를 다시 디스크에 써야 하므로 $T_0 + 2T_0 = 3T_0$의 디스크 I/O 비용이 발생한다. 마지막 세 번째는 I_1이 꽉 차있을 때이며, 이 경우는 I_1과 I_2를 병합하는 비용에 I_0의 내용을 I_1에 쓰는 데 드는 T_0의 디스크 I/O 비용이 추가로 발생한다. 일반적으로 I_i의 내용을 I_{i+1}에 병합할 때도 위와 유사한 분석을 적용할 수 있다.

Table 1. Disk I/O in the proposed structure

n	read	write	total
1		T_0	T_0
2	T_0	$2T_0$	$3T_0$
3		T_0	T_0
4	T_0	$2T_0$	$3T_0$
5	$4T_0$	$5T_0$	$9T_0$
6	T_0	$2T_0$	$3T_0$
7		T_0	T_0
8	T_0	$2T_0$	$3T_0$
9	$4T_0$	$5T_0$	$9T_0$
10	T_0	$2T_0$	$3T_0$
11	$8T_0$	$9T_0$	$17T_0$
12	T_0	$2T_0$	$3T_0$
…	…	…	…

Table 1은 I_0가 꽉 차서 I_0의 내용을 디스크에 저장된 역색인에 반영해야 하는 횟수 n의 증가에 따라 디스크에서 읽고 써야 하는 단문 ID의 개수를 나타낸다. I_0와 비어있는 I_1 간의 병합은 2번마다 한 번씩 발생하며, 이의 디스크 I/O 비용은 T_0이다. I_0과 반만 차 있는 I_1 간의 병합은 2번마다 한 번씩 발생하며, 이의 디스크 I/O 비용은 $3T_0$이다. I_1과 반만 차 있는 I_2 간의 병합은 4번마다 한 번씩 발생하며, 이의 디스크 I/O 비용은 $8T_0$이다. I_2와 반만 차 있는 I_3 간의 병합은 8번마다 한 번씩 발생하며, 이의 디스크 I/O 비용은 $16T_0$이다. 일반적으로 디스크에 저장된 I_i와 반만 차 있는 I_{i+1} 간의 병합은 2^{i+1}번 마다 한 번씩 발생하며, 이의 디스크 I/O 비용은 $2^{i+2}T_0$이다. 따라서 지금까지 들어온 단문의 총 개수를 N이라 할 때, 역색인의 갱신을 위해 디스크에서 읽고 쓰는 총 단문 ID의 개수 M_{new}는 다음과 같다.

$$M_{new} = (n/2)\cdot T_0 + (n/2)\cdot 3T_0 + (n/4)\cdot 8T_0 + \ldots$$
$$+ (n/2^i)\cdot 2^{i+1} T_0 + \ldots + (n/2^{\log n})\cdot 2^{\log n+1} T_0$$
$$= T_0 \sum_{i=1}^{\log n} 2n = 2T_0 n \log n$$

여기서 T_0는 상수이므로 $M_{new} = O(n\log n)$이다. 기존 방법의 디스크 I/O 비용은 $O(n^2)$이므로, 제안 방법은 기존 방법에 비해 디스크 I/O를 크게 줄임을 알 수 있다. 6장에서는 실제 실험을 통해 제안 방법의 성능을 보인다.

5.2 검색 비용 비교

본 절에서는 기존 방법과 제안 방법에서 3.1절에서 정의한 질의인 $q = (W, k)$를 처리하는 비용을 비교한다. 본 논문에서는 검색 비용을 디스크 읽기 횟수로 나타낸다.

1) 기존 방법의 검색 비용

$W = \{w_1, w_2, \ldots, w_i\}$ 이라 하자. 어떤 단어 w를 포함하고 있는 모든 문서들의 ID이 디스크에서 최대 B개의 블록에 걸쳐 저장된다고 하자. 이 경우, 최악의 상황에서는 각 $w \in W$에 대해 해시 함수를 사용하여 w에 대한 해시 테이블 엔트리를 찾아가는데 1번, 추가로 B개의 블록을 접근하는 비용이 발생한다. 따라서 기존 방법에서 $q = (W, k)$를 처리하기 위해 디스크를 읽는 횟수는 $i \cdot (B + 1)$가 된다.

2) 제안 방법의 검색 비용

제안 방법에서는 어떤 단어 w를 포함하고 있는 문서들의 ID가 I_1, I_2, ..., I_m에 나누어 저장될 수 있다. 최악의 경우는 각 단어 $w \in W$가 모든 I_1, I_2, ..., I_m에 나누어 저장된 경우이다. 이 경우 각 $w \in W$에 대해 각 I_1, I_2, ..., I_m에서의 해시 테이블 엔트리를 찾아가는데 1번씩 총 m번, 각 I_1, I_2, ..., I_m에서 추가로 평균적으로 B/m개의 블록을 접근하는 비용이 발생한다. 따라서 제안 방법에서 $q = (W, k)$를 처리하기 위해 디스크를 읽는 횟수는 최대 $i \cdot m \cdot (1 + B/m) = i \cdot (B + m)$이 된다. 한편 제안 방법에서 역색인의 수 m은 총 단문 개수가 N일 때 $T_0 + T_1 + \ldots + T_m = T_0 + 2T_0 + \ldots + 2^m T_0 = N$이므로 $m = \log(N/T_0 + 1) - 1$이 된다.

따라서 제안 방법은 일반적으로 기존 방법에 비해 검색 비용이 더 크다. 하지만 최악의 상황이 아닌 일반적인 상황에서 m은 그리 큰 수가 아니므로, 검색 비용에서 얻는 손해는 갱신 비용에서 얻는 이득을 고려했을 때 그리 크지 않다고 볼 수 있다.

6. 실험 결과

본 장에서는 실험을 통해 제안하는 역색인 구조의 갱신 성능이 기존 구조에 비해 효율적임을 보인다.

6.1 실험 방법

실험에서는 단문들이 데이터 스트림으로 유입되는 환경에서 역색인의 갱신 비용을 측정하였다. 단문은 가상의 데이터로 생성하였다. 각 단문은 평균적으로 5~20개의 단어로 구성되며, 각 단어는 10,000개의 단어로 이루어진 사전에서 무작위로 추출하였다. 본 실험에서는 단문들이 계속해서 유입될 때 이들을 반영하기 위해 역색인을 갱신하는 데 드는 단문 당 평균 소요 시간과 총 디스크 I/O 비용을 측정하였다. 본 실험에서는 다음 3가지 변수를 변화시켜가며 역색인의 갱신 성능을 측정하였다.

① 유입된 단문 개수 N: 데이터 스트림으로 들어온 단문의 개수를 1×10^6개, 2×10^6개, 3×10^6개, 4×10^6개까지로 증가시켜가며 역색인의 갱신 성능을 측정하였다.

② I_0의 크기 T_0: 메모리 상의 역색인인 I_0가 최대로 저장할 수 있는 단문 ID의 개수를 250,000개, 500,000개, 750,000개, 1,000,000개로 증가시켜가며 성능 변화를 측정하였다. 이것은 각각 1MB, 2MB, 3MB, 4MB의 메모리 크기에 해당한다.

③ 단문 당 평균 단어 개수 A: 단문들이 포함하고 있는 평균 단어 개수를 5개, 10개, 15개, 20개로 증가시켜가며 성능 변화를 측정하였다.

실험에는 CPU가 Intel Core i5-3570이고 메모리는 8GB이며, SATA 3.0 인터페이스의 7200RPM 회전속도를 가진 1TB 하드디스크가 설치된 컴퓨터를 사용하였다. 운영체제로는 Windows 7을 사용하였다.

6.2 실험 결과

Fig. 8은 T_0 = 250,000(1MB)이고 단문 당 평균 단어 개수가 10개일 때, 유입된 단문 개수를 1×10^6개에서 4×10^6개까지 증가시키며 제안 방법(Proposed)과 기존 방법(Naive)에서 역색인 갱신에 소요된 단문 당 평균 시간을 측정한 결과이다. Fig. 8에서 볼 수 있듯이 제안 방법은 기존 방법에 비해 단문 당 역색인 갱신에 소모되는 시간이 훨씬 적다. 또한 단문이 누적되면서 역색인의 크기가 커지면 커질수록 기존 방법은 갱신에 소모되는 시간이 급격히 증가되는 한편, 제안 방법은 천천히 증가됨을 볼 수 있다. 이것은 5.1절에서 분석한 바와 동일한 결과를 나타낸다. Fig. 8에서는 한 단문 당 갱신에 소모되는 시간이 최소 2.11배에서 최대 8.71배까지 차이가 남을 볼 수 있다. Fig. 9와 Fig. 10은 역색인을 갱신하기 위해 기존 방법과 제안 방법이 디스크로부터 읽은 데이터의 양과 디스크에 쓴 데이터의 양을 각각 단문 ID의 개수로 나타낸 것이다. Fig. 9와 Fig. 10에서 볼 수 있듯이 제안 방법의 디스크 I/O 비용은 기존 방법에 비해 훨씬 적다. 역색인의 갱신 비용은 디스크 I/O량에 비례하므로 그로 인해 Fig. 8과 같은 결과가 발생한 것으로 판단된다.

Fig. 11은 유입된 단문 개수가 2×10^6개이고 단문 당 평균 단어 개수가 10개일 때, T_0를 250,000(1MB)에서 1,000,000(4MB)까지 증가시키며 제안 방법과 기존 방법의 갱신 성능을 비교한 결과이다. Fig. 11에서 볼 수 있듯이 T_0가 커질수록 두 방법의 갱신 시간이 줄어드는 것을 볼 수 있다. 이것은 메모리 버퍼 역할을 하는 I_0의 크기가 커질수록 역색인의 갱신 횟수가 줄어들기 때문이다. 기존 방법의 경우 I_0가 꽉 찰 때마다 매번 역색인 전체를 읽고 써야하므로 T_0의 증가에 따라 갱신 횟수가 줄어들수록 효율성에 큰 영향을 받게 된다. 하지만 모든 경우에 대해 제안 방법은 기존 방법에 비해 빠른 갱신 성능을 보임을 알 수 있다. Fig. 11에서 단문 당 갱신에 소모되는 평균 시간은 최소 1.94배에서 최대 3.79배까지 차이를 보인다. Fig. 12와 Fig. 13은 Fig. 11의 실험에서 기존 방법과 제안 방법이 디스크로부터 읽은 데이터 양과 디스크에 쓴 데이터 양을 각각 단문 ID의 개수로 나타낸 것이다. 두 방법 모두 T_0가 증가할수록 디스크 I/O량은 감소하지만, 제안 방법이 훨씬 더 적은 양의 디스크 I/O을 발생시킴을 알 수 있다.

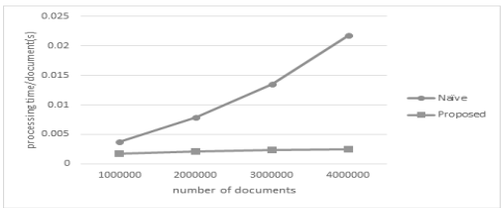

Fig. 8. Average update time per document
(T_0 = 250,000, A = 10)

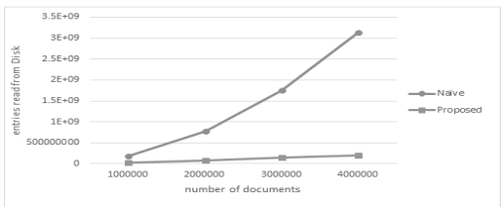

Fig. 9. The number of document IDs read from disk
(T_0 = 250,000, A = 10)

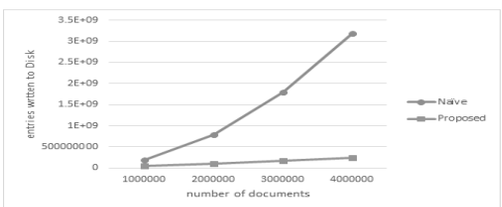

Fig. 10. The number of document IDs written to disk
(T_0 = 250,000, A = 10)

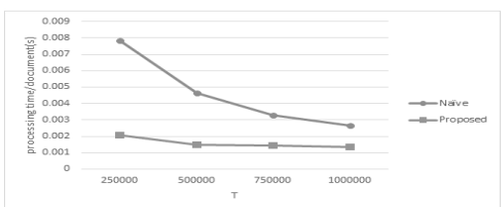

Fig. 11. Average update time per document
(N = 2×10^6, A = 10)

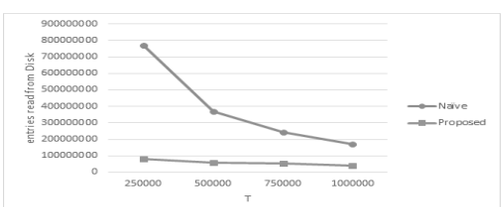

Fig. 12. The number of document IDs read from disk
(N = 2×10^6, A = 10)

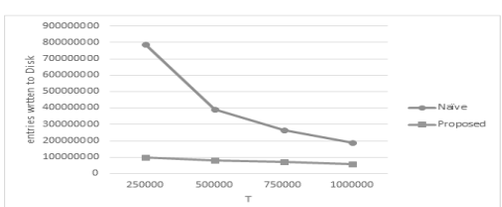

Fig. 13. The number of document IDs written to disk
(N = 2×10^6, A = 10)

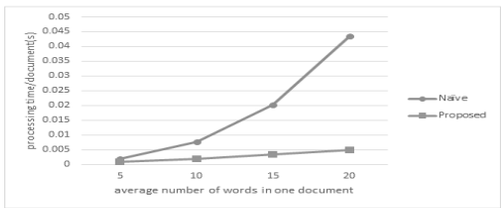

Fig. 14. The number of document IDs read from disk
(T_0 = 250,000, N = 2×10^6)

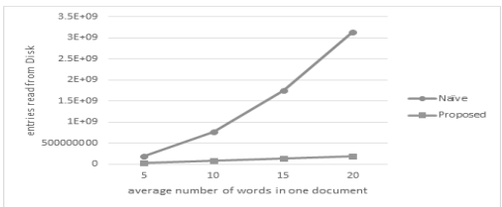

Fig. 15. The number of document IDs read from disk
(T_0 = 250,000, N = 2×10^6)

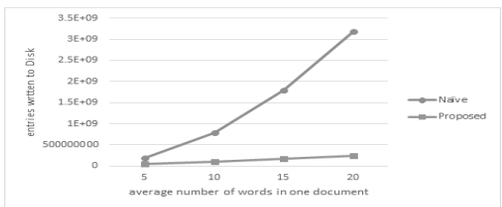

Fig. 16. The number of document IDs written to disk
(T_0 = 250,000, N = 2×10^6)

Fig. 14는 T_0 = 250,000(1MB)이고 유입된 단문 개수가 2×10^{6}개일 때, 단문 당 평균 단어 개수를 5개에서 20개까지 증가시키며 제안 방법과 기존 방법의 성능 변화를 측정한 결과이다. Fig. 14에서 볼 수 있듯이 두 방법 모두 단문 당 평균 단어 개수가 증가할수록 갱신 시간이 증가한다. 이것은 각 단문 당 역색인에 추가해야 하는 문서 ID의 개수가 증가하는 한편, 그로 인해 역색인의 크기도 더 빨리 커지기 때문이다. 역색인의 크기가 클수록 갱신 비용이 증가하게 된다. 하지만 앞 실험과 마찬가지로 제안 방법은 단문 당 평균 단어 개수에 상관없이 기존 방법에 비해 항상 좋은 갱신 성능을 보인다. 또한 Fig. 14의 실험 중 발생한 디스크 I/O 비용을 나타내는 Fig. 15와 Fig. 16 모두에서도 제안 방법은 항상 더 적은 디스크 I/O량을 발생시킴을 알 수 있다.

결론적으로 단문의 개수를 증가시키는 실험, T_0의 크기를 변화시키는 실험, 단문 당 평균 단어 수를 변화시키는 실험 모두에서 제안 방법은 기존 방법에 비해 훨씬 더 적은 디스크 I/O를 발생시킴을 확인하였으며, 그로 인해 실제 갱신 시간도 크게 줄어듦을 확인할 수 있었다.

7. 결 론

본 논문은 데이터가 계속해서 유입되는 환경에서 해당 데이터에 대한 효율적인 키워드 검색이 가능하도록 하는 역색

인 구조를 제안하였다. 제안하는 역색인 구조는 계속해서 유입되는 데이터를 빠르게 색인에 반영하기 위해 매우 적은 디스크 I/O 만으로 역색인 정보를 갱신할 수 있도록 해준다. 제안 방법은 역색인 정보를 크기가 지수적으로 증가하는 여러 역색인에 나누어 저장함으로써, 역색인을 갱신하기 위해 평균적으로 접근해야 하는 데이터량을 줄인다. 특히 제안 방법은 역색인들 간의 병합에 발생하는 디스크 I/O를 크게 줄이는 한편, 디스크에 최적화된 순차 쓰기만을 발생시킴으로써 디스크에 최적화된 갱신 성능을 보인다. 본 논문은 이론적인 분석을 통해 제안 방법의 성능을 보이는 한편, 실험을 통해 단문의 수, 메모리 사용량, 단문의 길이에 상관없이 항상 제안 방법이 기존 방법에 비해 적은 갱신 비용을 가짐을 보였다.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom., "Processing sliding window multi-joins in continuous queries over data streams," in *Proceedings of ACM SIGMOD-SIGACTSIGART Symposium on Principles of Database Systems (PODS)*, pp.1-16, June, 2002.

[2] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *ICDE*, pp. 1360-1369, 2012.

[3] S. Helmer and G. Moerkotte, "A performance study of four index structures for set-valued attributes of low cardinality," *The International Journal on Very Large Data Bases(VLDB)*, Vol.12, No.3, pp.244-261, 2003.

[4] C. Chen, F. Li, B. C. Ooi, and S. Wu, "TI: An efficient indexing mechanism for real-time search on tweets," in *SIGMOD*, pp. 649-660, 2011.

[5] Lingkun Wu, Wenqing Lin, Xiaokui Xiao, and Yabo Xu3, "LSII: An Indexing Structure for Exact Real-Time Search on Microblogs," in *ICDE*, pp.482-493, 2013.

[6] J. Zobel and A. Moat, "Inverted files for text search engines," *ACM Computing Survey*, Vol.38, No.2, July, 2006.

[7] D. Arroyuelo, S. Gonzalez, M. Oyarzun, and V. Sepulveda, "Document identifier reassignment and run-length-compressed inverted indexes for improved search performance," *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp.173-182, 2013.

[8] R. Baeza-Yates and B. Ribeiro-Neto, "Modern Information Retrieval: The Concepts and Technology behind Search," 2nd Edition, Addison-Wesley Professional, 2011.

[9] H. Yan, S. Ding, and T. Suel. "Inverted index compression and query processing with optimized document ordering," in *Proceedings of the 18th international conference on World Wide Web*, pp.401-410, 2009.

[10] Carolina Bonacic, Danilo Bustos, and Veronica Gil-Costa, "Multithreaded Processing in Dynamic Inverted Indexes for Web Search Engines," in *Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval*, pp.15-20, 2015.

[11] M. Stonebraker, "The case for partial indexes," *ACM-SIGMOD Record*, Vol.18, No.4, pp.4-11.

[12] P. Seshadri and A. N. Swami. 1995, "Generalized partial indexes," in *ICDE*, pp.420-427, December, 1989.

[13] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica, "Enhancing p2p file-sharing with an internet-scale query processor," in *VLDB*, pp.432-443, 2004.

[14] E. Adar, "User 4xxxxx9: Anonymizing query logs," in *Workshop on Query Log Analysis at the 16th World Wide Web Conference*, 2007.

[15] J. Lin and G. Mishne, "A study of 'churn' in tweets and real-time search queries," in *Proceedings of the Sixth International AAAI Conference on Weblogs and Social Media*, 2012.

[16] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log structured merge-tree (lsm-tree)," *Journal Acta Informatica*, Vol.33, No.4, pp.351-385, 1996.

박 은 주

e-mail : peunju92@naver.com

2014년 숙명여자대학교 컴퓨터과학부 (학사)

2016년 숙명여자대학교 컴퓨터과학부 (석사)

2016년~현 재 CJ올리브네트웍스 사원

관심분야 : 데이터베이스, 데이터스트림, 질의처리, 빅데이터

이 기 용

e-mail : kiyonglee@sookmyung.ac.kr

1998년 KAIST 전산학과(학사)

2000년 KAIST 전산학과(석사)

2006년 KAIST 전산학과(박사)

2006년~2008년 삼성전자 책임연구원

2008년~2010년 KAIST 전산학과 연구조교수

2010년~현 재 숙명여자대학교 컴퓨터과학부 부교수

관심분야 : 데이터베이스, 데이터스트림, 빅데이터, OLAP, 질의처리