스트링 B-트리의 효율적인 구현

윤주영° 박근수
서울대학교 컴퓨터공학부
{jyyoon°, kpark}@theory.snu.ac.kr

Effective Implementation of String B-trees

Joo Young Yoon° Kunsoo Park
School of Computer Science & Engineering, Seoul National University

요 약

스트링 B-트리는 외부 메모리에 저장된 문자열 데이터의 검색을 효율적으로 수행할 수 있는 자료 구조이다. 본 논문에서는 앞서서 연구된 새로운 분기 알고리즘을 이용하여 전체적인 새로운 스트링 B-트리의 구현 방법을 제시한다. 그리고 실험을 통하여 효율을 최대화하는 구현 방법을 논의한다.

1. 서 론

정보의 디지털화가 진행되면서 방대한 양의 문서가 디지털 데이터로 옮겨지고 있다. 이에 따라 이러한 데이터에서 필요한 정보를 찾아내기 위한 검색 기술의 중요성도 높아지고 있다. 메인 메모리 크기의 증가 속도가 문서의 양이 커지는 속도를 따라가지 못하기 때문에 메모리 내에서 처리할 수 없는 데이터가 점차 증가하고 있으며, 이러한 데이터를 처리하기 위해서 외부 메모리 자료 구조가 사용되고 있다. 외부 메모리의 속도는 메인 메모리에 비하여 매우 느리기 때문에 검색 속도를 빠르게 하기 위해서는 외부 메모리에 대한 접근의 횟수를 줄이는 것이 중요하다. 대표적인 외부 메모리 자료 구조로는 B-트리가 있으며, 그 변형으로 B^+-트리, 접두사 B-트리 등이 있다.

인덱스 자료 구조는 문서를 업데이트하는 경우보다 검색을 수행하는 경우가 많을 때 효과적인 자료 구조이다. 검색할 문서를 미리 처리하여 자료 구조를 생성하면 이를 이용하여 빠른 검색을 수행할 수 있다. 그러나 이러한 자료 구조는 외부 메모리에서 효과적으로 동작하도록 구현하기 어렵다. 대표적인 인덱스 자료 구조로는 Patricia 트라이, 접미사 트리, 접미사 배열 등이 있다.

Ferragina와 Grossi에 의해 고안된 스트링 B-트리[1]는 외부 인덱스 자료 구조로서, 앞의 두 자료 구조의 장점을 결합하여 여러 개의 문자열로 이루어진 방대한 데이터에서 효과적인 검색 작업을 수행할 수 있다. 이 자료 구조는 처리할 모든 문자열을 연결시킨 문자열 T에 대하여, T의 모든 접미사를 사전순서대로 정렬하여 B^+-트리에 저장하고 있다. 그리고 B^+-트리의 각각의 노드는 노드에 포함된 접미사들을 Patricia 트라이를 통해 관리함으로써 검색 및 업데이트를 효과적으로 수행한다. Patricia 트라이는 Ferguson이 고안한 비트 트리[2]를 이용하여 대체가 가능하다.

이 논문에서는 [3]에서 고안된 노드 구조와 분기 알고리즘을 이용하여 탐색 및 업데이트를 포함하는 전체적인 스트링 B-트리의 구현 방법을 제시한다. 새로운 구현은 기존의 스트링 B-트리와 동등한 수준의 검색을 지원하고, 보다 쉽게 구현할 수 있다. 그리고 실제 구현에서 조정 가능한 여러 값에 대한 실험을 통하여 효율을 최대화하는 구현 방법을 논의한다.

2. 스트링 B-트리의 자료 구조

스트링 B-트리의 자료 구조는 두 개의 파일로 이루어져 있다. 문자열 파일은 스트링 B-트리에 포함된 모든 문자열을 각각의 문자열 사이에 특수 문자를 삽입하여 연결시킨 문자열을 저장한다. 따라서 연결시킨 서열의 길이보다 작은 임의의 값은 스트링 B-트리에 포함된 어떤 접미사에 대한 문자열 파일에서의 포인터가 된다. B-트리 파일은 각각의 디스크 페이지마다 B-트리의 하나의 노드를 저장하고 있으며, 노드 내의 구조는 여러 가지 방법으로 구현이 가능하다. 여기에서는 [3]에서 소개된 노드 구조를 사용한다. 이 구조는 압축을 사용하는 Patricia 트라이 구조는 물론이고, 비트 트리를 사용한 구현과 비교해서도 비트 연산이 필요하지 않으므로 더 쉽고 효율적이다. 각각의 노드 $Node[x]$는 접미사 배열 SA_x를 저장하고 있으며, SA_x는 다음과 같은 값을 가진다.

- n_x : 노드에 포함된 접미사 포인터의 수.
- suf_x : $suf_x[i]$는 문자열 $s_x[i]$로의 포인터이며, $s_x[1] \le_L s_x[2] \le_L \cdots \le_L s_x[n_x]$이다.
- lcp_x : $lcp_x[i] = lcp(s_x[i-1], s_x[i])$ $(1 \le i \le n_x+1)$.
- lnc_x : $lnc_x[i] = s_x[i][lcp_x[i]+1]$ $(1 \le i \le n_x)$.
- $child_x$: 자식 노드로의 포인터. $Node[child_x[i]]$의 모든 문자열은 $s_x[i-1]$보다 사전순서상 느리고, $s_x[i]$보다 사전순서상 빠르다 $(1 \le i \le n_x+1)$.

또한, $s_x[0]$과 $s_x[n_x+1]$을 다음과 같이 정의한다. x가 루트이면, $s_x[0]$는 빈 문자열, $s_x[n_x+1]$는 Σ에 포함된 어떤 문자보다도 사전순서상 느린 특수문자로 정의한다. $Node[x]$가 루트가 아니면, $Node[x]$의 부모 노드를 $Node[y]$라고 하고 $Node[x]$가 $Node[y]$의 j번째 자식 노드라고 할 때, $s_x[0] = s_y[j-1]$, $s_x[n_x+1] = s_y[j]$로 정의한다.

3. 노드 내에서의 동적 연산

여기에서는 접미사 배열에서 연결, 분할, 삽입, 삭제의 네 가지 연산을 수행하는 방법을 설명한다.

3.1. SA-Concatenate($SA_{child_x[i]}$, $SA_{child_x[i+1]}$, $suf_x[i]$, c)

$Node[x]$의 접미사 배열 SA_x에서 i번째 문자열 $s_x[i]$와 $s_x[i]$의 좌우에 있는 자식 노드 $Node[child_x[i]]$, $Node[child_x[i+1]]$의 접미사 배열 $SA_{child_x[i]}$, $SA_{child_x[i+1]}$을 하나의 접미사 배열 SA_y로 연결하는 연산이다$(1 \le i \le n_x)$. $suf_x[i]$는 $s_x[i]$의 포인터이며, c는 $s_x[i]$에서 $Node[child_x[i]]$의 마지막 문자열과 다른 첫 번째 문자이다(즉, $c = s_x[i][lcp_{child_x[i]}[n_{child_x[i]}+1]+1]$).

새로운 접미사 배열 SA_y는 다음과 같은 값을 가진다.

$suf_y = suf_{child_x[i]} \cdot suf_x[i] \cdot suf_{child_x[i+1]}$, $lcp_y = lcp_{child_x[i]} \cdot lcp_{child_x[i+1]}$, $lnc_y = lnc_{child_x[i]} \cdot c \cdot lnc_{child_x[i+1]}$, $child_y = child_{child_x[i]} \cdot child_{child_x[i+1]}$, $n_y = n_{child_x[i]} + n_{child_x[i+1]} + 1$.

3.2. SA-Split(SA_x, i)

SA_x와 인덱스 $i(1 < i < n_x)$를 입력으로 하여, $Node[x]$의 문자열 중 $s_x[i]$보다 사전순서상 빠른 문자열들의 접미사 배열 SA_y와 $s_x[i]$보다 사전순서상 느린 문자열들의 접미사 배열 SA_z를 만들어 내는 연산이다. SA_y와 SA_z는 다음과 같은 값을 가진다.

$suf_y = suf_x[1...i-1]$, $\quad suf_z = suf_x[i+1...n_x]$,
$lcp_y = lcp_x[1...i]$, $\quad lcp_z = lcp_x[i+1...n_x+1]$,
$lnc_y = lnc_x[1...i-1]$, $\quad lnc_z = lnc_x[i+1...n_x]$,
$child_u = child_x[1...i]$, $\quad child_z = child_x[i+1...n_x+1]$,
$n_y = i-1$, $\quad n_y = i-1$.

3.3. SA-Insert(SA_x, X, i, l, $mode$, c, ch)

SA_x에 문자열 X를 삽입하는 연산이다. $i(1 \le i \le n_x+1)$는 X가 삽입될 인덱스이며, $l = max(lcp(X, s_x[i-1]), lcp(X, s_x[i]))$, 그리고 $mode$와 c는 다음과 같이 정의된다.

$$mode = \begin{cases} left & \text{if } lcp(X, s_x[i-1]) \ge lcp(X, s_x[i]) \\ right & otherwise \end{cases},$$

$$c = \begin{cases} X[l+1] & \text{if } mode = left \\ s_x[i][l+1] & otherwise \end{cases}.$$

만일 $Node[x]$가 내부 노드라면 X의 오른쪽 자식 노드가 삽입되는 것으로 가정하며, 그 포인터를 ch라고 한다.

X의 삽입은 $mode$의 값에 따라 다르게 진행된다.

- $mode$가 $left$인 경우 : $l' = lcp(s_x[i-1], s_x[i])$라 하면, $s_x[i-1] \le_L X \le_L s_x[i]$에 의하여 X와 $s_x[i]$의 길이 l'인 접두사는 항상 일치한다. 만약 $lcp(X, s_x[i-1]) > lcp(X, s_x[i])$이면 $X[l'+1] = s_x[i-1][l'+1] \ne s_x[i][l'+1]$에 의하여 $lcp(X, s_x[i]) = l'$이다. $lcp(X, s_x[i-1]) = lcp(X, s_x[i])$이면 자연스럽게 $lcp(X, s_x[i]) = l'$이다. 따라서 항상 $lcp(X, s_x[i]) = l'$이다.
- $mode$가 $right$인 경우 : $X[l'+1] = s_x[i][l'+1] \ne s_x[i-1][l'+1]$에 의하여 $lcp(X, s_x[i-1]) = l'$이다.

따라서 X의 포인터를 suf_X라고 하면, X의 삽입된 후의 접미사 배열 $SA_{x'}$은 다음과 같은 값을 가진다.

$suf_{x'} = suf_x[1...i-1] \cdot suf_X \cdot suf_x[i...n_x]$,

$$lcp_{x'} = \begin{cases} lcp_x[1...i-1] \cdot l \cdot lcp_x[i...n_x+1] & \text{if } mode = left \\ lcp_x[1...i] \cdot l \cdot lcp_x[i+1...n_x+1] & otherwise \end{cases},$$

$$lnc_{x'} = \begin{cases} lnc_x[1...i-1] \cdot c \cdot lnc_x[i...n_x] & \text{if } mode = left \\ lnc_x[1...i] \cdot c \cdot lnc_x[i+1...n_x] & otherwise \end{cases},$$

$child_{x'} = child_x[1...i] \cdot ch \cdot lcp[i+1...n_x+1]$.

3.4. SA-Delete(SA_x, i)

SA_x와 인덱스 $i(1 < i < n_x)$를 입력으로 하여, $Node[x]$의 문자열 중 i번째 문자열을 삭제하는 연산이다.

$lcp(s_x[i-1], s_x[i+1]) = \min(lcp(s_x[i-1], s_x[i]), lcp(s_x[i], s_x[i+1]))$

이므로 삭제 후의 접미사 배열 $SA_{x'}$은 다음과 같은 값을 가진다.

$suf_{x'} = suf_x[1...i-1] \cdot suf_x[i+1...n_x]$,

$$lcp_x = \begin{cases} lcp_x[1...i-1] \cdot lcp_x[i+1...n_x+1] & \text{if } suf_x[i] \ge suf_x[i+1] \\ lcp_x[1...i] \cdot lcp_x[i+2...n_x+1] & otherwise \end{cases},$$

$$lnc_{x'} = \begin{cases} lnc_x[1...i-1] \cdot lnc_x[i+1...n_x] & \text{if } suf_x[i] \ge suf_x[i+1] \\ lnc_x[1...i] \cdot lnc_x[i+2...n_x] & otherwise \end{cases},$$

$child_{x'} = child_x[1...i] \cdot lcp[i+2...n_x+1]$.

자식 노드의 포인터는 항상 삭제되는 문자열의 오른쪽 자식 노드가 삭제되는 것으로 가정한다.

4. 탐색 및 업데이트 방법

문자열 집합에서 어떠한 패턴 P가 나타나는 모든 위치를 탐색하기 위해서는 문자열 집합에 대한 스트링 B-트리를 생성하고, 생성된 스트링 B-트리에서 패턴 P를 탐색함으로써 원하는 결과를 얻을 수 있다. 여기에서는 앞에서 설명한 노드 내의 연산을 이용하여 스트링 B-트리에서 패턴을 탐색하는 방법과 업데이트하는 방법을 설명한다. 노드 내의 분기 알고리즘 SA-Search로는 [3]에서 소개된 분기 알고리즘을 사용한다.

4.1. 탐색

스트링 B-트리에서 길이가 m인 패턴 P가 나타나는 모든 위치를 찾기 위해서 SB-Search 연산을 수행한다. SB-Search의 슈도코드는 그림 1와 같다. $cont$는 SA-Search 연산이 P가 $Node[x]$의 범위 안에 포함될 수 있는 문자열일 때에만 정상적으로 동작하기 때문에 추가된 인자로, $s_x[0]$이 P를 접두사로 가지는 경우에 $true$값을 가진다. 이 탐색 알고리즘은 $O(\log_B N + \frac{m+occ}{B})$의 디스크 I/O를 필요로 한다.

```
SB-Search(P, x, l, mode, cont)
    Load SA_x from Node[x];
    if cont then (j, l, mode) := (0, lcp_x[1], left);
    else (j, l, mode) := SA-Search(SA_x, P, l, mode);
    if Node[x] is not a leaf then SB-Search(P, child_x[j], l, mode,
cont);
    if l ≥ m then
        do
            occurrence at suf_x[j];
            j := j + 1;
            if Node[x] is not a leaf then
                SB-Search(P, child_x[j], l, mode, true);
        while lcp_x[j] ≥ m and j ≤ n_x;
```

그림 1 SB-Search의 슈도코드

4.2. 업데이트

스트링 B-트리의 업데이트는 [4]에서 제시된 배치(batch) 삽입 알고리즘을 변형된 노드 구조에 맞게 수정하여 수행한다. 길이가 m인 문자열 Y에 대하여 Y의 모든 접미사를 사전순서대로 $Y_1, ..., Y_m$이라고 한다. 이러한 접미사들을 몇 개의 배치로 쪼개어 각각의 배치를 순서대로 다음의 알고리즘을 이용하여 루트에 삽입한다.

$Y_i, ..., Y_j$를 이번에 $Node[x]$에 삽입할 접미사의 배치라고 하면 각각의 접미사가 포함될 파티션 $K_h = \{X \mid s_x[h-1] <_L X \le_L s_x[h]\} + 1)(1 \le h \le n_x+1)$을 찾는다. 만일 $Node[x]$가 잎새 노드라면 $K = K_1 \cup s_x[1] \cup ... \cup s_x[n_x] \cup K_{n_x+1}$을 노드내의 문자열 집합으로 하는 접미사 배열을 생성하고, 분할 연산이 필요한 경우 수행하여 새로 생성된 두 접미사 배열 사이의 문자열을 반환한다. $Node[x]$가 내부 노드라면 각각의 K_h를 배치로 하여 재귀적으로 배치 삽입을 수행한다. 그 결과로 반환된 문자열의 집합 L_h에 대하여 $L = L_1 \cup s_x[1] \cup ... \cup s_x[n_x] \cup L_{n_x+1}$을 노드내의 문자열 집합으로 하는 접미사 배열을 생성하고, 분할 연산이 필요한 경우 수행하여 새로 생성된 두 접미사 배열 사이의 문자열을 반환한다.

이러한 업데이트 알고리즘은 $O(m \log_B (N+m))$의 디스크 I/O를 필요로 한다. 그러나 다음 장에서 제시될 실험 결과는 여러 개의 버퍼와 LRU 정책에 의하여 실제 업데이트가 적은 디스크 I/O 만으로 이루어질 수 있음을 보여준다.

5. 실제 구현상의 이슈

여기에서는 실제 구현에서 결정이 필요한 몇 가지의 값을 결정하기 위한 실험을 수행한다. 모든 실험에서 페이지의 크기로 32768byte를 사용하여 임의로 생성한 1024 길이의 DNA 문자열 1024개를 삽입한다. 페이지 버퍼의 관리는 LRU 정책을 사용하되, 루트부터 현재 열고 있는 노드까지의 길에 있는 노드는 모두 버퍼에 보관하도록 하였다.

5.1. 배치의 크기

배치 삽입 알고리즘에서 효율적인 배치의 크기를 알아내기 위하여 다양한 배치 크기를 사용하여 문자열을 스트링 B-트리에 삽입하는 실험을 수행하였다. 이 실험에서는 8개의 페이지 버퍼를 사용하였다. 실험 결과로 얻은 디스크 I/O의 수와 생성된 B-트리 파일의 크기는 그림 2와 같다. 이 결과에서 배치 크기에 따른 I/O의 수는 거의 차이가 나지 않는다. 그러나 배치의 크기가 클수록 생성된 B-트리 파일의 크기가 커지는 경향을 보여주고 있다. 이러한 경향성은 문자열의 길이와 문자열의 수를 다르게 하여 실험할 때에도 역시 나타난다. 따라서 불필요한 공간 낭비를 줄이기 위해서는 64 정도의 배치 크기가 효율적이다.

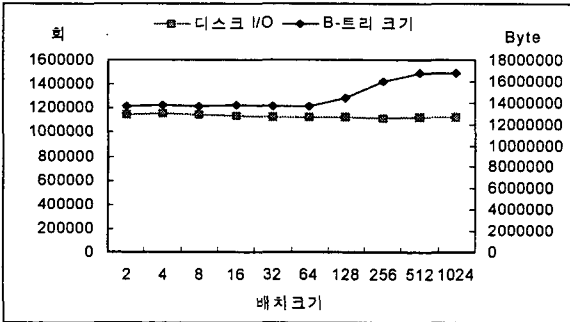

그림 2 배치 크기에 따른 디스크 I/O의 수와 B-트리 파일의 크기

5.2. 페이지 버퍼와 디스크 I/O

페이지 버퍼의 수가 디스크 I/O의 수에 주는 영향을 관찰하기 위한 실험을 수행하였다. 이 실험에서는 배치 크기로 256을 사용하였다. 버퍼의 수에 따른 디스크 I/O의 수는 그림 3과 같다. B-트리에 대한 디스크 I/O의 수는 버퍼의 수에 상관없이 거의 일정하다. 하나의 문자열을 삽입할 때 문자열의 접미사를 이미 정렬하고 사전순서대로 삽입하기 때문에, 새로운 접미사가 삽입될 노드들은 preorder로 방문된다. 따라서 이미 방문한 노드는 이번 삽입에서 다시 방문되지 않으므로 B-트리의 노드는 버퍼에 저장되어 있어도 다시 읽게 되는 경우가 거의 없다. 반면에 문자열 파일에 대한 접근은 랜덤 접근이므로 버퍼에 저장된 페이지가 많을수록 다시 읽어야 할 가능성이 줄어든다. 이에 따라 버퍼의 수가 두 배가 되면 문자열 I/O의 수는 약 절반이 되는 경향을 보인다.

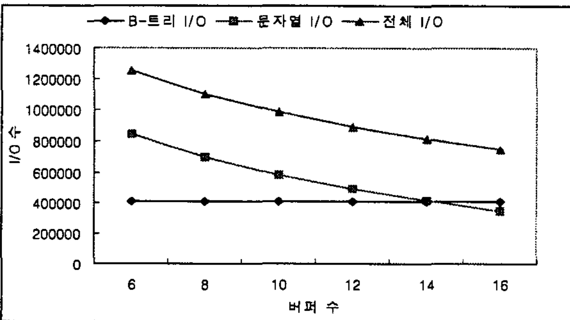

그림 3 버퍼의 수에 따른 디스크 I/O의 수

5.3. 문자열의 수와 디스크 I/O

앞에서 언급한 바와 같이 하나의 문자열을 삽입할 때의 B-트리의 노드에 대한 I/O의 수는 이미 최적화되어 있으므로 더 이상 줄일 수 없다. 그러나 한번의 삽입 연산에서 삽입하는 문자열의 수를 증가시킴으로써 전체 문자열들의 삽입에 필요한 B-트리의 I/O의 수를 줄일 수 있다. 문자열 사이에 Σ에 포함되지 않은 특수 문자를 끼워서 문자열들을 연결시켜 하나의 문자열로 만들고, 다시 이 연결된 문자열을 B-트리에 삽입하면, 연결에 사용된 모든 문자열을 B-트리에 삽입한 것과 동일한 결과를 얻을 수 있다. 따라서 연속적으로 여러 개의 문자열을 삽입할 때 문자열들의 길이의 합이 삽입 가능한 문자열의 최대 길이보다 짧다면 문자열을 연결시켜 삽입하는 것이 효율적이다. 16개의 페이지 버퍼와 256의 배치 크기로 실험을 수행하였다. 그림 4는 연결시킨 문자열의 수에 따른 디스크 I/O의 수이다. 연결시킨 문자열의 수가 2배가 되면 B-트리 I/O의 수는 약 절반이 되는 경향을 보이며, 문자열 I/O의 수도 약간 줄어든다.

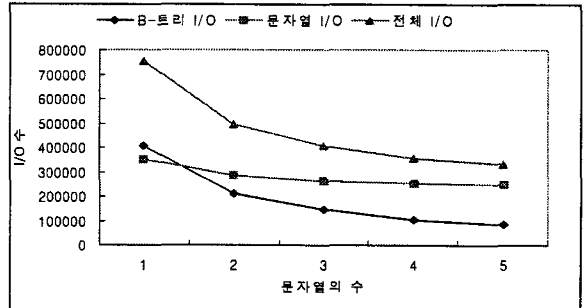

그림 4 연결시킨 문자열의 수에 따른 디스크 I/O의 수

6. 결 론

우리는 이 논문에서 [3]의 노드 구조와 분기 알고리즘을 이용하여 스트링 B-트리를 효율적으로 구현하였다. 새로운 구현 방법은 기존의 구현 방법과 비교할 때 각각의 연산 수행에서 디스크 I/O의 수에서 동등하다. 그리고 노드 내에서의 연산은 Patricia 트라이를 이용한 구현 방법은 물론이고, 비트 트리를 이용한 구현에 비해서도 비트 연산이 필요하지 않으므로 더 쉽고 효율적으로 구현된다. 구현상에서 결정이 필요한 값에 대해서는 실험을 통하여 결정하였다. 특히 메모리를 조절한 실험을 통하여 페이지 버퍼의 수를 두 배로 하면 문자열 파일에 대한 디스크 I/O의 수가 약 1/2배가 되고, 한번에 삽입하는 접미사의 수를 두 배로 하면 B-트리 파일에 대한 디스크 I/O의 수가 약 1/2배가 되는 사실을 확인하였다. 따라서 정해진 크기의 메모리를 사용하여 많은 양의 문자열을 처리하여 B-트리를 생성할 때에는 페이지 버퍼로 사용할 메모리와 패턴과 접미사 배열을 저장하기 위해 사용할 메모리의 비율을 적절하게 조절하면 보다 빠르게 B-트리를 생성할 수 있음을 보여주었다.

References

[1] P. Ferragina and R. Grossi, The string B-tree: A new data structure for string search in external memory and its applications, Journal of the ACM, 46(2):236-280, 1999.

[2] D. E. Ferguson, Bit-Tree: A data structure for fast file processing, Communications of the ACM, 35(6):114-120, 1992.

[3] J. C. Na and K. Park, Simple implementation of string B-trees, 11th Symposium on String Processing and Information Retrieval (SPIRE), Lecture Notes in Computer Science, vol.3246, Springer, pp.214-215, 1994.

[4] P. Ferragina and R. Grossi, Fast string searching in secondary storage: Theoretical developments and experimental results, In Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (Atlanta, Ga., Jan. 28-30), ACM, pp. 373-382, 1996.