중복제거 파일시스템에서 서머리 기반 인덱싱 기법

이중수*, 안창원*
*한국전자통신연구원
e-mail : joongsoo@etri.re.kr

A Method of Summary based Indexing in De-duplication File System

Joongsoo Lee*, Chang-Won Ahn*
*Electronics and Telecommunications Research Institute

요 약

중복제거 파일 시스템은 가상머신 이미지와 같이 서로 중복되는 데이터가 많은 파일에서 용량을 줄이기 위하여 많이 사용된다. 중복제거를 위하여 많은 경우 서머리 벡터와 인덱스를 함께 사용하고 있는데, 이는 메모리를 많이 소모하고 인덱스 구조에 따라 여러 번의 하드 디스크 접근을 해야 하는 한계가 있었다. 본 논문에서는 서머리 벡터를 인덱스 내에서 활용하고 하드디스크를 접근하는 횟수를 감소할 수 있는 인덱싱 기법을 제안한다.

1. 서론

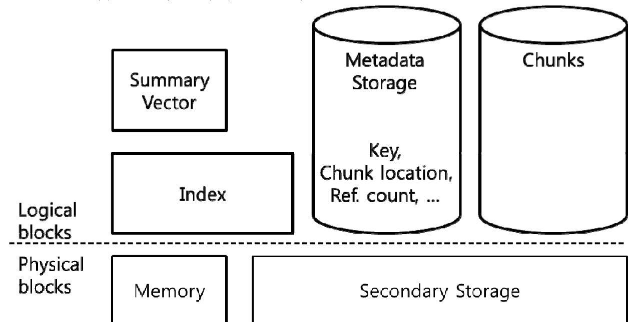
중복제거(de-duplication) 파일 시스템[1, 2]은 서로 유사한 파일의 부분을 각각 파일마다 저장하지 않고 한 번만 저장함으로써 스토리지의 용량을 감소시키는데 그 목적이 있다. 최근, 클라우드 컴퓨팅이 각광을 받으면서 많이 사용되는 가상머신에서는 OS, 애플리케이션 등 가상하드디스크 내의 많은 부분들이 중복되는데, 중복제거를 통해 효율적인 스토리지 활용이 가능하다.

중복제거를 위해 많이 사용하는 서머리 벡터는 key 가 저장 여부를 판단해 주는 매우 간단하지만 강력하고 효율적인 도구이다. 그러나, 서머리 벡터의 활용과는 별개로 인덱스를 활용하여 거짓 양성(false positive)을 확인해야 하는 경우가 있다. 데이터 용량이 커지면 인덱스가 메모리에만 저장되기 어려운 경우가 발생하여 성능 저하의 원인이 된다. 또한, 하드디스크에 저장된 인덱스를 따라 몇 번의 이동을 해야 하는지 한계가 정해지지 않아 반응 속도를 예측하기 어렵다.

본 논문에서는 별개의 자료구조로서 서머리 벡터를 유지하지 않고 인덱스 구조와 함께 유지하고 서머리 벡터의 결과와 연결하여 인덱스를 검색할 수 있도록 함으로써 메모리 효율성을 추구하였다. 또한, 최대 한 번의 하드디스크 접근이 일어나도록 함으로써 반응 시간이 예측 가능하도록 하였다.

2. 중복제거 파일 시스템

중복제거를 위해서는 파일을 작은 부분(chunk)로 나누고 각 부분들을 해쉬(hash) 함수를 사용하여 유일한 ID 를 생성하는 것이 일반적이다. 해쉬함수는 SHA-1 이나 SHA-512 와 같이 암호학적으로 검증된 함수가 흔히 사용된다. Chunk 에 대해 해쉬함수를 돌려본 다음 다른 key 값이 생성된다면, 다른 chunk 로 인식한다. 서로 다른 chunk 이지만, 같은 Key 가 생성될 가능성도 있으나 매우 낮다.



(그림 1) 중복 제거 엔진 블록 구성도

(그림 1)에 중복제거 파일 시스템의 중복제거 엔진에서 흔히 사용하는 블록 구성도를 도시하였다. 시스템의 반응 속도를 고려하여 메모리 용량을 충분히 설치할 수 있으나 모든 인덱스를 메모리에 올려 두고 사용하는 것은 불가능하다. 일부분 2 차 스토리지, 즉 HDD 등을 사용하여야 하는 경우 성능 저하가 크게 일어날 수 있다.

서머리 벡터(summary vector)는 (key, value) 쌍에 대한 검색에 있어 key 의 존재여부만 검사하기 위해 고안된 특별한 자료구조이다. 서머리 벡터 중 가장 많이 사용되는 블룸 필터(bloom filter)[3]는 비트 어레이와 여러 개의 해쉬 함수를 사용하는데, key 가 등록될 때는 각 해쉬 함수에 의해 매핑되는 모든 비트 어레이의 위치에 1 로 표시해 둔다.

Key 를 검색할 때는 해쉬 함수에 의해 매핑되는 모든 비트 어레이의 값에 하나라도 0 이 있으면, DB 에 존재하지 않는 key 임을 의미한다. 그러나, 모든 값이 1 이라 하더라도 존재하지 않는 경우, 즉 거짓 양성(false positive)이 존재할 수 있다.

메타데이터 저장소(Metadata storage)는 Chunk 의 위치, chunk 가 사용되는 횟수, 크기 등 chunk 를 관리하기 위해 필요한 데이터들을 담고 있다. 인덱스(index)는 메타데이터 저장소에 대한 인덱스이며, 검색이 일어나는 경우, 메타데이터 저장소까지 도달하고 나면 그 이후는 데이터를 다루는 차원에서의 오퍼레이션만 존재한다.

(그림 1)과 같은 구조에서, key 검색은 서머리 벡터에서 먼저 확인한 후, 존재한다는 확인을 하더라도 거짓 양성일 경우를 대비하여 인덱스에서 검색을 통해 최종 확인을 하는 과정을 거친다. 인덱스는 B-Tree 등을 사용하여 구현되는 경우가 대부분이므로 검색이 효율적이나 HDD 에 저장된 인덱스 등이 검색 과정에 혼합될 수밖에 없으며 이 때, 큰 속도 저하가 일어난다. 또한, 두 개의 상이한 자료구조를 메모리에 유지함으로써 발생하는 공간적인 오버헤드도 상당하다.

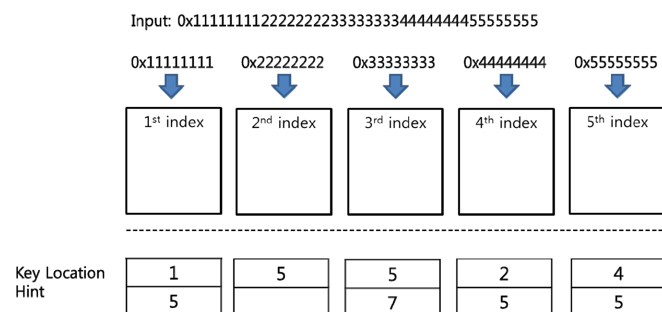
본 논문에서는 서머리 벡터와 인덱스를 혼합한 구조를 통해 서머리 벡터에서의 검색과 함께 인덱스 검색이 가능케 하는 기법을 제안하고자 한다.

3. 인덱싱이 가능한 서머리 벡터

3.1. 부분키와 서머리 벡터

하나의 key 는 n 개의 부분키로 나뉘어 검색이 이루어진다. 예를 들어, SHA-1 을 사용할 경우 160bit 크기의 해쉬된 결과가 나오는데, 이것을 5 개의 부분으로 나눌 경우 하나의 부분키는 32bit 씩 구성된다. 각 부분키 인덱스 내에는 b bit 크기의 부분키가 있고, 하나의 서머리 벡터 안에는 모두 k 개의 key 에 대한 서머리를 저장할 수 있다. 이 때 k 는 b 보다 작다. 서머리 벡터를 저장하는 방법은 부분키를 b 로 나눈 나머지(key_part mod k)에 대응하는 비트를 1 로 세팅하는 것으로 처리한다.

SHA-1 과 같은 성능이 좋은 해쉬 함수들은 해쉬를 한 결과의 일부분에도 여전히 랜덤 특성이 강하며, 연속균등분포를 보이기 때문에 key 를 여러 개의 부분으로 나누어서 각각 서머리 벡터의 필터로 기능하게 할 수 있다.



(그림 2) 부분 키에 의한 검색

(그림 2)에서 160bit 의 키 값을 다섯 개의 부분으로 나누고, 각각의 부분키 인덱스에서 검색한 결과를 활용하여 최종적인 검색 결과를 얻어내는 과정을 보이고 있다. 32bit 씩 쪼개진 부분키를 각각 인덱스에서 검색하면 서머리 벡터를 채울 때와 동일하게 모듈러 연산을 한 결과에 대응되는 비트가 1 로 세팅되어 있는지 검사하고, 1 로 세팅되어 있으면 그 결과로 key 가 저장된 위치에 대한 힌트를 돌려준다. Key 는 저장된 순서에 따라 크기가 늘어나는 형태로 저장된다고 가정하면 Key 위치에 대한 힌트가 빠를수록 먼저 저장된 키임을 의미한다. 여기서 힌트는 전체 저장 가능한 힌트 개수를 일정 영역으로 나눈 것이다.

이 힌트가 모두 일치하면 해당 키는 인덱스에 존재하는 것으로 인식할 수 있지만, 이 역시 거짓 양성(false positive)이 존재할 수 있어서 실제 검색은 서머리 벡터에 대응하는 키 블록을 확인해 봐야 한다.

3.2. 부분키 인덱스의 구조

부분키 내에서 key 를 찾아가는 방법으로 이진트리와 같은 자료구조를 사용할 수 있다. 본 논문에서는 검색의 용이를 위하여 prefix 가 겹치는 key 들을 그루핑하는 방법을 사용하였다. 부분키 인덱스 내에서는 x bit 의 prefix 를 기준으로 그룹을 형성한다.

하나의 서머리 벡터 엔트리가 저장할 수 있는 k 개씩 묶어서 key 블록에 저장하였다. Key 블록은 메모리 용량의 한계에 의해 HDD 에 저장되는 경우가 많을 것으로 판단되는데, 본 논문에서 제안하는 방법에 의하면 거짓 양성을 판단하기 위하여 한 번의 디스크 액세스가 있을 수 있다. 이는 기존에 B-Tree 를 따라 여러 번의 디스크 액세스를 할 가능성이 있었던 것에 비해 많은 성능 향상을 가져올 것으로 판단된다.

4. 결론

인덱스와 서머리 벡터를 결합함으로써 구현에 있어서 메모리 공간을 절약하고, 디스크 액세스 횟수가 예측 가능하도록 설계되어 블룸 필터와 B-Tree 를 사용하는 경우에 비해 효율적인 인덱스 구축이 가능하다. 향후 실험 결과로 두 가지 기법에 대해 정량적인 비교를 진행할 계획이다.

참고문헌

[1] Benjamin Zhu, Kai Li, and Hugo Patterson, “Avoiding disk bottleneck in the data domain deduplication file system,” in Proceedings of FAST’08, 2008.

[2] Sean Quinlan and Sean Dorward, “Venti: A New Approach to Archival Storage,” in Proceedings of FAST’02, pp. 89-101, 2002.

[3] Andrei Broder, Michael Mitzenmacher, Andrei Broder, and Michael Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” Internet Mathematics, pp.636-646, 2002.