논문 17-42-03-04 The Journal of Korean Institute of Communications and Information Sciences '17-03 Vol.42 No.03
https://doi.org/10.7840/kics.2017.42.3.566

Sorting Cuckoo: 삽입 정렬을 이용한 Cuckoo Hashing의 입력 연산의 성능 향상

민 대 홍•, 장 룡 호*, 양 대 헌**, 이 경 희°

Sorting Cuckoo: Enhancing Lookup Performance of Cuckoo Hashing Using Insertion Sort

Dae-hong Min•, Rhong-ho Jang*, Dae-hun Nyang**, Kyung-hee Lee°

요 약

키-값 저장소(key-value store)는 Redis, Memcached 등의 다양한 NoSQL 데이터베이스에 응용되어 그 우수성을 보였다. 그리고 키-값 저장소 응용프로그램은 대부분의 환경에서 삽입 연산(insert) 보다 탐색 연산(lookup)이 많이 발생하기 때문에 탐색의 성능이 중요하다. 하지만 기존의 응용프로그램은 해시 테이블을 링크 리스트(linked list) 형태로 유지하기 때문에 탐색 연산이 느릴 수 있다. 따라서 탐색 연산을 상수 시간 내에 완료할 수 있는 쿠쿠 해싱(cuckoo hashing)이 학계의 주목을 받기 시작했고, 그 후 메모리 사용률이 더 높은 버킷화 쿠쿠 해싱(Bucketized Cuckoo Hashing, BCH)이 제안되었다. 본 논문에서는 BCH 구조를 기반으로 하여 삽입 정렬 방법으로 데이터를 입력하는 Sorting Cuckoo를 소개한다. Sorting Cuckoo를 이용하면 데이터가 정렬된 상태에서 탐색을 수행하기 때문에 상대적으로 적은 메모리 접근을 통해 키의 존재 여부를 판단할 수 있으며, 메모리 점유율(load factor)이 높을수록 BCH보다 탐색의 성능이 좋아진다. 실험 결과에 의하면 Sorting Cuckoo는 메모리 점유율이 95%인 상황에서 BCH보다 천만 번의 negative 탐색(데이터가 존재하지 않는 탐색)에서는 최대 25%(약 1900만 회), 천만 번의 positive 탐색(데이터가 존재하는 탐색)에서는 최대 10%(약 400만 회)만큼 더 적은 메모리 접근을 이용하였다.

Key Words : Key-value stores, Cuckoo hashing, Insertion sort, Hash tables, Lookup performance

ABSTRACT

Key-value stores proved its superiority by being applied to various NoSQL databases such as Redis, Memcached. Lookup performance is important because key-value store applications performs more lookup than insert operations in most environments. However, in traditional applications, lookup may be slow because hash tables are constructed out of linked-list. Therefore, cuckoo hashing has been getting attention from the academia for constant lookup time, and bucketized cuckoo hashing (BCH) has been proposed since it can achieve high load factor. In this paper, we introduce Sorting Cuckoo which inserts data using insertion sort in BCH structure. Sorting Cuckoo determines the existence of a key with a relatively small memory access because data are sorted

※ 이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단 글로벌연구실사업의 지원을 받아 수행된 연구임(NRF-2016K1A1A2912757).
• First Author : Inha University Department of Computer Science Engineering, mang@seclab.inha.ac.kr, 학생회원
° Corresponding Author : Suwon University Department of Electrical Engineering, khlee@suwon.ac.kr, 정회원
* Inha University Department of Computer Science Engineering, jiyoo@seclab.inha.ac.kr, 학생회원
** Inha University Department of Computer Science Engineering, nyang@inha.ac.kr, 정회원
논문번호 : KICS2017-01-023, Received January 23, 2017; Revised February 16, 2017; Accepted February 16, 2017

in each buckets. In particular, the higher memory load factor, the better lookup performance than BCH's. Experimental results show that Sorting Cuckoo has smaller memory access than BCH's as many as about 19 million (25%) in 10 million negative lookup operations (key is not in the table), about 4 million times (10%) in 10 million positive lookup operations (where it is) with load factor 95%.

Ⅰ. 서 론

키-값 저장소(key-value store)는 구조가 간단하고 좋은 확장성을 지니고 있기 때문에 데이터의 저장이 필요한 많은 분야에 사용되고 있다. 최근에는 메모리의 가격 인하 및 클라우드 컴퓨팅의 고속화에 맞춰 Redis[1]와 같은 인메모리 데이터베이스(in-memory database)의 사용이 많아지고 있다[2]. Redis, Memcached[3] 등의 키-값 저장소는 분리 체인(separate chaining) 기법을 기반으로 충돌을 해결하는 해시 테이블을 사용하는데, 이를 메모리에 구현함으로써 데이터베이스 시스템의 효율성을 높였다. 하지만 Fan 등은 기존 Memcached에 분리 체인 기법 대신 버킷화 쿠쿠 해싱(Bucketized Cuckoo Hashing, BCH)[4]을 적용하여 성능을 향상시킨 MemC3[5]을 제시하였다.

쿠쿠 해싱(cuckoo hashing)[6]은 $O(1)$ 내에 탐색을 마칠 수 있는 특징을 가지고 있다. 쿠쿠 해싱을 최초로 제시한 Pagh 등은 각각 두 개의 해시 함수와 테이블을 사용하는 해시 테이블 구조를 제안하였고, 이후 많은 연구를 통해[4,7-9] 쿠쿠 해싱에서 BCH 형태로 변화해 왔다(2장 2절 참조). 또한 해시 테이블은 가장 기본적인 자료 구조 중 하나이기 때문에 학계에서 지속적인 연구가 진행되고 있다[10,11].

키-값 저장소의 주요 기능은 삽입(insert), 탐색(lookup), 갱신(update), 그리고 삭제(delete)이다. 그런데 삽입, 갱신, 삭제 연산은 모두 탐색 연산을 동반하기 때문에 탐색의 효율이 더욱 중요하다. 따라서 본 논문에서는 기존 BCH 구조를 바탕으로 탐욕(greedy) 방식 대신 데이터의 키를 기준으로 정렬하여 데이터를 입력함으로써 상대적으로 적은 메모리 접근으로 positive 탐색(데이터가 존재하는 탐색) 및 negative 탐색(데이터가 존재하지 않는 탐색)을 수행할 수 있는 Sorting Cuckoo를 제안한다. 실험 결과에 의하면 Sorting Cuckoo의 negative 탐색은 메모리 점유율 40%부터, positive 탐색은 80%부터 BCH보다 더 적은 메모리 접근 횟수를 기록하였으며, 메모리 점유율이 높아질수록 더 많은 차이를 보였다(4장 2절 참조). 특히, 메모리 점유율 95%일 때 negative 탐색은 약 25%, positive 탐색은 약 10%만큼의 메모리 접근 횟수가 감소한 것을 알 수 있다.

Ⅱ. 배경 지식

2.1 관련 연구

쿠쿠 해싱 이전에 사용되었던 해시 충돌 해결 알고리즘(예를 들면, 선형 탐사법(linear probing), 이차 탐사법(quadratic probing)[12] 등)은 메모리 점유율(load factor)이 높은 상황에서 삽입 및 탐색 연산을 수행하는데 많은 비용을 소모하였다. 또한 분리 체인 기법은 데이터가 들어갈 새로운 공간을 확보하여 해시 충돌을 해결하였다. 그 결과, 데이터를 입력할 때마다 추가적인 메모리를 사용하였고, 해시 충돌이 많이 발생한 곳에서 긴 탐색 시간이 요구되었다.

반면, 쿠쿠 해싱은 두 개의 해시 함수를 이용하여 접근한 곳에 반드시 데이터를 입력함으로써 탐색 연산을 상수 시간 안에 완료되도록 하였다. 하지만 메모리 점유율이 높은 상황에서 삽입 연산의 수행 시간이 느리다. 이러한 이유로 탐색 연산을 상수 시간 내에 완료하면서도 삽입 연산을 개선하기 위한 많은 연구가 있었다. 먼저, BCH는 버킷 내에 데이터의 입력이 가능한 공간을 늘림으로써 데이터를 밀어내는 상황을 줄이고, 메모리 효율성도 높였다. 그리고 높은 메모리 점유율을 유지하면서도 삽입 연산의 수행 시간을 감소시켰다. Cuckoo-Overlap[8]과 Cuckoo-Choose-K[9]는 버킷이 서로 분리된 상태가 아니라 겹쳐져 있는 상태라는 것에 착안하여 메모리 효율성을 더 높였고 데이터 입력 시간도 단축시켰다. 또한, Kuszmaul은 큐(queue) 등을 활용하여 특정 버킷에 데이터가 집중되는 상황을 줄이고, 데이터를 밀어내는(kick-out) 방식도 변형시켜 메모리 점유율이 높은 상황에서 삽입 연산의 성능을 향상시켰다[10].

입력 연산의 개선뿐만 아니라 응용 분야에 관한 연구도 진행되었다. MemC3는 기존 분리 체인 기반의 해시 테이블을 사용하던 Memcached를 변형하여 BCH 구조에 동시성(concurrent)을 증가시킬 수 있는 잠금(lock) 알고리즘과 데이터가 이용된 시간을 이용하는 LRU(Least Recently Used) 기법을 추가하여 삽입과 탐색 연산 모두 개선시켰다. MemC3는 메모리를

읽어오는 연산이 대부분인 상황을 가정했지만, Li 등은 특정 하드웨어를 통해 잠금 알고리즘을 변경하여 쓰기 연산이 더 많은 상황에서 쿠쿠 해싱의 성능 향상을 보이기도 하였다[13].

최근에는 BCH 구조의 변형과 SIMD(Single Instruction Multiple Data) 연산이 가능한 하드웨어를 이용하여 탐색 연산을 향상한 홀튼 테이블(Horton table)[11]이 제안되었다. 이론적으로 메모리 점유율 100%인 BCHT(Bucketized Cuckoo Hash Table)에서는 positive 탐색, negative 탐색을 수행할 때 각각 버킷을 1.5, 2.0회 접근하지만, 홀튼 테이블은 메모리 점유율 95%를 유지하면서 각각 1.18, 1.06회의 버킷을 참조하여 positive 탐색과 negative 탐색을 완료하였다.

2.2 버킷화 쿠쿠 해싱(Bucketized Cuckoo Hashing, BCH)

기존의 쿠쿠 해싱에서는 데이터가 들어갈 공간이 적었다. 테이블이 두 개 밖에 없었고 그 두 테이블에 반드시 데이터를 입력해야 하므로 다른 값들을 계속 밀어내는 상황이 발생 하였다. 결국 할당된 메모리를 50% 정도 밖에 사용하지 못하였고, 그 상황에서 데이터를 입력하면 무한 반복을 수행하는 문제가 발생하였다. 이를 해결하기 위해 테이블 공간을 두 배로 늘려 기존의 데이터 옮기는 재해싱(re-hashing) 방법을 사용해야 하였다.

더 많은 메모리를 사용하기 위해서 데이터가 들어갈 수 있는 공간(버킷, bucket)에 슬롯(slot)을 추가하는 BCH가 제안되었다. BCH 기법에서는 두 개의 테이블을 따로 유지하지 않고 하나의 테이블에서 서로 다른 두 개의 해시 함수를 사용하여 버킷에 접근하였다.

BCH 기법에서는 탐욕 방식으로 데이터를 삽입한다(그림 1 참조). 데이터 (K, V)가 주어지면 해시 함수(h_1)를 사용하여 첫 번째 버킷의 위치($h_1(K)$)를 계산하고, 그 버킷의 빈 공간(슬롯)에 데이터를 입력한다. $h_1(K)$에 빈 공간이 없으면, 두 번째 버킷($h_2(K)$)에서 같은 과정을 반복한다. $h_1(K)$, $h_2(K)$에 빈 슬롯이 없으면 임의로 한 슬롯을 밀어낸 후 그 자리에 데이터를 삽입한다. 다른 버킷에서 빈 슬롯을 발견할 때까지 밀어내기를 반복한다. 빈 슬롯을 발견하면 밀려난 데이터를 입력하고 삽입 연산이 종료된다.

BCH의 탐색은 $h_1(K)$, $h_2(K)$ 순으로 버킷에 접근하여 각 슬롯($S_1 \rightarrow S_4$ 순)에 K가 존재하는지 확인한다. $h_1(K)$에서 빈 슬롯을 찾은 경우에는 $h_2(K)$에 대한 탐색은 진행하지 않는다.

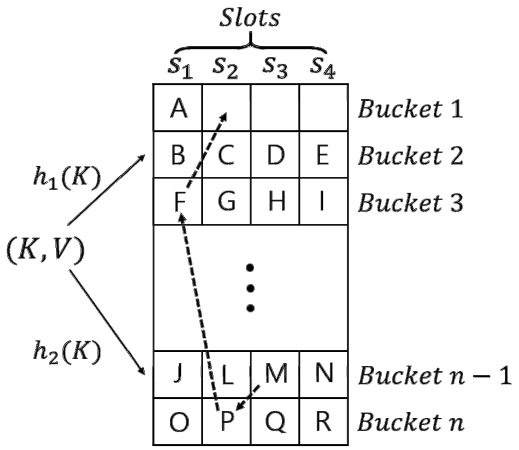

그림 1. 버킷화 쿠쿠 해싱의 삽입 연산 과정
Fig. 1. Insert process of bucketized cuckoo hashing

2.3 BCH의 문제점

2.3.1 Negative 탐색에서의 문제점

$h_1(K)$과 $h_2(K)$의 모든 공간에 K가 존재하지 않는 경우를 negative 탐색이라고 한다. 메모리 점유율이 낮을 때는 $h_1(K)$에 빈 슬롯이 존재할 가능성이 높아 $h_1(K)$의 슬롯(메모리)만 접근하면 K가 존재하지 않는 것을 알 수 있다. 하지만 점유율이 높을 때는 $h_1(K)$에 빈 슬롯이 존재할 확률이 낮으며, 최악의 경우 $h_1(K)$, $h_2(K)$의 모든 슬롯을 접근하여야 K가 존재하지 않는 것을 알 수 있다.

2.3.2 Positive 탐색에서의 문제점

$h_1(K)$ 혹은 $h_2(K)$에 K가 존재하는 경우를 positive 탐색이라고 한다. 메모리 점유율이 낮은 상황에서는 K가 $h_1(K)$에 존재할 확률이 높아 적은 메모리 접근으로 탐색을 완료할 수 있다. 그러나 점유율이 높을 때는 K가 $h_2(K)$에 존재할 수 있는데, 이 경우에서는 반드시 $h_1(K)$의 모든 슬롯을 접근하여 K가 존재하지 않는지 확인하여야 한다.

위에 제시한 것처럼 BCH 기법의 문제점은 메모리 점유율이 높은 상황에서 버킷에 K가 존재하지 않는지 확인하는 과정에서 발생한다. 따라서 상대적으로 적은 메모리 접근으로 K가 존재하지 않는 것을 확인할 수 있다면 BCH의 탐색 성능이 향상될 수 있다.

Ⅲ. Sorting Cuckoo

Sorting Cuckoo는 BCH 기법에서와 같은 탐욕 방식이 아닌 삽입 정렬(insertion sort) 방식을 이용하여 데이터 (K, V)을 입력한다. 각 버킷이 정렬된 상태에서 탐색 연산을 진행하기 때문에 모든 슬롯을 접근하지 않아도 데이터의 존재 여부를 판단할 수 있다. 따라서 메모리 접근 횟수를 줄여 탐색 연산의 성능 향상을 기대할 수 있다.

3.1 삽입

주어진 키(K)에 대하여 K를 입력할 수 있는 버킷 후보를 두 개의 해시 함수를 이용하여 계산 한다 ($h_1(K)$, $h_2(K)$). 데이터의 입력 과정은 삽입과 밀어내기 전략을 통해 진행된다.

3.1.1 삽입 전략

첫 번째 버킷($h_1(K)$)에 빈 슬롯이 존재하는지 확인한다. 즉, 마지막 슬롯(S_4)을 먼저 접근한다. 빈 슬롯의 존재 유무에 따라 두 가지 경우로 나뉜다.

(1) 빈 슬롯이 존재할 경우

$h_1(K)$에 입력할 공간이 남아 있는 것은 두 번째 버킷($h_2(K)$)에 데이터가 존재하지 않는 것을 의미한다. 따라서 $h_1(K)$ 내에서 삽입 정렬 알고리즘을 사용하여 K의 자리를 찾아서 입력하거나 K가 이미 존재한다면 값을 갱신한다.

(2) 빈 슬롯이 없는 경우

슬롯 S_4, S_1, S_2, S_3 순서대로 K의 갱신을 시도한다. $h_1(K)$에 K가 존재하지 않는 경우에는 $h_2(K)$에서 같은 전략으로 삽입 연산을 시도한다. 단, $h_2(K)$에도 빈 슬롯이 없고 K가 존재하지 않은 상황에서는 밀어내기 연산을 수행한다.

3.1.2 밀어내기(kick-out) 전략

K를 임의로 선택된 슬롯과 교체하고 해당 버킷을 재정렬(re-sort)한다. $h_1(K)$ 위치에서 밀려난 데이터는 $h_2(K)$으로, $h_2(K)$ 위치에서 밀려난 데이터는 $h_1(K)$으로 입력을 시도한다. 그 버킷에는 밀려난 데이터가 확실히 없으므로 빈 슬롯이 있는지만 확인하면 된다. 빈 슬롯이 있다면 삽입 연산이 완료되지만, 그렇지 않다면 빈 슬롯을 발견할 때까지 밀어내는 연산을 계속 반복한다. 특정 횟수 이상 밀어내기가 반복되면 입력 연산을 실패한 것이다.

Sorting Cuckoo는 이러한 삽입 전략으로 BCH와 마찬가지로 메모리 점유율이 95% 이상이 될 때가지 데이터의 입력이 가능하다(4장 2절 참조).

3.2 탐색

탐색은 버킷이 정렬($S_1 < S_2 < S_3 < S_4$)되어 있다는 점을 활용하여 S_1, S_4, S_2, S_3 순으로 비교 연산을 진행한다. 삽입 연산과 달리 S_1을 먼저 확인하는 이유는 메모리 점유율이 낮은 경우, S_4가 비어 있을 확률이 높고, 입력 연산의 결과가 Case 1, 2, 3에 해당할 가능성이 크다(4장 2절 참조).

3.2.1 첫 번째 버킷 탐색

(1) Positive 탐색의 경우

Positive 탐색의 경우(그림 2의 Case 1, 4, 5, 8, 9, 11)는 버킷의 범위 조건($S_1 \le K \le S_4$)을 만족하면서 S_1, S_4, S_2, S_3 순서로 슬롯에 접근하였을 때 K를 찾은 경우를 뜻한다. 최악의 경우 BCH와 같은 4번의 메모리 접근으로 탐색을 성공할 수 있다.

(2) Negative 탐색의 경우

Negative 탐색은 첫 번째 버킷에서 K를 찾지 못하고, 두 번째 버킷을 참조하지 않아도 되는 경우에 해당한다.

- S_1이 비어 있을 경우 탐색에 실패한다(그림 2의 Case 2).
- K가 S_1보다 작은 상황은 K가 첫 번째 버킷에 존재할 수 없다는 것을 의미한다. 만약 S_4가 동시에 비었다면 K는 두 번째 버킷에도 입력되지 않았다는 것을 의미하기 때문에 negative 탐색에 해당한다(그림 2의 Case 3).
- K가 S_1보다 크고 S_4가 비어있는 상황에서는 범위 조건을 위반(그림 2의 Case 7)하거나 비어있는 슬롯이 있거나(그림 2의 Case 6, 10) 마지막으로 탐색하는 S_3에서 K를 발견하지 못한 경우(그림 2의 Case 10)가 negative 탐색에 포함된다.

(3) 다음 버킷의 탐색이 필요한 경우

첫 번째 버킷에 데이터가 존재하지 않아 두 번째

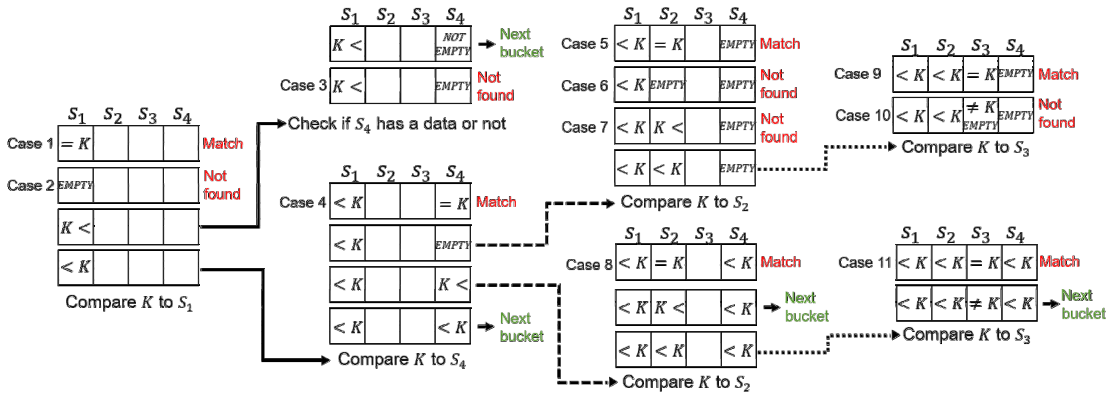

그림 2. 첫 번째 버킷 탐색
Fig. 2. Lookup process at first bucket

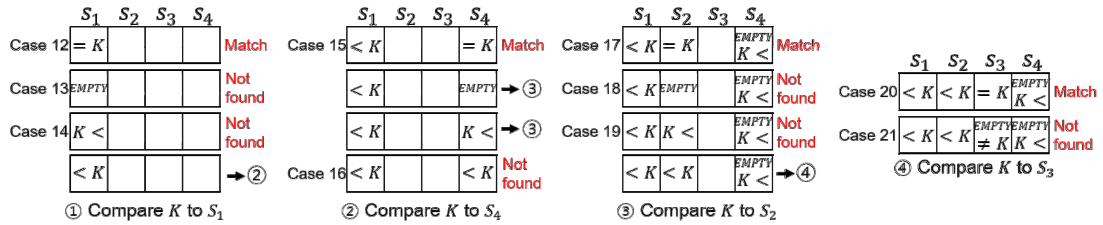

그림 3. 두 번째 버킷 탐색
Fig. 3. Lookup process at second bucket

버킷을 참조해야 하는 경우에 해당한다. 모두 S_4가 비어 있지 않아서 버킷 내 모든 슬롯에 데이터가 존재한다. 따라서 S_1, S_4을 참조하여 버킷의 범위를 빠르게 파악할 수 있다. 메모리 점유율이 높은 상황에서의 두 번째 버킷에 데이터가 존재하는 positive 탐색, 그리고 negative 탐색을 수행할 때 메모리 접근 횟수를 감소시킬 수 있다.

3.2.2 두 번째 버킷 탐색

첫 번째 버킷 탐색과의 차이점은 다음에 탐색할 버킷이 없다는 점이다. 따라서 첫 번째 버킷 탐색에서 다음 버킷을 탐색하는 경우에 해당하는 상황은 두 번째 버킷 탐색에서 모두 negative 탐색에 해당한다. 예를 들면 그림 2에서 첫 번째 버킷에서 K가 S_1보다 작은 경우에 두 번째 버킷에도 K가 존재할 가능성이 있었지만, 두 번째 버킷에서는 S_4을 확인할 필요 없이 negative 탐색에 포함된다(그림 3의 Case 14).

3.3 삽입 및 탐색 알고리즘

삽입, 탐색 연산의 고려사항을 포함하는 Sorting Cuckoo의 알고리즘은 Algorithm 1, 2에 기술하였다.

삽입 알고리즘(Algorithm 1)은 해시 함수(h_1, h_2)를 사용하여 해당 버킷의 위치(H_1, H_2)를 계산한 후, $BucketInsert(T, H, K, V)$ 함수를 호출하여 (K, V)을 입력하거나 값을 갱신한다. $BucketInsert(T, H, K, V)$ 함수는 버킷의 마지막 슬롯에 먼저 접근하여 데이터를 입력할 수 있는지 파악한 후, 마지막 슬롯이 비었다면 삽입 정렬을 이용하여 데이터를 입력하고 ($InsertionSort(T, H, K, V)$), 그렇지 않다면 K을 찾아 갱신하거나 K가 존재하지 않으면 다음 버킷으로 넘어가는 역할을 수행한다.

더 이상 사용할 해시 함수가 없으면 $Kickout(T, H, K, V)$ 함수를 호출하여 밀어내기 연산을 진행한다. 밀어내기는 (K, V)를 임의로 선택한 슬롯(R)과 교체한 후 밀려난 버킷이 아닌 곳을 찾는다. 그 버킷에 빈 슬롯이 있으면 데이터를 입력하고 삽입 연산이 종료되지만, 빈 슬롯이 없으면 밀어내기를 반복한다.

마찬가지로 탐색 알고리즘(Algorithm 2)도 해시 함

Algorithm 1: Insert

```
Input: key(K), value(V), hash table(T)
SET(T, K, V){
  H1←h1(K)
  BucketInsert(T, H1, K, V)
  H2←h2(K)
  BucketInsert(T, H2, K, V)
  Kickout(T, H2, K, V)
}

BucketInsert(T, H, K, V){
  if T[H][last] is empty then
    InsertionSort(T,H,K,V)
  else if T[H][last]= K then
    T[H][i].value = V
  else
    for i in [0, last−1] do
      if T[H][i].key = K
        T[H][i].value = V
        break
}
Kickout(T, H, K, V){
  while maxloop do
    R←rand(0, last)
    swap(T[H][R],KV)
    if H= h1(K)
      then H←h2(K)
    else H←h1(K)

    if T[H][last] is empty
      InsertionSort(T, H, K, V)
      break
  end while
}
```

수(h_1, h_2)를 통하여 해당 버킷의 위치를(H_1, H_2)를 계산한다. 이후 $BucketSearch(T, H, K)$ 함수를 호출하여 버킷 내에 K가 존재하는지 확인한다. K가 존재하면 그에 대응하는 값을 반환하고 존재하지 않으면 $null$을 반환한다. H_1에서 반환 받은 값(V)이 $null$이면 마지막 슬롯을 확인하여 다음 버킷으로 이동해야할지 판단한다. 이동할 필요가 없다면 $null$을 반환한다. 그리고 V가 $null$이 아니면 반환받은 값을 넘겨주고 탐색이 완료된다.

Algorithm 2: Lookup

```
Input: key(K), hash table(T)
Output: value(V) / null
GET(K){
  H1←h1(K)
  V←BucketSearch(H1, K)
  if V= null
    if T[H1][last] is empty
      return V
  else return V

  H2←h2(K)
  V←BucketSearch(T, H2, K)
  return V
}

BucketSearch(T, H, K){
  if T[H][first].key= K
    return T[H][first].value
  else if T[H][first] is (empty || > K)
    return null

  if T[H][last].key= K
    return T[H][last].value
  else if T[H][last]< K
    return null

  for i in [first+1,last−2] do
    if T[H][i].key= K
      return T[H][i].value
    else if T[H][i] is (empty || > K)
      return null

    if T[H][last−1].key= K
      return T[H][last−1].value
    else
      return null
}
```

다음 버킷(H_2)을 확인해야하는 상황이면 마찬가지로 $BucketSearch(T, H, K)$ 함수를 이용하여 K의 존재 유무를 파악한다. 차이점은 다음 버킷이 존재하지 않으므로 $null$을 반환 받으면 무조건 negative

탐색에 해당한다.

$BucketSearch(T, H, K)$ 함수는 버킷의 범위를 파악하기 위해 첫 번째 슬롯, 마지막 슬롯을 먼저 참조하고 두 번째 슬롯부터는 차례로 접근한다. 슬롯에 접근하여 K와 일치하면 그 값을 넘겨준다. 슬롯이 비어있거나, K가 버킷 안에 없다고 판단하면 $null$을 반환한다. 마지막 슬롯은 비어 있어도 앞의 슬롯을 참조해야 하므로 $null$을 반환하지 않는다. 마지막 이전 슬롯은 대소 비교를 시행하지 않고 K가 존재하는지만 확인한다.

Ⅳ. 실 험

4.1 실험 환경

실험 환경은 표 1과 같다.

표 1. 실험 환경
Table 1. Experimental environment

	description
CPU	Intel Core i5-4570 @ 3.2GHz
RAM	16GB
OS	Ubuntu 16.04.4 (Linux kernel 4.4)
Program Language	C (GCC-5.4.0)

4.1.1 실험 변수

실험에서는 버킷 당 슬롯 4개, 해시 함수 2개를 사용하는 버킷화 쿠쿠 해시 테이블((4, 2)-BCHT)을 사용하였다. 사용하는 해시 함수는 Jenkins가 제안한 one-at-a-time 함수[14]이다. 데이터를 입력하는 각 슬롯은 키 32-bit, 값 32-bit, 총 64-bit로 구성되어 있다. 그리고 입력과 탐색 실험 모두 BCHT에 할당한 메모리는 1GB이고, 각 실험마다 100번 반복하여 평균을 기록하였다.

4.1.2 실험용 데이터

모든 실험용 데이터는 32-bit 메르센 트위스터(Mersenne Twister)[15]를 PRNG(Pseudo Random Number Generator)로 사용하여 생성하였다. 삽입 연산의 성능 실험에서는 모두 다른 데이터를 테이블 메모리의 95%만큼 생성하였다. Positive 탐색은 메모리 점유율에 따라 입력한 데이터를, negative 탐색은 테이블에 입력하지 않은 데이터를 각각 천만 번 씩 저장하여 사용하였다. 그리고 같은 데이터를 이용하여 Sorting Cuckoo와 BCH에 적용하였다.

4.2 실험 결과 및 분석

4.2.1 삽입 연산 성능

메모리 점유율에 따른 삽입 연산을 수행하는 동안의 메모리(슬롯) 접근 횟수를 측정하였다. 메모리 점유율이 0%일 때부터 시작하여 10%(약 1300만 회)를 입력할 때마다, 그리고 95%까지 입력하였을 때 사용된 메모리 접근 횟수를 기록하였다.

그림 4는 메모리 점유율에 따른 Sorting Cuckoo와 BCH의 삽입 성능을 비교한 결과를 보여준다. Sorting Cuckoo를 이용한 삽입 연산은 데이터를 입력할 때 삽입 정렬 방식을 이용하고 밀어내기를 수행할 때마다 정렬이 필요하기 때문에 BCH보다 더 많은 메모리를 접근한 것을 알 수 있다. 메모리 점유율이 낮은 경우(0-70%)에는 빈 슬롯을 확인하는 과정과 삽입 정렬을 수행하는 중에 추가적인 메모리 접근이 발생하였고(약 1300만 회), 점유율이 높을 때(70-95%)는 밀어내기 연산이 많아져 버킷을 정렬하는 과정에서 더 많은 메모리 접근이 필요하였다. 그러나 Sorting Cuckoo의 목적은 탐색의 성능을 향상하는 것이고, 삽입 연산을 수행할 때에도 버킷에 데이터가 존재하는지 파악할 때 탐색 연산을 수행한다. 또한, Sorting Cuckoo에서 제안하는 삽입 방식은 기존 BCH와 같이 95% 이상의 메모리 점유율을 달성할 수 있다.

그림 5, 6은 슬롯 접근 순서에 따른 Sorting Cuckoo 기법을 이용하여 positive 및 negative 탐색 연산을 천만 번 씩 수행하였을 때 메모리 접근 횟수의 차이를 보여준다. 입력 연산에서는 빈 슬롯이 존재하는지 파악하는 것이 중요하지만 탐색 연산에서는 S_1, S_4, S_2, S_3 순으로 접근하는 것이 효과적이라는 것을 파악할 수 있다.

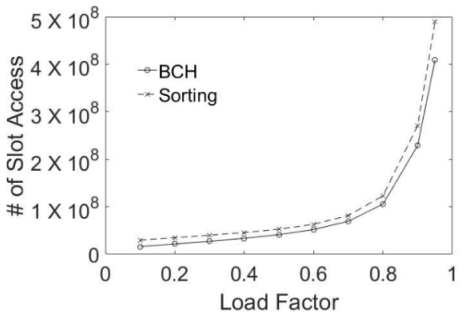

그림 4. 메모리 점유율에 따른 Sorting Cuckoo와 BCH 기법을 이용한 입력 연산 수행 시 슬롯 접근 횟수
Fig. 4. The number of insert slot accesses of Sorting Cuckoo and BCH depending on load factor

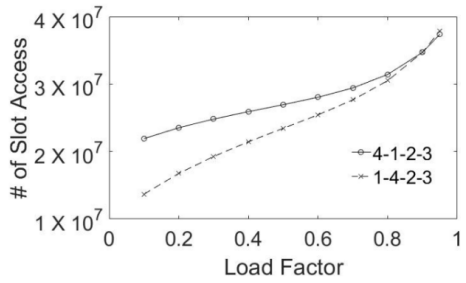

그림 5. 슬롯 접근 순서에 따른 Sorting Cuckoo를 이용한 positive 탐색 수행 시 슬롯 접근 횟수
Fig. 5. The number of positive lookup slot accesses of Sorting Cuckoo depending on order of slot access

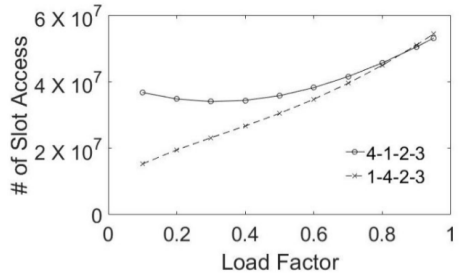

그림 6. 슬롯 접근 순서에 따른 Sorting Cuckoo를 이용한 negative 탐색 수행 시 슬롯 접근 횟수
Fig. 6. The number of negative lookup slot accesses of Sorting Cuckoo depending on order of slot access

4.2.2 탐색 연산 성능

(1) 수치적 분석

가정은 다음과 같다.

- 버킷의 모든 슬롯에 데이터가 입력되어 있다.
- 모든 데이터의 키는 균일한 확률을 가진다. 즉, 해시 함수 2개, 버킷 당 슬롯 4개일 때, 첫 번째 버킷에 있을 확률이 0.5, 두 번째 버킷에 있을 확률이 0.5이고, 각 슬롯에 위치할 확률은 0.25이다.

해당 버킷에 데이터가 존재할 경우에 접근하는 슬롯의 수(N_p)는 2.5로 BCH와 Sorting Cuckoo 모두 동일하다. 각 슬롯 마다 키가 존재할 확률이 같고, 접근하는 순서만 다르므로 다음과 같이 계산하였다.

$$N_p = 0.25 \times (1+2+3+4) = 2.5 \quad (1)$$

해당 버킷에 데이터가 존재하지 않을 경우에는 BCH는 반드시 4번의 슬롯 접근이 필요하다. 하지만 Sorting Cuckoo는 찾고자 하는 키가 버킷의 범위 내에 존재하는지(P_1), 범위보다 작은지(P_2), 범위보다 큰지(P_3)에 따라 나뉜다. 키의 위치에 대한 확률은 다음과 같다.

$$P_1 = P(k \in [n,m]) = \sum_{(n=1)}^{M-3} \sum_{(m=n+3)}^{M} (m-n+1)/M \approx 1/3 \quad (2)$$

$$P_2 = P(k \in [1,n)) = \sum_{(n=1)}^{M-3} \sum_{(m=n+3)}^{M} (n-1)/M \approx 1/3 \quad (3)$$

$$P_3 = P(k \in (m,M]) = \sum_{(n=1)}^{M-3} \sum_{(m=n+3)}^{M} (M-m)/M \approx 1/3 \quad (4)$$

위 식에서 M은 입력할 수 있는 키의 범위, n은 버킷 내 가장 작은 수, m은 버킷 내 가장 큰 수를 의미한다. 따라서 Sorting Cuckoo의 슬롯 접근 횟수(N_n)는 식 (5)와 같다. 찾고자 하는 키가 범위보다 작을 때는 슬롯을 1번 접근하고 범위 보다 클 때는 2번, 범위 내에 존재할 때는 3번 또는 4번인데, 그 확률은 균일하게 0.5라고 가정하였다.

$$N_n = 3.5 \times P_1 + 1 \times P_2 + 2 \times P_3 \approx 2.1667 \quad (5)$$

두 개의 버킷을 고려한 메모리 접근 횟수는 표 2와 같다. 메모리 점유율 100%일 때 positive 탐색은 $0.5 \times (4 - N_n)$회 (약 20%), negative 탐색은 $2 \times (4 - N_n)$회 (약 46%) 더 적게 메모리에 접근하여 탐색 연산을 수행한다.

표 2. positive 탐색과 negative 탐색 시 BCH와 Sorting Cuckoo의 슬롯 접근 횟수
Table 2. The number of slot accesses of BCH and Sorting Cuckoo for positive and negative lookup

	BCH	Sorting Cuckoo
positive lookup	$0.5 \times N_p + 0.5 \times (4 + N_p)$	$0.5 \times N_p + 0.5 \times (N_n + N_p)$
negative lookup	$4+4$	$N_n + N_n$

(2) 모의 실험 분석

Sorting Cuckoo와 BCH가 탐색 연산을 수행할 때 메모리 점유율에 따른 슬롯 접근 횟수의 변화를 측정하였다. 메모리 점유율 10%마다 그리고 95%에서 negative 탐색과 positive 탐색을 천만 번 씩 시도하였다.

• Negative 탐색: 그림 7은 Sorting Cuckoo와 BCH의 negative 탐색의 성능을 측정한 결과이다. 메모리 점유율이 40%가 될 때까지는 비슷한 성능을 보이지만, 점유율이 증가할수록 BCH에서 negative 탐색을 수행할 때 사용하는 메모리 접근 횟수가 Sorting Cuckoo보다 더 많아졌다. 왜냐하면 Sorting Cuckoo는 버킷의 범위를 빠르게 파악하여 K의 존재 여부를 확인하지만 BCH는 새로운 메모리를 접근할 때마다 K에 대한 일치 여부를 판단하는 연산이 발생하기 때문이다. 특히, 메모리 점유율이 95%인 경우에 Sorting Cuckoo의 메모리 접근 횟수는 BCH보다 약 25%(약 1900만 회) 더 적었다.

• Positive 탐색: 그림 8은 Sorting Cuckoo와 BCH 기법을 이용하여 positivie 탐색을 수행할 때의 성능을 나타낸 결과이다. 메모리 점유율이 0-80%까지는 마지막 슬롯을 확인하는 불필요한 연산 때문에 Sorting Cuckoo가 BCH보다 메모리 접근 횟수가 더 많다(메모리 점유율 30%일 때 최대 약 320만 회). 하지만 점유율이 높은 경우(80% 이상)에는 두 번째 버킷에 데이터가 존재할 가능성이 높아지면서, 버킷의 범위를 이용하여 키의 존재 유무를 빠르게 판단할 수 있기 때문에 Sorting Cuckoo의 메모리 접근 횟수가 더 작아진다. 95% 메모리 점유율에서 약 10%(약 400만 회)의 메모리 접근 횟수의 감소를 볼 수 있다.

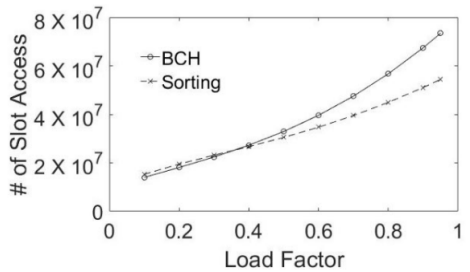

그림 7. 메모리 점유율에 따른 Sorting Cuckoo와 BCH 기법을 이용한 negative 탐색 연산 수행 시 슬롯 접근 횟수
Fig. 7. The number of negative lookup slot accesses of Sorting Cuckoo and BCH depending on load factor

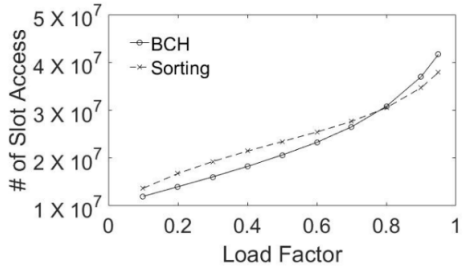

그림 8. 메모리 점유율에 따른 Sorting Cuckoo와 BCH 기법을 이용한 positive 탐색 연산 수행 시 슬롯 접근 횟수
Fig. 8. The number of positive lookup slot accesses of Sorting Cuckoo and BCH depending on load factor

수치적 분석보다 적은 메모리 접근 횟수 감소를 보이는 이유는 95% 메모리 점유율에서도 빈 슬롯을 가진 버킷이 존재할 가능성이 있고, 그 버킷에서의 탐색 연산은 BCH보다 Sorting Cuckoo가 더 많은 메모리를 접근하기 때문이다.

V. 결 론

본 논문에서는 BCH 구조를 기반으로 삽입 정렬 알고리즘을 이용하여 데이터를 입력하는 Sorting Cuckoo를 제안하였다. 버킷과 같은 작은 공간에서 정렬하는 간단한 연산을 통해 더 적은 메모리의 접근으로 키의 존재 유무를 알 수 있다. 그 결과, 기존 BCH의 이점인 메모리를 95% 이상 사용 가능하면서, 높은 메모리 점유율에서 negative 및 positive 탐색 연산의 메모리 접근 횟수를 감소시켰다. 실험 결과에 따르면 Sorting Cuckoo는 메모리 점유율 95%에서 BCH보다 negative 탐색은 25%, positive 탐색은 10%의 더 적은 메모리 접근 횟수를 보였다. 향후 연구에서는 Sorting Cuckoo를 이용하여 가장 많이 사용하고 있는 키-값 저장소인 Redis의 해시 테이블 구조를 변형하여 성능을 비교하는 연구를 진행할 예정이다.

References

[1] Redis, Retrieved Jan., 19, 2017, from https://redis.io/

[2] DB-Engines, Retrieved Jan., 19, 2017, from http://db-engines.com/en/ranking/

[3] Memcached, Retrieved Jan., 19, 2017, from http://memcached.org/

[4] R. Kutzelnigg, "An improved version of cuckoo hashing: Average case analysis of construction cost and search operations," *Math. Comput. Sci.*, vol. 3, no. 1, pp. 47-60, 2010.

[5] B. Fan, D. G. Andersen, and M. Kaminsky,

"MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," *The 10th USENIX Symp. NSDI 13*, pp. 371-384, 2013.

[6] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Eur. Symp. Algorithms*, Springer Berlin Heidelberg, pp. 121-133, Aug. 2001.

[7] U. Erlingsson, M. Manasse, and F. McSherry, "A cool and practical alternative to traditional hash tables," in *Proc. WDAS'06*, Jan. 2006.

[8] E. Lehman and R. Panigrahy, "3.5-way cuckoo hashing for the price of 2-and-a-bit," in *Eur. Symp. Algorithms*, pp. 671-681, Springer Berlin Heidelberg, Sept. 2009.

[9] E. Porat and B. Shalem, "A cuckoo hashing variant with improved memory utilization and insertion time," in *IEEE 2012 Data Compression Conf.*, pp. 347-356, Apr. 2012.

[10] W. Kuszmaul, "Brief announcement: Fast concurrent cuckoo kick-out eviction schemes for high-density tables," in *Proc. 28th ACM SPAA '16*, pp. 363-365, Jul. 2016.

[11] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen, "Horton tables: Fast hash tables for in-memory data-intensive computing," *USENIX ATC 16*, Jun. 2016.

[12] R. Jang, C. Jung, K. Kim, D. Nyang, and K. Lee, "Enhancing RCC(Recyclable counter with confinement) with cuckoo hashing," *J. KICS*, vol. 41, no. 6, pp. 663-671, Jun. 2016.

[13] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proc. ACM 9th Eur. Conf. Comput. Syst.*, p. 27, Apr. 2014.

[14] B. Jenkins, "A new hash function for hash table lookup," *Dr. Dobb's J.*, 1997.

[15] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM TOMACS*, vol. 8, no. 1, pp. 3-30, 1998.

민 대 홍 (Dae-hong Min)

2017년 2월 : 인하대학교 수학과 졸업
2017년 3월~현재 : 인하대학교 컴퓨터공학과 석사과정
<관심분야> 네트워크 보안, 암호이론, 인증프로토콜

장 룡 호 (Rhong-ho Jang)

2013년 8월 : 인하대학교 컴퓨터정보공학과 졸업
2015년 8월 : 인하대학교 컴퓨터정보공학과 석사
2015년 9월~현재 : 인하대학교 컴퓨터공학과 박사과정
<관심분야> 네트워크 보안, 정보보호, 무선 인터넷 보안

양 대 헌 (Dae-hun Nyang)

1994년 2월 : 한국과학기술원 과학기술 대학 전기 및 전자공학과 졸업
1996년 2월 : 연세대학교 컴퓨터과학과 석사
2000년 8월 : 연세대학교 컴퓨터과학과 박사
2000년 9월~2003년 2월 : 한국전자통신연구원 정보보호연구본부 선임연구원
2003년 2월~현재 : 인하학교 컴퓨터정보공학과 교수
<관심분야> 암호이론, 암호프로토콜, 인증프로토콜 무선 인터넷 보안, 네트워크 보안

이 경 희 (Kyung-hee Lee)

1993년 2월 : 연세대학교 컴퓨터과학과 졸업
1998년 8월 : 연세대학교 컴퓨터과학과 석사
2004년 2월 : 연세대학교 컴퓨터과학과 박사
1993년 1월~1996년 5월 : LG 소프트(주) 연구원
2000년 12월~2005년 2월 : 한국전자통신연구원 선임연구원
2005년 3월~현재 : 수원대학교 전기공학과 부교수
<관심분야> 바이오인식, 정보보호, 컴퓨터비전, 인공지능, 패턴인식