KIPS Trans. Comp. and Comm. Sys.
Vol.8, No.6 pp.127~138 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2019.8.6.127

Wall Cuckoo: A Method for Reducing Memory Access Using Hash Function Categorization

Seong-kwang Moon[†] · Dae-hong Min[††] · Rhong-ho Jang[†††] · Chang-hun Jung[†††] · Dae-hun NYang[††††] · Kyung-hee Lee[†††††]

ABSTRACT

The data response speed is a critical issue of cloud services because it directly related to the user experience. As such, the in-memory database is widely adopted in many cloud-based applications for achieving fast data response. However, the current implementation of the in-memory database is mostly based on the linked list-based hash table which cannot guarantee the constant data response time. Thus, cuckoo hashing was introduced as an alternative solution, however, there is a disadvantage that only half of the allocated memory can be used for storing data. Subsequently, bucketized cuckoo hashing (BCH) improved the performance of cuckoo hashing in terms of memory efficiency but still cannot overcome the limitation that the insert overhead. In this paper, we propose a data management solution called Wall Cuckoo which aims to improve not only the insert performance but also lookup performance of BCH. The key idea of Wall Cuckoo is that separates the data among a bucket according to the different hash function be used. By doing so, the searching range among the bucket is narrowed down, thereby the amount of slot accesses required for the data lookup can be reduced. At the same time, the insert performance will be improved because the insert is following up the operation of the lookup. According to analysis, the expected value of slot access required for our Wall Cuckoo is less than that of BCH. We conducted experiments to show that Wall Cuckoo outperforms the BCH and Sorting Cuckoo in terms of the amount of slot access in lookup and insert operations and in different load factor (i.e., 10%-95%).

Keywords : Open Addressing, Hash Table, Cuckoo Hashing, Key-Value Store, In-Memory Cache

월 쿠쿠: 해시 함수 분류를 이용한 메모리 접근 감소 방법

문 성 광[†] · 민 대 홍[††] · 장 룡 호[†††] · 정 창 훈[†††] · 양 대 헌[††††] · 이 경 희[†††††]

요 약

데이터 응답 속도는 사용자 경험과 직결되기 때문에 클라우드 서비스의 중요한 이슈이다. 그렇기 때문에 사용자의 요청에 빠르게 응답하기 위하여 인-메모리 데이터베이스는 클라우드 기반 응용 프로그램에 널리 사용되고 있다. 하지만, 현재 인-메모리 데이터베이스는 대부분 연결 리스트 기반의 해시 테이블로 구현되어 있어 상수 시간의 응답을 보장하지 못한다. 쿠쿠 해싱(cuckoo hashing)이 대안으로 제시되었지만, 할당된 메모리의 반만 사용할 수 있다는 단점이 있었다. 이후 버킷화 쿠쿠 해싱(bucketized cuckoo hashing)이 메모리 효율을 개선하였으나 삽입 연산시의 오버헤드를 여전히 극복하지 못하였다. 본 논문에서는 BCH의 삽입 성능과 탐색 성능을 동시에 향상시키는 데이터 관리 방법인 월 쿠쿠(wall cuckoo)를 제안한다. 월 쿠쿠의 핵심 아이디어는 버킷 내부의 데이터를 사용된 해시 함수에 따라 분리하는 것이다. 이를 통하여 버킷의 탐색 범위가 줄어들어 접근해야 하는 슬롯의 수를 줄일 수 있는데, 이렇게 탐색 연산의 성능이 향상되기 때문에 탐색 과정이 포함되어 있는 삽입 연산 또한 개선된다. 분석에 따르면, 월 쿠쿠에서의 슬롯 접근 횟수 기댓값은 BCH의 기댓값보다 작다. 우리는 월 쿠쿠와 BCH, 정렬 쿠쿠를 비교하는 실험을 진행하였으며, 각 메모리 사용률(10%-95%)에서 월 쿠쿠의 탐색 및 삽입 연산이 다른 기법보다 더 적은 슬롯 접근 횟수를 가지는 것을 보였다.

키워드 : 열린 주소, 해시 테이블, 쿠쿠 해싱, 키-값 저장소, 인-메모리 캐시

1. 서 론

인-메모리 데이터베이스(in-memory DB)는 데이터베이스의 응답 속도를 향상시키기 위해서 상대적으로 속도가 느린 디스크(HDD, SSD)에 데이터를 모두 저장하는 대신 빠른 메모리에 일부를 저장하여 응답하게 하는 기술이다. 여기서 메모리(DRAM)는 디스크 스토리지의 캐시(cache) 역할을 수행하며, 메모리의 데이터는 키-값(key-value) 구조를 가진다.

※ 이 성과는 2017년도 과학기술정보통신부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2017R1A2B4010657).
※ 이 논문은 2018년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No. 2018-0-00391, I/O 분포를 이용한 행위 기반의 랜섬웨어 탐지 기술).
† 준 회 원 : 인하대학교 컴퓨터공학과 석사과정
†† 비 회 원 : 인하대학교 컴퓨터공학과 석·박사 통합과정
††† 준 회 원 : 인하대학교 컴퓨터공학과 박사과정
†††† 정 회 원 : 인하대학교 컴퓨터공학과 교수
††††† 종신회원 : 수원대학교 전기공학과 부교수
Manuscript Received : February 15, 2019
First Revision : April 9, 2019
Accepted : April 17, 2019
* Corresponding Author : Kyung-hee Lee(khlee@suwon.ac.kr)

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/ licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

따라서 인-메모리 데이터베이스는 키-값 저장소(key-value store)로 알려져 있기도 하다. 클라우드 환경에서 데이터 서버의 빠른 응답속도는 좋은 사용자 경험(user experience)과 직결되어 있다. 따라서 구글(Google), 아마존(Amazon), 마이크로소프트(Microsoft) 등 많은 클라우드 서비스 제공자들이 인-메모리 데이터베이스를 웹, 모바일 어플리케이션 등 다양한 분야에 적용했으며 SaaS(software as a service) 형태로 서비스를 제공하기도 한다[1, 2].

대표적인 인-메모리 데이터베이스인 Redis와 Memcached는 연결 리스트(linked list) 기반의 해시 테이블(hash table)을 사용하여 해시 충돌을 처리하고 LRU(least recently used) 방식으로 메모리 캐시 데이터를 관리한다[3-5]. 하지만 연결 리스트 기반의 해시 테이블은 데이터의 삽입(insert)이 빠른 반면 체인의 길이에 따라서 탐색(lookup) 시간이 길어지므로 속도가 느리다. 특히, 테이블에 없는 데이터가 요청되었을 때에는 체인 전체를 탐색해야하기 때문에 매우 비효율적이다. 또한, 체인에 대한 정보를 저장하기 때문에 메모리 효율(memory efficiency)이 낮은 단점이 있다.

R. Pagh 등은 해시 테이블에서의 탐색 성능을 최적화하기 위해서 상수 시간($O(1)$)에 탐색 완료를 보장하는 쿠쿠 해싱(cuckoo hashing)을 제안하였다[6]. 그러나 최초로 소개된 쿠쿠 해싱은 탐색 속도가 빠른 반면 할당받은 메모리를 최대 절반까지만 유효하게 사용할 수 있어 메모리 효율성에 문제가 있었다. 그 뿐만 아니라 데이터를 삽입할 때 추가적인 연산이 필요하였기 때문에 성능이 좋지 않았다. 이후 등장한 버킷화 쿠쿠 해싱(bucketized cuckoo hashing, BCH)[7]은 데이터가 저장되는 단위인 슬롯(slot)을 여러 개로 묶은 버킷(bucket) 형태로 만들어 메모리 사용률 문제를 개선하였다. 하지만, 모든 면에서 좋은 성능을 보이는 것은 아니었다. BCH 또한 쿠쿠 해싱과 마찬가지로 데이터를 삽입할 때 많은 추가 연산이 필요하였으며, 버킷화 하여 저장하기 때문에 쿠쿠 해싱에 비해 탐색이 더 느렸다.

본 논문에서는 BCH를 기반으로 하는 새로운 데이터 저장 및 관리 방식인 월 쿠쿠(wall cuckoo)를 제안한다. 월 쿠쿠는 버킷에 월(wall)이라는 분리 지점의 개념을 추가한다. 월 변수를 기준으로 서로 다른 해시 함수(hash function)를 통해 삽입된 데이터들을 분리하여 관리하며, 이를 이용하여 삽입 및 탐색 시 필요로 하는 메모리 접근 횟수를 줄인다. 그 결과 BCH과 달리 메모리 사용률이 매우 높은 상황(95% 이상)에서도 뛰어난 삽입 및 탐색 성능을 유지할 수 있다.

본 논문에서는 월 쿠쿠의 성능을 검증하기 위하여 랜덤으로 생성된 데이터로 BCH 및 정렬 쿠쿠(sorting cuckoo)[8]의 비교실험을 진행하였다. 그 결과, 메모리 사용률(load factor)이 95%일 때 요청한 데이터가 테이블에 존재할 경우(positive lookup) 월 쿠쿠는 BCH보다 평균 36% 더 적은 메모리 접근 및 연산으로 탐색을 완료하였다. 또한 요청한 데이터가 테이블에 존재하지 않는 경우(negative lookup)에서도 평균 46.5%의 메모리 접근 및 연산 감소율을 보였다. 키(key) 기준으로 버킷내의 데이터를 정렬하는 정렬 쿠쿠와 비교하여도 각각 29.5%(positive lookup)와 20%(negative lookup)만큼의 메모리 접근 및 연산 감소율을 보였다. 월 쿠쿠는 탐색뿐만 아니라 삽입 연산에서도 좋은 성능을 보였다. 특히, 메모리 사용률이 높은 경우(90%-95%) BCH에 비해서 메모리 접근 및 연산이 38% 감소하였으며, 정렬 쿠쿠보다는 약 46.7% 감소한 것으로 나타났다.

2. 배경 지식 및 관련 연구

2.1 배경 지식

1) 초기 형태의 쿠쿠 해싱과 BCH

초기 형태의 쿠쿠 해싱에서는 키와 값의 쌍 (K, V)에 대하여 첫 번째 해시 함수에 의한 $h_1(K)$ 위치의 버킷(이후 첫 번째 버킷)과 두 번째 해시 함수에 의한 $h_2(K)$ 위치의 버킷(이후 두 번째 버킷)에 자료를 하나씩만 저장할 수 있다[6]. 그래서 메모리 사용률이 조금만 높아지더라도 기존 데이터를 다른 버킷으로 밀어내는 연산이 자주 수행되고, 그로 인해 밀어내기가 순환되어 반복되는 현상이 일어난다. 이 현상으로 인하여 초기 형태의 쿠쿠 해싱은 할당된 메모리의 절반 정도만을 유효하게 사용할 수 있다. 이 문제를 해결하기 위해서 밀어내기 연산을 줄일 필요가 있었고, 그 방법으로 제시된 것 중 하나가 한 버킷에 데이터가 들어갈 수 있는 슬롯을 여러 개 존재하도록 하는 방식인 BCH이다. 한 버킷 당 네 개의 슬롯을 가진다고 가정하면, 하나의 (K, V)에 대하여 자료가 저장될 수 있는 장소가 두 슬롯이 아닌 여덟 슬롯이 되기 때문에 밀어내기의 횟수가 줄어들어 유효 메모리 사용률이 향상되었다[9].

2) BCH에서의 삽입과 탐색 연산

기본 형태의 BCH에서는 데이터가 버킷에 입력되는 순서대로 버킷의 빈 슬롯 중 가장 앞에 있는 슬롯부터 저장되며, 첫 번째 버킷에 빈 슬롯이 없으면 두 번째 버킷에 삽입을 시도한다. BCH에서는 두 버킷 모두에 빈 슬롯이 없을 경우, 두 버킷 중 임의로 한 슬롯의 데이터를 선택하여 밀어내고 그 자리에 새로 넣을 데이터를 삽입한다.

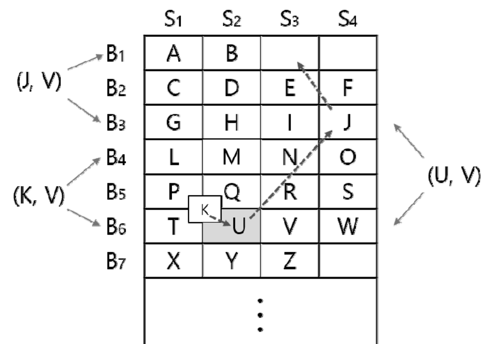


Fig. 1. Insert Process of Bucketized Cuckoo Hashing

Fig. 1은 BCH에서 밀어내기가 일어나는 삽입 과정의 예시를 보인다. B_n은 버킷을 나타내며, S_n은 각 버킷에 할당된 슬롯이다. 새로운 (K, V)를 입력하기 위해서 B_4와 B_6에 접근하지만 빈 슬롯이 존재하지 않아서 기존 데이터 중 하나를 임의로 선택하여 U를 밀어낸다. 밀려난 U는 다른 버킷인 B_3의 데이터 중 하나를 임의로 밀어내고, 그렇게 밀려난 J가 다른 버킷인 B_1에 가서 빈 슬롯에 입력된다. 이와 같이 연쇄적으로 밀어내기 연산이 반복되어 더 이상의 밀어내기 없이 삽입이 성공할 때까지 수행된다.

삽입 과정에서는 해당 버킷에 빈 슬롯이 있는지를 찾는 과정이 반드시 필요하다. 그런데 BCH에서는 어느 슬롯이 비어있을지에 대한 정보를 가지고 있지 않으므로 첫 번째 슬롯부터 차례대로 찾아야만 한다. 입력된 데이터가 적어서 버킷들이 많이 비어있는 상황이라면 적은 슬롯 접근만으로 빈 슬롯을 찾을 수 있지만, 메모리 사용률이 높아질수록 접근해야 하는 슬롯의 수가 많아진다.

탐색 과정 또한 데이터가 존재하는지를 찾기 위해서 버킷에 접근한 후 첫 번째 슬롯부터 마지막 슬롯까지 혹은 비어있는 슬롯을 발견할 때 까지 모두 확인하는 연산이 필요하다. 두 버킷을 모두 탐색해도 데이터를 찾지 못하면 데이터가 존재하지 않는 것이다.

2.2 관련 연구

쿠쿠 해싱은 빠른 탐색 성능을 자랑하는 반면, 삽입할 때의 밀어내기 반복에 의한 루프 현상(insert loop), 낮은 메모리 효율, 테이블에 존재하지 않는 경우의 탐색(negative lookup)시 많은 연산 필요 등 여러 방면에서 문제점을 가지고 있었다.

이 문제들을 극복하기 위하여 Y. Sun 등은 그래프 이론을 이용하여 삽입 시 밀어내기의 루프 현상을 감소시켰고(SmartCuckoo)[10], 다른 연구에서는 각 위치의 밀어내기 발생 횟수를 기록하여 가장 적은 횟수를 가진 위치의 데이터를 밀어내는 방법을 사용하였다(MinCounter)[11]. 이 두 방법은 비대칭적으로 일어나는 데이터 접근과 밀어내기를 방지함으로써 루프의 발생을 억제하였다. 하지만 SmartCuckoo는 삽입 연산은 개선하였으나 탐색 연산에 대해서는 성능 개선이 없었으며, MinCounter는 데이터 유형에 따라 유효 메모리 사용률이 약 70%에서 최대 85%에 그쳐 낮은 메모리 사용률의 한계를 완전히 극복하지 못하였다.

쿠쿠 해싱의 메모리 사용률 문제는 BCH의 등장으로 90% 이상까지 개선되었으며[9] 이후 BCH 구조를 기반으로 한 많은 알고리즘이 제안되었다. Lehman 등은 테이블에서의 버킷을 각각 독립적으로 할당하는 것이 아니라 서로 겹칠 수 있게 할당하여 메모리 사용률을 더 높인 Cuckoo-Overlap[12]를 제시하였다. 그리고 Porat 등은 한 번에 읽어올 수 있는 페이지 단위 안에서 버킷의 크기만 지키고 연속성은 따지지 않아서 더 유연하게 할당하고 겹칠 수 있게 한 Cuckoo-Choose-K[13]을 통하여 메모리 사용률과 삽입 시의 성능을 향상시켰다.

Breslow 등은 추가적인 블룸 필터(bloom filter) 방식으로 BCH에서의 테이블에 존재하지 않는 데이터를 탐색할 때의 성능을 향상시켰으며(Horton table)[14], D. Min 등은 버킷 내부의 데이터를 키를 기준으로 정렬된 상태로 유지하여 버킷 전체를 탐색하지 않고도 탐색을 완료할 수 있는 방법을 제시하였다[8]. 하지만 BCH 기반의 방법들은 탐색 연산에서는 좋은 성능을 보여주었지만, 메모리 사용률이 높을 때 삽입 연산의 큰 성능 저하까지는 해결하지 못하였다.

지금까지 제안된 연구들은 한 가지 면에서 좋은 성능을 보였으나 그것만으로는 충분하지 않다. 인-메모리 데이터베이스가 동작하는 클라우드 서버의 메모리 크기는 비용 문제로 인하여 예측되는 사용량에 비해 과다하게 구축하지 않을 것이므로, 항상 일정 이상의 메모리 사용률인 상태일 것이다. 그렇기 때문에 메모리 사용률이 높은 상황(90% 이상)에서도 삽입과 탐색 성능이 모두 좋은 방법이 필요하다.

3. 고속 삽입 및 탐색 방법: Wall Cuckoo

3.1 해시 함수를 구분하는 월(wall) 변수

BCH는 어떤 데이터를 탐색할 때 그 데이터가 위치할 수 있는 두 개의 버킷을 확인해야 한다. 따라서 데이터가 존재하지 않는 경우(negative lookup) 두 개 버킷의 모든 슬롯에 접근하므로 응답이 지연된다. 본 논문에서는 월 변수를 이용하여 응답 속도를 개선하는 데이터 관리 방법을 소개한다. 월 변수는 버킷마다 할당되며, 버킷에 삽입된 데이터를 사용된 해시 함수($h_1()$ 혹은 $h_2()$)기준으로 분리하고 그 지점을 나타내는 변수이다. 즉, 월보다 앞부분에 위치한 슬롯에는 $h_1(K)$에 의해 삽입된 데이터들이 위치하도록 하고, 월의 뒷부분은 $h_2(K)$에 의해 삽입된 데이터들이 위치하도록 한다. 따라서 월 변수의 값은 해시 함수를 기준으로 나눠진 데이터의 경계점을 의미한다. 이것을 이용하면 데이터를 탐색할 때 한 버킷에서 월 변수의 앞부분 혹은 뒷부분만 확인하기 때문에 BCH에 비해 더 적은 슬롯 접근 및 연산이 요구된다. 이 점은 삽입 연산에서도 성능 향상을 가져오는데, 삽입 연산 전에 데이터의 존재 여부를 확인하는 과정과 삽입할 수 있는 빈 슬롯을 찾는 과정이 포함되기 때문이다. 따라서 데이터 삽입 또는 갱신 연산의 슬롯 접근 횟수도 줄일 수 있다.

월 변수는 버킷의 가장 앞에서 시작하며, 새로운 삽입이 발생할 때마다 데이터의 경계점을 계산하고 월 변수를 갱신한다. BCH에서의 버킷 당 슬롯의 수는 일반적으로 8개를 넘지 않기 때문에, 월 변수는 1byte의 크기만으로도 충분히 그 위치를 나타낼 수 있다. 그러므로 적은 추가 메모리를 할당하는 것으로도 탐색과 갱신 또는 삽입의 성능 향상 효과를 얻는다.

3.2 탐색 연산

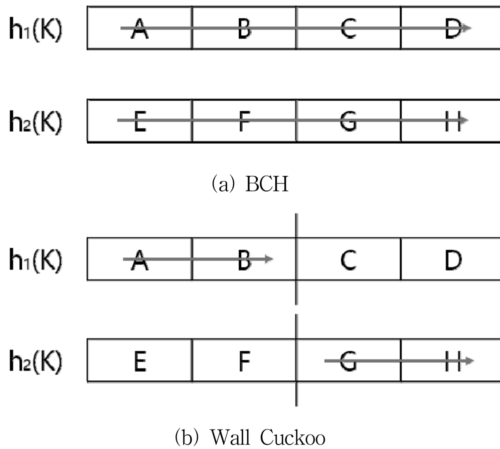
Algorithm 1은 해시 함수(h_1, h_2)를 사용하여 해당 데이터가 어느 버킷에 저장되어있을지 버킷 위치(H_1, H_2)를 구하여 탐색 연산을 시행하는 과정을 나타낸다.

Algorithm 1: Lookup(key)

```
GET(K){
  H1 ← h1(K)
  V ← BucketSearch(H1, K, 0, Wall − 1)
  if V == inconclusive then
    H2 ← h2(K)
    V ← BucketSearch(H2, K, Wall, last)
      return V
  else
    return V
}

BucketSearch(H, K, start, end){
  for i in [start, end] do
    if T[H][i].key == K then
      return T[H][i].value
    else if T[H][i] is empty then
      return null

  if H == H1 then
    return inconclusive
  else
    return null
}
```



(a) BCH

(b) Wall Cuckoo

Fig. 2. Comparing the Search Range of (a) BCH and (b) Wall Cuckoo

Fig. 2(a)는 BCH의 탐색 과정을, Fig. 2(b)는 월 쿠쿠의 탐색 과정의 한 예를 나타낸다. A, B, E, F는 첫 번째 해시에 의하여 버킷에 입력되었고, C, D, G, H는 두 번째 해시에 의하여 버킷에 입력되었다고 가정한다. $h_1(K)$과 $h_2(K)$에 의한 버킷을 모두 참조해야 하는 탐색일 경우에 BCH는 Fig. 2(a)와 같이 모든 슬롯에 접근해야 한다. 반면, 월 쿠쿠는 버킷 내부의 원소가 월로 구분되어있기 때문에 Fig. 2(b)와 같이 월을 기준으로 탐색 범위를 좁힘으로써 슬롯 접근 횟수를 줄인다. 첫 번째 버킷에는 $h_1(K)$로 접근하므로 데이터가 존재한다면 월의 앞부분에서 찾을 수 있다. 그러므로 Algorithm 1의 $BucketSearch(H_1, K, 0, Wall-1)$와 같이 첫 번째 버킷에서는 처음부터 월의 앞부분까지만 탐색한 후, 일치하는 슬롯이 없을 경우 두 번째 버킷을 찾는다. 두 번째 버킷에는 $h_2(K)$로 접근하므로, 데이터가 존재한다면 월의 뒷부분에 저장되어 있다. 따라서 두 번째 버킷에서는 $BucketSearch(H_2, K, Wall, last)$를 사용하여 월의 뒤에서부터 찾으며, 버킷의 끝까지 탐색하였을 때에도 일치하는 슬롯이 없거나 비어있는 슬롯이 먼저 발견되는 경우 찾는 데이터가 존재하지 않는다는 것을 알 수 있다.

3.3 삽입 연산

같은 키를 가지는 데이터가 중복해서 삽입되는 것을 막기 위하여 데이터를 삽입하기 전에 해당 데이터가 이미 테이블에 존재하는지를 찾는 탐색을 먼저 수행해야 한다. 탐색 결과가 존재한다면 값을 갱신하고, 존재하지 않는다면 해당 버킷들에 삽입할 빈 슬롯이 있는지 찾는다. 월 쿠쿠에서는 빈 슬롯이 항상 월의 뒷부분에만 존재할 수 있기 때문에 빈 슬롯을 찾는 연산을 처음부터가 아닌 월의 뒤부터 찾도록 개선하였다. 이로 인한 빈 슬롯 찾기에서의 슬롯 접근 횟수 이득이 크기 때문에 월 쿠쿠는 삽입 연산에서도 성능이 향상되었다.

Algorithm 2: Insert(key, value)

```
SET(K, V){
  if WallLookup(K) == null then
    H1 ← h1(K)
    BucketInsert(H1, K, V)
    H2 ← h2(K)
    BucketInsert(H2, K, V)
    Kickout(H1, H2, K, V)
  else
    T[H][found].value = V
}

BucketInsert(H, K, V){
  for i in [wall, last] do
    if T[H][i].key == empty then
      insert(H, K, V)
      break
}
```

Algorithm 2는 해시 함수(h_1, h_2)에 의한 버킷 위치(H_1, H_2)에 따라 어떻게 삽입 연산을 수행하는지를 나타낸다. 앞에서 언급한 대로 $BucketInsert(H, K, V)$ 안에서 빈 슬롯을 찾

는 연산이 월의 뒤부터 찾도록 개선되어 있다.

첫 번째 버킷에서 $BucketInsert(H_1, K, V)$를 시도한 후 데이터를 삽입할 빈 슬롯을 찾지 못하면 두 번째 버킷에서 $BucketInsert(H_2, K, V)$를 시도한다. 두 시도가 모두 실패하면 $Kickout(H_1, H_2, K, V)$이 일어나서 데이터의 입력이 완료되거나 정해진 최대 반복 횟수가 될 때까지 삽입 시도를 반복하며, 입력하지 못하였을 경우 삽입은 실패한다.

빈 슬롯에 자료를 삽입하는 $insert(H, K, V)$와 두 버킷 모두에 빈 슬롯이 없을 때 일어나는 $Kickout(H_1, H_2, K, V)$ 과정은 아래에서 자세히 설명한다.

1) 빈 슬롯이 존재할 경우

Fig. 3은 삽입 연산을 수행할 때 접근한 버킷에 빈 슬롯이 존재할 경우의 예시를 보인다. 새로 삽입되는 데이터가 $h_1(K)$에 의한 데이터이면 NEWA, $h_2(K)$에 의한 데이터이면 NEWB라고 표기하였다. $h_1(K)$에 의하여 첫 번째 버킷에 NEWA를 삽입할 때는 월의 앞부분에 넣어야 하는데, $h_2(K)$에 의한 다른 데이터가 이미 입력되어있는지에 따라 Fig. 3(a)와 Fig. 3(b)로 나뉜다.

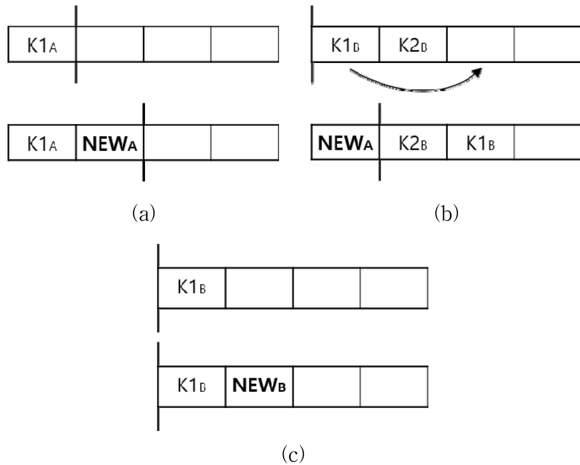


Fig. 3. Example of the Insertion Process when There Exist an Empty Slot in the Bucket

Fig. 3(a)의 경우처럼 해당 버킷이 비어있거나 $h_1(K)$ 값으로만 이루어져 있다면, 비어있는 슬롯 중 가장 앞에 위치한 슬롯에 NEWA를 삽입한 후 월의 위치를 뒤로 옮긴다. Fig. 3(b)의 경우와 같이 해당 버킷에 $h_2(K)$에 의한 다른 데이터가 이미 입력되어 있다면, 월 바로 뒤에 위치한 K1B를 비어있는 슬롯 중 가장 앞에 위치한 슬롯으로 옮긴 후 그 자리에 NEWA를 삽입하고 월의 위치를 뒤로 옮긴다. Fig. 3(c)의 경우 $h_2(K)$에 의하여 두 번째 버킷에 NEWB를 삽입하는데, $h_2(K)$에 의한 데이터를 월의 뒷부분에 넣는 것이므로 월의 변화가 일어나지 않는다. 버킷의 비어있는 슬롯 중 가장 앞에 위치한 슬롯에 NEWB를 삽입한다.

2) 빈 슬롯이 존재하지 않을 경우

$h_1(K)$과 $h_2(K)$ 위치의 두 버킷에 데이터의 입력이 가능한 빈 슬롯이 존재하지 않으면 새 데이터를 삽입하기 위하여 기존 데이터의 밀어내기 연산을 수행한다. 새로 입력되는 데이터는 두 버킷 중 한 곳에 위치할 수 있으므로 여덟 개의 슬롯 중 임의의 한 슬롯을 밀어내기 대상으로 선택한다. 밀려난 대상은 다른 버킷으로 이동해야만 하므로 $h_1(K)$에 의한 위치의 버킷이었다면 $h_2(K)$에 의한 위치의 버킷으로 밀려나고, $h_2(K)$에 의한 위치의 버킷이었다면 $h_1(K)$에 의한 위치의 버킷으로 밀려난다.

Fig. 4에서는 접근한 버킷에서 슬롯의 데이터를 밀어내야 하는 경우에 따른 월의 변화와 슬롯의 변화를 보인다. 새로 삽입되는 데이터가 $h_1(K)$에 의한 데이터이면 NEWA, $h_2(K)$에 의한 데이터이면 NEWB라고 표기하였으며, 음영으로 표시된 슬롯은 기존 데이터 중 밀려날 대상으로 선택된 슬롯을 나타낸다.

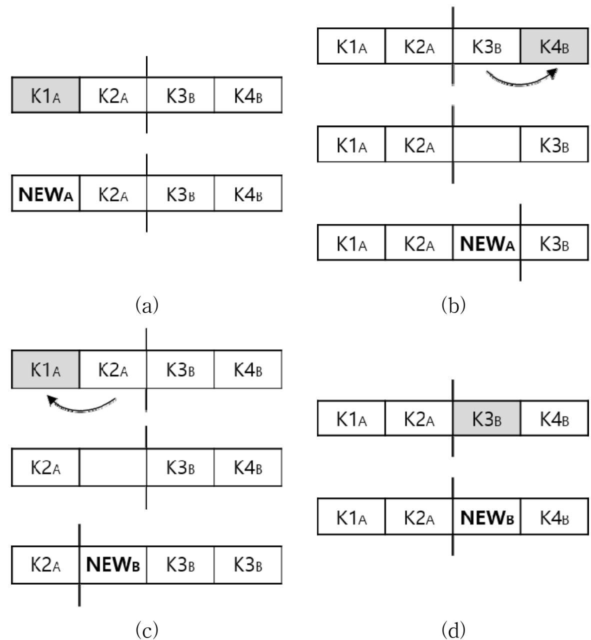


Fig. 4. Example of the Insertion Process when Kick-out Process is Needed

a) 삽입할 데이터가 월의 앞부분에 위치해야 할 경우

Fig. 4(a)와 Fig. 4(b)는 삽입할 데이터 NEWA가 월의 앞부분에 위치하는 경우의 예시이다. Fig. 4(a)의 경우, 밀어낼 대상이 월의 앞부분에 있던 데이터인 K1A이기 때문에 월의 변화가 없으므로 추가 조치 없이 밀어내고 NEWA를 삽입한다. Fig. 4(b)의 경우, 밀어낼 대상이 월의 뒷부분에 있던 데이터인 K4B이기 때문에 월 바로 뒤의 데이터인 K3B를 밀어낼 대상으로 선택된 K4B 슬롯에 옮긴 후, NEWA를 월 바로 뒤에 삽입하고 월을 한 칸 뒤로 조정한다.

b) 삽입할 데이터가 월의 뒷부분에 위치해야 할 경우

Fig. 4(c)와 Fig. 4(d)는 삽입할 데이터 NEWB가 월의 뒷부분에 위치하는 경우의 예시이다. Fig. 4(c)의 경우, 밀어낼 대상이 월의 앞부분에 있던 데이터인 K1A이기 때문에 월 바로 앞의 데이터인 K2A를 밀어낼 대상으로 선택된 K1A 슬롯에 옮긴 후, NEWB를 월 바로 앞에 삽입하고 월을 한 칸 앞으로 조정한다. Fig. 4(d)의 경우, 밀어낼 대상이 월의 뒷부분에 있던 데이터인 K3B이기 때문에 월의 변화가 없으므로 추가 조치 없이 밀어내고 NEWB를 삽입한다.

밀려난 데이터의 다른 버킷에 빈 슬롯이 존재하면 삽입이 완료되지만, 빈 슬롯이 없으면 다시 해당 버킷에서 밀어낼 대상을 임의로 정하여 밀어내는 과정이 반복된다. 이 과정을 더 이상 밀려나는 데이터가 없이 빈 슬롯에 입력이 성공할 때까지 혹은 정해진 최대 수만큼 반복하며, 빈 슬롯을 찾지 못하면 삽입은 실패한다.

4. 실험 및 분석

이 장에서는 이론 분석과 실험 분석을 통해 본 논문에서 제안하는 월 쿠쿠의 슬롯 접근 횟수의 감소 효과를 입증한다. 이론 분석 부분에서는 삽입 및 탐색연산의 슬롯 접근 횟수 기대치를 분석함으로써 월 쿠쿠의 평균 슬롯 접근 횟수가 BCH보다 작다는 것을 분석한다. 그리고 실험 부분에서는 메모리 사용률이 높은(90% 이상) 환경에서도 월 쿠쿠가 BCH 또는 정렬 쿠쿠(sorting cuckoo)보다 더 좋은 성능을 가지고 있다는 것을 보인다.

4.1 실험 방법 및 파라미터

Table 1은 실험을 진행한 환경을 보여주고 있다. 모든 실험에서는 버킷 당 4개의 슬롯을 할당하였으며, 각 슬롯은 32bit의 키와 32bit의 값으로 구성된다. 키와 값을 저장하기 위해 1GB의 메모리를 할당하였으며, 해시 함수는 2개의 Jenkins 해시[15]를 사용하였다. 실험에 사용한 데이터셋은 메르센 트위스터(Mersenne Twister)[16]을 이용하여 임의로 생성하였다.

Table 1. Experimental Environment

Item	Description
CPU	Intel Core i5-6600 @ 3.3GHz
RAM	16GB
OS	Windows 10
Language	C (Visual Studio 2015 14.0.25431.01)

Table 2는 실험에서 사용된 수치들을 보여준다. 키와 값의 크기가 합하여 23byte의 크기이므로, 1GB를 할당하기 위한 총 슬롯의 수는 227개이다. 모든 실험은 메모리 사용률이 10%부터 95%까지인 상황에서 진행하였으며, 탐색 연산 실험에서는 데이터가 테이블에 있는 경우(positive lookup)와 데이터가 없는 경우(negative lookup)를 나누어서 실험을 진행하였다. 월 쿠쿠의 우수성을 증명하기 위해 BCH 및 정렬 쿠쿠와 비교실험을 진행하였으며, 실험으로 얻은 결과는 100회씩 수행한 결과의 평균값이다.

Table 2. Experiment Workload

	Value
# of bucket	33,554,432 (2^{25})
# of insert	13,421,772 (per 10%, ~90%; 2^{27}×0.1) 6,710,886 (90~95%; 2^{27}×0.05)
# of lookup	each 10,000,000 (per 10%, ~90%) each 10,000,000 (95%)

4.2 이론적 분석

1) 탐색 연산

우선 해시 테이블의 메모리 사용률은 100%이며, 임의의 값들이 균일하여 전체 버킷에 $h_1(K)$에 의해 입력되어있는 값과 $h_2(K)$에 의해 입력되어있는 값의 비율이 1:1을 이루고 있다고 가정한다. 그러면 밀어내기 연산에 의해 임의의 위치에서 데이터가 밀려날 때 월의 위치가 조정되므로, 전체 버킷에서의 월의 위치 분포는 정규 분포를 따르게 된다. 월 변수의 값이 양 끝인 0과 4일 때의 확률을 각 α라 하고, 1과 3일 때를 각 β라 하며, 중간에 있는 2일 때를 γ라 하자. 그러면 버킷의 처음부터 월 앞까지 탐색하는 슬롯 접근 횟수의 기댓값(N)은 아래와 같다.

$$\begin{aligned} N &= (0\times\alpha)+(1\times\beta)+(2\times\gamma)+(3\times\beta)+(4\times\alpha) \\ &= 4\alpha+4\beta+2\gamma = 2(2\alpha+2\beta+\gamma) = 2 \end{aligned} \quad (1)$$

그러므로 각 버킷에서 처음부터 월 앞까지 탐색하는 평균 슬롯 접근 횟수의 기댓값은 2이며, 월 뒤에서 끝까지 탐색하는 기댓값도 2가 된다.

a) 데이터가 테이블에 존재하지 않는 경우(negative lookup)

BCH에서는 요청한 데이터가 테이블에 존재하지 않는 경우 버킷 내의 모든 데이터에 대해 비교연산을 한다. 따라서 한 버킷에서의 슬롯 접근 횟수의 기댓값(BCH_{n1})은 버킷 내부의 슬롯 수 만큼인 4이며, 두 버킷을 모두 확인할 때의 기댓값(BCH_{n2})은 8이다. 반면에 월 쿠쿠에서는 월을 기준으로 서로 다른 해시함수를 사용하여 삽입된 데이터들이 분리되어 있으므로 월을 기준으로 앞부분 혹은 뒷부분만 확인한다. Equation (1)에 의하여 각 버킷 당 슬롯 접근 횟수의 기댓값($Wall_{n1}$)은 2씩이 되므로 두 버킷을 모두 확인할 때의 기댓값($Wall_{n2}$)은 평균 4이다. 그러므로 메모리 사용률이 100%일 때 월 쿠쿠의 negative 탐색 시 슬롯 접근 횟수는 BCH 대비 약 50%만큼 감소한다.

b) 데이터가 테이블에 존재하는 경우(positive lookup)

데이터가 테이블에 존재하는 경우 BCH에서는 한 버킷에서 슬롯 당 데이터의 존재 확률이 25%이다. 하지만 월 쿠쿠에서는 Equation (1)에 의하여 평균적으로 버킷 당 두 번의 슬롯 접근만을 하게 되고, 그러므로 한 버킷에서 접근하는 슬롯 당 데이터 존재 확률은 50%이다. 따라서 BCH에서의 positive 탐색일 때 한 버킷에서의 슬롯 접근 수 기댓값 (BCH_{p1})와 월 쿠쿠에서의 positive 탐색 일 때 한 버킷에서의 슬롯 접근 수 기댓값($Wall_{p1}$)은 아래와 같다.

$$BCH_{p1} = 0.25 \times (1+2+3+4) = 2.5 \tag{2}$$

$$Wall_{p1} = 0.5 \times (1+2) = 1.5 \tag{3}$$

두 해시 함수를 갖는 BCH에서는 탐색 대상의 데이터가 두 버킷 중 한 곳에 저장되어 있다. 그러므로 첫 번째 버킷에서 찾는 것을 성공하였을 때와 첫 번째 버킷에서 찾지 못해 두 번째 버킷까지 찾게 되었을 때를 고려하여 기댓값을 계산한다. 실험의 가정에 의해 두 버킷에 데이터가 존재할 확률이 같을 때, 두 버킷의 접근을 모두 고려한 BCH에서의 positive 탐색 시 슬롯 접근 수 기댓값(BCH_{p2})과 월 쿠쿠에서의 positive 탐색 시 슬롯 접근 수 기댓값($Wall_{p2}$)은 아래와 같다.

$$\begin{aligned} BCH_{p2} &= 0.5 \times BCH_{p1} + 0.5 \times (BCH_{n1} + BCH_{p1}) \\ &= 0.5 \times 2.5 + 0.5 \times (4 + 2.5) = 4.5 \end{aligned} \tag{4}$$

$$\begin{aligned} Wall_{p2} &= 0.5 \times Wall_{p1} + 0.5 \times (Wall_{n1} + Wall_{p1}) \\ &= 0.5 \times 1.5 + 0.5 \times (2 + 1.5) = 2.5 \end{aligned} \tag{5}$$

즉, 메모리 사용률이 100%일 때 월 쿠쿠의 positive 탐색은 BCH 대비 약 44%의 슬롯 접근 횟수 감소를 기대할 수 있다.

2) 삽입 연산

삽입 연산에서는 입력하고자 하는 데이터가 존재하는지를 확인하는 탐색 연산 과정, 데이터가 존재하지 않았을 때 빈 슬롯을 찾는 과정, 빈 슬롯이 없을 경우 밀어내는 과정이 차례로 수행된다. 월 쿠쿠는 밀어낼 대상을 선정하는 방법이 BCH와 동일하기 때문에 같은 메모리 사용률에서의 삽입 연산에서 버킷의 접근 수에는 서로 차이가 없다고 가정한다. 그러므로 각 버킷에서의 밀어내기 연산 과정과 빈 슬롯을 찾는 과정을 중점으로 분석한다.

a) 메모리 사용률이 높지 않을 때

여기에서는 메모리 사용률이 높지 않은 상황 중 50% 수준일 때를 가정하여 분석한다. 즉, 평균적으로 각 버킷에는 절반인 두 슬롯에만 데이터가 저장되어 있다. 이 경우 빈 슬롯이 충분하여 밀어내기 연산은 거의 일어나지 않는다. 따라서 삽입 연산에 사용된 슬롯 접근의 수는 탐색 연산 1회와 빈 슬롯 찾기 연산 1회를 수행하는 데 필요한 슬롯 접근의 수이다.

새로운 데이터를 삽입하는 경우, 우선 탐색 연산 과정에서 negative 탐색이 수행된다. BCH는 첫 버킷부터 차례대로 접근하여 검사하므로 빈 슬롯을 만나서 찾는 데이터가 존재하지 않음을 확인하는데 평균 3회의 슬롯 접근 횟수를 갖는다. 월 쿠쿠의 경우, 메모리 사용률이 높지 않다는 것은 빈 슬롯이 많이 남아있기 때문에 두 번째 버킷까지 가지 않고 첫 번째 버킷에 입력되어있는 데이터가 대부분인 상황이다. 그러므로 월에 의한 슬롯 접근 횟수 감소 효과가 나타나지 않아서 BCH와 마찬가지로 평균 3회의 접근 횟수를 갖는다.

탐색 연산 후 수행되는 빈 슬롯 찾기 연산을 보면 3.3절에 언급하였듯이 메모리 사용률이 높지 않은 경우에는 빈 슬롯이 존재하는 버킷이 많아서 처음 접근한 버킷에 Fig. 3(a)와 같이 입력될 확률이 높다. 빈 슬롯을 찾기 위해서 BCH는 처음부터 차례대로 찾으므로 평균 3회의 슬롯 접근이 필요하지만, 월 쿠쿠는 월의 바로 뒤에서 빈 슬롯을 찾을 확률이 높으므로 이 경우 1회의 슬롯 접근만을 필요로 한다.

그러므로 메모리 사용률이 약 50%일 때 BCH는 빈 슬롯을 찾기까지 평균 약 6회의 슬롯 접근이 필요하지만, 월 쿠쿠는 슬롯 접근 횟수가 평균 약 4회가 되어 BCH에 비해 약 33.3%의 슬롯 접근 감소 효과를 기대할 수 있다.

b) 메모리 사용률이 높을 때

메모리 사용률이 높은 경우에는 밀어내기 연산이 자주 일어나며, 이 때 여러 버킷에 접근하며 삽입을 시도한다. 반면 데이터의 존재 여부를 찾기 위해 처음에 이루어지는 탐색 연산은 한 번만 수행하므로, 전체 연산에 미치는 영향이 적다. 그렇기 때문에 메모리 사용률이 높을 때는 각 버킷에서의 밀어내기 연산과 빈 슬롯 찾기 연산만을 고려하여 비교한다.

밀어내기 연산이 일어나는 경우는 3.3절의 Fig. 4에 해당한다. Fig. 4에서 (a)-(d)의 각 상황이 일어날 확률이 같다고 가정하므로, 50%의 확률로 데이터 이동으로 인한 추가 슬롯 접근이 한 번 발생한다. 즉, 월 쿠쿠는 BCH에 비해 삽입 연산 시 버킷 접근 한 번 당 0.5회만큼의 추가 슬롯 접근이 요구된다. 하지만 월은 평균적으로 버킷의 중간에 존재하기 때문에, 월 쿠쿠는 빈 슬롯은 언제나 월 뒤에 존재한다는 특성을 이용할 수 있다. 그리하여 월 쿠쿠의 해당 버킷에 빈 슬롯이 있는지 찾기 위한 슬롯 접근은 BCH에 비해 절반 수준이다. 이것을 고려하면 높은 메모리 사용률에서의 월 쿠쿠와 BCH의 버킷 당 슬롯 접근 횟수(Slot Accesses Per Bucket; SAPB)는 아래와 같은 연관성을 가진다.

$$SAPB_{Wall} = \frac{SAPB_{BCH}}{2} + 0.5 \tag{6}$$

메모리 사용률이 높은 상황에서 BCH는 버킷의 모든 슬롯을 접근해야 하므로 빈 슬롯을 찾기까지 접근하는 한 버킷 당 4회의 슬롯 접근이 필요하지만, 위의 식에 의하면 월 쿠쿠는 2.5회의 슬롯 접근 기댓값을 가진다. 그러므로 월 쿠쿠는 BCH에 비해 약 37.5%의 슬롯 접근 횟수의 감소 효과를 기대할 수 있다.

4.3 실험 분석

1) 탐색 연산 슬롯 접근 횟수 실험

BCH와 정렬 쿠쿠, 그리고 월 쿠쿠를 비교하였다. 메모리 사용률이 10%일 때부터 10% 간격으로 positive 탐색과 negative 탐색을 각각 1000만 회씩 시도하여 슬롯 접근 횟수를 측정하였고, 90%일 때까지 측정한 후에 추가로 95%일 때 측정하여 기록하였다. Fig. 5와 Fig. 6에 그 결과를 그래프로 나타냈다. 두 그래프에서 X축은 메모리 사용률을 나타내고, Y축은 슬롯 접근 횟수를 백만 단위로 나타낸다.

a) 데이터가 테이블에 존재하지 않는 경우(negative lookup)

Fig. 5는 메모리 사용률에 따른 세 기법의 negative 탐색 성능 비교 결과를 보여준다. Negative 탐색은 버킷에 일치하는 데이터를 찾을 수 없거나 빈 슬롯을 발견하는 경우인데, positive 탐색과 마찬가지로 에 의한 값들이 채워지기 시작하는 약 50%시점 이후에 탐색해야하는 슬롯의 범위가 줄어드는 것이 효과를 보이기 시작하여 메모리 사용률이 높을수록 다른 알고리즘들과 큰 격차를 보인다. 메모리 사용률 95%에서 월 쿠쿠의 슬롯 접근 횟수는 BCH에 비해서는 약 46.5% (3400만 회) 적었으며, 높은 메모리 사용률에서 BCH에 비해 적은 슬롯 접근을 보이는 정렬 쿠쿠보다도 약 20%(1500만 회) 더 적은 슬롯 접근 횟수를 갖는다. 메모리 사용률이 100%일 때 탐색 당 슬롯 접근 기댓값이 4회인데, 95% 사용률에서 1000만 회 탐색에 약 3930만 회의 접근을 보이므로 이론적 분석과 비슷한 값이 나타난다.

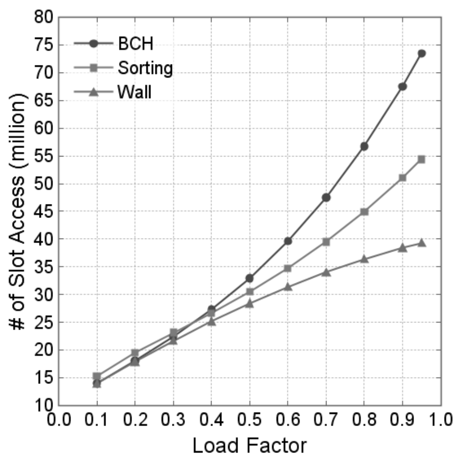


Fig. 5. Comparison of the Number of Slot Accesses in Negative Lookup Process

b) 데이터가 테이블에 존재하는 경우(positive lookup)

Fig. 6은 메모리 사용률에 따른 세 기법의 positive 탐색 성능 비교 결과를 보여준다. 40%정도까지는 비어있는 슬롯이 있는 버킷이 많아서 $h_1(K)$에 의한 데이터 삽입이 주로 일어나기 때문에 월의 앞부분만 찾는 월 쿠쿠의 탐색 방식이 큰 우위를 보이지 않는다. 하지만 $h_2(K)$에 의한 값들이 채워지기 시작하는 약 50%시점 이후로는 탐색해야하는 슬롯의 범위가 줄어드는 것이 효과를 보이기 시작하여 메모리의 사용률이 높을수록 BCH와 큰 격차를 보인다. 메모리 사용률 95%에서 월 쿠쿠의 슬롯 접근 횟수는 BCH에 비해서는 약 36%(1500만 회) 적었으며, 정렬 쿠쿠보다도 약 29.5%(1100만 회) 더 적은 횟수를 갖는다. 앞선 이론적 분석에 따르면 메모리 사용률이 100%일 때 탐색 당 슬롯 접근의 기댓값이 2.5회인데, 95% 사용률에서 1000만 회 탐색에 약 2670만 회의 접근을 보이므로 이론적 분석과 비슷한 값이 나타남을 확인할 수 있다.

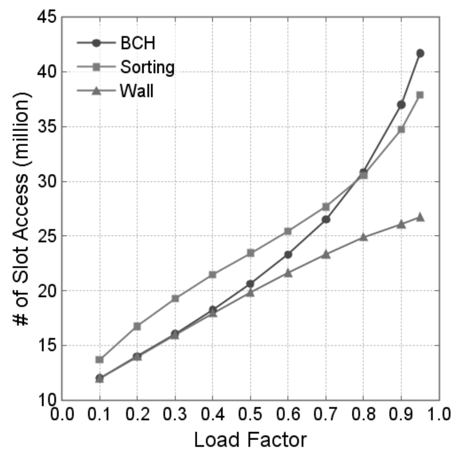


Fig. 6. Comparison of the Number of Slot Accesses in Positive Lookup Process

2) 삽입 연산 슬롯 접근 횟수 실험

BCH와 정렬 쿠쿠, 그리고 빈 슬롯을 버킷의 처음부터 찾는 월 쿠쿠(Fig. 7의 from first)와 월의 뒤부터 빈 슬롯을 찾도록 개선한 월 쿠쿠(Fig. 7의 from Wall)를 비교하였다. 메모리 사용률이 0%일 때부터 10%만큼씩(약 1340만 회) 추가로 삽입하는 데 필요한 슬롯 접근 횟수를 측정하였으며, 90%일 때까지 측정한 후에 추가로 95%일 때까지 삽입한 슬롯 접근 수를 측정하여 기록하였다. Fig. 7의 그래프에서 X축은 메모리 사용률을 나타내고, Y축은 슬롯 접근 횟수를 백만 단위로 나타낸다.

Fig. 7을 보면 월 쿠쿠에서 삽입 시 빈칸을 찾는 과정을 버킷의 처음부터 찾는 방식으로 할 경우 BCH와 크게 다르지 않은 성능을 보인다. 90% 이하에서는 데이터의 중복 방지를 위한 탐색 연산에서의 성능 향상이 존재하지만, 95%에서는 밀어내기 연산에서의 월 이동에 따른 추가 슬롯 접근에 의해 성능이 악화된다. 하지만 빈 슬롯은 항상 월의 뒷부분에 존재한다는 특징을 기반으로 하여 빈 슬롯 찾기를 월의 뒤부터 찾도록 개선하면 삽입 연산 시 슬롯 접근이 모든 구간에서 향상됨을 알 수 있다. 메모리 사용률이 50%일 때 삽입 연산 수행 시 월 쿠쿠의 슬롯 접근 횟수가 BCH에 비해서 약 37%(3000만 회)가량 적음을 알 수 있는데, 이론적 분석에서의 33.3% 감소 기대와 비슷한 결과가 나타난다. 그리고 메모리 사용률이 높은 90%에서 95%까지 입력한 횟수를 비교하면, 삽입 연산에서 BCH보다 많은 연산량을 갖는 정렬 쿠쿠보다 약

46.7%(2억 4500만 회) 더 적은 슬롯 접근 횟수를 가진다. BCH에 비해서는 38%(1억 7600만 회) 적은데, 이론적 분석에서 도출된 37.5% 감소 예상과 거의 같은 결과를 보인다.

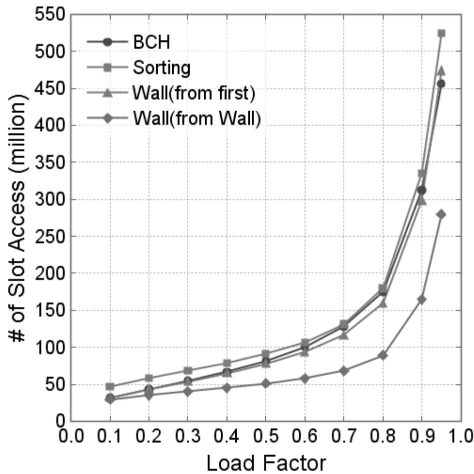


Fig. 7. Comparison of the Number of Slot Accesses in Insert Process

3) 수행 시간 실험

슬롯 접근 비교와 같은 조건으로 실험하였다. 메모리 사용률이 0%일 때부터 10% 간격으로 데이터를 삽입하였으며, 각 10%마다 1000만 회씩 탐색을 시도하며 시간을 측정하였고 90%일 때까지 측정한 후에 추가로 95%일 때 측정하여 기록하였다. Fig. 8과 Fig. 9에는 각각의 탐색 결과를, Fig. 10에는 삽입 결과를 그래프로 나타냈다. 두 그래프에서 X축은 메모리 사용률을 나타내고, Y축은 밀리초 단위의 시간을 나타낸다.

탐색 결과를 나타내는 Fig. 8과 Fig. 9를 보면, 처음부터 성능의 향상이 예측되던 슬롯 접근 횟수의 비교와는 달리 실제로는 약 50% 전후에서부터 성능이 좋아진다. 월 변수에 따로 접근해야하는 오버헤드가 존재하기 때문인데, 메모리 사용률이 커지며 슬롯 접근을 크게 절약할 수 있는 시점부터는 성능이 향상된다. Negative 탐색의 결과를 보이는 Fig. 8을

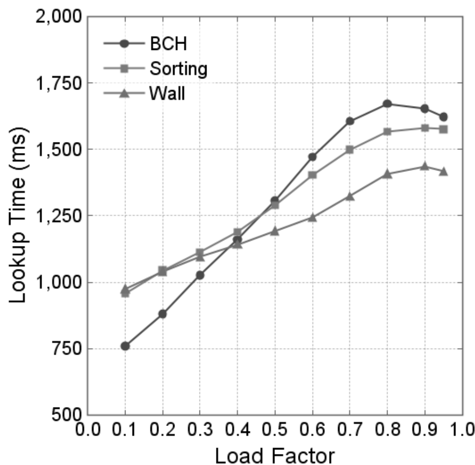


Fig. 8. Comparison of Negative Lookup Time

보면 메모리 사용률이 60%에서 95%인 구간에서 약 12%에서 17%의 시간 감소를 보인다. Positive 탐색의 결과를 보이는 Fig. 9에서는 메모리 사용률이 60%에서 95%인 구간에서 약 5%에서 12%의 시간 감소를 보인다.

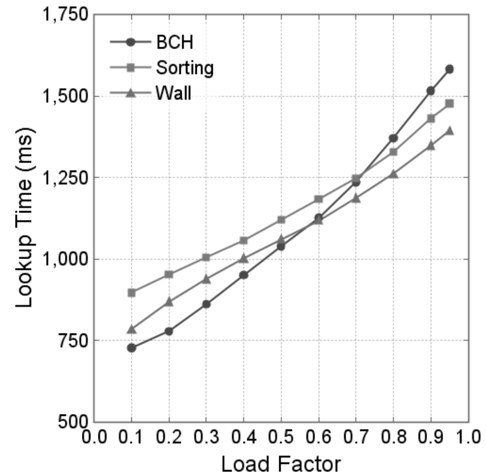


Fig. 9. Comparison of Positive Lookup Time

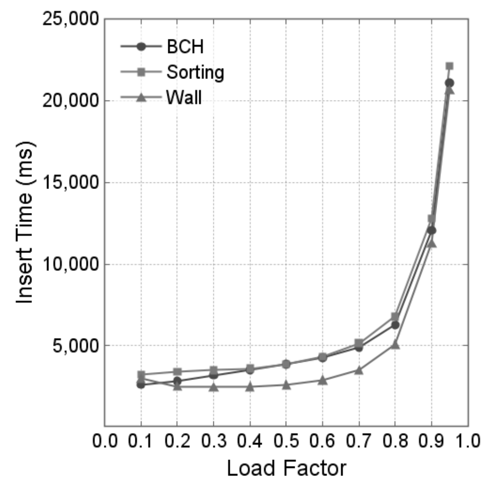


Fig. 10. Comparison of Insert Time

삽입 결과를 나타내는 Fig. 10을 보면, 처음 10% 외에는 성능의 저하를 보이지 않는다. 메모리 사용률이 30%에서 80% 구간은 약 20%에서 30%의 시간 감소를 보이며, 90% 이상에서는 약 5%의 시간 감소를 보인다. 높은 메모리 사용률에서 상대적으로 시간 감소 수치가 적은데, 슬롯 접근에 걸리는 시간보다는 밀어내기가 발생하며 일어나는 다른 버킷으로의 메모리 접근과 데이터의 이동에 걸리는 시간 비율이 더 크기 때문이다.

4) 삽입/탐색 연산 비율 비교 실험

탐색, 삽입만의 비교뿐만 아니라 삽입과 탐색이 일어나는 비율에 따라 얼마나 월 쿠쿠의 성능 이점이 있는지를 실험하였다. 메모리 사용률이 어느 정도인지에 따라 성능이 크게 좌

우되므로, 앞선 실험들과 같이 데이터 삽입을 0%부터 95%까지 시행하였다.

Table 3. Experiment Workload

Insert : Lookup	Total # of Operation
1 : 9	127,506,841 : 1,147,561,568
2 : 8	127,506,841 : 510,027,364
3 : 7	127,506,841 : 297,515,962
4 : 6	127,506,841 : 191,260,262
5 : 5	127,506,841 : 127,506,840
6 : 4	127,506,841 : 85,004,560
7 : 3	127,506,841 : 54,645,788
8 : 2	127,506,841 : 31,876,710
9 : 1	127,506,841 : 14,167,426

삽입 연산의 수가 일정하므로 Table 3과 같이 그에 따라 총 탐색의 수를 조정하였으며, 메모리 사용률 10%마다 1:1 비율로 positive와 negative 탐색을 시행하였다.

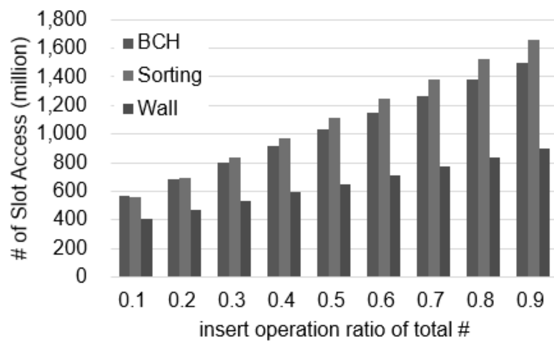


Fig. 11. Comparison of the Number of Slot Accesses in Different Insert/lookup Ratio

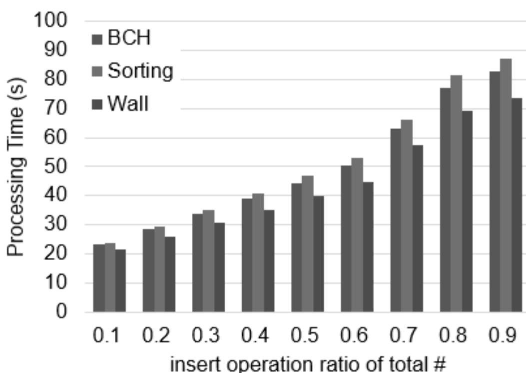


Fig. 12. Comparison of Time in Different Insert/Lookup Ratio

Fig. 11과 Fig. 12는 삽입과 탐색의 총 수가 가장 적은 삽입 비율 9 : 탐색 비율 1의 결과에 맞추어 표준화한 결과를 나타낸 그래프이다.

슬롯 접근 횟수와 시간 비교 모두에서 삽입의 비율이 높을수록 월 쿠쿠의 효율이 높음을 알 수 있다. Fig. 11의 슬롯 접근 횟수 비교에서는 30%에서 40%정도의 감소량을 보이며, Fig. 12의 시간 비교는 7%에서 11%정도의 감소를 보인다. 모든 메모리 사용률에서 균등하게 삽입과 탐색이 일어나므로 BCH와 정렬 쿠쿠에 비해 전반적으로 좋은 성능을 보이는 삽입 연산의 비율이 높을 때 상대적인 성능이 좋다.

앞선 4.3절의 3) 시간 실험에 따르면 높은 메모리 사용률에서는 삽입 연산의 시간 감소 수치가 적지만 탐색 연산은 높은 사용률일수록 효율이 좋아진다. 쿠쿠 해싱은 탐색이 상수 시간에 이뤄진다는 특성상 주로 탐색 위주의 데이터베이스에서 사용될 것이므로, 높은 메모리 사용률에서 탐색이 많은 환경에서도 좋은 결과를 보인다.

4.4 다른 해싱 알고리즘과의 비교 분석

해시 테이블 알고리즘에는 쿠쿠 해싱 외에도 효율적이라고 알려진 알고리즘들이 존재하는데, 대표적으로 멀티쓰레딩 환경에서의 락 프리(lock-free) 알고리즘이 있다. 이러한 여러 해싱 알고리즘과 쿠쿠 해싱 알고리즘의 분석에 관한 연구가 진행된 바 있다[17]. 이에 따르면 삽입에서는 동시 처리에 맞게 구조를 변경한 enhanced Cuckoo hashing이 가장 성능이 좋고, 탐색에서는 enhanced chained hashing이 가장 뛰어나다.

쿠쿠 해싱은 탐색 성능이 중요하므로 탐색 결과를 살펴보면, 70% 이상일 때의 탐색 성능은 쿠쿠 해싱과 체인 해싱이 큰 차이가 없다. 해당 쿠쿠 해싱에도 이 논문에서 제안하는 월 변수를 적용하면 탐색 범위를 줄일 수 있을 것이므로 성능을 향상시켜 체인 해싱에 비해 뒤지는 성능을 보완할 수 있을 것이다. 삽입 연산에서는 쿠쿠 해싱이 이미 가장 우수하다고 결과가 나왔지만, 월 변수를 적용하면 버킷과 슬롯에 접근하는 시간이 줄어들어 그 성능을 더욱 향상시킬 수 있을 것이다.

4.5 월 쿠쿠의 한계점 분석

한 버킷 내의 슬롯들은 데이터의 지역성 때문에 CPU의 캐시에 캐싱되기 쉽고, 따라서 버킷 내의 슬롯들에 대한 접근 시간 지연은 버킷 간의 접근 시간에 비해 적다. 4.3절의 수행 시간 비교에서 이론적인 분석보다 효율이 좋지 않게 나왔던 이유이다.

또한, 월 변수가 별개의 변수로 유지되어 접근하는 데에 오버헤드가 발생하는 점도 한계가 있어 개선의 여지가 있다. 추가적으로 요구되는 메모리의 양은 적지만, 최적화를 거치지 않은 구조로는 효율을 더 높이기 어렵다. 위에서 언급한 캐싱 관련 이슈를 고려하여 버킷 구조를 개선해서 이 오버헤드를 줄일 수 있다면 성능이 더 향상될 수 있음을 기대한다.

5. 결 론

본 논문은 버킷화 쿠쿠 해싱(BCH)의 삽입 및 탐색 성능을 향상하는 월 쿠쿠(wall cuckoo)를 제안하였다. 월 쿠쿠는 입력된 순서대로 버킷에 저장하는 기존의 BCH과 달리, 데이터를

삽입할 때 사용된 해시 함수를 기준으로 데이터를 분리하여 저장한다. 그리고 그 경계점의 위치를 저장하는 윌이라는 추가 변수를 유지하여 데이터를 탐색할 때 접근해야 하는 슬롯의 범위를 줄인다. 삽입 연산 또한 탐색 연산이 수반되기 때문에 마찬가지로 성능이 향상된다. 본 논문에서는 슬롯 접근 횟수의 기댓값 분석을 통해 윌 쿠쿠가 BCH보다 효율적이라는 것을 보였고, BCH와 정렬 쿠쿠(sorting cuckoo)와의 비교 실험을 통해 성능을 입증하였다. 실험 결과에 따르면 윌 쿠쿠는 메모리 사용률이 95%인 상황에서 BCH보다 데이터가 존재하는 탐색(positive 탐색)은 36%, 데이터가 존재하지 않는 탐색(negative 탐색)은 46%만큼 더 감소한 슬롯 접근 횟수를 보였다. 또한 BCH의 탐색 성능을 향상시킨 정렬 쿠쿠에 비해서도 positive 탐색은 29.5%, negative 탐색은 20%만큼 더 적은 슬롯 접근 횟수를 보였으며, 삽입 연산에서도 BCH에 비해 38%만큼 더 감소한 슬롯 접근 횟수를 보였다. 그리고 수행 시간을 측정한 실험에서도 실제로 약 10%가량의 성능 향상을 보였다. 따라서 윌 쿠쿠는 윌 변수를 유지하는 적은 추가 메모리만을 사용하여 삽입과 탐색에서의 성능 향상을 얻을 수 있었다. 윌 쿠쿠는 슬롯 접근 횟수를 감소시키기 때문에, 멀티쓰레딩 환경에서도 메모리에 접근할 때의 락 시간을 줄여 성능 향상에 기여할 수 있다. 향후 연구에서는 사용하는 해시 함수가 두 개가 아닐 때에도 적용이 가능하도록 발전시키고, 메모리 스토리지 기술인 Memcached에 윌 쿠쿠의 구조를 적용하는 연구를 진행할 예정이다.

References

[1] DB-Engines, DB-Engines Ranking [Internet], http://db- engines.com/en/ranking/.

[2] Amazon Web Services, Definition of Key-value Database [Internet], https://aws.amazon.com/ko/nosql/key-value/.

[3] Redis, OBJECT Subcommand [Internet], https://redis.io/commands/object.

[4] Memcached [Internet], https://memcached.org/.

[5] Memcached GitHub, Memcached/memcached.h [Internet], https://goo.gl/BK7nkQ, Line 462.

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

[7] 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.

[8] D. H. Min, R. H. Jang, D. H. Nyang, and K. H. Lee, "Sorting Cuckoo - Enhancing Lookup Performance of Cuckoo Hashing Using Insertion Sort -," *The Journal of Korean Institute of Communications and Information Sciences*, Vol.42, No.3, pp.566-576, Mar. 2017.

[9] U. Erlingsson, M. Manasse, and F. McSherry, "A Cool and Practical Alternative to Traditional Hash Tables," *in Proc. WDAS'06*, Jan. 2006.

[10] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, "SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems," *USENIX Annual Technical Conference*, pp.553-565, Santa Clara, CA, USA, July 2017.

[11] Y. Sun, Y. Hua, D. Feng, L. Yang, P. Zuo, S. Cao, and Y. Guo, "A Collision-Mitigation Cuckoo Hashing Scheme for Large-Scale Storage Systems," *IEEE Trans. Parallel Distrib. Syst.*, Vol.28, No.3, pp.619-632, 2017.

[12] 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.

[13] 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.

[14] 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*, pp.281-294, Jun. 2016.

[15] J. Bob, A Hash Function for Hash Table Lookup [Internet], http://www.burtleburtle.net/bob/hash/doobs.html.

[16] 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.

[17] E. H. Kim and M. S. Kim, "Enhanced chained and Cuckoo hashing methods for multi-core CPUs," *Cluster Computing*, Vol.17, No.3, pp.665-680, 2014.

문 성 광

https://orcid.org/0000-0003-3499-1984

e-mail : skm0622@gmail.com

2017년 인하대학교 컴퓨터공학과(학사)

2017년～현 재 인하대학교 컴퓨터공학과 석사과정

관심분야 : 네트워크 보안

민 대 홍

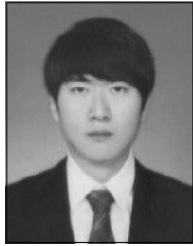
http://orcid.org/0000-0001-9451-312X

e-mail : mang@isrl.kr

2017년 인하대학교 수학과(학사)

2017년～현 재 인하대학교 컴퓨터공학과 석・박사 통합과정

관심분야 : 네트워크 보안, 무선 인터넷 보안, SDNe



장 룡 호

http://orcid.org/0000-0002-3417-6851
e-mail : r.h.jang211@gmail.com
2013년 인하대학교 컴퓨터정보공학과(학사)
2015년 인하대학교 컴퓨터정보공학과(석사)
2015년~현 재 인하대학교 컴퓨터공학과 박사과정
관심분야 : 네트워크 보안, 무선 인터넷 보안, SDN

정 창 훈

http://orcid.org/0000-0001-6299-1207
e-mail : jcptk677@gmail.com
2017년 인하대학교 컴퓨터정보공학과(석사)
2017년~현 재 인하대학교 컴퓨터공학과 박사과정
관심분야 : 인증 프로토콜, 네트워크 보안, 정보보호

양 대 헌

http://orcid.org/0000-0001-5183-891X
e-mail : nyang@inha.ac.kr
1994년 한국과학기술원 전기 및 전자공학과(학사)
1996년 연세대학교 컴퓨터과학과(석사)
2000년 연세대학교 컴퓨터과학과(박사)
2000년~2003년 한국전자통신연구원 정보보호연구본부 선임연구원
2003년~현 재 인하대학교 컴퓨터공학과 교수
관심분야 : 암호이론, 암호프로토콜, 인증프로토콜, 무선 인터넷 보안, 네트워크 보안

이 경 희

http://orcid.org/0000-0001-5669-1216
e-mail : khlee@suwon.ac.kr
1993년 연세대학교 컴퓨터과학과(학사)
1998년 연세대학교 컴퓨터과학과(석사)
2004년 연세대학교 컴퓨터과학과(박사)
1993년~1996년 LG 소프트(주) 연구원
2000년~2005년 한국전자통신연구원 선임연구원
2005년~현 재 수원대학교 전기공학과 부교수
관심분야 : 바이오인식, 정보보호, 컴퓨터비전, 인공지능, 패턴인식