Coputer Algorithms
Project - 02

이름 김태연
학번 201511058
이메일 kim77ty@dgist.ac.kr

요 약

본 프로젝트를 통해서 현대의 secondary memory 의 동작에 대해서 알아보고, 그의 abstract 된 모델인 DGIBox 의 property 들을 고려하며 write 와 read 를 할 수 있도록 key 값들을 mapping 해주는 알고리즘을 만드는 것을 목표로 한다. 기본적으로 direct addressing 방법인 dictionary 와 hash 를 이용한 알고리즘을 구현 한 후, 메모리와 DGIBox operation 개수 등의 성능테스트를 한 후, 각 알고리즘의 장단점을 알아보고, 이러한 점들을 보완하기 위한 dictionary 와 hash 의 장점을 뽑아서 쓰는 새로운 알고리즘인 ty 알고리즘을 제안한다.

1. 서론

DGIBox 라는 현대의 second storage 의 축약된 모델을 이용하여 value 를 저장하고, 읽어 오기 위한 알고리즘을 구현한다. 기본적으로 secondary storage 는 한 번 접근을 할 때 램에 저장된 정보보다 cost 가 많이 들기 때문에, 최대한 DGIBox 의 접근을 최소화하는 것을 목표로 하며, 가능하다면 저장해주기 위해 사용하는 알고리즘의 메모리도 최대한 줄이고자 한다.

DGIBox 는 총 151552 개의 bucket 으로 구성되어 있으며, 해당 bucket 안에는 각각 128 개의 slot 들이 index 로 있다. 그리고 DGIBox 는 다음의 세가지의 연산을 지원한다. 1. Set(bid, index, value) 2. Get(bid, index) 3. Empty(bid)

Set 연산의 경우는 인풋으로 받은 value 를 알맞은 bucket id 와 index 의 주소에 저장해주는 역할을 하며, Get 연산은 인풋으로 받은 주소에 저장되어 있는 값을 출력해주는 역할을 하고, Empty 는 인풋으로 받은 bucket 의 모든 값들을 지워주는 기능을 한다. 이러한 기능을 지원하는 DGIBox 에는 두가지의 property 가 있으며, 첫번째는 같은 bucket 에서는 항상 ascending order 로만 set 의 연산이 가능하다는 것이고, 두번째는 각각의 bucket 에 대해서 overwrite 가 안 된다는 것이다.

이 DGIBox 의 구현을 위해 기본적으로 두 가지의 방법을 사용할 것이며, Dictionary 는 말 그대로 어떠한 key 값에 대해 value 나 어떠한 정보를 저장해주는 방법이며, [1] hash fuction 은 임의의 길이의 입력 메시지를 고정된 길이의 출력값으로 압축시키는 함수를 의미한다.

[1] 해시 함수의 정의, 네이버 지식백과

2. 본문

2.1 Dictionary Algorithm

기본적으로 2D-array 를 만들어 각각의 key 값에 대해서 저장되는 bucket_id 와 index 의 주소의 위치를 알 수 있게 구현을 하였으며, Dictionary 의 알고리즘이 가장 효율적으로 작동하게 하기 위해서는, bucket 이 가득차고, 하나의 bucket 을 cleaning 해줄 때 가장 의미 없는 정보들을 많이 포함하고 있는 bucket 을 선택할 수 있어야 한다. 이를 위해서는 각 bucket 에 대해서 의미 있는 정보가 몇 개가 저장되어 있는지 알아야 하는데, 이는 존재하는 dictionary 를 이용해서 count 를 해 줄 수는 있지만, 이는 cleaning 을 한 번 할 때마다 dictionary 전체를 scan 해줘야 하기 때문에 매우 비효율적이다. 그래서, 메모리를 조금 더 사용하여, count 라는 array 를 만들었으며, count 는 각각의 bucket 에 대해서 유의미한 정보의 개수를 저장해주도록 하였다. 그리고 cleaning 을 할 bucket 을 이렇게 찾고, cleaning 을 해줄 때도, 그 bucket 에 어떠한 key 값이 mapping 이 되는지를 모두 찾아줘야 하기 때문에 이 부분에서 역시 dictionary 만 이용하면 전체를 한 번씩 scan 을 해줘야 되게 된다. 그래서 이 부분에서도 checker 라는 array 를 추가해주어서, 각각의 bucket_id 와 index 의 주소에 대해서 어떠한 key 값이 mapping 이 되어 있는지를 저장해주어서 메모리를 더 쓰지만, 더욱 효율적으로 작동을 하도록 해주었다.

2.2 Hash Algorithm

인풋으로 들어온 key 의 값을 slot 의 수, 즉 128 로 나눈 몫을 bucket_id 로, 나머지를 index 로 mapping 해주는 두개의 hash 함수를 만들어 준다. 그리고 그 함수의 결과로 mapping 된 주소에 set 을 시도하고, set 이 제대로 작동되지 않는 경우에는 cleaning 이라는 작업을 하게 되며, cleaning 에서는 해

당 bucket 에 있는 value 들을 get 하여 모두 임시로 저장을 한 후에, 다시 DGIBox 의 property 에 맞게 순서대로 set 을 해주는 역할을 한다.

2.3 Ty Algorithm

아직 각 알고리즘에 대해서 성능평가는 하지 않았지만, dictionary algorithm 은 메모리를 매우 많이 쓰고, hash algorithm 은 operation 의 수가 매우 많을 것이라는 것을 짐작할 수 있다. 이에 대해서 dictionary 와 비슷한 효율로 동작을 하며, 메모리는 10% 정도만 쓰는 것을 목표로 만든 알고리즘이 Ty Algorithm 이다.

이 알고리즘은 기본적으로 hash 와 dictionary 의 장점을 모아 놓은 알고리즘이다. Dictionary 의 경우는 cleaning 의 수를 최소화해주는 알고리즘이며, hash 는 많은 메모리를 쓰지 않는다는 장점과 함께, hash 함수로 인해 거의 무작위로 bucket 에 들어가지기 때문에 인풋 데이터의 질에 크게 performance 가 크게 영향을 받지 않는다는 장점이 있다.

Ty 알고리즘의 동작을 설명하면, 일단 인풋으로 받은 key 의 값을 전체 bucket 의 수로 나눈 나머지 값을 받는 hash 함수를 이용하여 bucket_id 로 mapping 을 해준다. 이 함수를 이용하면 전체적으로 각각의 bucket 으로 key 값들이 고르게 들어가게 되며, 하나의 bucket 당 최대 111 개의 유의미한 정보만을 저장하게 되어 bucket 에 저장할 공간이 부족할 경우도 없고, 한 번 cleaning 을 할 때 최소 17 개 정도의 공간이 확보되는 것을 보장할 수 있다. 다음으로 각각의 bucket_id 로 mapping 이 된 value 들은 mapping 이 되는 순서대로 0 번째 index 부터 저장이 된다. 이를 위해서는 전체 slot 의 수만큼의 메모리 공간이 필요하지만, dictionary 에서와는 다르게, 각각의 index 에 저장되는 숫자가 0~127, 또는 없을 경우에 -1 이기 때문에 각각 C 에서의 char 로 변수를 지정하여 1Byte 의 공간만이 필요하게 된다. 그리고 dictionary 와 마찬가지로 counts 라는 array 를 추가적으로 만들어서 bucket 에 저장된 value 들의 개수를 저장하여 cleaning 여부의 확인을 조금 더 효율적으로 할 수 있게 한다.

2.4 각 알고리즘들의 성능 비교

먼저 각 알고리즘들이 쓰는 메모리들을 비교해보면 다음의 표와 같다.

표 1 알고리즘별 사용 메모리

	메모리 사용한 변수 명	총 사용 메모리
Dictionary algorithm	ary, checker, count, memory, memory1	212MB
Hash algorithm	temp	512Byte
Ty algorithm	jadress, counts, mem	17MB

dictionary 에서 정의된 array 들에 대해서 먼저 살펴보면, ary 라는 배열은 dictionary 에 해당하는 부분으로, integer 형의 2D array 를 선언해주어 (2^24)*2*4(Byte)의 메모리를 차지하고 있으며, checker 라는 배열은 해당 주소에 대한 key 값들을 저장하고 있으며, BUCKET_NUM*SLOT_NUM*4 = 151552*(2^7)*4(Byte)의 메모리를 사용하게 되며, count 라는 배열은 각 bucket 에 유의미한 정보의 수를 저장하여, BUCKET_NUM*4 = 151552*4(Byte)의 메모리를 사용하고, memory 와 memory1 이라는 배열은 cleaning 을 해줄 때 임시로 데이터를 저장해주는 역할을 하는 메모리로 각각 SLOT_NUM*4 = 4*2^7(Byte)의 메모리를 차지하게 된다. 이 사용되는 메모리들을 모두 합하면 약 212MB 의 메모리를 사용한다는 것을 알 수 있다. 그리고 동작을 조금 더 효율적으로 하게 해주기 위한 배열들을 빼고, 필수적인 배열만의 메모리 사용량을 계산해보면, ary 의 메모리만 필요하게 되므로 대략 128MB 의 메모리가 필요하다는 것을 알 수 있다.

다음으로 hash 에 대해서 보면, temp 라는 array 하나만 사용하며, 이는 cleaning 할 때 정보들을 임시로 저장해주기 위한 메모리로 약 512Byte 만을 사용한다는 것을 알 수 있다.

마지막으로 ty 알고리즘을 보면, jadress 라는 배열은 각각의 key 에 대해서 index 의 주소를 저장해주는 역할을 하기때문에, MAX_KEY*1 = 2^24(Byte)의 메모리를 사용하며, counts 라는 배열은 bucket 에 따라 유의미한 정보의 수를 나타내고 이 역시 1Byte 로 써질 수 있으므로, BUCKET_NUM*1 = 151552(Byte)의 메모리를 필요로 한다. 마지막으로 mem 이라는 배열은 cleaning 할 때 정보를 임시로 저장해주는 array 로, SLOT_NUM*4 = 4*2^7(Byte)의 메모리를 차지하게 된다. 이 메모리들을 모두 합치면 약 17MB 가 되고, 이 경우 역시 counts 라는 배열은 알고리즘 동작을 위해 필수적인 메모리는 아니라는 것을 알 수 있다. 즉, ty 알고리즘과 dictionary 알고리즘을 비교하면 총 메모리를 비교했을 때에는 약 8%정도의 메모리만 쓴다는 것을 알 수 있고, 필수적인 메모리만을 비교했을 때에는 약 13%정도의 메모리를 쓴다는 것을 알 수 있다.

다음으로 각 알고리즘에서 write 한 번 당 평균 operation 의 수를 비교하면 다음과 같다.

표 2 알고리즘별 operation 수

		Trace1	Trace2	Trace3
Dictionary Algorithm	Set count	170,966,375	308,424,941	346,357,870
	get count	70,818,776	208,736,865	246,830,579
	Empty count	1,188,152	2,265,715	2,563,244
	operation per write	2.41	5.16	5.92
Hash Algorithm	Set count	10,802,646,655	10,802,646,913	10,802,646,913
	get count	10,737,418,240	10,737,418,368	10,737,418,240
	Empty count	83,886,080	83,886,081	83,886,080
	operation per write	214.81	214.81	214.81
Ty Algorithm	Set count	701,499,434	701,505,005	701,494,808
	get count	597,713,229	597,706,062	597,730,023
	Empty count	5,398,516	5,398,450	5,398,666
	operation per write	12.96	12.96	12.96

위의 표를 구한 방법은, 알고리즘 내에서 set, get, empty 의 연산이 나오면 그 수를 한 번씩 더하면서 세주었고, 총 수에서 read 를 할 때는 항상 get 을 한 번씩 하기 때문에 get 의 수에서 read 가 호출된 수를 뺀 후 호출된 수를 모두 더하여 write 가 호출된 수로 나누어서 write 한 번당 operation 의 평균 수를 구하였다.

일단, dictionary 와 hash 를 비교하면, write 연산 한 번당 operation 의 수에서 상당히 많은 차이가 난다는 것을 알 수 있다. 하지만, dictionary 의 경우는 trace 파일에 따라서 operation 의 평균 수가 2 배 이상 커지는 등 데이터의 질이 performance 에 크게 영향을 끼친다는 것을 볼 수 있다.

다음으로, ty 알고리즘의 write 한 번당 operation 의 수를 보면, 일단 trace 에 상관 없이 거의 일정하게 나오고, dictionary 보다는 비효율적이지만, 그래도 어느 정도 효율적으로 특히, 기존 hash 알고리즘에 비해서는 약 6%정도의 operation 만 쓰는 등 성능면에서도 좋은 모습을 보여준다는 것을 알 수 있다.

2.5 결론

일단, 모든 알고리즘에 대해서 모든 연산마다 DGIBox 의 check 기능을 이용해서 제대로 연산이 이루어지고 있는지 확인을 하고 싶었지만, check 할 때 연산속도가 매우 느려서 모두 확인을 해 주지는 못했고, 모든 trace 파일과 알고리즘에 대해서 1,000,000 번에 한 번 씩 check 를 해 주어 이상이 없는 것을 확인하였으며, 조금 더 정확한 확인을 위해서 trace3 파일에 대해서는 10,000 번에 한 번, trace2 파일에 대해서는 100 번에 한 번 check 를 해 주었고, 그 결과 모두 이상이 없다는 것을 확인하였다. 이를 통해서, 비록 전체에 대해서 다 확인을 해보지는 못했지만, 3 개의 알고리즘이 모두 정상적으로 작동한다는 것을 알 수 있었다.

추가적으로 dictionary algorithm 의 구현에 대해서 이야기를 해보면, dictionary 를 구현하기 위해서는 이론상으로는 key 에 대해서 bucket_id 와 index 를 저장해주는 dictionary 하나만 있으면 된다. 하지만, 이만을 이용해서 구현해본 결과, cleaning 을 한 번 할 때마다 전체의 dictionary 를 두번씩 scan 을 해줘야 되게 된다. 이는 secondary memory 의 접근으로는 이어지지 않지만, 내부 연산의 과정이 상당히 길어지는 결과를 만들게 된다. 실제로, 처음에 dictionary 만을 이용해서 dictionary algorithm 을 구현하였을 때에는 trace1.bin 파일 하나에 대해서 모든 연산을 해주고, 1,000,000 번에 한 번씩 check 를 해주어도 약 25 시간의 run 타임이 걸렸다. 그에 반해, 추가적인 메모리를 이용하여 구현하였을 때에는 같은 환경에서 실행해본 결과 약 20 분 내의 시간이 걸린다는 것을 확인할 수 있었다. 이는 사실상 dictionary algorithm 이 정상적으로 작동하기 위해서는 128MB 의 메모리 뿐만 아니라 추가적인 메모리도 필요로 한다는 것을 보여준다고 할 수 있다.

다음으로 ty 알고리즘에 대해서 생각을 해보면, Open address 나 binary search tree 등의 조금 더 다양한 이론들을 사용할 수도 있었겠지만, 다른 방법을 사용하면 메모리의 사용향이 input 에 크게 영향을 받을 수 있을 것 같다고 판단을 하여, dictionary 와 hash 만을 이용했으며, 일단 기존의 목표였던 10% 의 메모리보다는 조금 더 사용을 하지만, 기준에 따라 10% 이하라고 볼 수도 있고, 10%근처의 메모리만을 사용하면서 구현을 했다는 점에 의의가 있고, 알고리즘의 complexity 의 경우는 앞서 언급했듯이 write 한 번당 12.96 번 정도의 operation 이 이루어진다는 것을 구했는데, 이 부분 역시 dictionary 와 비교하면 많은 편 이라고 할 수도 있겠지만, memory 를 거의 0%로 사용하는 hash 의 알고리즘과 비교를 해보면 사용하는 메모리에 비해서는 매우 효율적으로 작동한다는 것을 알 수 있다. 즉, 메모리는 dictionary 의 13%정도, complexity 는 hash 의 6% 정도의 성능을 내는, dictionary 와 hash 의 단점을 모두 해결한 알고리즘을 제안하였다고 할 수 있다.

참고문헌

[1] 본 레포트 양식은 국내 학술대회 논문을 참고하였습니다.
[2] Introduction to Algorithms (Thomas H. Cormen)
[3] 컴퓨터 알고리즘 강의노트
[4] 네이버 지식백과