memcached의 slab chunk 크기 조정을 통한 효율적인 메모리 사용

손주형*, 이민재**, 김태일***, 강수용****
한양대학교 컴퓨터공학부
*e-mail:homrunball21@naver.com, **e-mail:dlekst@nate.com,
***e-mail:cardia87@naver.com, ****e-mail:sykang@hanyang.ac.kr

Effectively Using Memory throughout Adjustment for Slab Chunk Size of Memcached

Ju-Hyoung Son*, MinJae Lee**, Tae Il Kim***, SooYong Kang****
Dept of Computer Science and Engineering, HanYang University

요 약

memcached는 기존 RDBMS의 한계를 극복하기 위해 나온 소프트웨어 중 하나이다. memcached는 여러 장점들이 있어 많은 곳에서 활용되지만 주의할 점이 있어 사용 시 몇 가지 요소들을 적절히 설정하여 사용해야 한다. memcached는 기존메모리 할당방식 대신 slab allocater를 통해 메모리를 관리하여 입력되는 데이터의 크기에 맞춰 slab class의 chunk에 저장된다. 입력되는 데이터의 크기에 따라 저장된다는 memcached의 특성 때문에 slab chunk 크기를 조정하는 요소를 상황에 맞게 정해준다면 보다 효율적인 메모리 사용을 할 수 있다는 것을 실험을 통해 검증해보았고, 이 실험을 통해 나온 결과는 memcached를 사용하는 많은 분야에서 응용될 수 있을 것이다.

1. 서론

최근 업계에서는 기존 DataBase(DB) 관리도구의 데이터 수집, 저장, 관리, 분석의 역량을 넘어서는 대량의 데이터 집합인 빅 데이터(Big Data) 이슈에 직면하면서 이를 유연하고 저비용으로 처리할 수 있는 DB의 필요성이 대두되는 상황이다. 이에 맞춰, 전통적인 관계형 데이터베이스 관리 시스템(Relational Data Base Management System)의 한계를 극복하기 위해 분산가능성에 중점을 두고, 수평적이며 확장성을 갖춘 비 관계형 DBMS가 등장하고 있다. NoSQL(NOT ONLY SQL)라고도 불리는 비 관계형 DBMS에는 Memcached, Redis, Casandra, MongoDB 등이 대표적이며, 이미 관련 시장에 널리 사용되고 있는 중이다 [1]. 특히 memcached는 페이스북, 유투브, 아마존, 위키피디아, 소스포지, 트위터, 뉴욕타임즈, 하테나, 구글 App engine 등 많은 대형 사이트에서 사용한다.

2. 비 관계형 DBMS, NoSQL

모든 노드(node)는 같은 시간에 같은 데이터를 보여줘야 하는 일관성(Consistency)과, 일부노드가 다운되더라도 다른 노드에 영향을 끼치지 않는 유효성(Availability)에 중점을 둔 RDBMS로는 네트워크 발전으로 인해 발생한 대용량 데이터의 read/write 제약과 다양한 형태의 데이터 변화의 한계, 클라우드(Cloud) 분산 환경과 웹 환경에서의 확장성에 한계를 갖고 있다 [2]. NoSQL은 이런 한계들을 극복하기 위해 일관성과 유효성은 보장하지 않더라도 분산 환경을 지원하도록 하나의 단위서비스를 추가할 때 성능의 향상을 보이는 수평적 확장성을 보장한다. 주요 특징으로는 오픈소스를 기반으로 하고, 테이블(table)간 조인(join)이 없으며, 노드의 추가, 삭제, 데이터 분산에 유연하고, 고정된 스키마(schema) 없이 키(key)와 값(value)의 쌍으로 구성되어 모델링(modeling), 쿼리(query)에 유연하여 대용량 데이터처리에 특화되어 있다.

3. memcached

memcached는 DB호출, API(Application Program Interface) 호출 또는 페이지 렌더링(page rendering) 등의 결과를 임의의 데이터(문자열 또는 객체)화하여 작은 덩어리 형태로 저장하는 'in-memory 키-값 저장소'이다. 이름 그대로 메모리 캐시를 관리해주는 데몬으로, 고성능 분산메모리 객체 캐싱 시스템 또는 분산 데이터 캐시 소프트웨어라고 정의한다.

memcached를 이용해 데이터를 저장할 때에는 중요한 메모리 자원을 활용하기에 간단하게 조직하고, 최소의 응답반응을 얻기 위해 데이터들을 최대한 간편하게 조작하여 키-값 쌍으로 저장 한다 [3]. 그리고 지정된 키로 정보를 요청하여 정보를 획득, 업데이트, 삭제 등을 수행한다. memcached는 기본적으로 객체를 저장하는데 사용되지만, string과 같은 직렬화 될 수 있는 어떤 변수라도 저장할 수 있다 [6]. memcached에 데이터를 넣을 때는 표1에 정리한 4개의 인자를 사용한다. 그리고 이 인자들을 통해 작업을 수행하는 일반적인 함수는 다음 표 2와 같다.

<표 1> memcached에 데이터 input type

인자	특성
유일키 (key)	캐시 내부의 데이터 검색용 레코드 자체가 고유 아이디인 경우 그 자체가 키로서 사용가능하다. (예: 세션)
저장변수 (value)	직렬 화되어 저장되고 비 직렬 화되어 검색 될 수 있다면 어떤 유형이든 가능하다.
compaction	압축여부(압축에 의한 오버헤드와 메모리 저장 공간에 대해 생각한다.)
cache expire time (expiry)	캐시가 종료될 때 시간을 지정한다. 0으로 지정된 경우 캐시에서 절대 지워지지 않는다.

<표 2> memcached에서 일반적으로 사용되어지는 함수들

함수	특성
get(key)	지정된 키를 memcached에서 정보획득. 키가 존재하지 않으면 오류를 리턴.
set(key, value, expiry)	캐시에서 ID키를 사용하여 지정된 값을 저장. 키가 이미 존재하는 경우 업데이트됨. 만료의 기본기간은 30일이며 단위는 초 단위.
add(key, value, expiry)	키가 존재하지 않는 경우 캐시에 추가. 키가 이미 존재하는 경우 오류를 리턴.
replace(key, value, expiry)	지정된 키의 값을 업데이트. 키가 존재하지 않는 경우에 오류를 리턴.
delete(key, time)	캐시에서부터 키/값 쌍을 삭제. 시간을 제공하는 경우, 지정된 기간 동안 이키로 새 값을 추가하는 것이 차단.
flush_all	캐시에서 모든 현재 항목을 무효화(또는 만료)

memcached에 저장된 정보는 memcached가 셧다운(shut down)되거나 캐시메모리의 고갈, 항목의 만료 경우가 발생할 때까지 영구 저장한다.

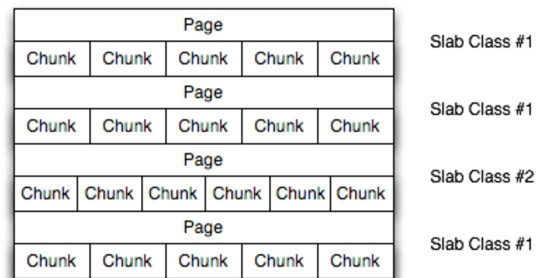
활용되는 예를 살펴보면 DB에 저장된 값에 매우 잦은 접근(Access)이 필요할 경우, 이를 캐시해서 활용하는 것이 훨씬 유용하므로 자주 사용되어지는 값들은 memcached에서 관리하는 것이 효율적이어서 이 때 사용한다. 또한 대표적인 예로 웹 프로그램에서 세션처리에도 활용가능하다. PHP(Personal Hypertext Preprocessor)의 경우 세션을 저장하는 곳이 디스크라면 여유분의 메모리를 활용해 memcached에 저장하는 것이 훨씬 편리하여 많이 쓰인다. 그리고 특정 홈페이지 DB에서 읽혀지지만 매번 달라지지 않는 콘텐츠의 경우에도 처음 로드 시 DB에 읽어서 memcached에 저장하고, 다른 페이지에서 이를 활용할 수가 있다. 이처럼 memcached는 disk나 DB와 같이 상대적으로 느린 소스에서부터 정보를 로드하고 처리하는 것을 방지하도록 자주 사용하는 정보를 저장하기 위한 캐시이다.

반면에 memcached는 사용할 때 주의점이 존재한다. 데이터가 자주 변할 가능성이 낮은 구간에 적용한다면 비약적인 성능의 향상을 가져올 수 있겠지만 데이터가 자주 변경될 가능성이 있는 곳에 가져다 붙이면 공연히 데이터를 캐시에 넣거나, 넣은 데이터가 맞는 데이터인지 검증해야하는 오버헤드가 생긴다. 또한 스왑이 일어나지 않을 정도로 메모리는 여유 있다는 가정 하에 개발이 되었더라도 운영 시 시스템의 메모리를 모니터링해서 스왑이 일어나지 않도록 해야 한다. 메모리가 부족해 스왑이 일어나 메모리에 저장했던 콘텐츠가 스왑 디스크에 저장되면, 심각한 속도 저하가 있기 때문이다.

3.1. memcached의 slab allocator

전통적인 메모리할당 방식으로 malloc과 free를 통해 힙(heap)영역에서 메모리를 관리하는 것 대신 memcached는 슬랩 할당자(Slab Allocator)를 통해 메모리를 관리 한다 [5]. 슬랩 할당(Slab Allocation)은 memcached에서의 메모리 사용을 최적화하고, 메모리 단편화(Fragmentation)를 막아준다. 왜냐하면 메모리에 1MB의 페이지(page)들을 잡아 슬랩(Slab)에 할당하고, 이 슬랩은 곧 다수의 같은 크기 청크(chunk) 들로 나눠져서 결국 캐시에 어떤 값을 저장하려 할 때, memcached가 저장할 값의 크기를 측정하고, 이 항목과 적절한 크기를 포함하는 슬랩을 할당해주어 그 슬랩 내부의 청크에 저장되기 때문이다. 저장할 새 항목이 기존 청크의 크기보다 크다면 적절한 크기의 청크로 나누어진 새로운 슬랩이 만들어 지기도하고, 새 항목이 기존 청크에 적합하지만 빈 청크가 없다면 새로운 슬랩이 만들어 진다 [6]. 기존 저장되어 있던 항목이 업데이트되어 크기가 커지더라도 같은 방식으로 적당한 슬랩이 생성되어 재 할당된다. 슬랩 내부 청크에 항목들이 다 차있는 상태에서 비슷한 크기의 새로운 항목이 저장되려 할 때에는 슬랩 클래스 내부에서 LRU(Least Recently Used) 알고리즘에 의해 저장되어 활용 되지 않는 청크를 선정하여 업데이트 한다. 그림 1은 위 같은 과정을 통해 할당된 memcached 메모리할당 결과를 나타낸 것이다.

(그림 1) memcached에서의 메모리 할당 [5]



예를 들어 슬랩 내부에 가장 작은 청크의 초기 크기가 88byte(값 저장 공간 40byte, 플래그(flag)와 키(key)데이터 저장 공간 48byte)일 때, 처음 저장될 항목이 이보다 작다면 바로 저장되지만, 만약 저장해야할 데이터의 크기가 이보다 크다면 청크의 크기가 청크 크기성장요인(chunk size growth factor)에 의해 데이터크기에 맞춰 충분히 저장할 수 있는 크기를 갖는 새 슬랩을 생성하여 저장한다. 초기 청크의 크기가 104byte이고, 청크 크기 요인이 1.25라면 다음 청크 크기는 104*1.25가 되어 약 130byte가 되는 것이다.

4. 실험

memcached에 저장될 데이터의 크기에 따라 어떤 슬랩에 저장될 것인지가 정해진다는 것은 주의 깊게 살펴볼 필요가 있다. 예를 들어 청크 크기의 기본(default)값은 48byte인데 30byte의 데이터가 들어온다면 18byte의 공간이 남은 채로 그 청크는 LRU가 발생할 때까지 데이터를 받지 못한다. 이 때, 청크마다 내부단편화문제도 발생하게 되고, 청크가 좀 더 데이터에 맞게 충분히 용량이 적지 못해 더 나중에 발생했을 LRU작동이 잦아져 청크에 I/O(Input, Output)가 늘어나 오버헤드가 발생한다. 이러한 문제점에 착안하여 memcached의 slab allocation을 조정한

다면 좀 더 효율적인 사용이 가능할 것 같았다. 슬랩 클래스는 페이지 크기와 청크 크기성장요인으로 결정된다는 점, 또한 데이터크기와 청크 크기가 차이가 많지 않아야 한다는 점에서 초기 청크 크기는 적당한 수준에서 작은 값을 갖게 하고, 청크 크기 성장요인만 변화를 주어가면서 실험을 통해 청크 write 횟수를 분석을 해보기로 하였다.

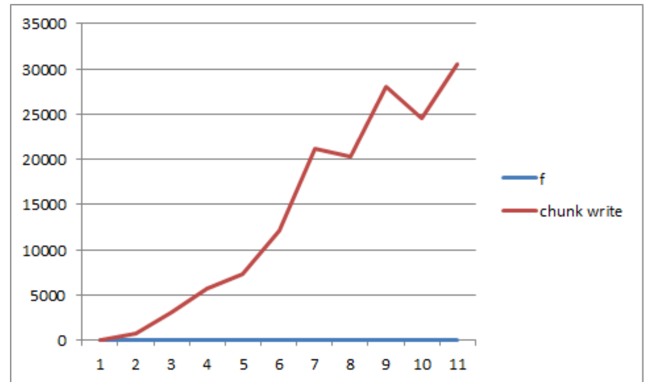
실험은 1부터 15까지의 임의의 숫자를 1부터 10개까지 임의의 개수로 가질 수 있는 트랜잭션(transaction)들을 5만개 생성하여 텍스트파일로 저장하고, 이를 다시 memcached에 set하여 각각 f값이 다른 상태에서 같은 데이터 환경에서 실험해보았다. input데이터들의 값이 그리 크지 않기 때문에 memcached의 주요 인수(parameter)들인 m(memcached의 서버용량)과 n(청크의 초기크기)은 각각 4MB, 24byte로 두고, 청크 크기 성장요인인 f값만 변화를 주어 실험을 진행하였다.

<표 3> memcached 실험 결과(m=4MB, n=24byte)

f (청크 크기성장요인)	Eviction (Chunk write)	슬랩 클래스 종류 개수	full상태의 슬랩 수
1.06	0	13	0
1.065	729	11	1
1.07	3040	11	1
1.075	5725	10	2
1.1	7265	8	4
1.15	12153	6	4
1.2	21182	5	3
1.25	20313	4	2
1.5	28095	3	3
2	24517	2	4
3	30584	1	4

실험 중 f값에 따른 eviction, 즉 청크write 수의 변화가 현격한 부분은 f값의 세밀한 조정이 필요했다. 표 3에는 다른 모든 조건은 동일하고, f값의 변화에 따른 memcached-tool의 stat과 display에서 명시하는 것을 나타내었다. 표에서 확인할 수 있듯이 f의 값이 1.1을 초과할 시 Eviction이 만의 자릿수의 값을 갖는 것을 확인할 수 있다. 그리고 f값이 커서 청크 크기가 비약적으로 커짐에 따라 데이터를 저장할 슬랩 클래스의 종류도 적어지고 full상태인 슬랩의 수도 많아진다는 것을 확인할 수 있었다. 이는 입력데이터의 수가 5만개라는 것을 감안했을 때 상당히 큰 수치이고, 이 실험을 통해 f의 값이 클 경우 메모리 사용이 비효율적이라는 것을 확인할 수 있었다. 위 표의 결과를 도식화하여 좀 더 효과적인 실험 결과를 표현하기 위해 그래프를 그린 것이 그림 2이다.

(그림 2) f값에 따른 Chunk write 횟수



5. 결론

memcached는 기존 RDBMS의 한계를 극복하기 위해 등장한 비 관계형 DBMS중에서도 많은 곳에서 사용되고 있는 소프트웨어이다. 하지만 이는 캐시라는 저장소의 성질 때문에 효과적으로 사용하기 위해서는 특징에 맞는 사용법이 존재한다. 데이터가 자주 변할 가능성이 낮은 구간에 적용한다면 비약적인 성능의 향상을 가져올 수 있겠지만 데이터가 자주 변경될 가능성이 있는 곳에 가져다 쓰면 공연히 데이터를 캐시에 넣거나, 넣은 데이터가 맞는 데이터인지 검증해야하는 오버헤드가 생기므로 주의해야 한다. 또한 memcached는 전통적인 메모리 할당 방식이 아닌 슬랩 할당자를 통해 메모리를 관리하여 슬랩 내부에 일정한 크기들의 청크들을 갖으며, 데이터의 크기에 맞는 청크를 가진 슬랩으로 데이터를 저장하게 된다. 캐시메모리의 특성상 자주 쓰이고, 상대적으로 크기가 작은 데이터들이 저장된다는 것을 감안하여 본 논문에서는 memcached를 사용할 때 설정해야할 값 중 청크 크기를 조밀하게 정해주었고, 결국 오버헤드를 줄인 효율적인 사용방법을 제안하였다. 그리고 시뮬레이션을 통해 Eviction과 청크 크기성장요인과의 관계를 명시하여 성능의 향상을 확인하였다. 이러한 사실을 통해 memcached를 보다 효율적으로 사용할 수 있기에 최근의 빅 데이터 이슈 대안으로 memcached를 사용하는 많은 분야에 응용될 수 있을 것으로 생각된다.

참고문헌

[1] Tatsuya Sasaki "빅 데이터 시대를 위한 NoSQL 핵심 가이드" 로드북

[2] nosql-cap-theorem, http://i-bada.blogspot.kr

[3] B. Fitzpatrick "Distributed Caching with Memcached"

[4] Jithin Jose, Hari subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur-Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, Dhabaleswar K. Panda "Memcached Design on High Performance RDMA Capable Interconnects"

[5] Jure Petrovic "Using Memcached for data distribution in industrial environment"

[6] Memory Allocation within memcached, http://dev.mysql.com/