LevelDB 캐시 구조 및 성능 분석

홍수빈○, 최민국, 유시환, 최종무
단국대학교
{subinhong0109, mgchoi, seehwan.yoo, choijm}@dankook.ac.kr

LevelDB cache structure and performance analysis

Subin Hong○, Min-guk Choi, Seehwan Yoo, Jongmoo Choi
Dankook University

요 약

key-value 쌍으로 데이터를 저장하는 LevelDB는 LSM-tree 구조를 기반으로 데이터를 효율적으로 저장한다. 하지만 계층적 구조로 인해 읽기 작업의 성능이 저하된다. 이에 LevelDB는 빠른 읽기 작업의 성능을 위해 SSTable을 캐싱하는 인덱스 캐시와 데이터를 캐싱하는 블록 캐시를 사용한다. 본 논문에서는 SSTable의 구조와 SSTable의 key-value 데이터를 읽는 작업에서 캐시 사용을 분석하였다. 또한 인덱스 캐시의 개수와 블록 캐시의 크기를 조절하여 성능 변화를 관찰하였다. 실험 결과 데이터 로드 크기에 따라서 인덱스 캐시 개수에 대한 읽기 성능 향상이 관찰되었고, 블록 캐시의 크기가 증가함에 따라 읽기 성능 향상이 관찰되었다.

1. 서 론

수많은 데이터의 양이 전 세계로부터 생겨나면서 지금 관계형 데이터베이스로는 이루어 낼 수 없는 과제들이 생겨나고 있다. 빅데이터가 커지고 있는 디지털 경제 동향을 따라 경쟁 차별화 요소에서 앞서 나가기 위하여 수많은 회사에서 NoSQL을 사용하고 있다[1]. 대표적으로 구글은 NoSQL 기반 데이터베이스인 LevelDB를 개발했다[2]. 많은 현대 NoSQL 데이터베이스 시스템들이 LSM-tree를 선택하였으며[3], LevelDB 또한 마찬가지이다. LSM-tree는 tree의 특정 레벨에서 merge sort를 통해 변경사항을 계층적으로 일괄처리하여 효율성을 높인다. 또한 out place update 방식을 채택하여 항상 여러 페이지 블록을 새 위치에 순차적으로 기록한다[4].

이처럼 LevelDB는 LSM-tree 구조를 기반으로 쓰기 작업에는 효율적이나, LSM-tree의 계층적 구조로 인하여 읽기 작업에 여러 레벨을 탐색해야 하는 문제가 존재한다. 이에 LevelDB는 Disk I/O를 줄이기 위해 인덱스 캐시와 블록 캐시를 이용한다. 인덱스 캐시는 Disk에 존재하는 SSTable 파일의 인덱스 블록을 캐싱하며, 블록 캐시는 SSTable의 필터 블록과 데이터 블록을 캐싱한다. 본 논문의 2장에서는 SSTable의 구조를 설명하며, SSTable의 읽기 작업 순서와 이에 따른 두 가지 유형의 캐싱을 설명한다. 3장에서는 캐시 구조를 설명한다. 4장에서 여러 종류의 워크로드에서 캐시 개수와 크기를 조절하면서 성능 변화에 대한 실험을 진행한다.

*본 연구는 과학기술정보통신부 및 정보통신기획평가원의 2022년도 SW중심대학사업의 결과로 수행(2017-0-00091)과 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구이고 (No.2021-0-01475) 2022년도 정부(과학기술정보통신부)의 재원으로 한국연구재단 중견연구자지원사업의 지원을 받아 수행된 연구임(No.2022R1A2C100605011)

2. SSTable

2.1 SSTable 구조

그림1은 LevelDB의 디스크에 key-value를 저장하는 형식인 SSTable을 논리적 관점에서 본 구조이다. 데이터 블록은 key-value가 저장되는데 사용된다. 필터 블록은 데이터 블록의 블룸 필터가 저장된다. 메타 인덱스 블록은 필터 블록의 인덱스를 저장한다. 이와 마찬가지로 인덱스 블록은 모든 데이터 블록의 인덱스 정보를 저장한다. 풋터(Footer)은 메타 인덱스 블록과 인덱스 블록의 인덱스 정보를 저장한다.

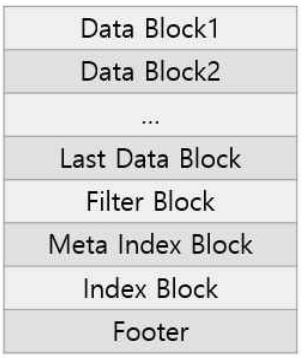

그림 1. SSTable 구조

2.2 SSTable 읽기 작업

그림2는 LevelDB의 읽기 작업과 SSTable을 읽을 경우의 캐시 사용을 보인다. LevelDB의 읽기 연산 수행 시, 먼저 메모리에 존재하는 Memtable에서 데이터를 탐색한다. 만약 찾지 못했다면 Immutable Memtable에서 데이터를 탐색(Memtable::Get)한다. 메모리에서 데이터를 찾지 못했다면, SSTable이 존재하는 디스크의 첫 번째 레벨인 L0부터 마지막 레벨까지 순차적으로 데이터를 탐색(Version::Get)한다.

SSTable의 key-value 데이터를 읽으려 할 때, 인덱스 블록을 통해서 찾으려는 key가 속한 데이터 블록을 찾는다. 이후 메타

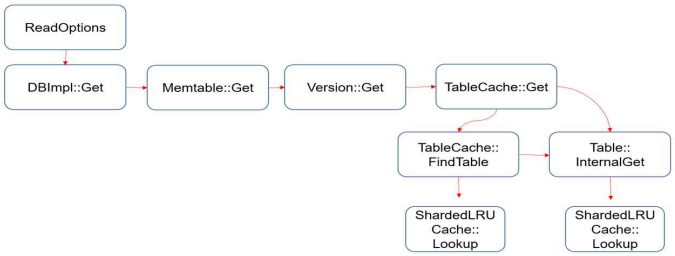

그림 2. LevelDB 읽기 흐름

필터 블록을 통해서, 필터 블록에서 해당 데이터 블록의 블룸 필터를 찾는다. 해시 연산 이후에 블룸 필터가 참이라면, 데이터 블록을 읽고, 데이터 블록에서 찾으려는 key-value를 찾는다. 이때 인덱스 캐시는 SSTable의 인덱스를 캐싱한 것이고, 블록 캐시는 데이터 블록과 필터 블록 등 인덱스 블록을 제외한 나머지 SSTable의 블록들을 캐싱한다.

3. LevelDB 캐시 구조

인덱스 캐시와 블록 캐시를 이루는 캐시 구조는 그림 3과 같다. 다만 각 캐시의 노드에 담기는 값이 인덱스인지 블록 데이터인지만 다르다. LevelDB의 캐시는 LRU(Least Recently Used) 정책을 활용한다. LRU란 데이터를 제거할 때 가장 오랫동안 사용하지 않은 데이터를 제거하는 정책이다.

LevelDB는 캐시를 샤딩을 통해 16개의 샤드로 나누어 관리한다. 읽기 연산 시 샤드 단위로 상호 배제 잠금(Lock)을 수행하며, 이는 여러 스레드가 접근하는 것을 줄여 다중 스레드 조회 속도를 높인다. 각 샤드는 해시 테이블과 두 개의 이중 연결 리스트를 사용한다. 해시 테이블을 사용하여 노드에 빠른 접근이 가능하며, 이중 연결 리스트를 사용하여 새로운 노드와 오래된 노드의 양방향 탐색을 통해 삽입과 삭제가 용이하다.

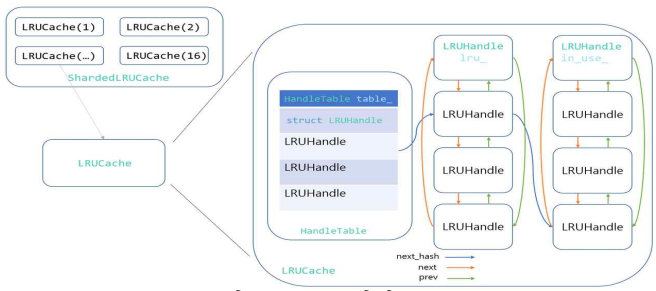

그림 3. LRU 캐시 구조

4. 실험

4.1 실험 환경

인스턴스	AWS EC2(t2.micro)
vCPU*	1
메모리	1GiB
스토리지	EBS 전용

표 1. 실험 환경

실험 환경은 표1과 같이 구성하였으며 Amazon Elastic Compute Cloud(Amazon EC2)를 사용하였다. key의 크기는 각 16byte이고, value의 크기는 각 100byte로 설정하였다. LevelDB의 자체 벤치마크인 db_bench 활용하여, fillrandom을 통해 100MB의 데이터를 로드하였다. 데이터를 읽는 read/seek의 경우에는 1MB의 데이터를 읽도록 하였다. LevelDB 벤치마크 파라미터 중 디폴트가 1000개인 인덱스 캐시의 개수를 조절하는 옵션 open_files을 사용하였고, 디폴트가 4MB인 블록 캐시의 크기를 조절하는 옵션 cache_size를 이용하였다. readhot / seekrandom/ readrandom의 워크로드에 대한 실험을 진행하였다.

4.2 인덱스 캐시 읽기 성능 분석

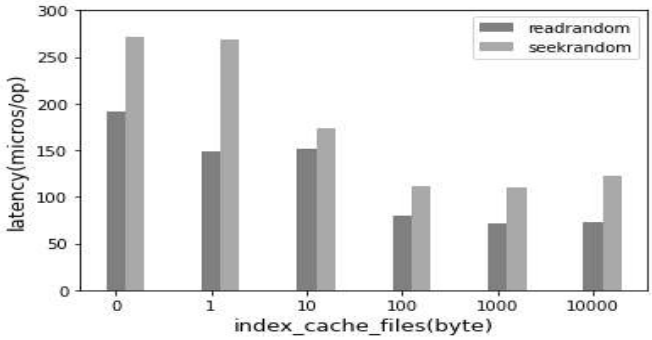

그림 4. 인덱스 캐시 수에 따른 읽기 성능 비교

옵션 open_files를 통해 인덱스 캐시를 0개부터 10000개까지 10배씩 높이며 실험하였고, 블록 캐시는 디폴트 값인 4MB를 사용하였다. 그림 4에서는 인덱스 캐시의 개수가 증가할수록 readrandom과 seekrandom 모두 latency가 감소한다. 이때 100개 이상부터는 인덱스 캐시의 개수변화에 따른 성능변화가 미미한데, 이는 전체 SSTable의 개수가 100개 이하이므로, 더 이상 캐싱할 SSTable의 블록이 없기 때문이다.

4.3 블록 캐시 읽기 성능 분석

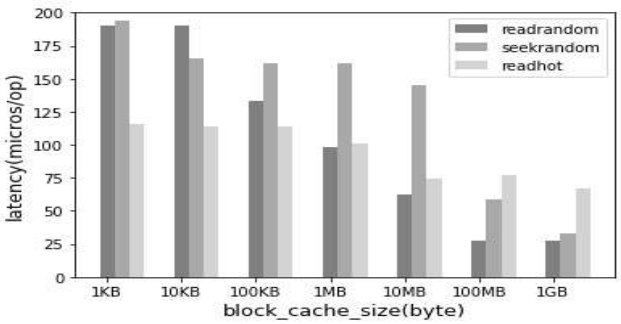

그림 5. 블록 캐시 크기에 따른 읽기 성능 비교

옵션 cache_size를 통해 블록 캐시 크기를 1KB ~1GB까지 10배씩 높여가며 실험하였고, 테이블 캐시는 디폴트 값인 1000개로 사용하였다. 그림 5에서는 모든 워크로드에서 블록 캐시 크기가 증가할수록 latency가 감소하는 경향을 보인다. 다만 워크로드에 따라 블록 캐시 크기 증가에 따른 성능 변화 양상이 다르다.

readhot은 random하게 특정 1퍼센트의 데이터만 읽으므로, 워킹 셋이 작다. 따라서 작은 캐시에서도 효과적이나, 블록 캐시 크기 증가에 따른 성능 향상은 미미하다. readrandom은 random 하게 데이터를 읽기 때문에 readhot보다 워킹 셋이 크다. 따라서 블록 캐시 크기 증가에 따른 성능 향상이 뚜렷하다. seekrandom은 모든 레벨을 탐색하여, iterator를 구성해야 하

므로 readrandom보다 워킹 셋이 크다. 따라서 캐시 크기가 작은 10KB ~ 10MB의 경우에는 워킹셋을 만족하지 못해, 캐시 크기 증가에 따른 성능 향상이 미미함이 관찰되었다.

5. 결론

LevelDB에서 인덱스 캐시의 개수와 블록 캐시의 크기에 따라서 읽기 성능에 얼마나 영향을 미치는지 분석하였다. 결과적으로 모든 작업(읽기 워크로드)에서 인덱스 캐시의 개수가 증가하면서 성능이 향상했다. 하지만 인덱스 캐시의 수가 인덱스의 수 보다 많은 경우에는 성능 변화가 미미하였다. 블록 캐시의 크기에 따라서는 모든 작업에서 성능이 향상하였으나, 성능 변화 양상이 각 읽기 작업의 특징에 따라서 다름을 실험을 통해 관찰하였다.

이후에는 LevelDB 캐시 구조의 샤딩 또한 읽기 작업의 성능 향상에 큰 영향을 미치는 요소이기에 샤딩이 어떤 기준에 의해서 캐시를 16개로 나누어 관리하는지, 무슨 방식으로 효율적인 로드 밸런싱을 지키며 각 캐시에 데이터가 삽입되는지에 대해서 연구할 것이다.

참고문헌

[1] Why successful enterprises rely on NoSQL,https://www.couchbase.com/resources/why-nosql

[2] LevelDB, https://github.com/google/leveldb

[3] Mao, Qizhong, Efficient Storage Design in Log-Structured Merge (LSM) Tree Databases, 2022, abstract

[4] Patrick O'Neil, The Log-Structured Merge-Tree (LSM-Tree), Acta Informatica, 1996, page-7