JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 23, No. 5, pp. 921-927, May. 2022

Check for updates

모바일 엣지 컴퓨팅을 위한 효율적인 키-값 스토리지 엔진의 설계 및 평가

한 혁
동덕여자대학교 컴퓨터학과 부교수

Design and Implementation of an Efficient Key-Value Storage Engine for Mobile Edge Computing

Hyuck Han

Associate Professor, Department of Computer Science, Dongduk Women's University, Seoul 02748, Korea

[요 약]

유비쿼터스 환경에서 사용자가 자동차와 같은 이동하는 장치와 연관된 서비스에 접근하려면 그 장치가 위치하는 영역을 담당하는 모바일 엣지 서버에 이동 장치의 정보가 등록되어 있어야 한다. 모바일 엣지 환경에서 이동 장치의 정보를 관리하기 위해 키-값 스토리지 엔진을 사용하는데 기존의 키-값 스토리지 엔진은 모바일 엣지 환경에서 이동하는 장치의 특성을 최대로 활용하고 있지 못 하다. 본 논문에서는 장치의 이동성을 고려하기 위해 2계층 키-값 스토리지 엔진을 제안한다. 비교적 짧은 시간 동안에 엣지에 유지되어야 하는 장치들은 상위 계층에서 그리고 긴 시간 동안 유지되는 장치들은 하위 계층에서 관리된다. 이를 통해 스토리지 엔진의 성능을 높이고 쓰기 증폭을 줄였으며 이러한 기법이 구현된 스토리지 엔진은 기존의 키-값 스토리지 엔진보다 최대 23배 이상 높은 성능을 달성한 것으로 나타났다.

[Abstract]

In order for a user to access a service related to a moving device, such as a car, the information of the moving device is registered in a mobile edge server in charge of the area in which the device is located. Key-value storage engines are used to manage moving device information in the mobile edge environment. However, conventional key-value storage engines do not exploit the characteristics of moving devices. In this article, we propose a two-level key-value storage engine to consider the device mobility. Devices that are managed at the edge for a relatively short time is kept at the upper layer and devices that are maintained for a long time are managed at the lower layer. This scheme improves the performance of the storage engine and reduces write amplification, and our experimental results show that the storage engine with our scheme achieves up to 23 times better performance than the existing key-value storage engine.

색인어 : 모바일 엣지, 키-값 스토리지 엔진, 로그 기반 스토리지, 2계층 저장소, 이동 장치

Keyword : Mobile Edge, Key-Value Storage Engine, Log-Structured Storage, Two-Level Storage, Moving Device

http://dx.doi.org/10.9728/dcs.2022.23.5.921

Received 12 April 2022; **Revised** 04 May 2022
Accepted 12 May 2022

***Corresponding Author; Hyuck Han**

Tel: +82-2-940-4687
E-mail: hhyuck96@dongduk.ac.kr

Ⅰ. 서 론

엣지 컴퓨팅은 사용자, 장치 또는 데이터 소스의 위치를 고려해서 그 근처에서 컴퓨팅을 수행하는 모델을 의미한다. 즉, 엣지 컴퓨팅은 다양한 위치에서 데이터 연산/처리, 제어 처리를 분산시킨다. 최근 IoT 장치들이 폭발적으로 증가함에 따라 장치의 진단, 에너지 관리 등의 작업을 엣지 컴퓨팅을 통해 수행하는 일이 빈번하며, 장치에서 취득한 많은 양의 데이터를 엣지 서버에서 관리한다. 또한, 장기적인 데이터 분석을 기반으로 실시간 모니터링 및 오프라인 의사결정을 수행한다.

그림 1과 같은 클라우드-엣지 구조는 다양한 이동 장치를 관리하는 실제 환경을 보여준다. 자동차와 드론과 같은 단말 장치는 센서를 이용해서 데이터를 취득하고 취득한 데이터를 엣지 계층으로 전송한다. 또한, 엣지 계층으로부터의 명령을 처리하고 그 결과를 전송한다. 엣지 계층의 게이트웨이는 복수의 이동 장치로부터 연속적으로 송신되는 데이터를 저장하며 저장된 데이터에 대해서 실시간 질의를 처리한다. 또한 사용자 요청에 의해 이동 장치에 대한 명령을 전달하고 처리된 결과를 사용자에게 전송한다.

이러한 환경에서 장치가 이동하면서 데이터와 제어 명령 처리 결과를 전송하는 모델은 이동 장치를 관리하는 엣지 서버가 실시간으로 변경된다는 것을 고려해야 한다. 그림1에서 Car#1과 Drone#1이 이동함에 따라 데이터를 전송하는 엣지 서버가 edge1/edge2에서 각각 edge2/edge4로 변경됨을 보여준다. 이 때 기존의 엣지 서버에서 관리하고 있는 장치 데이터들은 클라우드 컴퓨팅 서버 혹은 목적지 엣지 서버로 전달되어 관리될 수 있다. 그리고 기존의 엣지 서버는 이동된 장치가 생산한 데이터들을 삭제한다. 본 연구에서는 엣지 컴퓨팅의 게이트웨이(엣지 서버 혹은 엣지 노드)에서 이동하는 장치의 데이터 및 정보를 삽입 및 삭제를 포함하는 데이터 관리에 대해 초점을 맞춘다. 엣지 컴퓨팅에서 장치 데이터를 실시간으로 관리하기 위해 로그 구조 병합 트리(LSM tree)에서 기반의 키-값 스토리지 엔진들을 주로 사용한다 [1][2].

이러한 LSM tree는 쓰기 집약적인 워크로드에서 뛰어난 성능을 보인다. 그러나 기존의 LSM tree 기반 키 값 스토리지 엔진은 엣지 컴퓨팅 환경에서 이동하는 장치의 데이터 워크로드 및 정보 특성을 충분히 반영하고 있지 못하다. 엣지 서버의 관점에서 이동하는 장치의 데이터는 해당 서버에서 삽입되어 일정 시간 동안에 관리된다. 이 과정에서 인덱싱을 위해 저장 장치로부터 데이터들을 읽어서 정렬을 하고 정렬된 데이터들을 다시 저장 장치로 쓰는 LSM tree의 compaction 및 merge 연산을 반복적으로 수행한다.

그리고 장치가 다른 엣지 서버가 관리하는 영역으로 이동하면 그 장치는 클라우드 서버 혹은 다른 엣지 서버에서 관리되고 기존의 엣지 서버에서는 해당 데이터가 삭제된다. 이동하는 장치들이 생산한 데이터들은 엣지 서버에 추가되어 저장 장치에 써지면 삭제될 때까지 데이터 변경이 거의 없음에도 compaction 및 merge 연산으로 여러 번 저장 장치에 써진다.

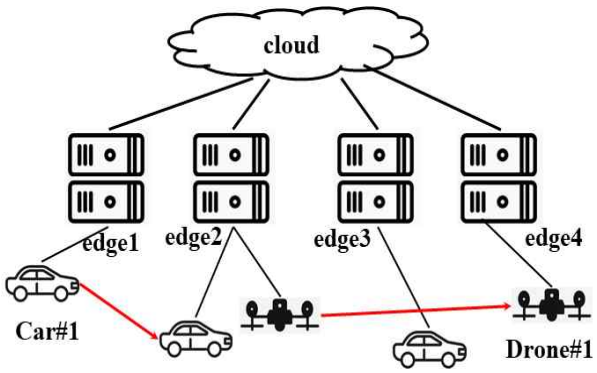

그림 1. 엣지 컴퓨팅 개요
Fig. 1. Overview of Edge Computing

이러한 점이 엣지 서버에 장착된 저장 장치의 성능을 최대로 활용하지 못 하게 하여 키-값 스토리지 엔진의 최대 성능을 제한하며, 저장 장치의 수명을 단축시킬 수 있다. 본 논문에서는 이 문제를 다루기 위해 이동 장치들의 특성을 고려하여 키-값 스토리지 엔진을 최적화하는 기법을 설계하고 평가하고자 한다.

제안하는 기법은 장치의 이동성을 고려한 최적화를 위해 2단계 관리 기법을 제안한다. 이 기법은 장치가 엣지 서버에 등록되어 관리될 때 기존의 키-값 스토리지 엔진이 아니라 장치별 로그 기반 저장소에서 일정 시간 동안에만 관리된다. 본 논문에서는 이 시간을 관리 시간으로 표시한다. 관리 시간 동안에는 장치에 대한 데이터 및 정보는 장치별 로그 기반 저장소에서 접근이 가능하다. 만약 이 장치가 관리 시간 안에 다른 영역으로 이동하면 관리 시간 후에 다른 장치의 정보들과 함께 삭제된다. 만약 관리 시간 후에도 이 장치가 여전히 같은 영역에 있다면 그 데이터들은 모두 기존의 키-값 스토리지 엔진으로 추가되어 관리된다. 이렇게 함으로써 키-값 스토리지 엔진의 쓰기 증폭을 줄여서 성능을 높일 수 있다.

본 논문의 기여는 다음과 같다. 먼저, 이동 장치를 고려해서 엣지 환경에서 사용되는 키-값 스토리지 엔진을 최적화하기 위해 2단계로 데이터를 관리하는 최적화 기법을 제안하고 제안하는 최적화 기법을 구글이 개발한 LevelDB[3]에 구현하였다. 그리고 장치에서 발생하는 센서 데이터의 워크로드를 모사하는 벤치마크를 이용해서 구현된 키-값 스토리지 엔진을 평가하였고, 평가 결과를 통해 제안하는 기법이 기존 기법보다 최대 23배 정도 우수한 성능을 보였다.

본 논문은 다음과 구성을 가진다. 2장에서는 관련 연구들 요약 및 설명하여 본 연구와의 차별성을 제시한다. 3장에서는 엣지 서버의 키-값 스토리지 엔진을 위한 최적화 기법을 제시한다. 4장에서는 제안된 기법을 구현한 키-값 스토리지 엔진의 성능 평가 결과를 보여준다. 5장에서는 논문의 결론과 향후 연구 계획을 제시한다.

II. 관련 연구

이 장에서는 병합 및 커밋 과정에서 키-값 스토리지 엔진의 성능을 향상시키는 선행 연구를 요약하고자 한다. 또한, 이 연구와 관련된 기존의 엣지 컴퓨팅에서의 데이터 관리 시스템을 요약한다.

2-1 관련 연구

LSM tree 기반 키-값 스토리지 엔진의 병합 오버헤드를 줄이거나 제거하는 것은 성능 향상을 위한 중요한 문제이다. [4, 5, 6]에서는 병합 중에 데이터를 다시 쓰는 횟수를 줄이기 위해 파일의 그룹화 정책을 제안했다. 이러한 방식은 데이터 파일 간의 중복되는 키 범위를 검사하여 겹치는 범위가 큰 데이터 파일을 같은 그룹에 넣는다. [7][8]의 연구에서는 키-값 스토리지 엔진의 데이터 처리 성능을 높이기 위해 로그 구조 해시 테이블을 제안했다. 로그 구조 해시 테이블을 사용한 키-값 스토리지 엔진은 put 연산을 통해 저장되는 데이터는 정렬하지 않고 로그 구조 파일에 저장된다. 그리고 데이터의 로그 구조 파일 내의 위치와 키를 메인 메모리 내의 해시 테이블로 관리하여 get 연산을 고속으로 처리할 수 있게 한다. 그러나 관리해야할 데이터의 양이 많아지면 메인 메모리 내의 해시 테이블의 크기도 커지게 된다. 또한 해시 인덱스를 사용하기 때문에 범위 질의를 처리하기가 어렵다.

[9][10]은 장치 혹은 센서 데이터를 시간 범위별로 관리하는 방법을 제안하였다. 시간 범위별 데이터 파티션에 대해 메모리 인덱스 구조를 유지하여 시간 범위 질의를 효과적으로 처리한다. [11][12]의 연구가 제안하는 키-값 스토리지 엔진은 키-값 데이터는 로그 구조 파일에 추가하며 키와 해당 키-값 데이터가 로그 구조 파일에서의 위치를 값을 LSM tree에서 관리한다. 이런 구조는 병합 오버헤드 줄일 수 있지만 이동 장치가 생산하는 데이터의 특징을 고려하고 있지 않다.

[13]의 연구에서는 키-값 데이터의 저장 연산을 커밋할 때 일정 동안의 시간을 두어 여러 개의 커밋 연산들을 하나의 그룹으로 처리하는 방법을 제안하였다. [14]의 연구에서는 키-값 스토리지 엔진에서 로그 쓰기 연산을 완료하기 전에 트랜잭션 처리를 완료하는 것을 허용하거나 로그 쓰기 연산 중에 다른 트랜잭션들이 수행될 수 있도록 하여 성능을 향상 시킨다. [13,14]의 연구들이 키-값 스토리지 엔진의 성능을 개선하지만 이동하는 장치의 데이터 특성에 초점을 맞추지는 않는다.

2-2 엣지 컴퓨팅에서의 데이터 관리 시스템

엣지 컴퓨팅은 기존 클라우드 컴퓨팅의 단점 (네트워크 지연 시간)을 극복하기 위한 새로운 컴퓨팅 패러다임으로 부상했다. 많은 연구 및 개발팀이 엣지 디바이스 혹은 엣지 장치의 데이터 관리 시스템을 적극적으로 연구해 왔다. [15]의 연구에서는 엣지 컴퓨팅 환경의 낮은 네트워크 지연 시간을 활용하는 EdgeKV라고 불리는 범용 키-값 스토리지 엔진을 제안했다. 데이터 복제를 위해 근처 엣지 노드를 사용하고 데이터 일관성을 보장한다.

DataFog [16] 및 FogStore [17] 연구에서는 여러 엣지 노드에서의 질의 처리를 지원하기 위해 위치 및 타임스탬프를 사용하는 분산 인덱싱 방식이 제안되었다. 이러한 인덱싱 체계를 가지고 있지 않은 다른 키-값 스토리지 엔진보다 시공간 범위 쿼리(예: 차량 궤도)에 대한 낮은 실행 시간을 보여주었다.

[18]-[20]에서는 리소스가 부족한 임베디드 컴퓨팅 장치에 최적화된 키-값 스토리지 엔진을 제안하여 데이터를 엣지 서버로 전송하기 전에 장치에서 처리할 수 있도록 했다. 그러나 메모리 사용량을 최소화하는 설계 정책으로 엣지 서버에서 사용할 경우 실시간 질의에 효율적으로 대응하기가 어렵다. Apache IoTDB [2]는 고성능 데이터 수집 및 분석을 지원하는 IoT용 풀 스택 시계열 데이터베이스이다. 스토리지 컴포넌트뿐만 아니라 IoT 데이터를 인덱싱하는 방식도 갖추고 있다. 병합 오버헤드는 없지만 서로 다른 장치의 데이터가 동일한 스토리지 그룹에 저장될 경우 스토리지 그룹에서 락 오버헤드가 발생할 수 있다. 앞서 언급한 연구들은 엣지 컴퓨팅을 위한 효율적인 데이터 처리 아키텍처를 설계하는 데 초점을 맞추었지만 이동하는 장치의 데이터를 효과적으로 관리하는 것에 초점을 맞추고 있지 않다.

III. 2단계 관리 기법의 설계 및 구현

3-1 LSM Tree 개요 및 연구 동기

이 장에서는 많은 키-값 스토리지 엔진들의 근간이 되는 LSM tree 구조와 이동하는 장치의 데이터를 처리할 때의 문제를 설명한다. 키-값 스토리지 엔진은 데이터를 키-값 쌍으로 읽고 저장하는 소프트웨어이다. 스토리지 엔진에서 관리되는 키는 데이터의 식별자이며, 값은 저장 장치에서 관리될 데이터이다. 그림 2는 LSM tree 기반 키-값 스토리지 엔진의 구조를 보여준다. LSM tree에는 보통 memtable이라고 불리는 메모리 내 쓰기 버퍼가 있다. 이 버퍼는 키-값 스토리지 엔진의 입력 키-값 데이터를 키 별로 정렬 및 관리한다. Memtable에 데이터를 저장하는 공간이 부족한 경우 memtable에서 관리하는 데이터는 Stored Sequence Table(SST) 파일에 기록된다. SST 파일은 가장 최근에 저장 장치에 써지는 데이터가 레벨 0으로 생성된다. 그리고 각 레벨의 저장 한도를 초과하면 다음 레벨의 SST 파일들과 병합 프로세스가 수행된다. 즉, 병합 작업은 저장 장치로부터 현재 레벨과 다음 레벨의 SST 파일들의 데이터들을 읽고, 그 데이터들을 병합하고, 다시 저장 장치로 쓰는 작업이기 때문에 쓰기 증폭이 발생한다.

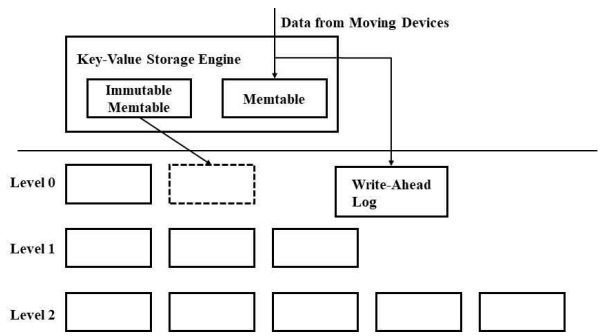

그림 2. LSM Tree 구조
Fig. 2. Architecture of LSM Tree

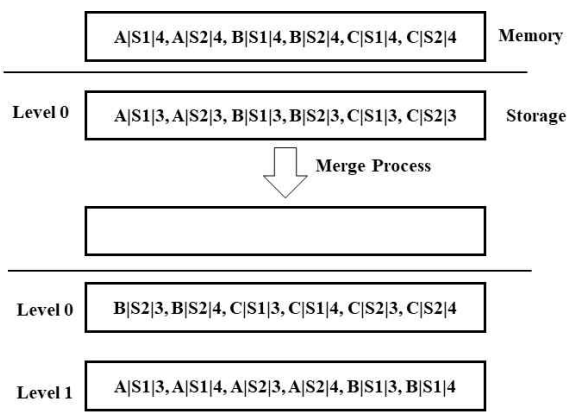

그림 3. LSM Tree의 쓰기 증폭
Fig. 3. Write Amplification in LSM Tree

그림 3은 자동차 A, B, C의 센서 S1, S2가 시간 3과 4에서 생성한 데이터가 저장될 때 발생하는 LSM tree의 쓰기 증폭을 보여준다. 이 예는 키-값 데이터 중에서 키만 보여주며 키는 자동차 식별자, 센서 식별자, 그리고 데이터 발생 시간의 조합이다. 시간 4에서 처리해야 할 데이터들이 memtable의 공간을 다 쓰게 되어 레벨 0의 SST 파일들의 데이터들과 병합해야 한다. 이 때 시간 3에서 저장한 데이터들은 변경 사항이 없음에도 모두 저장 장치에 새로 써져야 하기 때문에 쓰기 증폭이 발생하게 된다. 이후 만약 자동차 B가 엣지 서버에게 관리하는 영역에서 벗어나게 되면 자동차 B의 데이터는 모두 삭제되어야 하고 이 경우에도 자동차 A와 자동차 C에서 발생한 데이터들은 변경이 없음에도 다시 저장 장치에 써지는 경우가 생긴다.

3-2 2단계 관리 기법

그림 4는 움직이는 장치들을 고려하는 엣지 환경을 위한 스토리지 엔진 구조를 보여준다. 제안하는 구조는 상위 수준의 로그 구조 저장소들과 하위 수준의 LSM tree 기반의 저장소로 나누어지며 모든 저장소들은 모두 메모리 상의 오브젝트 매핑 테이블(Object Mapping Table)이라는 해시 테이블로 관리된다.

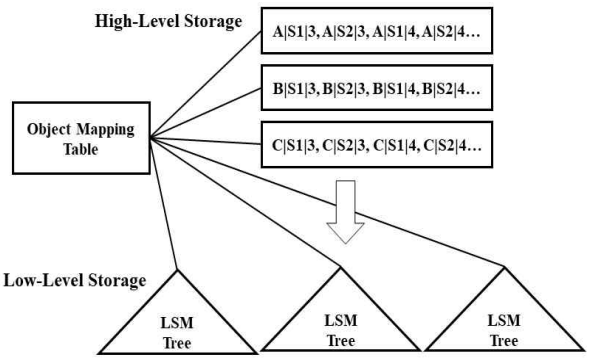

그림 4. 2단계 데이터 관리 기법
Fig. 4. Two-Level Data Management Scheme

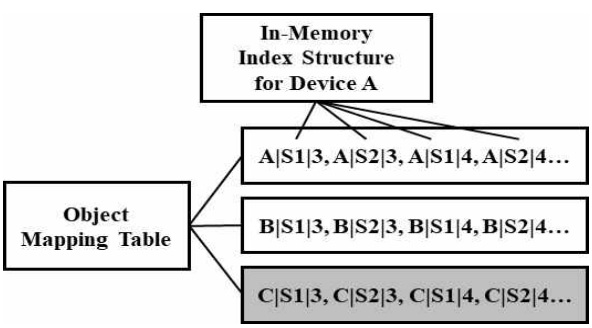

그림 5. 상위 수준 저장소
Fig. 5. High-Level Storage

상위 수준 저장소에는 해당 엣지 서버가 관리하는 영역에 새로운 장치가 진입하여 생성하는 데이터들이 저장되어 관리된다. 그리고 이 데이터들은 미리 지정한 특정한 시간 동안에만 로그 구조 저장소에서 관리되고 이후에 저장 공간을 회수한다. 상위 수준의 로그 구조 저장소들은 엣지 서버가 관리하는 영역에 있는 장치별로 생성된다. 그림의 예에서는 장치 A, B, C에서 생성된 센서 (S1과 S2) 데이터들이 각각의 로그 구조 저장소에서 관리된다.

상위 수준 저장소에서 관리되던 데이터들은 해당 장치가 엣지 서버에 처음으로 추가된 후부터 특정 시간 동안만 관리된다. 본 논문에서는 이 기간을 관리 시간이라 부른다. 장치들은 관리 시간 안에 엣지 서버가 관리하는 영역에서 벗어날 수도 있고 그렇지 않을 수도 있다. 장치가 관리 시간 안에 엣지 서버의 영역에서 벗어난다면 그 장치가 생산한 데이터들은 엣지 서버에서 관리되지 않아도 되며 해당 장치의 데이터를 관리한 로그 구조 저장소의 공간은 한 번에 회수될 수 있다. 만약 장치가 관리 시간 이후에도 엣지 서버의 영역에 있다면 해당 장치의 데이터들은 모두 LSM tree로 재구조화 돼서 관리된다. 앞서 설명한 바와 같이 엣지 노드에 관리된 추가된 최신 장치들은 로그 구조 저장소에 저장이 되고 오래된 장치들은 LSM tree에서 관리되기 때문에 짧은 시간 동안에 엣지 서버에 관리되다가 영역을 이동한 장치들이 생산한 데이터들로 인해 생길 수 있는 쓰기 증폭을 최소화할 수 있다.

```
1: procedure PUT(key, value)
2:     mapping_info = object_mapping_table.get(key);
3:     if mapping_info == NULL then
4:         log_structured_storage = alloc_log_structured_storage(key);
5:         log_structured_storage.put(key, HIGH, log_structured_storage);
6:         mapping_info = alloc_mapping_info(key, value);
7:         object_mapping_table.put(key, mapping_info);
8:     else
9:         object_storage = mapping_info.get_storage();
10:        object_storage.put(key, value);
11:    end if
12: end procedure
13: procedure GET(key)
14:    mapping_info = object_mapping_table.get(key);
15:    if mapping_info == NULL then
16:        return NULL;
17:    else
18:        object_storage = mapping_info.get_storage();
19:        return object_storage.get(key);
20:    end if
21: end procedure
```

그림 6. 제안된 키-값 스토리지 엔진의 연산 처리 절차

Fig. 6. Query Processing in the Proposed Key-Value Storage Engine

그림 5는 상위 수준 저장소의 구조를 보여준다. 상위 수준 저장소는 로그 구조 저장소를 바탕으로 관리된다. 로그 구조 저장소는 연속 쓰기 연산을 활용하기 때문에 데이터 저장 및 변경 성능이 우수하다. 데이터 검색 및 읽기 연산을 효율적으로 하기 위해 장치 별로 데이터 검색 구조를 메모리 상에 추가로 유지하며 데이터 검색 구조는 데이터의 키와 데이터의 로그 구조 저장소 상의 위치를 관리한다. 데이터 검색 구조가 포인트 질의에 최적화되어야 하면 해시 테이블과 같은 구조를 사용하며 범위 검색을 고려하여 트리 구조를 사용할 수 있다.

상위 수준 저장소에 많은 장치가 관리된다면 메모리 사용량이 증가하게 된다. 따라서 적절한 메모리 사용률을 위해 관리 시간 이후에도 유지되어야할 장치의 데이터들은 모두 기존의 키-값 스토리지 엔진이 사용한 LSM tree에서 관리하도록 한다. 그림의 예에서는 장치 C의 데이터가 관리 시간 이후에 유지되어야하고 이 데이터들은 LSM tree에서 관리된다. 이를 위해서 로그 구조 저장소의 데이터를 LSM tree로 삽입한 후에 로그 구조 저장소의 공간과 검색 인덱스 공간을 회수한다.

그림 6은 제안된 기법이 적용된 키-값 스토리지 엔진에서 Put/Get 연산의 처리 절차를 보여준다. Put 연산을 시작하면 오브젝트 매핑 테이블에서 매핑 정보를 추출한다. (2줄) 만약 매핑 정보가 존재하지 않는다면 새로운 장치로 간주하여 로그 구조 저장소를 할당한다. 할당된 저장소에 키와 값을 저장하고 오브젝트 매핑 정보를 생성한다. 그리고 생성된 오브젝트 매핑 정보를 오브젝트 매핑 테이블에 저장한다. (3-7줄) 매핑 정보가 존재하면 매핑 정보에서 로그 구조 저장소 혹은 LSM tree 기반 저장소 정보를 추출하고 해당 저장소에 키와 값을 저장한다. (8-11줄)

Get 연산도 오브젝트 매핑 테이블에서 매핑 정보를 추출하는 것으로 시작한다. (14줄) 만약 매핑 정보가 존재하지 않는다면 해당 키에 대한 값을 저장하고 있지 않으므로 NULL을 리턴한다. (15-16줄) 매핑 정보가 존재하면 매핑 정보에서 로그 구조 저장소 혹은 LSM tree 기반 저장소를 추출하고 해당 저장소로 Get 연산을 전달하여 그 결과를 리턴한다. (17-20줄)

Ⅳ. 실험 및 평가

4-1 실험 환경

제안하는 키-값 스토리지 엔진의 성능을 평가하기 위해 Intel(R) Xeon(R) Core(TM) i7-8700 CPU, 32GB의 메모리, 512 GiB Samsung 970 PRO Flash SSD를 장착한 컴퓨터를 이용하였다. 이 시스템의 CPU는 총 6개의 코어를 가지고 있으며 하이퍼쓰레딩을 활성화하여 12개의 하드웨어 쓰레드 환경에서 성능을 평가하였다. 운영체제는 Linux 커널 5.13을 사용하였다. 본 연구에서 제안한 2단계 키-값 스토리지 엔진을 Google의 LevelDB 상에서 구현하였다. (2Lev-LevelDB) 그리고 2Lev-LevelDB 시스템과 기존의 LevelDB와 (Orig-LevelDB) 비교하였다. 성능 평가를 위해 센서 데이터 벤치마크인 TPCx-IoT 워크로드를 이용하였으며 TPCx-IoT는 99.995%의 Put 연산과 0.005%의 Get 연산으로 구성되어 있다. 키는 장치 및 센서 식별자 그리고 데이터 생성 시간으로 구성되어 있으며 값은 1KB의 랜덤 문자열 데이터이다.

4-2 실험 결과

평가를 위해 쓰레드를 1개부터 12개까지 변화시켜가면서 실험을 수행했다. 이동 장치는 제안하는 키-값 스토리지 엔진의 상위 수준 저장소에서 관리 시간이 끝나기 전에 90%의 확률로 해당 엣지 노드에서 관리하는 영역에서 벗어나는 것을 가정했다.

그림 7은 제안하는 2Lev-LevelDB와 Orig-LevelDB의 성능평가 결과이다. 2Lev-LevelDB는 쓰레드의 수가 8일 때 성능의 가장 우수하며 초당 48만개의 TPCx-IoT 연산을 수행한다. 그러나, Orig-LevelDB는 쓰레드의 수가 1일 때 성능이 가장 좋으며 초당 4만 정도의 TPCx-IoT 연산을 수행한다. 그리고 쓰레드의 수가 증가함에 따라 성능이 떨어지며 쓰레드의 수가 12일 때 초당 2만개의 연산을 수행한다. 2Lev-LevelDB의 경우 쓰레드의 수가 1일 때 Orig-LevelDB보다 성능이 우수한 이유는 2Lev-LevelDB의 상위 수준 저장소가 장치별로 로그 구조 저장소로 관리되기 때문이다. 즉, 쓰기 성능에 있어서 LSM tree 기반 저장소보다 유리하기 때문이다.

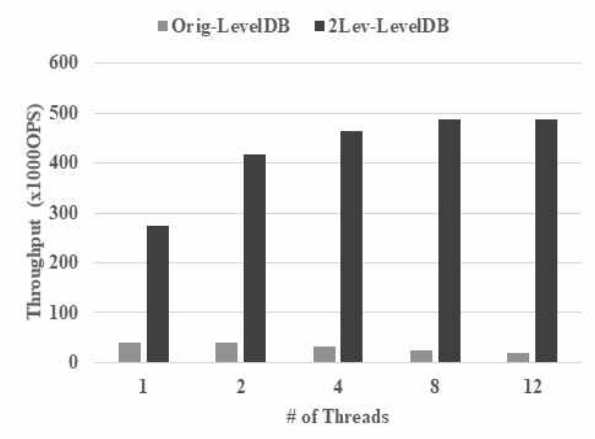

그림 7. TPCx-IoT 성능 평가 결과
Fig. 7. TPCx-IoT Performance Results

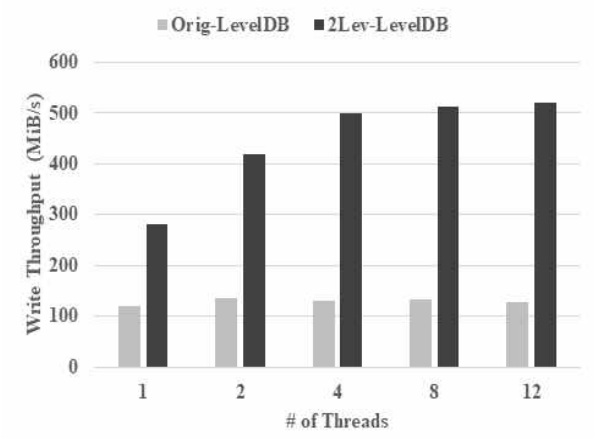

그림 8. TPCx-IoT의 디스크 쓰기 사용량
Fig. 8. Disk Write Throughput for TPCx-IoT

그림 8은 2Lev-LevelDB와 Orig-LevelDB의 저장 장치 대역폭 사용량을 측정한 그래프이다. Orig-LevelDB는 쓰레드의 수와 상관없이 120~130MiB/s 정도의 대역폭을 사용한다. 저장 장치의 대역폭 사용량과 Orig-LevelDB의 성능을 고려해서 쓰기 증폭을 계산하면 쓰레드의 수가 1일 때 3 정도이며 쓰레드의 수가 12로 증가하게 되면 6.4 정도로 커지게 된다. 따라서 이 결과를 통해 기존의 키-값 스토리지 엔진들이 엣지 환경에서 저장 장치의 대역폭의 상당 부분을 쓰기 증폭으로 낭비하는 것을 확인할 수 있다.

반면 2Lev-LevelDB의 경우에는 제안된 2단계 관리 기법으로 입출력 대역폭을 효율적으로 사용하며 쓰기 증폭도 낮다. 쓰레드의 수가 1일 때 279MiB/s 정도의 대역폭을 사용하고 쓰레드의 수가 12일 때 475MiB/s 정도의 대역폭을 사용한다. 그리고 쓰기 증폭은 쓰레드의 수의 상관없이 1.1 정도로 측정되었다.

V. 결 론

본 논문에서는 엣지 컴퓨팅 환경을 고려하여 키-값 스토리지 엔진의 성능을 개선하였다. 기존의 키-값 스토리지 엔진에서 활발히 사용하고 있는 LSM tree는 쓰기 집약적인 워크로드에 적합하지만 엣지 컴퓨팅 환경에서의 이동 장치의 특성을 고려하지 못 하고 있다. 즉 장치가 이동함에 따라 장치를 관리하는 엣지 노드가 변경되는 것을 고려하지 않는다. 따라서 본 연구에서 이동 장치들이 장치별 로그 기반 저장소에서 특정 시간 동안 관리되고 그 시간이 지난 후에도 관리되어야 하는 이동 장치의 데이터를 기존의 LSM tree 기반의 저장소에서 관리하는 2단계 관리 기법을 제안하였다. 그리고 제안한 기법을 Google이 개발한 LevelDB에 구현하였다. 구현된 시스템을 TPCx-IoT 벤치마크를 사용해서 비교 평가하였다. 실험 결과를 통해 제안된 기법이 최대 23배의 성능 향상 효과가 있음을 보였다. 향후에는 제안된 기법을 EdgeX foundry project에 적용하여 성능 평가를 진행할 예정이다.

감사의 글

본 논문은 2021년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것입니다.

참고문헌

[1] Linux Foundation, EdgeX Foundary Project, https://www.edgexfoundry.org

[2] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. A. McGrail, P. Wang, et al., “Apache IoTDB: Time-Series Database for Internet of Things”, VLDB Endowment, Vol. 13, No. 12, pp 2901–2904, August 2020. https://doi.org/10.14778/3415478.3415504

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

[4] P. Raju, R. Kadekodi, V. Chidambaram, I. Abraham, "PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees", the 26th Symposium on Operating Systems Principles (SOSP), pp. 497–514, October 2017. https://doi.org/10.1145/3132747.3132765

[5] F. Pan, Y. Yue, J. Xiong, “dCompaction: Delayed Compaction for the LSM-tree”, International Journal of Parallel Programming, Vol. 45, No. 6, pp. 1310–1325, December 2017. https://doi.org/10.1007/s10766-016-0472-z

[6] F. Mei, Q. Cao, H. Jiang, J. Li, "SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter", the 9th ACM Symposium on Cloud Computing (SoCC), pp.

477-489, October 2018.
https://doi.org/10.1145/3267809.3267829

[7] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, M. Barnett, “FASTER: A Concurrent Key-Value Store with In-Place Updates”, the 2018 ACM International Conference on Management of Data (SIGMOD), pp. 275-190, May 2018.
https://doi.org/10.1145/3183713.3196898

[8] D. Xie, B. Chandramouli, Y. Li, D. Kossmann, “FishStore: Faster Ingestion with Subset Hashing”, the ACM International Conference on Management of Data (SIGMOD), pp 1711-1728, Jume 2019.
https://doi.org/10.1145/3299869.3319896

[9] M. P. Andersen and D. E. Culler, “BTrDB: Optimizing Storage System Design for Timeseries Processing,” the 14th Usenix Conference on File and Storage Technologies, pp. 39-52, February 2016.
https://dl.acm.org/doi/10.5555/2930583.2930587

[10] Y. Yang, Q. Cao, and H. Jiang, “EdgeDB: An Efficient Time-Series Database for Edge Computing,” IEEE Access, Vol.7, pp 142295-142307, September 2019.
https://doi.org/10.1109/ACCESS.2019.2943876

[11] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Wisckey: Separating Keys from Values in SSD-Conscious Storage,” ACM Transactions on Storage, Vol.13, No.1, pp 1-28, February 2017. https://doi.org/10.1145/3033273

[12] H. H. Chan, C. J. M. Liang, Y. Li, W. He, P. P. Lee, L. Zhu, Y. Dong,Y. Xu, Y. Xu, and J. Jianget al., “HashKV: Enabling Efficient Updates in KVStorage via Hashing,” USENIX Annual Technical Conference 2018, pp 1007-1019, July 2018. https://doi.org/10.1145/3340287

[13] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, “Group commit timers and high volume transaction systems,” the 2nd International Workshop on High Performance Transaction Systems, pp. 301-329, 1987. https://doi.org/10.1007/3-540-51085-0_52

[14] R. Ramakrishnan and J. Gehrke, “Database management systems”, Osborne/McGraw- Hill, 2000.

[15] K. Sonbol, O. Ozkasap, I. Al-Oqily, M. Aloqaily, “Edgekv: Decentralized, scalable, and consistent storage for the edge”, Journal of Parallel and Distributed Computing, Vol. 144, pp. 28–40. October 2020.
https://doi.org/10.1016/j.jpdc.2020.05.009

[16] H. Gupta, Z. Xu, U. Ramachandran, “DataFog: Towards a Holistic Data Management Platform for the IoT Age at the Network Edge”, the 2018 USENIX Workshop on Hot Topics in Edge Computing (HotEdge), pp. 1-6 , July 2018.

[17] H. Gupta, U. Ramachandran, “Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access”, the 12th ACM International Conference on Distributed and Event-based Systems, pp. 148–159, June 2018. https://doi.org/10.1145/3210284.3210297

[18] S. Fazackerley, E. Huang, G. Douglas, R. Kudlac, R. Lawrence, “Key-Value Store Implementations for Arduino Microcontrollers”, the 2015 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), pp. 158–164, May 2015,
https://doi.org/10.1109/CCECE.2015.7129178

[19] I. Kopestenski, P. Van Roy, “Achlys: Towards a Framework for Distributed Storage and Generic Computing Applications for Wireless IoT Edge Networks with Lasp on GRiSP”, the 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 875–881, March 2019.
https://doi.org/10.1109/PERCOMW.2019.8730773

[20] Suman Nath and Aman Kansal. "FlashDB: dynamic self-tuning database for NAND flash". the 6th international conference on Information processing in sensor networks (IPSN '07). pp. 410–419, April 2007. https://doi.org/10.1109/IPSN.2007.4379701

한혁(Hyuck Han)

2006년 : 서울대학교 대학원 (공학석사)
2011년 : 서울대학교 대학원 (공학박사 -분산시스템)

1999년～2003년: 델타정보통신
2011년～2012년: 서울대학교
2012년～2014년: 삼성전자
2014년～현 재: 동덕여자대학교 컴퓨터학과 조교수/부교수
※관심분야 : 분산시스템(Distributed System), 운영체제 (Operating System), 데이터베이스시스템 (DBMS) 등