https://doi.org/10.5392/JKCA.2021.21.07.127

산업용 IoT 환경을 위한 고성능 키-값 저장소의 설계 및 평가

Design and Evaluation of a High-performance Key-value Storage for Industrial IoT Environments

한혁
동덕여자대학교 컴퓨터학과

Hyuck Han(hhyuck96@dongduk.ac.kr)

요약

산업용 IoT 환경에서 센서들은 감지하고 있는 대상의 데이터들을 연속으로 생성하며, IoT 게이트웨이에 전달한다. 따라서 대량의 실시간 센서 데이터를 관리하는 것은 IoT 게이트웨이에 필수적인 기능이며 이러한 센서 데이터를 관리하기 위해 키-값 스토리지 엔진들이 널리 사용되고 있다. 그러나 IoT 게이트웨이에 사용되는 키-값 스토리지 엔진들은 산업용 IoT 환경에서 생성되는 센서 데이터들의 특징을 고려하지 않고 있으며 이 때문에 제한된 성능을 보인다. 본 논문에서는 산업용 IoT 환경에서 센서 데이터의 특징을 활용하여 키-값 스토리지 엔진을 최적화한다. 제안하는 최적화 기법은 키-값 스토리지 엔진의 입력인 키를 분리하여 계층적인 색인화 작업을 하는 것이다. 이를 통해 과도하게 발생하는 쓰기 증폭을 줄이며 성능을 향상 시킬 수 있다. 이러한 최적화 기법을 LevelDB에 구현하였으며, 제안하는 기법을 평가하기 위해 TPCx-IoT의 워크로드를 사용했다. 실험 결과에 따르면 제안하는 기법의 성능은 기존의 방법보다 21배 더 좋으며 이는 제안하는 기법이 산업용 IoT 환경에서 데이터 수집을 고속으로 처리할 수 있음을 보인다.

■ 중심어 : | 산업용 사물 인터넷 | 센서 데이터 | 키-값 스토리지 엔진 | LSM Tree | 쓰기 증폭 |

Abstract

In industrial IoT environments, sensors generate data for their detection targets and deliver the data to IoT gateways. Therefore, managing large amounts of real-time sensor data is an essential feature for IoT gateways, and key-value storage engines are widely used to manage these sensor data. However, key-value storage engines used in IoT gateways do not take into account the characteristics of sensor data generated in industrial IoT environments, and this limits the performance of key-value storage engines. In this paper, we optimize the key-value storage engine by utilizing the features of sensor data in industrial IoT environments. The proposed optimization technique is to analyze the key, which is the input of a key-value storage engine, for further indexing. This reduces excessive write amplification and improves performance. We implement our optimization scheme in LevelDB and use the workload of the TPCx-IoT benchmark to evaluate our proposed scheme. From experimental results we show that our proposed technique achieves up to 21 times better than the existing scheme, and this shows that the proposed technique can perform high-speed data ingestion in industrial IoT environments.

■ keyword : | Industrial IoT | Sensor Data | Key-Value Storage Engine | LSM Tree | Write Amplification |

* 이 논문은 2019년도 동덕여자대학교 학술연구비 지원에 의하여 수행된 것임.

접수일자 : 2021년 03월 25일
수정일자 : 2021년 04월 26일
심사완료일 : 2021년 04월 26일
교신저자 : 한혁, e-mail : hhyuck96@dongduk.ac.kr

I. 서 론

IoT 분야는 사용자 중심의 개인용/가정용 IoT 분야 뿐만 아니라, 산업용 IoT 분야도 급속도로 성장하고 있다. 스마트 전력망, 스마트 공장 등 다양한 분야에서 IoT 장치가 사용되고 있으며, 장치가 센서를 통해 생성한 데이터는 실시간으로 수집 및 분석을 통해 유의미한 데이터로 사용된다. 예를 들어 스마트 공장 환경에서 생산 장비에 부착된 센서가 생성하는 데이터를 분석하여 생산 장비가 고장 나기 전에 결함을 미리 예측할 수 있다. 이러한 예측을 위해 각 생산 장비에 장착된 센서 값들을 실시간으로 수집 및 분석할 수 있는 시스템이 필요하다.

산업용 IoT 환경에서 이기종의 장치와 네트워크 환경을 반영하기 위하여 다양한 시스템 구조들이 소개되었는데, 3계층 시스템 구조가 실시간 데이터를 수집 및 분석하기 위한 방법으로 소개되었다[1]. 이 구조는 End tier, Edge tier, Datacenter tier로 구성되어 있다. End tier의 장치들은 센서들을 가지고 있으며 센서들이 생성하는 아날로그 신호를 디지털 데이터로 변환하여 다양한 통신 방법을 통해 Edge tier로 전달한다. Edge tier의 IoT 게이트웨이는 전달받은 센서 데이터를 실시간으로 저장 및 분석한다. 그리고 센서 데이터의 저장을 위해 키-값 스토리지 엔진이 널리 사용되고 있다. Datacenter tier에서는 여러 IoT 게이트웨이에서 관리 중인 데이터들을 일 혹은 주 별로 수집하며 수집된 데이터를 바탕으로 의사 결정을 위한 복잡한 질의 처리가 수행된다.

산업용 IoT에서 각 센서가 생성한 데이터는 시간에 따라 연속적인 특성을 가지고 있다. IoT 게이트웨이의 키-값 스토리지 엔진 관점에서는 센서 데이터는 일반적으로 데이터 생성된 시간, 센서 식별자, 센서 값으로 구성된다. 그리고 한 번 생성되어 저장된 센서 데이터는 다시 변경되지 않으며 실시간 분석을 위해 조회 연산이 이루어진다.

IoT 게이트웨이에서 실시간 대용량의 센서 데이터를 저장하기 사용하는 키-값 스토리지 엔진들은 일반적으로 LSM tree[2]를 기반으로 하고 있으며 LSM tree는 쓰기 집중적인 환경에서 우수한 성능을 보인다. 그러나 기존의 LSM tree 기반의 키-값 스토리지 엔진은 산업용 IoT 환경에서 실시간 센서 데이터를 저장하는 IoT 게이트웨이의 워크로드를 충분히 반영하고 있지 못 하다. 즉, IoT 게이트웨이 관점에서 센서 데이터는 한 번 저장되면 다시 업데이트 되지 않음에도 데이터를 처리하는 LSM tree의 merge 연산으로 인해 같은 센서 데이터가 반복적으로 스토리지 장치에 써진다. 이러한 문제로 IoT 게이트웨이에 장착된 스토리지 장치의 성능을 최대로 활용하지 못 하여 키-값 스토리지 엔진의 성능이 제한되며, 스토리지 장치의 수명이 단축될 수 있다. 본 논문은 이와 같은 문제를 해결하기 위해 센서 데이터의 특성을 반영하여 IoT 게이트웨이의 키-값 스토리지 엔진에 적합한 최적화 기법을 설계하고 구현하고자 한다.

제안하는 기법은 키-값 스토리지 엔진에 저장되는 센서 데이터를 인덱스하는 키가 여러 개로 이루어져 있으며 각각의 키의 속성이 범주형 혹은 숫자형이라는 것을 이용한다. 즉, 센서 데이터가 생성되는 시간은 숫자형이며 개별 시간 혹은 시간 범위를 바탕으로 검색 질의가 처리된다. 반면, 센서 식별자는 개별 센서 혹은 특정 장치의 센서들과 같은 형태로 검색 질의가 처리된다. 따라서 이러한 상이한 키 속성들을 활용하기 위해 인덱스를 계층화하여 상위 수준 인덱스의 각 리프 노드가 하위 수준 인덱스의 루트를 가리키도록 한다. 즉, 상위 수준 인덱스의 검색 키로 분할 저장된 센서 데이터는 하위 수준 인덱스 키로 인덱싱된다. 저장 장치에서 센서 데이터를 저장할 때 두 가지 유형의 속성을 키-값 스토리지 엔진에서 하나의 복합 키가 아닌 개별 검색 키로 처리하는 것이다. 본 논문의 기여는 다음과 같다.

먼저, 산업용 IoT 환경에서 사용하는 키-값 스토리지 엔진에서 발생하는 쓰기 증폭을 줄이는 최적화 기법을 제안하였다. 다음으로는 제안된 기법을 구글의 LevelDB[3]에 구현하였고, 구현된 시스템과 산업용 IoT 환경의 표준 벤치마크인 TPCx-IoT[4]를 이용하여 제안하는 기법이 최대 21배 정도 성능 향상이 있음을 보였다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구와 관련된 다른 연구에 대해 설명한다. 3장에서는 키-값 스토리지 엔진을 위한 최적화 기법을 설명한다. 4장에

서는 구현한 키-값 스토리지 엔진의 성능 결과를 보여주며 5장에서는 향후 연구 방향을 제기하고 논문의 결론을 맺는다.

II. 관련 연구

키-값 스토리지 엔진의 성능을 개선하기 위한 많은 연구들이 있으며 그 연구들은 대부분 1) LSM tree의 merge의 오버헤드를 줄이거나 2) 커밋 처리 오버헤드를 줄이는 것으로 나눌 수 있다. [5][6]의 연구에서는 데이터 수집 처리량을 향상시키기 위해 로그 구조 해시 테이블을 제안했다. 이러한 구조에서 들어오는 키-값 데이터는 정렬 없이 로그 파일에 추가되며, 로그 파일에서 데이터를 쉽게 검색할 수 있도록 메인 메모리 내에 해시 인덱스가 유지된다. 그러나 데이터가 많아지면 메모리 내 해시 인덱스 크기가 급격히 증가한다. 또한 범위 질의 처리의 어려움으로 인해 산업용 IoT 환경에서 사용하기가 어렵다. [7][8]의 연구에서는 센서 데이터를 시간 간격별로 분할하고 모든 파티션에 대해 메모리 내 인덱스 구조를 유지하는 시계열 데이터베이스를 제안했다. 시간 범위 질의 처리에는 유리하지만 인덱스 구조를 위해서는 큰 메모리 공간이 필요하다. [9][10]의 연구에서는 merge 오버헤드를 줄이기 위해 키-값 데이터는 로그 파일에 추가되며 LSM tree에서는 키와 해당 데이터의 로그 파일에서의 위치를 관리한다. 그러나 LSM tree의 키 순서가 로그 파일에 저장된 데이터 순서와 동일하다는 것이 보장되지 않기 때문에 시간 범위 질의를 처리할 때 제한된 성능을 얻는다.

키-값 스토리지 엔진의 커밋 처리 오버헤드를 줄이는 기법[11-13]들이 제안되었다. 그룹 커밋[11]은 커밋 연산을 수행하기 전에 대기 시간을 두어 여러 커밋 연산들의 로그 요청들을 하나의 로그 입출력 연산으로 처리한다. 비동기식 커밋[12]은 로그 입출력 연산이 완료될 때까지 기다리지 않고 트랜잭션을 완료하거나 다른 트랜잭션을 처리하도록 허용하여 트랜잭션 처리 성능을 개선한다. Aether[13]는 커밋 처리 연산을 고속으로 수행하기 위해 로그 데이터를 버퍼로 복사할 때 잠금을 사용하지 않게 하여 커밋 처리 성능을 높여준다. 이러한 연구들은 키-값 스토리지 엔진의 성능 개선이라는 측면에서 유사하지만, 본 논문과 같이 산업용 IoT 환경의 센서 데이터의 특징을 고려한 인덱스 구조를 제시하지는 않는다.

III. 센서 데이터를 고려한 인덱스 기법

1. 설계 동기

이번 장에서는 키-값 스토리지 엔진이 사용하는 LSM tree 구조와 센서 데이터를 저장하는 키-값 스토리지 엔진의 쓰기 증폭의 원인을 설명한다. 키-값 스토리지 엔진은 데이터를 키-값 쌍으로 데이터를 읽고 저장하는 저장소이다. 키는 데이터의 식별자이며, 값은 저장소에 저장할 데이터를 포함한다. [그림 1]는 LSM tree를 사용한 키-값 스토리지 엔진의 기본적인 형태를 보여주고 있다. LSM tree는 일반적으로 memtable이라고 부르는 in-memory 쓰기 버퍼를 가지고 있으며 memtable은 입력되는 키-값 데이터를 키로 정렬하여 관리한다. memtable에 여분의 공간이 없으면 메모리에 관리하는 데이터를 Stored Sequence Table (SST) 파일에 쓴다. SST 파일은 가장 최근 기록된 데이터부터 레벨 0에 생성되며, 각 레벨의 저장 크기 제한을 초과하면 다음 레벨의 SST 파일과 merge 과정을 수행하게 된다. 즉, merge 작업은 현재 레벨과 다음 레벨의 데이터를 읽고 병합하여 다시 디스크에 쓰는 작업이기 때문에 쓰기 증폭을 발생시킨다.

[그림 2]은 ‘A’, ‘B’, ‘C’ 센서가 생성하는 데이터를

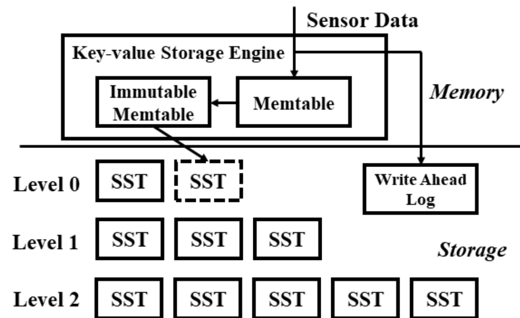

그림 1. LSM tree 구조

LSM tree 기반의 키-값 스토리지 엔진에 저장하는 과정을 보여준다. 이 그림 예에서 키만 표시하였으며 키는 센서 식별자('A', 'B', 'C')와 데이터 수집 시간의 조합이 되며 값은 센서 데이터이다. 시간 4에서 발생한 센서 데이터들이 memtable을 다 쓰게 되면, 레벨 0의 SST 파일의 데이터와 합친다. 합치는 과정에서 SST 파일의 데이터들을 정렬하여 다시 저장한다. 이 때, 시간 3에서 발생한 센서 데이터들은 이미 저장 장치에 써져 있고, 다시 merge 연산을 통해서 저장장치의 다른 위치에 다시 써지게 된다. 시간 3에서 발생한 센서 데이터들은 변경이 없음에도 LSM tree의 merge 연산 때문에 같은 데이터가 여러 번 써지게 되는 쓰기 증폭을 야기하게 된다.

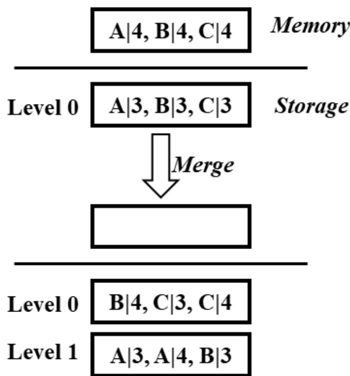

그림 2. 키-값 스토리지 엔진의 쓰기 증폭
(키만 표시하였으며 A, B, C는 센서 식별자, 숫자는 시간)

2. 최적화된 인덱스 구조 설계

[그림 3]는 IoT 환경을 고려하는 최적화된 인덱스 구조를 보여준다. 제안하는 구조는 해시 테이블 구조를 사용하여 상위 수준의 인덱스 구조를 구성한다. 그림에서는 상위 수준의 인덱스가 하나만 있지만 여러 개로 계층적으로 구성할 수 있다. 예를 들어, 스마트 공장 환경에서는 센서 식별을 위해 공장 식별자, Part 식별자, 머신 식별자, 센서 식별자 등으로 해시 테이블을 계층적으로 구성할 수 있다. 즉, 공장 매핑 테이블, Part 매핑 테이블, 머신 매핑 테이블, 센서 매핑 테이블이 각 인덱스 계층에서 해시 테이블로 사용된다. 키-값 스토리지 엔진은 여러 키를 인코딩한 복합키에서 공장 식별자, Part 식별자, 머신 식별자, 센서 식별자 등을 추출하여 각 테이블의 해시 키로 사용한다. 센서 데이터는 기존의 LSM tree로 되어 저장소에 저장된다. 따라서 기존 LSM tree과 마찬가지로 데이터를 고성능으로 쓸 수 있으며, [5][6][9][10]의 연구와는 다르게 범위 질의 처리도 LSM tree와 동일하다.

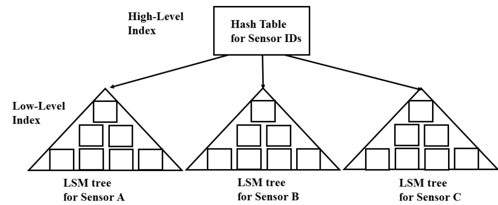

그림 3. 최적화된 인덱스 구조

이와 같은 계층적 구조에서는 수집된 데이터를 각 인덱스 도메인의 인덱스 키로 분류한다. 이러한 분류는 센서 데이터를 범주형 속성별로 분할한다. 동일한 상위 계층 식별자를 가진 모든 데이터만 해당 도메인 식별자의 하위 수준 인덱스 구조에 의해 관리된다. 예를 들어 Part 도메인에서 데이터는 머신 식별자 별로 분류되며, 머신 식별자가 동일한 데이터만 해당 머신에 대한 인덱스 구조로 전달되어 관리된다. 동일한 센서 데이터들은 하나의 LSM tree에 생성된 시간 순서대로 저장된다. 이와 같이 저장되면 LSM tree에 저장되는 데이터가 모두 정렬되어 있게 되며, 모든 SST 파일들 사이에 겹치는 영역이 생기지 않게 되어 merge 연산 중에 파일을 읽고 정렬하여 쓰는 과정을 수반하지 않고 SST 파일들의 레벨만 조정함으로써 merge 연산이 완료될 수 있다. 이를 통해 merge 시에 발생할 수 있는 쓰기 증폭을 획기적으로 줄일 수 있다. [그림 4]는 센서 A를 위한 LSM tree의 예를 보여준다. 시간 6에서 발생한 센서 데이터가 memtable에 쓰게 되면 memtable을 다 사용하게 된다. 이때, 레벨 0의 SST 파일의 데이터와 memtable의 데이터를 합치지 않고, memtable의 데이터를 레벨 0의 SST 파일에 쓰고, 레벨 0의 SST 파일은 레벨 1로 레벨만 조정함으로써 merge 연산을 끝낼 수 있고, 기존의 스토리지 장치에 쓰여 있던 데이터가 다시 써지는 쓰기 증폭 현상을 막을 수 있다.

[그림 5]은 제안된 기법에서 키-값 스토리지 엔진의

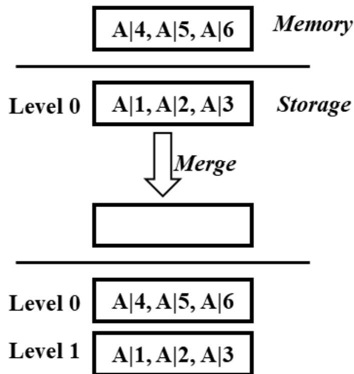

그림 4. 제안된 기법에서의 merge 연산

```
1: procedure PROCESS_PUT(k,v)
2:     sensor_id = extract_sensor_id(k);
3:     gen_time = extract_generation_time(k);
4:     if contains(sensor_mapping_table, sensor_id) = true then;
5:         sensor_tree = get(sensor_mapping_table, sensor_id);
6:         put(sensor_tree, gen_time, v);
7:     else
8:         sensor_tree = create_tree();
9:         put(sensor_tree, gen_time, v);
10:        put(sensor_mapp_table, sensor_id, sensor_tree);
11:    end if
12: end procedure
13: procedure PROCESS_GET(k)
14:    sensor_id = extract_sensor_id(k);
15:    gen_time = extract_generation_time(k);
16:    if contains(sensor_mapping_table, sensor_id) = true then;
17:        sensor_tree = get(sensor_mapping_table, sensor_id);
18:        return get(sensor_tree, gen_time);
19:    else
20:        return NULL;
21:    end if
22: end procedure
```

그림 5. 제안된 키-값 스토리지 엔진의 연산 처리 절차

주요 연산인 Put/Get의 절차를 보여준다. Put 연산을 시작하면 제안된 기법에서는 각 인덱스 계층별 식별자들을 모두 추출한다. (2-3줄) 만약 상위 계층에서 하위 계층 인덱스 구조를 찾을 수 있다면, 하위 계층 인덱스 구조로 Put 연산을 전달한다. 즉, 센서별 LSM tree에 센서 데이터의 생성 시간을 키로 하여 센서 데이터를 저장한다. (줄 5-6) 만약 하위 인덱스를 찾을 수 없다면 하위 인덱스를 생성하고 하위 인덱스에 Put 연산을 전달한다. 그리고 상위 인덱스에 생성된 하위 인덱스를 Put 연산을 이용해서 저장한다. (줄 8-10)

Get 연산도 Put 연산과 마찬가지로 각 인덱스 계층별 식별자들을 모두 추출한다. (14-15줄) 만약 상위 계층에서 하위 계층 인덱스 구조를 찾을 수 있다면, 하위 계층 인덱스 구조로 Get 연산을 전달하여 리턴한다. (17-18줄) 만약 하위 인덱스를 찾을 수 없다면 존재하지 않는 데이터이므로 널을 리턴한다. (20줄)

앞서 살펴본 하위 인덱스가 생성되는 경우는 새로운 센서, 머신, Part의 추가를 의미하므로 실제 산업용 IoT 환경에서 드물게 일어날 것이다. 즉, 상위 수준 인덱스 구조의 압도적인 연산은 Get 연산임을 알 수 있으며, 상위 수준 인덱스가 해시 테이블이므로 모든 Get 연산은 상수 시간 (O(1)) 안에 처리할 수 있다. 그리고 readers-writer lock[14] 혹은 read-copy update[15] 기법을 사용하여 상위 수준 인덱스의 연산의 병행성을 높인다.

IV. 성능 평가

최적화된 인덱스 구조를 평가하기 위해 4개의 Intel(R) Xeon(R) Gold 6152 CPU, 512GB의 메모리, 2.0 TiB Intel DC P4510 flash SSD를 장착한 서버를 이용하였다. 이 시스템은 총 80개의 코어를 가지고 있으며 하이퍼쓰레딩을 비활성화하였다. 운영체제는 Linux 커널 4.1.7을 사용하였다. 본 연구에서 제안한 인덱스 구조를 Google의 LevelDB에 구현하였으며 (Opt-LevelDB) 구현된 시스템과 기존의 Level (Def-LevelDB)과 비교하였다. 성능 평가를 위해 TPCx-IoT 벤치마크의 워크로드를 이용하였다. 이 워크로드는 99.995%의 센서 데이터 저장(Put) 연산과 0.005%의 조회(Get) 연산으로 이루어져 있다. 키는 센서 식별자와 센서 데이터의 생성 시간의 복합 키이며 값은 1KB의 랜덤 데이터이다.

실험을 위해 작업 쓰레드는 1개부터 64개까지 변화시켜가며 수행하였으며 쓰레드당 1개 혹은 10개의 센서에서 생성되는 데이터를 처리하도록 하였다. [그림 6]은 쓰레드당 1개의 센서의 데이터가 처리하는 경우의 성능 결과이다. Opt-LevelDB는 64 쓰레드일 때 성능이 가장 우수하며 초당 63만개의 TPCx-IoT 연산을 수행한다. 반면, Def-LevelDB는 1 쓰레드일 때 성능이 가장 우수하며 초당 약 15만개의 연산을 수행하고 쓰레드의 수가 증가할수록 오히려 성능이 감소하게 되어 64 쓰레드일 때는 초당 3.3만개의 연산을 수행한다.

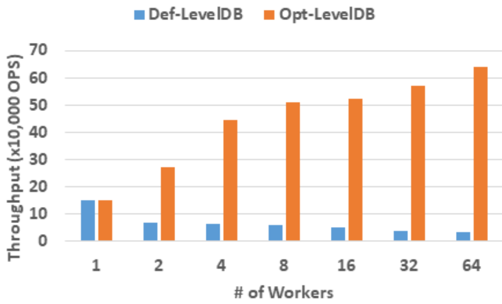

그림 6. TPCx-IoT 성능 평가 결과
(쓰레드당 1개의 센서에서 생성되는 경우)

Opt-LevelDB와 Def-LevelDB 모두 쓰레드가 1일 때는 성능이 비슷하며, 이 경우는 1개의 센서에서 생성되는 데이터만을 처리하여 merge 오버헤드가 없기 때문이다. Def-LevelDB는 여러 개의 센서에서 생성되는 데이터가 센서별로 구별되지 않아서 쓰기 증폭이 생기며 성능 또한 하락하게 된다. [그림 7]은 Opt-LevelDB와 Def-LevelDB의 스토리지 사용량을 보여준다. Def-LevelDB는 쓰레드의 수가 2 이상부터 200-300MB/s 정도 사용한다. 이는 Def-LevelDB의 성능 추이를 고려하면 쓰기 스토리지 사용량의 상당 부분이 쓰기 증폭으로 낭비되는 것으로 확인할 수 있다. 반면 Opt-LevelDB의 경우에는 제안된 기법으로 입출력 대역폭의 낭비를 줄여서 성능 개선 효과를 가지는 것을 알 수 있다.

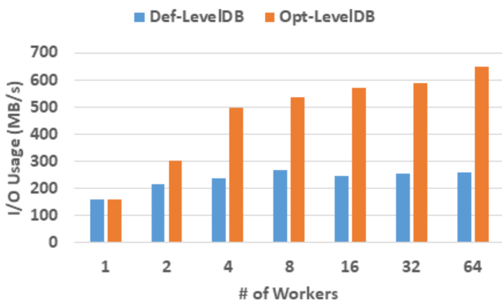

그림 7. TPCx-IoT의 입출력 사용량
(쓰레드당 1개의 센서에서 생성되는 경우)

[그림 8]는 쓰레드당 10개의 센서의 데이터들이 처리되는 경우의 성능 결과이다. Opt-LevelDB는 [그림 7]의 성능 추이 결과와 유사하며 초당 60만개의 연산을 수행한다. Def-LevelDB는 [그림 7]의 경우보다 더 나빠지며 이것은 센서의 수가 증가함에 따라 쓰기 증폭의 정도가 더 커지게 되기 때문이다. 64 쓰레드일 때 초당 2.7만개의 연산을 처리한다. 1개의 쓰레드일 때는 초당 6.4만개의 연산을 수행하며 [그림 7]의 경우보다 43% 정도 하락한 성능을 보인다. 이것은 10개의 센서 데이터가 섞이면서 쓰기 증폭을 야기하기 때문이다.

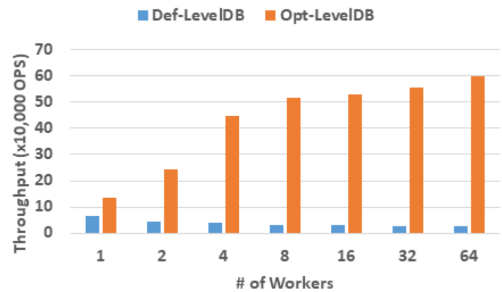

그림 8. TPCx-IoT 성능 평가 결과
(쓰레드당 10개의 센서에서 생성되는 경우)

V. 결론

본 논문에서는 산업용 IoT 분야의 IoT 게이트웨이에서 사용되고 있는 키-값 스토리지 엔진의 성능을 개선하였다. 키-값 스토리지 엔진에서 사용하고 있는 LSM tree 구조는 쓰기 집중적인 워크로드에 적합하지만 산업용 IoT 환경의 센서 데이터를 처리할 때는 추가적인 최적화 기법이 요구된다. 이를 위해 센서 데이터가 저장될 때 사용하는 키를 분해하여 센서별로 LSM tree로 구성하면 쓰기 증폭을 줄였다. 제안된 기법을 Google의 LevelDB에 구현하였다. TPCx-IoT 벤치마크의 워크로드를 이용하여 제안하는 기법을 평가하였으며 평가 결과에 따르면 제안하는 기법이 최대 21배의 성능 향상 효과를 보여준다. 향후에는 제안된 최적화 기법을 센서 데이터 수집 분야에서 사용되고 있는 시계열 데이터베이스 시스템과 같은 완전한 시스템에 적용하여 성능 평가를 수행할 예정이다.

참 고 문 헌

[1] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, “Industrial Internet of Things: Challenges, Opportunities, and Directions,” IEEE Transactions on Industrial Informatics, Vol.14, No.11, 2018.

[2] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. “The log-structured merge-tree (LSM-tree),” Acta Inf., Vol.33, No.4, 1996.

[3] https://github.com/google/leveldb

[4] M. Poess, R. Nambiar, K. Kulkarni, C. Narasimhadevara, T. Rabl, and H. Jacobsen, “Analysis of TPCx-IoT: The First Industry Standard Benchmark for IoT Gateway Systems,” IEEE ICDE, 2018.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, “FASTER: A Concurrent Key-Value Store with In-Place Updates,” ACM SIGMOD, 2018.

[6] D. Xie, B. Chandramouli, Y. Li, and D. Kossmann, “FishStore: Faster Ingestion with Subset Hashing,” ACM SIGMOD, 2019.

[7] M. P. Andersen and D. E. Culler, “BTrDB: Optimizing Storage System Design for Timeseries Processing,” USENIX FAST, 2016.

[8] Y. Yang, Q. Cao, and H. Jiang, “EdgeDB: An Efficient Time-Series Database for Edge Computing,” IEEE Access, Vol.7, 2019.

[9] 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, 2017.

[10] 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 ATC, 2018.

[11] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, “Group commit timers and high volume transaction systems,” HPTS, 1987.

[12] R. Ramakrishnan and J. Gehrke, *Database management systems,* Osborne/McGraw- Hill, 2000.

[13] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki, “Aether: a scalable approach to logging,” VLDB, 2010.

[14] B. B. Brandenburg and J. H. Anderson, “Reader-Writer Synchronization for Shared-Memory Multiprocessor Real-Time Systems,” 21st Euromicro Conference on Real-Time Systems, 2009.

[15] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole, “RCU Usage In the Linux Kernel: Eighteen Years Later,” SIGOPS Oper. Syst. Rev. Vol.54, 2020.

저 자 소 개

한 혁(Hyuck Han) 정회원

▪ 2003년 8월 : 서울대학교 컴퓨터공학부(공학사)
▪ 2006년 2월 : 서울대학교 컴퓨터공학부(공학석사)
▪ 2011년 2월 : 서울대학교 컴퓨터공학부(공학박사)
▪ 2011년 3월 ~ 2012년 8월 : 서울대학교 컴퓨터공학부 박사후 연구원
▪ 2012년 9월 ~ 2014년 2월 : 삼성전자 메모리 사업부 책임연구원
▪ 2014년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 조교수/부교수

<관심분야> : 데이터베이스 시스템, 병렬 프로그래밍, 분산 시스템