스토리지 분리화 환경에서 스토리지 서버 기반 키-값 스토어 설계 및 구현

박여현[1], 이창규[1], 김경표[2], 박성순[2,3], 김영재[1]
[1]서강대학교 컴퓨터공학과, [2]글루시스, [3]안양대학교 컴퓨터공학과
{yeohyeon, changgyu, youkim}@sogang.ac.kr, {kpkim, sspark}@gluesys.com

Design and Implementation of a Storage Server-based Key-Value Store in a Storage Disaggregation Environment

Yeohyeon Park[1], Chang-Gyu Lee[1], Kyungpyo Kim[2], Sung-Soon Park[2,3], Youngaje Kim[1]
[1]Sogang University, [2]Gluesys, Anyang University

요 약

스토리지 분리화는 계산 노드와 스토리지 노드를 분리하는 클러스터 구축 기법으로 스토리지 노드 관리가 용이할 뿐만 아니라 스토리지 확장성이 높은 장점을 가진다. 스토리지 분리화 환경에서 계산 노드는 스토리지 노드와 고속의 네트워크로 연결되며 NVMe-oF (NVMe-over Fabrics) 프로토콜을 사용하여 스토리지 노드의 블록 디바이스에 I/O를 수행한다. 본 연구에서는 이러한 스토리지 분리화 환경에서 고속의 스토리지 노드 기반 키-값 스토어 (Key-Value Store) 설계를 제안한다. 본 연구에서 제안하는 설계 구조는 키-값 스토어 엔진이 계산 노드에서 구동되는 전통적인 계산 노드 기반 구조에서 탈피하여 스토리지 노드에서 키-값 스토어 엔진을 구동한다. 사용자는 경량의 키-값 API를 사용하여 원격 키-값 스토어로 키-값 요청을 수행한다. 특히 경량의 스토리지 엔진 구현을 위해 간단한 Hash 구조 기반 인덱스 구조를 채택하였으며 SPDK (Storage Performance Development Kit)에 BDEV 모듈로 구현하였다. 제안하는 구조는 계산 노드의 커널 I/O 스택을 완전히 우회하여 높은 I/O 성능을 제공한다. 실험 평가를 위해 dbbench의 쓰기 워크로드인 'Fillsequential'에 대하여 계산 노드 기반 RocksDB와 비교 평가한 결과, 제안하는 스토리지 노드 기반 키-값 스토어가 쓰기 응답시간에서 매우 우수한 성능 향상을 보였다.

1 서론

전통적인 데이터센터는 CPU, 메모리, 가속기, 스토리지를 모두 포함하는 서버들을 네트워크로 연결한 클러스터 구조로 되어 있다. 하지만 최근 데이터센터 규모가 커짐에 따라 기존 데이터 센터의 클러스터 구조는 확장성 한계에 직면하고 있다 [1, 2]. 따라서 모든 자원들을 단일 서버에 포함하는 서버 중심 (Server-centric) 구조에서 각 서버를 자원의 특성에 따라 클러스터를 파티셔닝하여 자원의 풀을 구성하는 자원 중심 (Resource-centric) 구조로의 전환 시도가 활발하게 진행되고 있다. 특히, 스토리지 분리화 (Storage Disaggregation)는 계산 노드와 스토리지 노드를 물리적으로 분리하여 각각 계산 노드 풀과 스토리지 노드 풀로 구성한다. 계산 노드와 스토리지 노드는 고속의 네트워크로 연결되며, 계산 노드는 NVMe-oF 프로토콜을 사용하여 스토리지 노드에 읽기와 쓰기를 수행한다.

한편, 최근에는 빅데이터 저장소로 간단한 키-값 인터페이스를 제공하며 저장 및 관리가 용이한 키-값 스토어가 활발하게 사용되고 있다. 앞서 언급한 스토리지 분리화 환경에서 사용자는 전통적인 키-값 스토어를 계산 노드에서 구동하고 NVMe-oF를 통하여 네트워크 I/O를 수행한다. 최근 RocksDB [3]나 LevelDB [4]와 같은 LSM-tree [5] 기반 키-값 스토어는 쓰기에 최적화된 빅데이터 저장소로 널리 사용되고 있다. 스토리지 분리화 환경에서 전통적으로 LSM-tree 기반 키-값 스토어는 계산 노드에서 구동되며 다음과 같은 이유들로 높은 처리 지연시간과 꼬리 지연시간 (Tail Latency) 문제를 가진다.

- 첫째, 키-값 스토어는 계산 노드의 파일시스템 상에서 구동되므로 I/O 수행시 계산 노드의 커널 I/O 스택 (파일시스템, 블록계층)을 통과하는 컨텍스트 스위치와 인터럽트 처리 오버헤드를 가진다.
- 둘째, LSM-tree 기반 키-값 스토어의 빈번한 컴팩션 작업 수행은 I/O 증폭 문제를 동반하여 I/O 성능 저하를 초래한다.

본 논문에서는 앞서 언급한 전통적인 계산 노드 기반 키-값 스토어의 I/O 성능 문제를 최소화 하기 위해 **스토리지 노드 기반 키-값 스토어** 설계를 제안한다. 우리가 제안하는 스토리지 노드 기반 키-값 스토어는 스토리지 엔진이 스토리지 노드에서 구동되기 때문에 계산 노드의 커널 영역 I/O 스택 오버헤드를 회피할 수 있다. 또한, 스토리지 노드에서 경량의 Hash 기반 스토리지 엔진을 구동하여 기존 LSM-tree (Log-Structured Merge-tree) 기반의 키-값 스토어의 컴팩션 작업으로 인해 발생하던 I/O 증폭을 최소화 한다. 우리가 제안하는 시스템의 프로토타입을 위하여 (i) 사용자가 계산 노드에서 키-값 요청을 수행할 수 있는 키-값 API 개발, (ii) NVMe-oF 상에 키-값 명령을 전달할 수 있는 NVMe 스펙 확장, (iii) 스토리지 노드의 SPDK (Storage Performance Development Kit) [6] 소프트웨어 계층에 Hash 기반 키-값 스토리지 엔진을 직접 개발하였다. 우리가 제안 시스템의 성능 평가를 위해 대표적인 키-값 스토어인 RocksDB의 벤치마크 툴인 dbbench를 사용하였다. 본 논문에서 제안하는 스토리지 엔진에 사용되는 NVMe-oF 기반 키-값 요청을 지원하기 위해 dbbench가 키-값 API를 사용하도록 수정하였고 'Fillsequential' 쓰기 워크로드를 사용하여 평가하였다. 그 결과, 본 연구에서 제안하는 스토리지 노드 기반 키-값 스토어는 계산 노드 기반 RocksDB 보다 지연시간 측면에서 매우 높은 성능 향상을 보였다.

이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2020-0-00104, 저지연 입출력 집약적 엣지 데이터 처리를 위한 스토리지 모듈 기술 개발)

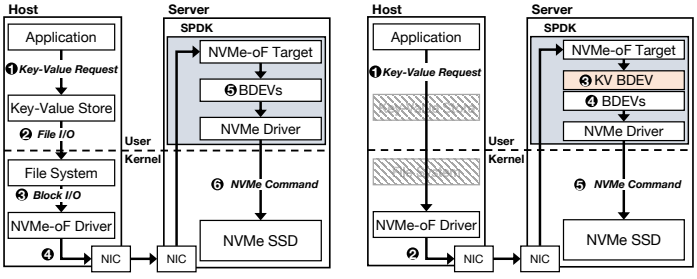
2 배경 지식

2.1 키-값 스토어

키-값 데이터는 단순한 키-값 메소드를 사용하는 비관계형 데이터이다. 키-값 인터페이스를 제공하는 키-값 스토어는 대용량 데이터의 저장과 조회가 편리하여 널리 사용되고 있다. 특히, 높은 쓰기 성능을 지원하는 LSM-tree기반의 키-값 스토어인 RocksDB 및 LevelDB의 사용이 증가하고 있다. LSM-tree 기반의 키-값 스토어 [4, 3, 7]는 빠른 쓰기 성능을 제공하기 위해 메모리에 MemTable을 두고 키-값을 캐싱한다. MemTable의 크기가 임계값을 넘으면 영구저장장치로 일괄 플러시 (Flush) 한다. Memtable은 SSTable (Sorted String Table)이라는 인덱스 파일 형태로 파일 시스템에 저장된다. SSTable은 각 레벨 (Level)에 저장되어 있으며 가장 최근에 플러시 된 SSTable은 레벨 0에 위치한다. 레벨은 수용 가능한 SSTable의 총합 크기가 정해져 있으며, 레벨이 높아수록 수용 가능한 SStable의 총합 크기가 증가한다. 컴팩션 (Compaction)은 SStable을 병합하는 과정이다. 레벨에 저장된 SSTable이 레벨의 크기를 넘어서면 컴팩션을 통해 다음 레벨에 SSTable와 병합한다.

2.2 Storage Performance Development Kit (SPDK)

SPDK [6]는 스토리지 응용의 소프트웨어 오버헤드를 줄이기 위해 제안된 소프트웨어 툴과 라이브러리를 제공한다. SPDK는 커널 I/O 스택 오버헤드인 컨텍스트 스위치와 인터럽트를 회피하기 위해 NVMe 드라이버를 사용자 영역으로 이동시키고 폴링 (Polling) 기반의 I/O 처리 방식을 채



(a) 계산 노드 기반 설계 (b) 스토리지 노드 기반 설계

그림 1: 키-값 스토어 구조

택하여 높은 I/O 성능을 제공한다. SPDK의 I/O 처리는 기본 모듈 단위인 BDEV의 집합으로 정의되며 각 BDEV는 스토리지 서비스의 불가분한 최소 기능들을 정의한다. BDEV는 크게 두 종류로 나누어지며 NVMe-oF 타겟, NVMe 드라이버 등 실질적 I/O 처리를 담당하는 BDEV와 Compression과 같이 다른 BDEV에 I/O를 변환 및 전달하는 VBDEV로 구성된다. 사용자가 SPDK I/O 처리 과정 중에 특정 기능을 추가하고자 하는 경우, NVMe-oF 타겟과 NVMe 드라이버 BDEV들의 사이에 사용자 정의 BDEV를 생성하여 연결한다.

3 스토리지 노드 기반 키-값 스토어 설계 및 구현

3.1 계산 노드 기반 키-값 스토어

그림 1는 두가지 NVMe-oF 기반 키-값 스토어의 설계 구조를 보여준다. 그림 1(a)는 전통적인 계산 노드 기반 키-값 스토어 구조이며 그림 1(b)는 본 연구에서 제안하는 스토리지 노드 기반 키-값 스토어 구조이다.

그림 1(a)에서 사용자의 키-값 요청은 다음과 같은 과정으로 수행된다. ❶ 응용은 키-값 스토어 (Key-Value Store)에 읽기 또는 쓰기 키-값 요청을 전달한다. ❷ 키-값 스토어는 키-값 요청을 파일 I/O 요청으로 변환하여 커널 영역 파일 시스템에 전달한다. ❸ 파일 시스템은 파일 I/O 요청을 블럭 I/O 요청으로 변환하여 NVMe-oF 드라이버에 전달한다. ❹ NVMe-oF 드라이버는 블럭 I/O를 NVMe 명령어로 변환한 뒤, NVMe-oF 프로토콜을 이용하여 스토리지 노드에 전달한다. ❺ 스토리지 노드는 SPDK를 사용하여 NVMe-oF 타겟과 NVMe 드라이버를 사용자 공간에서 구동한다. 따라서 스토리지 노드에 도착한 NVMe 명령어는 SPDK 내 NVMe-oF 타겟으로 전달된다. BDEV는 NVMe-oF 타겟으로부터 전달받은 블럭 I/O를 사용자 공간 NVMe 드라이버로 요청한다. ❻ NVMe 드라이버는 NVMe SSD에 블럭 I/O를 전달한다. NVMe SSD가 블럭 I/O 처리를 완료하면 동일한 경로를 거쳐 응용에 완료를 알린다.

3.2 스토리지 노드 기반 키-값 스토어

그림 1(b)는 본 논문에서 제안하는 스토리지 노드 기반 키-값 스토어 설계를 보여준다. 사용자의 키-값 요청은 다음과 같은 과정으로 수행된다.

❶ 응용은 키-값 API를 이용하여 NVMe-oF 드라이버에 키-값 요청을 전달한다. 그림2(a)의 계산 노드 기반 키-값 스토어와 달리 키-값 스토어와 커널 영역 파일 시스템을 거치지 않는다. ❷ NVMe-oF 드라이버는 키-값 요청을 키-값 NVMe 명령어로 변환한 뒤, NVMe-oF 프로토콜을 이용하여 스토리지 노드에 전달한다. ❸ 스토리지 노드에 도착한 키-값 NVMe 명령어는 SPDK 내 NVMe-oF 타겟을 지나 스토리지 노드 기반 키-값 스토어인 KVBDEV로 전달된다. KVBDEV는 키-값 NVMe 명령어를 블럭 I/O로 변환하여 NVMe BDEV에 전달한다. ❹ NVMe BDEV는 전달받은 블럭 I/O를 사용자 공간 NVMe 드라이버로 요청한다. ❺ NVMe 드라이버는 NVMe SSD에 블럭 I/O를 전달한다. NVMe SSD가 블럭 I/O 처리를 완료하면 동일한 경로를 거쳐 응용에 완료를 알린다.

그림 2는 키-값 스토리지 엔진 BDEV가 포함된 SPDK 구조와 동작을 설명한다. SPDK는 코어에 이벤트를 할당하여 처리한다. 각 코어에는 이벤트 루프가 할당되며, 이벤트 루프는 리엑터, 이벤트 큐로 구성된다. 리엑터는 코어의 쓰레드로 이벤트를 처리한다. 각 BDEV는 이벤트 형태로 이벤트 큐에 삽입되어 실행된다.

SPDK에 KVBDEV가 추가된 후 I/O가 처리되는 과정은 다음과 같

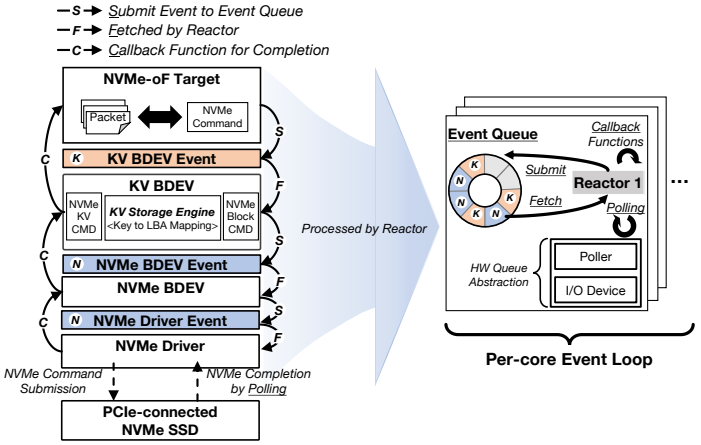


그림 2: SPDK 기반 키-값 스토리지 엔진 구조

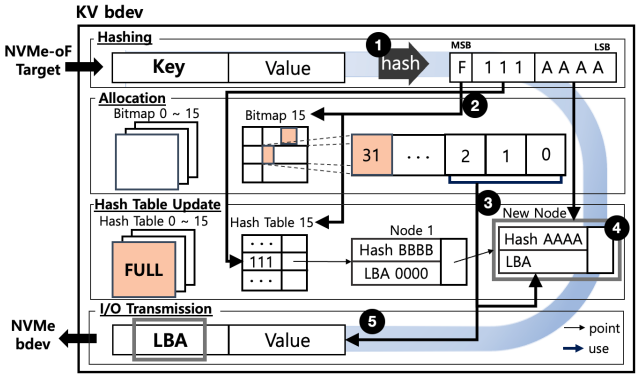


그림 3: KVBDEV 설계 및 동작 방식

다. KVBDEV는 NVMe-oF 타겟과 NVMe BDEV 사이에 연결하였다. NVMe-oF 타겟 BDEV는 이벤트에 키-값 NVMe 명령어 정보와 KVBDEV를 담아서 이벤트 큐에 제출한다. KVBDEV에 리엑터가 할당되면 KVBDEV가 실행된다. KVBDEV는 키-값 NVMe 명령어를 NVMe 블럭 I/O로 변환한다. KVBDEV의 실행이 종료될 때, 블럭 I/O 정보와 NVMe BDEV를 이벤트에 담아 이벤트 루프에 제출한다. NVMe BDEV 또한 동일하게 동작한다. NVMe 드라이버가 리엑터를 할당받아 실행을 완료하면 콜백을 이용해 NVMe BDEV에 알린다. 동일하게 NVMe BDEV는 KVBDEV에, KVBDEV는 NVMe-oF 타겟에 콜백을 이용하여 실행 완료를 알린다.

3.3 키-값 API 및 NVMe 명령어 확장

사용자 수준의 응용 프로그램이 스토리지 프로토콜인 NVMe를 활용하여 키-값 요청을 보내도록 하기위해 본 논문에서는 NVMe I/O Passthrough 명령어를 활용한 키-값 API 라이브러리를 구현하였다. 해당 키-값 API에서 사용하는 키-값 NVMe 명령어는 iLSM-SSD의 KV NVMe 확장 명령어를 차용하여 Put, Get, Delete 등의 요청에 대해 NVMe의 Vendor Specific Opcode을, 키를 위해 Starting LBA 영역을, 그리고 값을 전송하기 위해 기존 블록 데이터 전송에 사용한 PRP List를 사용한다 [8].

3.4 SPDK 기반 Hash 구조 스토리지 엔진

LSM-tree에서 컴팩션 수행시 발생하는 I/O 증폭 문제를 경감하기 위해 컴팩션 과정이 필요 없으며 간단한 구조를 가진 Hash 자료구조를 채택한 스토리지 엔진을 SPDK에 BDEV (KVBDEV)로 구현하였다. Hash 구조 스토리지 엔진은 다음과 같은 세가지 주요 모듈로 구성된다. (i) 키 해시 함수, (ii) NVMe SSD의 빈공간 관리를 위한 비트맵 배열, (iii) 키-값 쌍의 NVMe SSD 상의 논리 블럭 주소를 관리하는 해시 테이블. 해시테이블은 키 별 노드를 저장하는 자료구조다.

비트맵 배열과 해시 테이블은 공유 자료구조로 mutex lock을 사용하여 임계구역 보호가 필요하다. 하지만 배열과 테이블 전체를 하나의 락으로 보호하는 것은 락 컨텐션 (Lock Contention)으로 인해 병렬 처리 확장성에

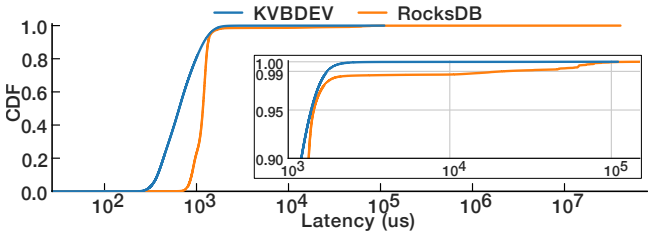


그림 4: 지연시간에 대한 CDF (Cumulative Distribution Function) 비교

한계가 있다. 따라서 락 컨텐션을 줄이고 성능을 향상시키기 위하여 비트맵 배열과 해시 테이블을 15개로 파티셔닝 하여 파티션 단위로 임계구역 보호를 하였다. 비트맵 배열과 해시 테이블의 파티션은 해시값을 이용해 선택한다.

그림 3는 KVBDEV 설계 및 동작 방식을 설명한다.❶ NVMe-oF 타겟으로 부터 키-값 쓰기 요청을 받으면 해시함수를 이용하여 키의 4 B 크기 해시값을 생성한다. 빠른 탐색을 위해 해시값으로 시작 비트맵 배열 파티션, 시작 해시 테이블 파티션, 해시테이블의 인덱스를 선택한다. ❷ 해시값의 MSB부터 4개 비트는 16개 비트맵 배열 파티션과 16개 해시 테이블 파티션에서 각각의 시작 파티션을 선택할 때 사용된다. 다음 12개 비트는 해시 테이블 내 인덱스를 결정한다. 그림 3에서 해시값의 MSB로 15번 비트맵 배열 파티션과 15번 해시 테이블 파티션을 선택한 후, 다음 12개 비트로 15번 해시 테이블의 인덱스 $111_{(16)}$을 선택한다. ❸ 비트맵 배열 파티션은 저장장치 전체 용량을 16개로 나눈 뒤 각 파티션 내 빈 블럭을 표시한 자료구조로, 먼저 탐색할 구간을 시작 비트맵 배열 파티션으로 설정한다. 해시값으로 결정한 시작 비트맵 배열 파티션 에서 First-fit 알고리즘을 사용하여 필요한 갯수 만큼 연속된 빈 블럭들을 찾으면 시작 블럭의 논리 블럭 주소를 할당받는다. 그림 3에서는 15번 비트맵 배열 파티션에서 세개의 연속된 빈 블럭을 찾아 시작 블럭의 논리 블럭 주소를 할당받는다. 만약 해당 비트맵 배열 파티션에 할당 가능한 연속된 빈 블럭이 없으면 다음 번호의 비트맵 배열 파티션을 탐색한다. 현재 구현은 연속 블럭 할당 방식으로 외부 단편화 문제가 발생할 수 있다. ❹ 해시 테이블의 인덱스에는 노드들이 연결되어 있다. 노드의 엔트리는 해시값과 논리블럭주소를 저장한다. 인덱스에 연결된 노드의 해시값을 탐색하여 같은 해시값을 가진 노드가 없으면 새로 추가한다. 이와 같은 방식으로 키 별 노드를 해시 테이블에 저장한다. 읽기는 같은 해시값을 가진 노드에서 논리 블럭 주소를 얻는다. 만약 같은 해시값을 가진 노드가 없을 경우, 응용에 읽기 실패를 알린다. ❺ 논리 블럭 주소와 값을 쓰기 I/O로 변환하여 NVMe BDEV로 전달한다. NVMe BDEV는 전달받은 I/O를 NVMe Driver로 요청한다.

4 실험 및 평가

4.1 실험 환경

실험을 수행한 환경은 다음과 같다. 계산 노드는 10 코어 20 쓰레드 서버, 스토리지 노드는 10 코어 20쓰레드 서버와 Samsung 970 EVO 500 GB NVMe SSD를 사용하였다. 두 서버는 10 Gbps 네트워크로 연결하였으며, 스토리지 노드는 SPDK v.21.10을 사용하였고, 스토리지 기반 키-값 스토어 또한 동일 버전에 구현하였다. 표 1은 실험을 수행한 계산 노드와 스토리지 노드의 상세 세부 스펙을 설명한다. 각 노드는 동일한 스펙을 가진다.

CPU	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
Memory	32GB DDR4
Network	10 Gbps Ethernet
OS	Ubuntu 20.04.
SPDK	v.21.10 (스토리지 노드)
RocksDB	v.6.23 (계산 노드)

표 1: 계산 노드와 스토리지 노드의 하드웨어, 소프트웨어 세부 상세 설명

다음과 같은 두개 시스템에 대한 비교 평가를 수행한다.

- **계산 노드 기반 키-값 스토어 (RocksDB):** 본 시스템은 전통적으로 계산 노드에서 키-값 스토어가 구동되는 시스템이다. 우리는 키-값 스토어로 RocksDB를 사용하였다. 스토리지 노드 기반 키-값 스토어와 동일하게 SPDK를 이용하여 스토리지 노드의 블럭 I/O 스택을 우회한다.
- **스토리지 노드 기반 키-값 스토어 (KVBDEV):** 본 논문에서 제안하는 시스템으로 해시 구조 기반 스토리지 엔진을 SPDK에 통합하여 스토리지 노드에서 구동되는 시스템이다.

실험에 사용한 워크로드는 dbbench 벤치마크의 쓰기 워크로드인 'Fillsequential'이며 계산 노드는 15개 쓰레드, 스토리지 노드는 19개 쓰레드를 사용하였다. 계산 노드 기반 설계에서 RocksDB는 32MB buffer size를 이용하였다. 계산 노드 및 스토리지 노드 기반 설계 모두 전송 쓰레드당 100만개 씩 총 1500만개의 키-값 쌍을 전송하였으며, 키-값 쌍의 크기는 각각 4 B, 32 KB를 사용하였다.

4.2 결과 및 분석

평가 분석을 위해 우리는 지연시간 (Latency)과 처리율을 측정하였다. 그림 4는 두개 시스템의 지연시간을 누적 분포 함수로 표현한 결과이다. 계산 노드 기반 키-값 스토어의 평균 지연시간은 1835 us이며 스토리지 노드 기반 키-값 스토어의 평균 지연시간은 735 us으로 계산노드 기반 키-값 스토어 대비 60% 감소하였다. 특히, 99 Percentile의 지연시간은 계산 노드 기반 키-값 스토어가 21224 us, 스토리지 노드 기반 키-값 스토어는 1853 us으로 91%가 감소하였다. 이것은 평균 지연시간 감소보다 꼬리지연 시간 감소의 폭이 높은 것을 의미하며, 제안하는 스토리지 노드 기반 키-값 스토어가 기존 방식보다 꼬리 지연시간 감소에 매우 큰 효과가 있음을 확인하였다. 평균 지연시간 및 꼬리 지연시간 감소를 통해 Hash 구조 스토리지 노드 기반 키-값 스토어에서 기존 계산 노드 기반 RocksDB에서 발생하던 LSM-tree의 컴팩션 과정을 완전히 제거했을 뿐만 아니라 계산 노드의 커널 우회로 I/O 스택 오베헤드를 최소화 하였음을 알 수 있다.

구분	계산 노드 기반	스토리지 노드 기반
처리율 (MB/s)	255	638

표 2: 처리율 비교

표 2은 두 시스템의 평균 처리율을 보여준다. 계산노드 기반 키-값 스토어의 평균 처리량은 255 MB/s이며 스토리지 노드 기반 키-값 스토어의 평균 처리량은 638 MB/s로 183% 증가하였다.

5 결론 및 향후 연구

본 연구에서는 스토리지 분리화 환경에서 기존 계산 노드 기반 키-값 스토어 설계 패러다임에서 벗어나 스토리지 노드 기반 키-값 스토어의 설계를 제안한다. 특히, 스토리지 노드에 Hash 구조 키-값 스토리지 엔진을 SPDK BDEV로 설계 구현하였다. 기존 계산 노드 기반 RocksDB와 쓰기 성능 평가를 수행하였으며 dbbench의 쓰기 워크로드인 'Fillsequential' 대하여 Hash 구조 서버 기반 키-값 스토어가 낮은 쓰기 응답 시간을 보임을 확인하였다. 본 연구는 쓰기 성능에 대한 평가만 진행한 한계가 존재한다. 향후에 읽기 성능 평가를 수행하여 스토리지 노드 기반의 접근의 우수성을 살펴볼 계획이다.

참고 문헌

[1] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 33–48, USENIX Association, July 2020.

[2] E. S. Brian Cho, "Taking advantage of a disaggregated storage and compute architecture." https://databricks.com/session/taking-advantage-of-a-disaggregated-storage-and-compute-architecture, 2020.

[3] Facebook, "RocksDB." http://rocksdb.org.

[4] Google, "LevelDB." https://github.com/google/leveldb, 2021.

[5] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[6] "SPDK." https://spdk.io/, Accessed 2020-10-10.

[7] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, p. 35–40, 2010.

[8] C.-G. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park, "iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics," in *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '19, pp. 384–395, 2019.