JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 21, No. 8, pp. 1501-1506, Aug. 2020

Check for updates

# PlexDB: LSM-tree 기반 Key-Value 저장소의 데이터 중복제거를 위한 효율적인 컴팩션 알고리즘

서 용 철[1] · 박 성 훈[2*]
[1]충북대학교 컴퓨터공학과 박사과정
[2]충북대학교 컴퓨터공학과 교수

# PlexDB: Efficient Compaction Algorithm for Deduplication of LSM-tree based Key-Value Store

**Yong-Cheol Seo[1] · Sung-Hoon Park[2*]**

[1]Doctoral Course, Department of Computer Engineering, Chungbuk National University, Cheongju 28644, Korea
[2]Professor, Department of Computer Engineering, Chungbuk National University, Cheongju 28644, Korea

**[요 약]**

NoSQL 데이터베이스의 유형인 Key-Value 저장소는 전통적인 SQL 데이터베이스에 비해 높은 읽기 및 높은 쓰기 처리량을 제공한다. LSM-tree는 KV 저장소에서 소규모 쓰기 성능을 향상시키기 위해 널리 사용되는 데이터 구조이다. 그러나 컴팩션 작업으로 인해 많은 쓰기 증폭을 유발하고 있다. 따라서 tiered LSM-tree는 각 레벨의 LSM-tree 구조를 여러 하위 레벨으로 나누어 쓰기 증폭을 줄인다. 최근 KV 항목은 많은 응용프로그램에 의해 자주 업데이트되며, KV 저장소에 있는 동일한 KV 항목들이 여러 버전으로 존재하기 때문에 쓰기 증폭과 공간 증폭을 많이 발생한다. 낮은 쓰기 증폭과 공간 증폭을 동시에 달성하기 위해 tiered LSM-tree를 이용한 새로운 컴팩션 알고리즘을 제안하고 새로운 KV 저장소인 PlexDB로 구현하였다. 본 실험을 통해 PlexDB는 LevelDB와 LSM-trie를 이용해 낮은 쓰기 증폭과 공간 증폭에 대한 성능을 비교 분석한다.

**[Abstract]**

Key-Value store, a type of NoSQL database, provides higher read and higher write throughput than traditional SQL databases. LSM-tree is a widely used data structure for improving small write performance in KV store. However, the compaction operation is causing a lot of write amplification. Therefore, tiered LSM-tree divides the LSM-tree structure of each level into several sub-levels to reduce write amplification. Recently, KV items are frequently updated by many applications, and write amplification and space amplification occur because the same KV items in the KV store exist in multiple versions. To achieve low write amplification and space amplification at the same time, a new compaction algorithm using tiered LSM-tree is proposed and implemented with KV store, PlexDB. Through this experiment, PlexDB uses LevelDB and LSM-trie to compare and analyze low write amplification and space amplification performance.

**색인어 :** LSM-tree, 키-값 저장소, 데이터 중복제거 압축, 쓰기 증폭, 공간 증폭

**Key word :** LSM-tree, Key-Value Store, Deduplication Compaction, Write Amplification, Space Amplification

**http://dx.doi.org/10.9728/dcs.2020.21.8.1501**



**Received** 22 July 2020; **Revised** 05 August 2020
**Accepted** 11 August 2020

***Corresponding Author; Sung-Hoon Park**

**Tel:** +82-43-261-3269
**E-mail:** ycseo@cbnu.ac.kr

## Ⅰ. 서 론

KV(Key-Value) 저장소는 일반적인 NoSQL 데이터베이스다. 확장성과 유연성으로 인해 소셜 네트워킹 분석, 전자 상거래 및 클라우드 스토리지 등 많은 데이터 집약적 응용 프로그램에서 기본 데이터 스토리지로 널리 사용되고 있다. KV 저장소는 insert, update, delete, get과 같은 간단한 인터페이스를 사용하여 상위 레벨 응용 프로그램에 대한 지속적인 백업 및 빠른 데이터 검색을 제공한다. 기본 스토리지 장치와 사용자의 작업량을 고려하여 KV 저장소는 최상위에서 실행되는 응용 프로그램 요구 사항을 충족하도록 서로 다른 데이터 구조를 구성할 수 있다. Log Structured Merge-tree(LSM-tree)[1]는 leveled LSM-tree[2], tiered LSM-tree[2]가 대표적으로 구현되어 있으며 최근 워크로드의 변화로 인해 BigTable, Cassandra 및 RocksDB와 같은 KV 저장소가 많이 사용되고 있다.

지난 10년 동안 워크로드는 더욱 더 쓰기 작업에 우위를 두게 되었다. 예를 들어, Yahoo[3]의 일반적인 KV 저장소 워크로드에서 읽기는 총 요청의 80%~90%를 차지하는 반면 2012년에는 50%로 감소한다. 또한, 많은 시스템이 읽기 요청을 흡수하기 위해 Redis 및 Memcached와 같은 내부 캐시를 사용하므로 읽기 요청의 작은 부분만 최종적으로 Back-End의 KV 저장소에 처리된다. 예를 들어 Facebook의 사진 캐싱 시스템에서 90%의 읽기 요청은 Font-End Cache에 흡수되는 반면 Back-End의 KV 저장소에서 제공하는 읽기 요청은 10%에 불과하다.

본 논문에서는 levelDB[4]를 기반으로 구현된 새로운 KV 저장소인 PlexDB에서 공간 증폭을 줄이면서 쓰기 증폭을 낮게 유지하는 새로운 컴팩션 방법인 레벨 데이터 중복제거 컴팩션(LDC: Level Duduplication Compaction) 알고리즘을 제안한다. 실험을 위해 YCSB(Yahoo! Cloud Serving Benchmarking)를 사용하여 주어진 워크로드를 통해 PlexDB, levelDB 및 LSM-trie를 각각 수행하였으며 쓰기 증폭 및 공간 증폭의 성능을 비교 분석한 결과 LDC를 적용한 PlexDB가 성능 향상에 효과가 있음을 확인하였다.

## Ⅱ. 관련 연구

KV 저장소에는 빈번한 업데이트로 인해 여러 버전의 KV 항목이 존재할 수 있으며 이는 스토리지의 공간 증폭(space-amplification)을 발생시킨다. 특히 tiered LSM-tree를 사용하고 있는 KV 저장소의 쓰기 주도적인 응용 프로그램에서 빈번한 업데이트는 동일한 레벨의 서로 다른 하위 레벨에 KV 항목을 여러 버전으로 저장하여 leveled LSM-tree보다 공간 증폭을 많이 발생시킬 수 있다.

Leveled LSM-tree는 $L_0, L_1, ..., L_n$이라는 여러 레벨을 유지하며 각 레벨의 크기는 기하급수적으로 증가한다. 신규 항목은 $L_n$에 먼저 추가되는 방식이다. $0 \le i \le n-1$인 경우, $L_i$가 초과하면 컴팩션이 실행되어 $L_i$의 KV 항목을 $L_{i+1}$로 병합하고 정렬 상태를 유지한다. 컴팩션은 KV 항목이 $L_n$을 제외한 각 레벨에서 정렬되도록 한다. 그러나 많은 I/O 리소스를 소비하고 높은 쓰기 증폭(write amplification)을 제공하여 사용 중인 응용 프로그램의 성능을 저하 시킨다. 컴팩션의 쓰기 증폭을 완화하기 위해 tiered LSM-tree는 데이터가 각 수준에서 완전히 정렬되어야 한다는 제약 조건이 있다. 완전하게 정렬된 하위 레벨이 아닌 각 레벨에서 여러 개의 정렬된 하위 레벨을 유지한다. 각 레벨에서 완전히 정렬된 데이터의 제약 없이 tiered LSM-tree로 컴팩션하면 쓰기 증폭이 크게 감소한다.

LSM-tree의 프로세스를 이해하기 위해 LevelDB를 예로 들어 작동하는 방식을 자세히 설명한다. LevelDB는 그림 1과 같이 메모리와 보조 저장소 두 가지로 구성된다.

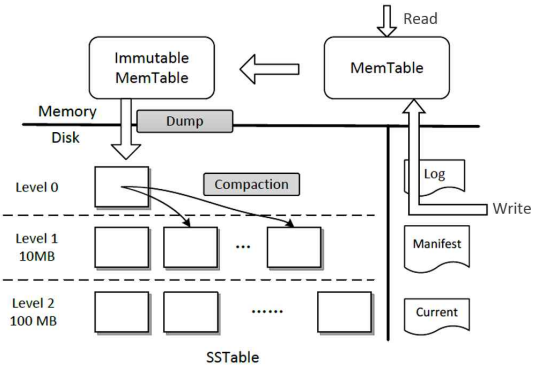



**그림 1.** LevelDB의 아키텍처
**Fig. 1.** Architecture of LevelDB

메모리의 구성 요소는 Memtable과 Immutable Memtable로 나뉘며, 보조 저장소의 구성 요소는 여러 레벨인 $L_0$(Level 0), $L_1$(Level 1), $L_2$(Level 2)로 나뉜다. 각 레벨에는 KV 항목들이 SSTable 파일에 저장된다. 추가되는 KV 항목들은 skiplist[5]에 구성된 Memtable에 먼저 삽입된다. Memtable이 가득 차면 모든 KV 항목은 읽기 전용 Immutable Memtable로 변환된다. $L_0$이 채워지면 컴팩션 연산은 $L_0$에서 $L_1$까지 데이터를 컴팩션하며, $L_0$에서 $L_1$까지 SSTable을 메모리로 읽고 KV 항목을 정렬하고 분리된 Key 범위를 가진 새로운 SSTable 파일을 생성하여 최종적으로 그림 2와 같이 $L_1$에 기록한다. 다른 레벨들은 동일한 컴팩션 규칙을 따른다. 각 레벨의 주요 범위는 기본적으로 동일하고 $L$의 크기가 $L_{i-1}$의 대략 10회 정도 되므로 컴팩션은 큰 쓰기 증폭을 발생한다. LevelDB에 6 또는 그 이상의 레벨이 있는 경우 쓰기 증폭 비율이 기하급수적으로 커질 수 있다[6]. 본 논문에서 쓰기 증폭을 줄이기 위한 관련 작업은 LSM-trie[6] 및 tiered LSM-tree를 대상으로 한다.

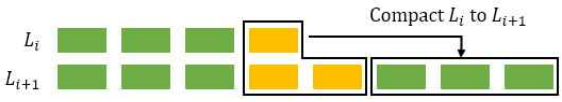



**그림 2.** Leveled LSM-tree의 컴팩션
**Fig. 2.** Compaction of Leveled LSM-tree

Tiered LSM-tree와 leveled LSM-tree의 핵심 차이점은 tiered LSM-tree가 leveled LSM-tree의 정렬된 레벨을 대체하기 위해 다수의 정렬된 하위 레벨을 사용하며 컴팩션 연산은 $L$의 모든 하위 레벨을 병합 및 정렬하여 $L_{i+1}$의 하위 레벨을 생성한다는 것이다. 하위 레벨은 컴팩션에 포함되지 않기 때문에 그림 3과 같이 쓰기 증폭을 크게 줄일 수 있다. 그러나 tiered LSM-tree는 자주 업데이트되는 워크로드(예: Facebook[7] 및 Yahoo[8])에서 leveled LSM-tree 더 훨씬 높은 공간 증폭이 발생한다.

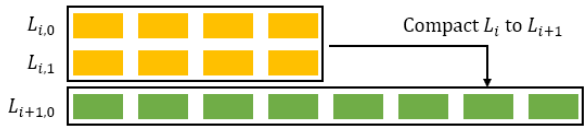



**그림 3.** Tiered LSM-tree의 컴팩션
**Fig. 3.** Compaction of Tiered LSM-tree

Nutanix는 업데이트 작업의 비중이 매우 높으며, 일부 워크로드의 경우 Key가 평균 9회 이상 업데이트된다[9]. 데이터 크기가 증가함에 따라 높은 공간 증폭으로 보조 기억장치의 비용이 크게 증가할 것이다. 또한, Bloom filter[10] 및 Index를 포함하여 더 많은 메타 데이터를 생성하므로 공간 증폭은 더 심해진다. 추가되는 메타 데이터를 메모리에 캐시 할 수 없는 경우 데이터를 검색하기 위해 보조 저장 장치의 액세스가 추가로 발생하고 이는 읽기 성능 저하로 이어진다.

## Ⅲ. 제안하는 레벨 데이터 중복제거 알고리즘

Tiered LSM-tree는 구조화된 KV 저장소에서는 레벨에 저장된 데이터의 크기가 레벨의 크기 제한에 도달했을 때에만 정통적인 컴팩션이 실행된다. 레벨의 모든 하위 레벨이 기록되면 해당 레벨에 대한 컴팩션 작업이 실행된다. 컴팩션 작업은 해당 레벨에서 모든 KV 항목을 읽고 정렬한 다음 컴팩션된 SSTable 파일들을 다음 레벨의 최소 Index가 있는 빈 하위 레벨에 다시 작성한다. 그러나 tiered LSM-tree에서 전통적인 컴팩션 방식은 오래된 데이터의 양을 무시한다. KV 항목이 자주 업데이트되는 응용 프로그램에는 동일한 KV 항목의 오래된 복사본이 많이 있다. 이러한 오래된 복사본이 제때 수집될 수 없다면 시스템에 큰 공간 증폭을 초래하여 저장 공간에 상당한 낭비를 초래한다.

본 논문에서는 PlexDB에서 KV 저장소의 공간 증폭을 줄이기 위해 기존 컴팩션과 함께 동작하는 LDC(Level-Duduplication Compaction)을 제안한다.

표 1.

Algorithm 1. Level deduplication compaction algorithm

```
1:  procedure isLevelCompaction()
2:      if (isLevelSizeCompaction()) then
3:          SetLevelSizeCompaction()
4:          SetOutputLevelSizeCompaction()
5:      else if (isLevelDeduplicationCompaction())
6:          SetLevelDeduplicationCompaction()
7:          SetOutputLevelDeduplicationCompaction()
8:      end
9:  end procedure
```

알고리즘 1과 같이 LDC는 오래된 데이터가 사전 정의된 임계값에 도달하면 실행된다. 오래된 복사본의 비율을 감지하기 위한 여러 개의 모니터를 설정했으며 각각의 모니터는 레벨과 연관되어 있다. 모니터가 임계값에 도달한 레벨에서 오래된 복사본의 비율을 감지하면 LDC가 실행된다. LDC의 설계시 고려해야 할 사항은 모니터가 가벼워야 하며 CPU와 메모리 모두 추가적인 오버헤드가 없어야 한다는 것이다. 이 문제를 해결하기 위해 알고리즘 2와 같이 count-distinct problem에 대한 알고리즘인 HyperlLogLog[11]로 모니터를 구현하였다.

표 2.

Algorithm 2. Compaction detect monitoring algorithm

```
1:  procedure MONITOR COMPACTION()
2:      while (isStaleCompaction) do
3:          compaction level = StaleCompactLevel
4:          output level = Output.level
5:          compactionResult result = Compacton(compaction level)
6:          compactionBuild(result)
7:          HLL[compaction level].StaleClear
8:          HLL[output level].Update(result)
9:      end
10: end procedure
```

HLL은 메모리에 무시할 수 있는 데이터만 유지하며 이를 사용하여 해당하는 다중 집합의 고유한 요소 수를 근사화 할 수 있다. 상기 집합에 요소를 추가할 때 HLL은 먼저 추가 요소의 해시 값을 계산한 다음 해시 값에 따라 HLL의 데이터를 업데이트한다. 또한, 두 개 이상의 다중 집합으로부터 각각의 HLL을 결합하여 고유한 원소의 수를 얻을 수 있다. HLL은 각 레벨의 고유키 수를 계산하는 데 사용되며 각 레벨을 HLL 인스턴스로 설정한다. 오래된 복사본은 제때에 Garbage- Collection이 되지 않으므로 다른 하위 레벨과 레벨 사이에 동일한 키를 확산시킨 많은 KV 항목이 있을 수 있다.

레벨에서 오래된 복사본의 비율을 설명하기 위해 식 (1)과 같이 정의할 수 있다.

$$duplicate\_rate = unique\_keys \ / \ total\_keys \qquad (1)$$

total_keys는 메타 데이터에서 얻을 수 있는 레벨의 Key의 총 수이고 unique_keys는 레벨의 동일한 키가 한 번만 카운트되는 Key의 수로 동일한 Key가 HLL에 의해 한 번만 카운트되는 고유한 Key의 수이다.

duplicate_rate는 레벨에서 동일한 키의 평균을 의미하며 이는 오래된 복사본의 비율을 반영한다. duplicate_rate가 클수록 오래된 복사본의 비율이 높아진다. 해당 duplicate_rate가 deduplicate_threshold에 도달하면 시스템은 이 수준에서 컴팩션을 실행하고 HLL 인스턴스를 재설정한다. 컴팩션의 프로세스 중에 출력 레벨과 연관된 HLL인스턴스가 업데이트 된다. HLL을 업데이트하려면 시스템에서 컴팩션 결과에 속하는 각 Key의 Hash 값을 계산해야 하므로 많은 CPU 리소스가 소비된다. 이를 피하기 위해 전통적인 KV 저장소의 읽기 성능을 향상시키기 위해 표준 구성 요소인 Bloom filter[9]에서 생성된 Hash 값을 재사용한다. 컴팩션 작업중에 Bloom filter도 삽입 Key의 Hash 값을 계산해야 하므로 HLL은 CPU에 추가 오버헤드를 발생시키지 않는다. tiered LSM-tree의 전통적인 컴팩션과 달리 컴팩션 레벨에서 하위 레벨의 절반 이상이 비어있는 경우 LDC에 의해 컴팩션된 SSTable 파일은 컴팩션 레벨에서 첫 번째 하위 레벨로 기록될 것이다. 그렇지 않으면 기존 방식을 따른다.

## IV. 알고리즘 성능 평가 및 분석

### 4-1 실험 환경

KV 저장소인 tiered LSM-tree를 기반으로 PlexDB를 구현하였다. PlexDB는 각 레벨의 데이터 볼륨과 각 레벨의 오래된 복사본의 비율을 기반으로 컴팩션을 실행한다.

본 논문에서는 PlexDB를 LevelDB와 LSM-tree를 이용하여 컴팩션의 쓰기 증폭에서 공간 증폭 및 시스템 워크로드를 갖는 총 처리량을 계산하였다. 실험은 Ubuntu 서버에서 수행되었으며 세부적인 구성은 표 3과 같다.

**표 3.** 성능 평가 시스템의 환경
**Table 3.** Environment of Performance Evaluation System

| | |
|---|---|
| OS | Ubuntu Server 18.04 LTS 64Bit |
| CPU | Intel Xeon CPU E5-2620 (2.0GHz 2 * 6 Cores) |
| RAM | 48GB |
| SDD | SATA3 512GB |

YCSB[7]를 사용하여 각 Key가 10 byte이고 Value가 200 byte인 20GB의 쓰기 전용 워크로드를 생성한다. 또한, 워크로드에는 높은 비율의 업데이트 작업으로 채워지며, 각 KV 항목은 Nutanix의 워크로드와 일치하는 평균 9회 업데이트를 진행한다. 생성된 KV 항목을 PlexDB, LevelDB, LSM-trie에 넣은 후 쓰기 증폭과 공간 증폭을 각각 기록한다. 실험에서는 PlexDB에서 deduplicate_threshold에 대한 4가지 설정을 통해 성능 분석을 하였다.

### 4-2 비교 분석 결과

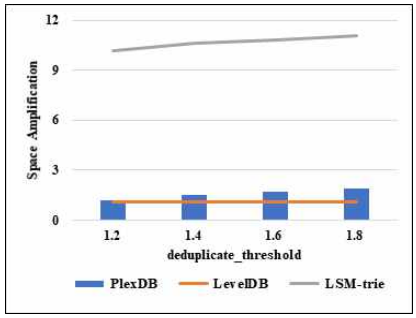



**그림 4.** 20GB 워크로드에서 PlexDB, LevelDB 및 LSM-trie의 공간 증폭
**Fig. 4.** Space Amplification between PlexDB, LevelDB and LSM-trie under 20GB Workload

그림 4와 같이 LevelDB의 공간 증폭은 1.2에 불과하지만 LSM-trie의 공간 증폭은 11보다 매우 큰 것을 확인하였다. 그 이유를 두 가지로 요약할 수 있다. 첫째 LSM-trie는 tiered LSM-tree를 기반으로 하고 있어 공간 사용률은 $O(T)$이고 LSM-tree는 공간 사용률이 $O(\frac{T+1}{T})$이다. 따라서 LSM-trie는 LSM-tree에 비해 상대적으로 높은 공간 증폭을 유발한다[12]. 둘째 LSM-trie는 고정된 KV 테이블 설계를 채택하고 있는데, 이는 많은 KV 테이블의 저장 공간이 충분히 활용되지 못하여 공간 증폭을 더 많이 유발된다는 것을 의미한다. PlexDB의 경우 deduplicate_threshold가 증가함에 따라 공간 증폭이 증가하고 있다. 그 이유는 deduplicate_threshold가 컴팩션을 제어하기 때문이며 deduplicate_threshold가 작을수록 공간 증폭을 견딜 수 있는 용량이 적고, deduplicate_threshold가 1.2로 설정되면 공간 증폭은 LevelDB와 거의 동일하다. deduplicate_threshold가 1.8로 설정되면 PlexDB의 공간 증폭은 약 2로 여전히 LSM-trie의 공간 증폭보다 훨씬 낮다.

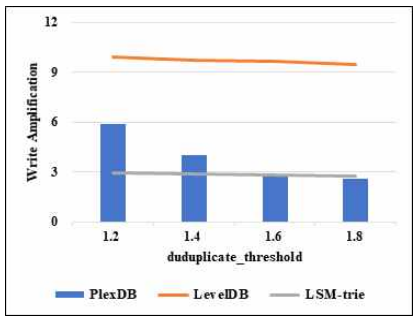



**그림 4.** 20GB 워크로드에서 PlexDB, LevelDB 및 LSM-trie의 쓰기 증폭
**Fig. 4.** Write Amplification between PlexDB, LevelDB and LSM-trie under 20GB Workload

쓰기 증폭 결과는 그림 4와 같으며 LevelDB의 쓰기 증폭은

10에 가깝다는 것을 실험을 통해 확인 하였다. PlexDB의 경우 deduplicate_threshold가 증가함에 따라 쓰기 증폭이 감소하고, 작은 deduplicate_threshold는 더 높은 쓰기 증폭이 이어지는 컴팩션 빈도를 증가시킨다. 또한, deduplicate_threshold가 1.6을 초과하면 PlexDB 및 LSM-trie의 쓰기 증폭은 거의 동일하다는 것을 알수 있다.

100GB의 대용량 데이터 세트로 각 Key를 20회 업데이트하는 실험을 수행하였다. PlexDB의 deduplicate_threshold를 1.6으로 고정시킨 후 생성된 KV 항목을 3개의 KV 저장소에 넣었다. 20GB 워크로드와 비교하여 그림 5와 같이 3개의 KV 저장소 모두 쓰기 증폭이 증가한다. KV 저장소에 KV 항목이 더 많은 레벨(데이터 볼륨이 더 많음)이 많아 쓰기 증폭을 유도하기 때문이다.

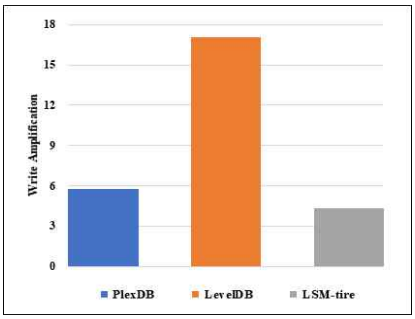



**그림 5.** 100GB 워크로드에서 PlexDB, LevelDB 및 LSM-trie의 쓰기 증폭 비교

**Fig. 5.** Compare Write Amplification of PlexDB, LevelDB and LSM-trie on 100GB Workload

그림 6과 같이 LSM-trie의 공간 증폭이 18에 가까이 도달한다는 것을 확인했으며 이는 20GB의 작업부하를 가진 것보다 훨씬 크다. LevelDB와 PlexDB의 공간 증폭은 2에 가깝기만 하다.

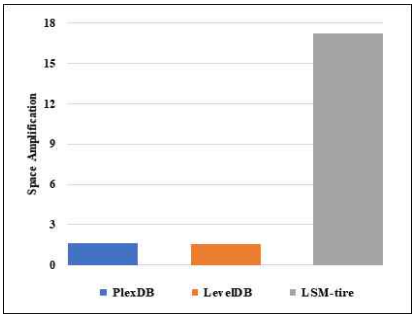



**그림 6.** 100GB 워크로드에서 PlexDB, LevelDB 및 LSM-trie의 공간 증폭 비교

**Fig. 6.** Compare Space Amplification of PlexDB, LevelDB and LSM-trie on 100GB Workload

또한, 그림 7과 그림 8에서와 같이 각각 표시된 쓰기 처리량과 읽기 처리량을 평가한다. LSM-trie를 사용한 쓰기 처리량은 쓰기 증폭을 가지고 있고, SSD에 대한 소규모 쓰기를 피하는 WAL(Write-Ahead-Logging)을 채택하지 않기 때문에 가장 높다[13]. 3개의 KV 저장소는 모두 Bloom filter가 장착되어 있으며, 각 Key는 16 bit로 할당되어 있다. 읽기 처리량을 측정하기 위해 Bloom filter를 포함한 메타 데이터를 캐싱하기 위한 메모리 공간 사용을 1GB로 제한하고, 직접적인 I/O를 통해 SSD에서 KV 항목을 검색했다. Bloom filter의 활용으로 시스템은 SSD에서 KV를 검색하는 불필요한 액세스를 피할 수 있다.

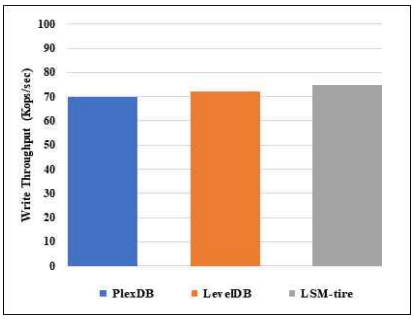



**그림 7.** 100GB 워크로드에서 PlexDB, LevelDB 및 LSM-trie의 쓰기 처리량 비교

**Fig. 7.** Compare Write Throughput of PlexDB, LevelDB and LSM-trie on 100GB Workload

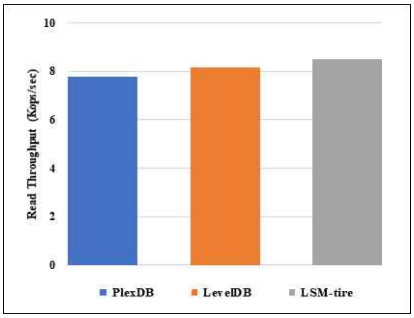



**그림 8.** 100GB 워크로드에서 PlexDB, LevelDB 및 LSM-trie의 쓰기 처리량 비교

**Fig. 8.** Compare Read Throughput of PlexDB, LevelDB and LSM-trie on 100GB Workload

## Ⅴ. 결 론

본 논문에서는 여러 버전의 동일한 데이터가 감지될 때 컴팩션을 실행하는 tiered LSM-tree에 대한 LDC를 제안한다. 본 제안은 쓰기 증폭과 공간 증폭간의 균형을 이루며 이상적인 조건에서 tiered LSM-tree의 쓰기 증폭과 leveled LSM-tree의 공간 증폭이 낮게 발생하여 향상된 시스템 성능을 보였다.

향후 연구는 새로운 KV 저장소로 구현중인 PlexDB에서 데이터의 읽기 액세스 기능을 향상시키기 위해 새로운 Bloom filter 할당 방식을 설계하여 시스템의 성능을 최적화 하는 연구를 진행할 예정이다.

## 참고문헌


[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, The log-structured merge-tree (LSM-tree), Acta Informatica, Vol.33, No.4, pp.351-385, 1996.

[2] N. Dayan, M. Athanassoulis, and S. Idreos, Monkey: Optimal navigable key-value store, Proceedings of the 2017 ACM International Conference on Management of Data, pp. 79-94, 2017.

[3] R. Sears and R. Ramakrishnan, bLSM: a general purpose log structured merge tree, Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 217-228, 2012.

[4] LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values[Internet]. Available: https://github.com/google/leveldb.

[5] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Communications of the ACM, Vol.33, No.6, pp.668-676, 1990.

[6] X. Wu, Y. Xu, Z. Shao, and S. Jiang, LSM-trie: An LSM-tree-based ultra-large key-value store for small data items, 2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15), pp. 71-82, 2015.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, Workload analysis of a large-scale key-value store, Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, pp. 53-64, 2012.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, Benchmarking cloud serving systems with YCSB, Proceedings of the 1st ACM symposium on Cloud computing, pp. 143-154, 2010.

[9] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores, 2017 Annual Technical Conference, pp. 363-375, 2017.

[9] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, Vol.13, No.7, pp.422-426, 1970.

[10] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunie r, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in Discrete Mathematics and Theoretical Computer Science, pp. 137-1 56, 2007.

[11] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, Incremental organization for data recording and warehousing, VLDB, Vol. 97, pp. 16-25, 1997.

[12] C. Luo and M. J. Carey, LSM-based storage techniques: a survey, The VLDB Journal, Vol.29, No.1, pp.393-418, 2020.

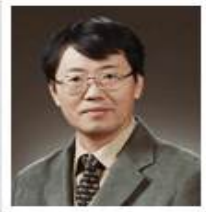
[13] SQLite. Write-Ahead Logging[Internet]. Available: https://www.sqlite.org/wal.html.



**서용철**(Yong-Cheol Seo)

2004년 : 충주대학교 컴퓨터공학과 (공학사)
2018년 : 충북대학교 컴퓨터공학과 (공학석사)

1997년～2018년: 자화전자(주), 수석 연구원
2018년～현 재: 충북대학교 대학원 컴퓨터공학과 (박사과정)
※관심분야 : 분산 처리, 분산 데이터베이스, 미들웨어



**박성훈**(Sung-Hoon Park)

1993년 : 인디애나대학교 컴퓨터공학과 (공학석사)
2000년 : 인디애나대학교 컴퓨터공학과 (공학박사)

1994년～1996년: ㈜ 두산컴퓨터 연구소 소장
1994년～2004년: 남서울대학교 컴퓨터공학과 교수
2004년～현 재: 충북대학교 컴퓨터공학과 교수
※관심분야 : 분산/모바일/유비쿼터스 컴퓨팅, 알고리즘, 컴퓨터 이론, 미들웨어