RocksDB WAL Overhead 분석

성한승*, 이두기*, 박상현* †
*연세대학교 컴퓨터과학과
{ hssung, Edoogie, sanghyun }@yonsei.ac.kr

Overhead Analysis of WAL on RocksDB

Hanseung Sung*, Doogie Lee*, Sanghyun Park* †
*Dept. of Computer Science, Yonsei University

요 약

RocksDB 는 데이터를 Key-Value 쌍으로 다루는 Key-Value 데이터베이스 시스템이며 효율적으로 데이터를 저장하기 위한 자료구조로 Log-Structured Merge-Tree 를 사용하고 있다. 이에 더하여, 데이터 베이스의 지속성을 위해 WAL 방식으로 로깅을 한다. 이러한 특징들로 인해 RocksDB 는 신속하고 효과적인 데이터 처리와 지속성 보존이 가능하여 지속적으로 화두가 되고 있는 데이터베이스 시스템이다. 그러나 RocksDB 는 WAL 로깅으로 인한 오버헤드가 발생한다.

본 논문에서는 RocksDB 에서 발생하는 WAL 오버헤드를 측정하여 WAL 로깅이 차지하는 오버헤드를 분석하였으며, 차세대 비 휘발성 메모리인 NVRAM 을 통해 오버헤드가 얼마나 개선 될 수 있는지 분석하였다. 분석을 통해 로깅 오버헤드는 성능 저하에 상당한 비중을 차지하고 있으며, 디바이스의 쓰기 속도에 따른 로깅 오버헤드의 차이를 발견 하였다.

1. 서론

빅데이터 시대인 지금은 센서 데이터, SNS 데이터와 같이 정형화하기 어려운 비정형 데이터가 방대하고 다양하게 생성되고 있다. 하지만 기존의 릴레이션 형태로 데이터를 저장하고 관리하는 관계형 데이터베이스로는 비정형 데이터를 다루기가 힘들다. 따라서, 이러한 비정형 데이터를 다루기 위해 Key-Value 쌍의 형태로 데이터를 저장하는 MongoDB[4], Cassandra[5], HBase[6], BigTable[7], LevelDB[8]와 같은 Key-Value 데이터베이스 시스템이 등장하였다.

그 중에서도 RocksDB[1-3]는 LSM-Tree 구조를 바탕으로 컬럼 패밀리, 블룸 필터 등을 추가하여 높은 데이터 처리속도를 얻게 되었다. 높은 데이터 처리속도와 함께 RocksDB 는 데이터가 반영되기 전에 로그를 먼저 기록하는 WAL(Write Ahead Logging)방식을 채택하여 데이터의 지속성까지 보장하였다. 하지만 이로 인하여 WAL 수행 오버헤드가 발생하게 되었다.

본 논문에서는 RocksDB 에서 발생하는 로깅 오버헤드 분석을 통해 로깅을 개선했을 때, 최대 성능이 어느 정도로 향상될 수 있는지 실험을 통하여 측정하고, 더 나아가 WAL 로깅에 NVMe, NVRAM 등 좀더 높은 성능의 장치를 활용했을 때 오버헤드가 어느정도 개선될 수 있는지 간략히 살펴보았다.

본 논문의 구성은 다음과 같다. 2 장에서는 Key-Value 데이터베이스 시스템에서 대표적으로 사용되는 구조인 LSM-Tree 구조와 RocksDB 그리고 RocksDB 에서 채택하고 있는 로깅 방법인 WAL 로깅에 대하여 설명을 하며, 이에 더하여 비 휘발성 메모리인 NVRAM 에 대한 설명과 RocksDB 의 성능 개선을 위해 선행되었던 기존 연구에 대해 설명한다. 3 장에서는 WAL 로깅에 대한 문제점과 함께 4 장에서는 RocksDB 의 WAL 오버헤드를 측정하고 NVRAM 을 통한 성능 향상을 측정하고 분석한다. 마지막으로 5 장에서는 결론을 맺는다.

2. 관련 연구

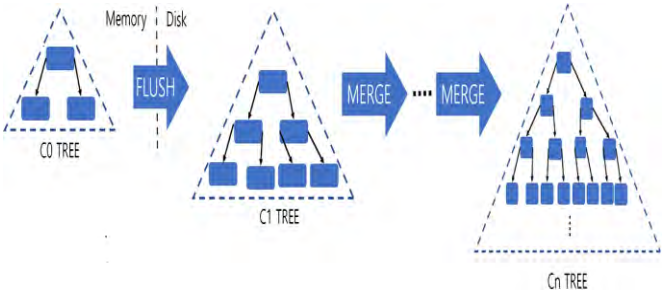
2. 1 LSM-Tree (Log-Structured Merge-Tree)

LSM-Tree(Log-Structured Merge-Tree)는 RocksDB, Cassandra, HBase, LevelDB 와 같은 Key-Value Store 에서 대표적으로 사용하는 자료 구조이며, 연속적인 쓰기연산을 수행하는 워크로드를 위해 설계되었다. LSM-Tree 구조는 (그림 1) 와 같다. LSM-Tree 구조는 하나의 In-Memory 데이터 구조와 여러 개의 Disk 에 저장을 위한 Append 방식의 데이터 구조로 이루어져

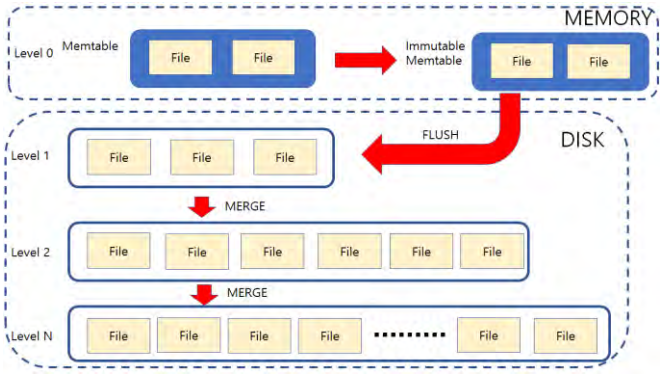
* 이 논문은 2015 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임
(NRF-2015R1A2A1A05001845).

† 교신 저자: sanghyun@yonsei.ac.kr

있다. LSM-Tree 는 삽입연산이 수행되면 먼저 메모리 영역에 데이터를 저장한다. 메모리의 일정 용량까지 데이터가 쌓이게 되면 메모리의 내용을 디스크로 Flush 한다. 이때, Flush 되는 데이터는 Disk 에 저장되어 있던 기존 데이터와 병합 정렬을 하여 기록된다. 만약 디스크의 영역의 각 레벨이 임계점을 넘게 되면 병합 정렬을 실행하여 하위 레벨을 생성해낸다. LSM-Tree 자료 구조는 디스크를 위한 별도의 로그가 필요하지 않으며 임의적 순서로 데이터를 쓰지 않고 순차적으로 데이터를 쓰는 것을 유도한다는 장점이 있다. 하지만, 데이터를 조회할 때, LSM-Tree 에서 주어진 데이터가 트리 내의 어느 위치에 있는지를 알 수 없어서 데이터를 찾기 위해서는 상위 레벨부터 순차적으로 검색해야 한다는 단점이 있다. 즉, 만일 트리에 존재하지 않은 데이터에 대하여 읽기 연산 요청될 경우에는 디스크에 데이터가 없음에도 데이터를 확인하기 위해 모든 레벨의 모든 파일을 읽어야 한다. 이러한 단점 외에도, LSM-Tree 구조를 사용하는 시스템에서 업데이트 된 데이터는 우선 메모리 영역에 기록된 다음 한꺼번에 디스크에 저장이 되므로 만약 디스크에 기록되기 전에 시스템 오류가 생길 경우, 저장되지 않은 메모리 영역의 데이터가 사라지는 위험성이 있다.



(그림 1) LSM-Tree 구조



(그림 2) RocksDB 구조

2. 2 RocksDB

RocksDB 는 Facebook 에서 고속 스토리지용으로 개발한 임베디드 Key-Value 데이터베이스 시스템이다. Google 에서 오픈소스로 공개한 LevelDB(v1.5)를 기반으로 하였으며, Key-Value 형식으로 데이터를 저장한다. RocksDB 구조는 (그림 2)에 표현되어 있다. RocksDB 에 데이터의 삽입 연산 요청이 들어오면 데이터를 메모리에 기록하기 전에 우선적으로 로그 파일에 로그를 기록한다. 로그를 기록한 다음 메모리 영역에 있는 'Memtable' 에 데이터를 저장한다. 쓰기요청이 계속되어 Memtable 에 데이터가 일정 용량까지 기록된 경우, Memtable 은 변경이 불가능한 Immutable Memtable (Read-Only Memtable)로 변경된다. Immutable Memtable 이 가득 차게 되면 Disk 영역으로 flush 가 일어난다. Flush 가 일어날 때, Memtable 의 파일은 key 순서에 따라 정렬이 되어 SST 파일(Static Sorted File)로 바뀐다. SST 파일은 디스크 영역에서 compaction 을 통해 업데이트 되며 한번 생성된 SST 파일은 사라지지 않는다. 따라서 하위 레벨에 상주하는 SST 파일일수록, 상위 레벨의 SST 파일보다 오래된 데이터가 위치한다. LSM-Tree 구조를 사용하는 RocksDB 는 높은 공간 효율성과 더 나은 쓰기 성능을 보장한다.

2. 3 WAL (Write-Ahead-Logging)

트랜잭션 수행도중에 시스템 오류 또는 전원 차단과 같은 문제가 발생 했을 때, 정상적으로 완료되었지만 아직 디스크에 반영되지 않고 버퍼에 남아있는 데이터는 유실된다. 시스템이 재부팅 된 다음 데이터베이스 시스템이 복구를 수행할 때, 트랜잭션이 어떠한 연산을 했는지 기록이 남아있지 않다면 정상적으로 복구를 수행 할 수 없다. 따라서 데이터베이스 시스템이 정상적으로 복구를 수행할 수 있도록 트랜잭션이 어떤 갱신 연산을 수행했는지 기록을 하는 로그를 사용한다. 로그를 기록하는 방법에는 Redo, Undo 그리고 Redo/Undo 로깅이 있으며 WAL(Write-Ahead-Logging)규칙에 따라 로그가 기록된다. WAL(Write-Ahead-Logging)은 트랜잭션으로 인해 변경된 데이터가 디스크에 기록되기 전에 먼저 관련된 로그를 로그 파일에 기록하는 규칙이다.

2. 4 NVRAM (Non-Volatile Random Access Memory)

NVRAM(Non-Volatile Random Access Memory)은 전원이 차단되거나 재부팅을 하면 메모리의 데이터가 사라지는 휘발성 메모리인 DRAM 과는 달리, 전원이 공급되지 않아도 저장된 정보를 유지 할 수 있는 비 휘발성 메모리이다. 현재 NVRAM 은 FeRAM, MRAM, PRAM 과 같이 메모리 자체가 비 휘발성의 특징을 가진 메모리와 휘발성 메모리인 DRAM 에 베터리 백업을 이용하여 비 휘발성을 띄게 된 메모리로 나뉜다. FeRAM, MRAM, PRAM 과 같은 메모리는 아직까지는 주 기억장

치로 사용하기에는 용량이 작고 단가가 비싸다는 단점이 있다. 이를 개선하고자 많은 연구와 개발이 지속되고 있다[13].

2. 5 RocksDB 성능 향상에 대한 기존 연구

RocksDB 의 성능 저하 요소인 compaction 오버헤드와 로깅 오버헤드를 개선하여 성능을 향상시키기 위한 연구와 분석이 지속적으로 이뤄지고 있다.

Key 와 value 를 분리시키고 value 의 위치를 가리키고 있는 포인터를 합친 연구[14]와 SSD 디바이스를 활용한 연구[9, 15]와 같이 compaction 오버헤드를 감소시켜 성능을 향상시키는 연구가 진행되었다. Compaction 오버헤드를 줄이는 연구 이외에도 파일 시스템 마다 얼마나 성능이 차이가 있는지 분석한 연구[11], 저장 장치의 종류를 바꿔보며 RocksDB 의 성능을 분석한 연구[10]와 동기와 옵션을 변경하며 동기화 옵션에 따른 RocksDB 의 성능을 분석한 연구[12]와 같이 데이터를 디스크에 기록하는 것과 관련된 연구들이 진행되었다.

3. RocksDB WAL(Write-Ahead-Logging) 오버헤드

LSM-Tree 구조는 디스크를 위한 별도의 로그를 기록하지 않는다. 그렇기 때문에, 전원 공급차단과 같은 문제가 발생하면 메모리영역에 있는 데이터들은 사라지게 된다. LSM-Tree 구조를 사용하는 RocksDB 는 데이터의 지속성을 위해 WAL 로깅을 사용한다. 하지만 추가적인 쓰기 연산으로 인하여 오버헤드가 발생하게 되었다.

본 논문에서는 WAL 로깅이 없는 이상적인 경우의 성능과 실제로 SSD 에 WAL 로깅으로 인한 오버헤드를 분석하고자 한다. 추가적으로, 차세대 비 휘발성 메모리인 NVRAM 을 활용하여 로깅 오버헤드의 개선정도를 파악하여 NVRAM 의 활용 가능성을 확인하고자 한다.

<표 1> 실험 환경

OS	CentOS 7.3.1611 (x86_64)
CPU	Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
RAM	64GB
NVRAM	NV1600 Flashtec(TM) NVRAM
NVME	Intel SSDPEDMD800G4 DC P3700 800GB
SSD	Crucial MX200 250GB SATA 2.5 256GB * 2

<표 2> 블록사이즈 4k 기준 디바이스 임의 쓰기 성능

Device	Write(KB/s)
SSD	86459
NVMe	174440
NVRAM	189395

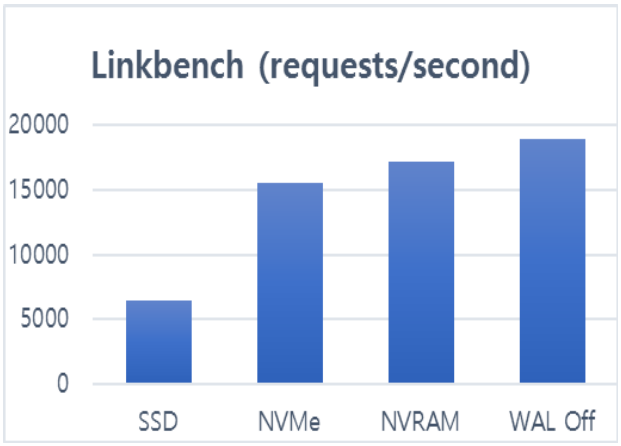
4. 실험결과 및 분석

4.1 실험 환경

본 논문에서는 로깅 오버헤드를 분석하기 위해 WAL 로깅을 했을 때와 WAL 로깅을 하지 않았을 때를 구분하여 실험하였다. 로그 파일 디렉토리는 SSD 와 NVMe 를 기반으로 진행하였으며 디바이스의 성능은 하드웨어 성능 평가 벤치마크인 fio 벤치마크[16]를 사용했다. 블록사이즈가 4k 일 때, 초당 얼마를 쓰는지 측정하였으며 디바이스의 쓰기 성능은 <표 2>와 같다. 추가적으로, DRAM 에 외부 베터리를 연결한 비 휘발성 메모리인 NVRAM 을 로그 파일 디렉토리로 활용하여 RocksDB 의 로깅 성능 향상도를 분석하였다.

모든 성능 측정은 Linkbench 벤치마크[17]와 TPC-C 벤치마크[18]에서 진행하였다. Linkbench 벤치마크는 만든 Facebook 에서 Social workload 에 대한 벤치마크 모델이다. TPC-C 벤치마크는 트랜잭션 처리 성능 평가 위원회에서 발표한 벤치마크로 도매상 전산 환경을 시뮬레이션 하는 OLTP 벤치마크이다.

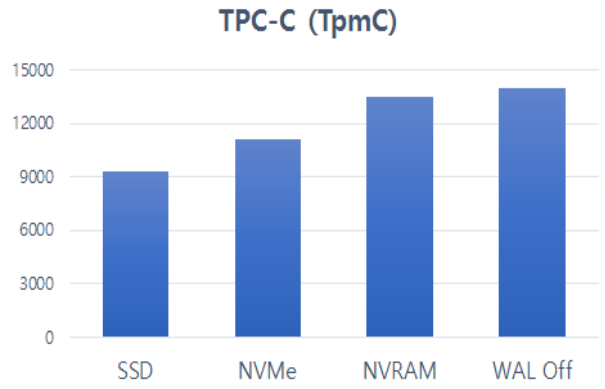
Linkbench 에서는 1000 만개의 Key-Value 쌍을 대상으로 하였으며 사용된 데이터의 크기는 3.5GB 이다. TPC-C 벤치마크에서는 warehouse 크기 200 에 concurrent client 의 수를 10 개로 설정하였으며 사용된 데이터의 크기는 약 7GB 이다.



(그림 3) Linkbench 실험 결과

<표 3> Linkbench 성능 결과

Device	Requests/second
SSD	6455
NVMe	15580
NVRAM	17174
WAL Off	18977



(그림 4) TPC-C 실험 결과

<표 4> TPC-C 성능 결과

Device	Requests/second
SSD	9270
NVMe	11100
NVRAM	13490
WAL Off	13977

4.2 실험 결과

SSD, NVMe 그리고 NVRAM 를 기반으로 두 벤치마크 실험을 한 결과, 두 벤치마크에서 비슷한 양상을 보였다. 디바이스 별로 WAL 로깅 성능 차이가 발생했다.

<표 3>과 <표 4>에서 보이듯이, 두 실험을 통해 WAL 로깅을 한 상태에서 SSD, NVMe, NVRAM 의 순으로 시간당 처리량이 높았으며, WAL 로깅을 하지 않았을 때 가장 높은 성능을 보였다.

일반 SSD 를 사용하여 WAL 로깅을 했을 때에는 다른 디바이스에 비해 상대적으로 낮은 쓰기 연산 속도로 인해 로그를 기록하는 속도가 오래 걸리므로 성능이 낮게 측정되었다. NVMe 는 일반 SSD 보다 빠른 쓰기 성능을 보유하고 있다. 따라서 NVMe 에서의 로깅 오버헤드가 SSD 에서의 로깅 오버헤드보다 더 적게 측정되었다. 마찬가지로 세 디바이스 중 가장 좋은 쓰기 연산 성능을 보유한 NVRAM 은 빠른 연산 속도로 인해 로깅 오버헤드가 가장 낮게 측정이 되었다. WAL 로깅을 하지 않았을 때는 로그를 기록하는 추가적인 쓰기 연산이 필요하지 않으므로 로깅 오버헤드가 발생하지 않았으며 벤치마크를 사용한 성능 평가에서 가장 높게 측정되었다.

5. 결론

본 논문에서는 성능 측정을 통하여 RocksDB 에서 발생하는 WAL 로깅 오버헤드를 알아보았고, 각기 다른 쓰기 성능을 보유한 디바이스를 통해 로깅 오버헤드가 얼마나 개선될 수 있는지 알아보았다.

WAL 로깅을 했을 때와 WAL 로깅을 하지 않았을 때의 결과를 통해 로깅 오버헤드가 성능 저하에 상당한 비중을 차지하고 있다는 것을 알 수 있다. 이에 더하여, 디바이스의 쓰기 성능이 WAL 로깅 오버헤드에 중요한 요인인 것을 파악 할 수 있다. 또, 실제로 NVRAM 을 활용하여 향후 연구에 사용될 NVRAM 의 가능성을 확인하였다.

향후에는 NVRAM 과 함께 WAL 로깅 알고리즘 개선을 하여 WAL 오버헤드를 더 감소시키는 연구를 진행하고자 한다.

참고문헌

[1] RocksDB, http://rocksdb.org/
[2] RocksDB, https://github.com/facebook/rocksdb
[3] Dong, Siying, et al. "Optimizing Space Amplification in RocksDB."
[4] MongoDB, https://www.mongodb.com/
[5] Cassandra, http://cassandra.apache.org/
[6] HBase, https://hbase.apache.org/
[7] Bigtable, https://cloud.google.com/bigtable/
[8] LevelDB, http://leveldb.org/
[9] 김태일, “SSD 를 활용한 Key-Value Database System 의 성능향상 연구”, 한양대학교, 2013 년
[10] 안미진, 오기환, 강운학, 이상원. (2014). RocksDB 의 동기화 성능 비교. 한국정보과학회 학술발표논문집, , 1516-1518.
[11] 이동윤, 한상훈, 정기식, 김진수. (2015). Key-value 시스템의 파일 시스템 별 성능 차이 분석. 한국정보과학회 학술발표논문집, , 1795-1797.
[12] 박연수, 오기환, 이종백, 강운학, 이상원. (2014). 동기화 옵션에 따른 RocksDB 성능 평가. 한국정보과학회 학술발표논문집, , 1731-1733.
[13] Mittal, Sparsh, and Jeffrey S. Vetter. "A survey of software techniques for using non-volatile memories for storage and main memory systems." *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016): 1537-1550.
[14] Lu, Lanyue, et al. "WiscKey: Separating Keys from Values in SSD-conscious Storage." *FAST*. 2016.
[15] Yang, Fei, et al. "Optimizing NoSQL DB on Flash: A Case Study of RocksDB." *Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), 2015 IEEE 12th Intl Conf on*. IEEE, 2015.
[16] http://freecode.com/projects/fio
[17] https://github.com/facebookarchive/linkbench
[18] TPC-C, http://www.tpc.org/tpcc/