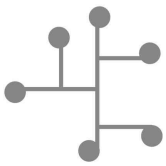


데이터 병렬성을 활용한 로그 파일 재구축 부하 개선 연구

A Study on Improvement of Log File Reconstruction Overhead Utilizing Data Parallelism

서주연(Juyeon Seo)[1] 성한승(Hanseung Sung)[2]
최원기(Won Gi Choi)[3] 박상현(Sanghyun Park)[4]

요 약

휘발성 장치인 DRAM을 주 저장소로 사용하는 인메모리 데이터베이스인 Redis는 신속한 데이터 처리의 장점을 지녔지만, 데이터 유실 위험이 크다. Redis는 데이터 유실 방지를 위해, 수행한 명령어를 로그 파일에 덧붙여 기록하는 AOF 기법을 제공한다. 하지만 계속되는 기록으로 인하여 로그 파일이 기약 없이 증가하는 현상이 발생한다. Redis는 로그 파일 재구축 기법을 제공하여 로그 파일 급증을 방지하지만, 부가적인 메모리 사용과 데이터 처리 지연이 요구되어 성능 및 자원 측면에서 부담이 가중된다. 본 연구에서는 로그 파일 재구축 시 발생하는 부하를 개선하기 위해 데이터 병렬성을 활용한 로그 파일 재구축 기법을 제안한다. 또한, 재구축 부하의 주요 원인이 되는 AOF Rewrite 버퍼 없이 재구축 작업을 수행하여, 메모리 사용량과 데이터 처리 성능을 향상시킨다. 제안하는 모델은 memtier-benchmark를 통해 평가되었으며, 재구축 부하가 크게 개선되었음을 확인하였다.

주제어: 로그 파일 재구축, 데이터 병렬성, 인메모리 데이터베이스, 레디스

1 연세대학교 컴퓨터과학과, 석사과정.
2 연세대학교 컴퓨터과학과, 석사 후 연구원.
3 연세대학교 컴퓨터과학과, 박사과정.
4 연세대학교 컴퓨터과학과, 교수, 교신저자.
+ 이 논문은 2020년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원(IITP-2017-0-00477, (SW스타랩) IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발)과 국토교통부의 스마트시티 혁신인재육성 사업의 지원을 받아 수행된 연구임
+ 논문접수: 2020년 11월 02일, 최종 심사완료: 2020년 12월 19일, 게재승인: 2020년 12월 26일.

Abstract

Redis, an in-memory database that uses DRAM which is a volatile device as its main repository, has the advantage of rapid data processing but the risk of data loss is high. To prevent the data loss, Redis provides AOF method that appends log records for performed commands to the log file. However, there is a constant increase in the log file due to continuous records. Although Redis provides log file reconstruction methods to prevent sudden increase in the log file, additional memory usage and latency in data processing are required, adding to the burden in terms of performance and resources. In this research, we propose the log file reconstruction method using data parallelism to improve the overhead occurred in the log file reconstruction. Also, we perform reconstruction without AOF Rewrite buffer that is the major cause of reconstruction overhead through our method, so the memory usage and data processing performance are improved. The proposed model is evaluated via memtier-benchmark and we confirmed the reconstruction overhead has been considerably improved.

Keywords: Log File Reconstruction, Data Parallelism, In-memory Database, Redis

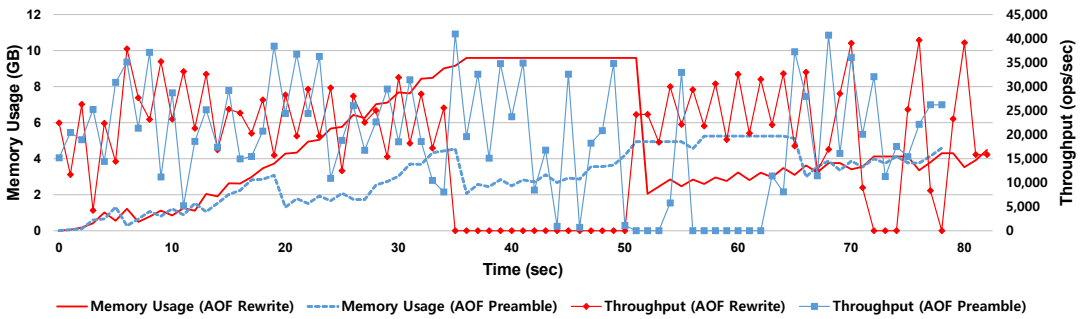
1. 서 론

IMKVS(In-Memory Key-Value Store) 또는 MMKVS(Main-Memory Key-Value Store)로 불리는 메모리 기반 데이터베이스는 HDD(Hard Disk Drive)와 SSD(Solid State Drive) 같은 디스크를 주 저장소로 사용하는 기존 시스템들과 달리, DRAM(Dynamic Random-Access Memory)을 주 저장소로 채택한 시스템을 지칭한다. 주 저장소로 DRAM이 사용됨에 따라, 메모리 기반 데이터베이스에 저장된 모든 데이터는 디스크에 기록되지 않고 메모리 계층에 상주한다. 메모리 기반 데이터베이스에서는 디스크 연산을 수행하지 않으므로, 매우 빠른 데이터 쓰기 및 조회가 가능하다는 이점이 있다. 하지만 전원 공급이 중단되면 저장된 데이터가 모두 사라지는 DRAM의 성질인 휘발성으로 인하여, 인메모리 데이터베이스에 저장된 데이터의 지속성을 보장하지 못하는 문제가 존재한다.

Aerospike[1], SAP HANA[2], Memcached[3]와 같은 메모리 기반 데이터베이스인 Redis[4]는 우수한 데이터 처리 성능 외에도 데이터 저장을 위한 다양한 자료구조와 기능을 제공하여, 실시간 데이터 처리가 요구되는 환경에서 가장 높은 사용률을 보이고 있다[5]. 일부 메모리 기반 데이터베이스는 사용 목적 및 개발 컨셉과 같은 이유로 인하여 데이터 지속성 방법을 제공하지 않는 반면, Redis는 DRAM의 휘발성으로부터 데이터를 보존하기 위한 두 가지 데이터 지속성 기법을 제공한다.

Redis에서 제공하는 데이터 지속성 기법 중 하나인 RDB(Redis Database)는 일정한 간격으로 스냅샷을 생성하는 기법으로, 우수한 로깅 및 복구 성능의 장점이 있다. 그러나 일정 주기마다 동작하는 RDB의 동작 방식으로 인하여, 데이터가 유실될 여지가 높다. AOF(Append-Only File)는 또 다른 지속성 방법으로써, 데이터셋을 변경하는 명령이 요청되었을 때, 해당 명령에 대한 로그 레코드를 로그 파일에 계속해서 덧붙여 나가는 방법이다. AOF 기법은 데이터셋 변경이 발생할 때마다 로그 파일에 기록하기 때문에 데이터 지속성을 보장할 수 있다. 그러나 로그 레코드를 파일의 끝에 계속해서 덧붙이기 때문에, 사용자의 요청이 계속되면 로그 파일의 크기가 지속적으로 증가하는 문제가 발생한다.

AOF 로그 파일이 과도하게 증가하는 것을 방지하기 위해, Redis는 AOF 파일을 재구성하는 두 가지 기법을 제공한다. 첫 번째 재구축 방법인 AOF Rewrite는 저장된 데이터셋을 구성하는 데 필요한 로그 레코드만을 다시 작성하여 AOF 파일 크기를 축소시킨다. AOF-USE-RDB-PREAMBLE이라 불리는 두 번째 방법은 AOF와 RDB를 혼합 사용하여 로그 파일을 재구축한다. 이러한 재구축 기법을 통



<그림 1> AOF Rewrite와 AOF-USE-RDB-PREAMBLE의 오버헤드 측정

해 로그 파일 크기에 관한 문제는 해결하였지만, 그림 1에 나타나 있듯이, 파일 재구축을 위한 부가적인 메모리 사용과 잦은 디스크 입출력이 요구되어, 시스템 자원 및 데이터 처리 성능 측면에서 심각한 부하를 유발한다. Redis의 데이터 처리 지연은 실시간 데이터 처리가 가능하다는 이점을 상실시키며, 저장 용량이 한정적인 DRAM에 모든 데이터를 저장하기 때문에, 메모리 사용량 증가는 저장소로서의 운영에 대한 부담을 가중시킨다. 따라서 로그 파일 재구축 작업 부하에 대한 개선이 필요하다.

본 논문에서는 로그 파일 재구축으로 인하여 발생하는 데이터 처리 성능 감소와 메모리 사용량 증가 현상을 개선하기 위해, 데이터 병렬성을 활용한 로그 파일 재구축 기법을 제안한다. 해당 기법은 자식 프로세스가 로그 파일 재구축 작업을 수행할 때 다수의 스레드를 할당하여, 로그 레코드 생성 및 기록 작업을 병렬로 수행함으로써 재구축 작업에 소요되는 시간을 단축시킨다. 또한, 메모리 사용량 급증과 빈번한 디스크 입출력의 원인이 되는 AOF Rewrite 버퍼 사용을 배제하여, 메모리 사용량을 절감시킴과 동시에 과도한 디스크 입출력을 방지하여 데이터 처리 성능을 향상시킨다. 본 논문에서 제안하는 기법을 통해, 현재 다수의 기업에서 사용하고 있는 Redis뿐만 아니라, AOF 방식을 채택하는 인메모리 데이터베이스에 적용하여 재구축 부하를 줄임으로써, 고가용성을 제공할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 Redis의 지속성 방법과 로그 파일 재구축 기법 및 관련 연구에 대하여 설명한다. 3장에서는 본 논문에서 제안하는 기법인 PAOF(Parallel Append-Only File)에 대해 소개한다. 4장에서는 PAOF 기법의 성능 평가를 위해, 기존 로그 파일 재구축 기법과 비교 실험을 진행한 결과를 해석한다. 마지막으로 5장에서는 결론과 향후 연구에 대해 설명한다.

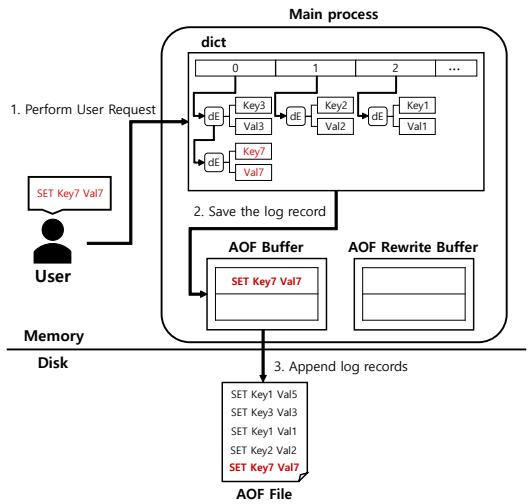
2. 배경 및 관련 연구

2.1 Append-Only File (AOF)

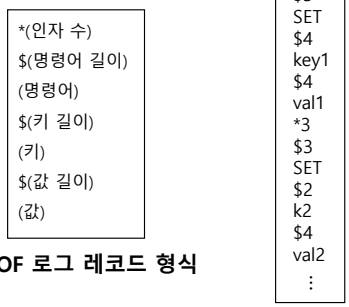
AOF 기법은 SET, DEL과 같이 데이터셋을 변경하는 명령을 수행하였을 때, 해당 명령에 대한 로그 레코드를 텍스트 형태로 파일에 덧붙여 기록하는 방법이다. AOF 로깅의 동작과정은 그림 2와 같다. Redis 서버로 데이터셋을 변경하는 명령이 요청되었을 경우, 해당 명령을 수행하고 수행한 명령에 대한 로그 레코드를 생성한다. 생성된 로그 레코드는 AOF 버퍼에 저장된다. Redis는 1초 간격마다 *fsync* 함수를 호출하여 AOF 버퍼에 저장된 로그 레코드를 AOF 파일에 기록한다.

일정 간격마다 동작하여 데이터 유실 위험이 있는 RDB와 달리, AOF는 매초마다 동작하기 때문에 데이터 지속성 보장이 가능하다. 따라서 Redis는 AOF 기법을 기본 지속성 방법으로 채택하고 있다.

시스템 오류로 인해 서버가 비정상 종료되는 경우, Redis는 디스크에 존재하는 로그 파일을 읽어 데이터를 복구한다. RDB 복구 방식은 RDB 파일에 기록된 키-값 로그 레코드를 하나씩 읽어 데이터셋



<그림 2> AOF 로깅 과정



(a) AOF 로그 레코드 형식

(b) AOF 파일 예시

<그림 3> AOF 로그 레코드 형식 및 예시

이 저장되는 공간에 바로 복구를 진행한다. 반면, AOF 복구 방식은 AOF 파일로부터 로그 레코드를 한 명령어씩 차례대로 읽고 수행하여, 서버가 종료된 시점까지의 데이터셋을 복구한다. 기록된 명령어 로그 레코드에 대한 복구 작업을 처음부터 다시 수행하기 때문에 복구 시간이 RDB에 비해 많이 소요된다.

AOF 기법은 데이터셋을 변경하는 명령어를 수행할 때마다 동작하므로, 하나의 데이터에 대한 여러 개의 로그 레코드가 존재할 수 있다. 그러나 RDB는 일정한 간격마다 동작하며, 하나의 키-값 데이터에 대해 한 개의 로그 레코드가 작성된다. RDB 로그 레코드는 값 데이터의 종류, 키, 값 형태로 구성된다. 이와 달리, AOF 로그 레코드는 그림 3과 같이, 명령어, 키, 값 외에도 각 요소의 길이에 대한 정보가 추가적으로 포함된다. RDB는 값 데이터를 압축하여 기록하는 반면, AOF는 값 데이터를 원본 그대로 기록하기 때문에 생성되는 로그 레코드가 RDB에 비해 비교적 크다.

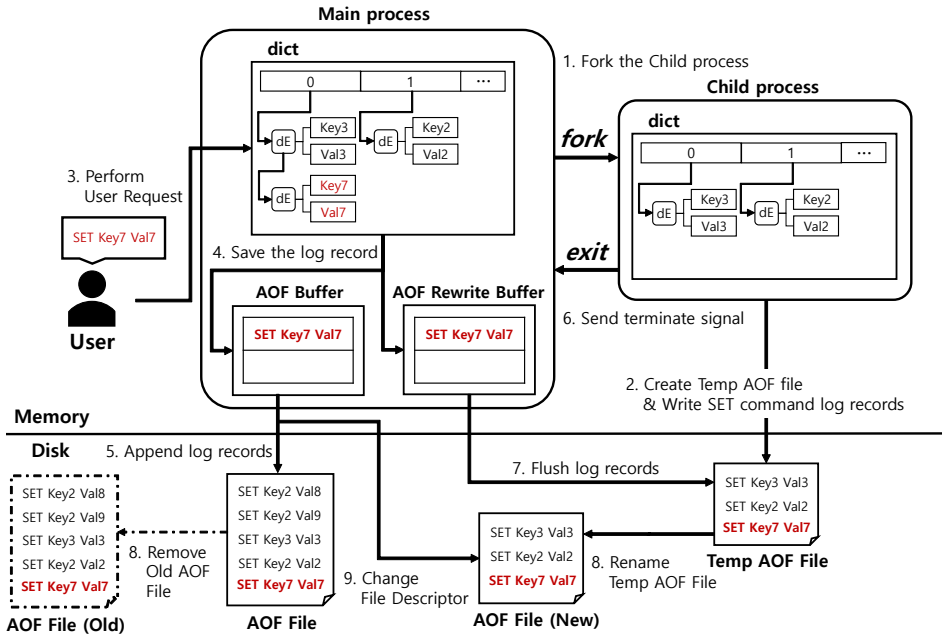
데이터셋 변경에 대한 로그 레코드를 덧붙여 기록하는 방법을 통해, 데이터 지속성을 보장할 수 있게 되었지만, 비교적 크기가 큰 로그 레코드가 계속하여 기록됨에 따라 로그 파일의 크기가 급격히 증가하는 문제가 발생한다. 무분별한 AOF 파일의 증가는 복구 성능 저하의 원인이 되며, 최악의 경우 시스템 파일 크기 제한을 초과하여 Redis가 동작하지 못하는 상황을 초래한다.

2.2 AOF Rewrite

Redis는 AOF 파일의 기약 없는 증가를 방지하기 위해, AOF Rewrite 기법을 제공한다. 해당 기법은 현재 데이터셋을 구축하는 데 필요한 로그 레코드만을 남겨 AOF 파일 크기를 감소시킨다. AOF Rewrite는 AOF 파일의 크기가 특정 임계값에 도달하면 최초로 동작한다. Redis는 AOF 파일 크기의 기본 임계값을 64 MB로 설정하였다.

로그파일 재구축을 위한 AOF Rewrite의 동작 과정은 그림 4와 같다. AOF 파일의 크기가 임계값을 초과하여 AOF Rewrite 작업이 호출되면, 메인 프로세스는 AOF 재구축 작업을 위한 자식 프로세스를 생성한다. 싱글 스레드 기반의 키-값 저장소인 Redis의 메인 프로세스가 AOF 파일 재구축 작업을 수행할 경우, 해당 작업을 수행하는 동안에는 사용자로부터 요청된 명령어는 처리되지 못하고 지체된다. 따라서 데이터 처리 성능 저하를 방지하기 위해, 자식 프로세스를 생성하여 AOF 재구축 작업을 수행한다.

메인 프로세스에 의해 생성된 자식 프로세스는 Temp AOF 파일을 생성한다. 그다음, 현재 시점까지 저장된 데이터셋의 각 엔트리마다 SET 명령어 로그 레코드를 생성하여, Temp AOF 파일에 기록한다. 자식 프로세스의 작업이 진행되는 동안 메인 프로세스는 사용자가 요청한 명령어를 수행한다. 메인 프로세스는 수행한 명령어에 대한 로그 레코드를 생성하여, AOF 버퍼와 AOF Rewrite 버퍼에 기록한다. AOF 버퍼의 로그 레코드는 기존 AOF 로깅 방식과 같이 *fsync* 주기에 따라 AOF 파일에



<그림 4> AOF Rewrite 동작 과정

기록되고, AOF Rewrite 버퍼의 로그 레코드는 자식 프로세스의 작업이 완료될 때까지 유지된다.

AOF Rewrite가 트리거 된 시점까지 저장된 데이터셋에 대한 로그 레코드 작성이 완료되면, 자식 프로세스가 종료된다. 자식 프로세스 종료 이후, 메인 프로세스는 AOF Rewrite 버퍼에 저장된 로그 레코드를 Temp AOF 파일에 기록한다. 그다음, Temp AOF 파일의 이름을 AOF 파일로 변경하고, AOF 버퍼의 파일 서술자(File Descriptor)를 새로 생성된 AOF 파일로 변경하면 AOF Rewrite 작업이 완료된다. 이후 Redis는 요청된 데이터 변경 명령에 대해 AOF 로깅 작업을 수행한다. 첫 번째 AOF Rewrite 작업이 완료된 이후부터는 AOF 파일 크기가 이전 종료 시점의 AOF 파일 크기의 2배가 되었을 때 다시 동작한다.

AOF Rewrite 기법을 통해 AOF 파일의 크기를 줄일 수 있게 되었지만, 해당 작업 동안 메모리 사용량이 급격하게 증가하고 데이터 처리 성능이 낮아지는 현상이 발생한다 (그림 1). Redis는 데이터 일관성 유지를 위해, AOF Rewrite가 동작 중 생성된 로그 레코드를 AOF Rewrite 버퍼에 추가적으로 저장한다. 동일한 로그 레코드가 두 버퍼에 중복으로 저장되기 때문에 메모리 사용량이 증가한다. 또한, AOF Rewrite 버퍼에 저장된 로그 레코드들은 자식 프로세스의 작업이 종료될 때까지 유지되기 때문에 메모리 점유가 발생한다.

AOF Rewrite 버퍼는 메모리 부하와 더불어 과도한 디스크 입출력 발생의 원인이 된다. AOF Rewrite 작업에서 자식 프로세스의 작업이 완료되면, AOF Rewrite 버퍼에 저장된 로그 레코드가 Temp AOF 파일에 기록된다. AOF Rewrite 버퍼에 저장된 로그 레코드를 기록하는 작업은 다량의 디스크 입출력을 유발하여, 상당한 시간이 소요된다. 이러한 작업을 메인 프로세스가 수행하기 때문에, 해당 작업을 수행하는 동안 사용자로부터 요청된 명령어는 처리되지 못하고 지연된다.

2.3 AOF-USE-RDB-PREAMBLE

AOF-USE-RDB-PREAMBLE(이하 AOF Preamble)은 AOF Rewrite 기법에서 발생하는 재구축 부하를 개선하기 위해 제안된 AOF 파일 재구축 기법이다. AOF Preamble의 가장 큰 특징은 AOF 기법과 RDB 기법을 동시에 활용한다는 것이다.

AOF Preamble은 AOF Rewrite와 동일한 동작 조건을 가진다. 먼저, AOF Preamble이 호출되면 RDB 작업을 수행할 자식 프로세스를 생성한다. 생성된 자식 프로세스는 Temp AOF 파일을 만든 다음, 생성한 파일에 현재 저장된 데이터셋에 대한 RDB 로그 레코드를 기록한다. 이때, 메인 프로세스는 사용자로부터 요청된 명령어를 처리하며, 처리한 명령어에 대해 AOF 로깅을 수행한다. AOF Rewrite와 마찬가지로, 생성된 로그 레코드는 AOF 버퍼와 AOF Rewrite 버퍼에 중복되어 저장된다.

RDB 작업을 완료한 자식 프로세스는 메인 프로세스로 종료 신호를 전송한다. 메인 프로세스는 자식 프로세스를 종료한 다음, AOF Rewrite 버퍼에 저장된 로그 레코드를 Temp AOF 파일에 기록하는 작업을 수행한다. AOF Rewrite 버퍼에 저장된 로그 레코드를 기록하는 *flush* 작업이 완료되면, Temp AOF 파일의 이름을 AOF 파일로 변경하는 작업을 수행하여 AOF Preamble 작업을 마무리한다. AOF Preamble 역시 AOF 파일 크기가 이전 작업 종료 시점의 AOF 파일 크기의 2배가 되었을 때 다시 호출된다.

AOF Preamble은 빠른 작업 및 복구 속도를 가진 RDB의 이점을 활용하여, AOF Rewrite 동작 중 발생하는 재구축 부하를 개선한다. AOF Preamble은 AOF와 RDB를 혼합하여 활용하므로, AOF Preamble에 의해 재구축된 로그 파일에는 AOF 로그 레코드와 RDB 로그 레코드가 공존한다. 그러므로 AOF Preamble을 통해 생성된 파일을 사용하여 데이터 복구를 수행할 때, RDB 복구와 AOF 복구 방법을 모두 사용한다.

RDB의 이점을 활용하여 재구축 작업의 부하가 어느 정도 개선되었지만, AOF Preamble에서도 재구축 작업으로 인한 부하가 여전히 남아있다 (그림 1). AOF Rewrite 작업과 같이, AOF Preamble 작업에서도 AOF Rewrite 버퍼 중복 저장과 *flush* 작업이 수행된다. 이로 인해, 메모리 사용량이 증가하고 데이터 처리 성능 저하되는 현상이 발생한다.

2.4 관련 연구

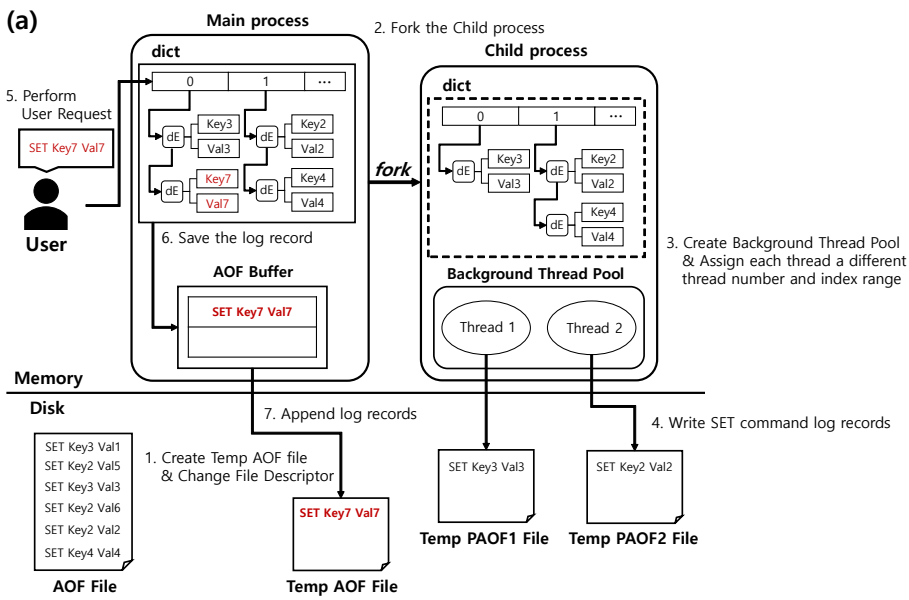
데이터 유실 위험으로부터 데이터를 보존하기 위한 지속성 방법이 제공되고 있으나, 심각한 성능 저하가 발생한다. 지속성 작업 부하의 원인을 분석하고 다양한 기법을 제안함으로써, 성능 저하를 완화하기 위한 연구가 계속해서 진행되고 있다. [6-8]은 Redis의 지속성 방법을 사용하며 발생하는 성능 저하의 원인을 분석하여, 기존 지속성 기법에 대한 개선의 필요성을 부각하였다. [6, 7]에서는 RDB와 AOF에 대한 자세한 서술과 함께, 서로 다른 워크로드에 대한 CPU 사용률, 메모리 사용량과 같은 다양한 부하를 측정하여, 각 기법의 로깅 및 복구 성능을 평가하였다. 또한, AOF Rewrite 기법을 사용할 때 메모리 부하와 동시에 데이터 처리 성능이 급격히 저하됨을 보여, 재구축 부하의 심각성을 강조하였다[8]. CP-Redis[9]는 저장소로 사용되는 Redis의 효용 가치를 높이기 위해, 데이터 압축 저장 기법과 함께 비압축식 병렬 스냅샷 생성 기법에 대한 연구를 진행하였다. CP-Redis는 데이터 병렬성을 활용하여 RDB 작업 성능을 향상시켰으나, 기존 RDB 작업 동작과 동일하게 일정한 주기마다 동작하여, 여전히 데이터 유실 위험이 남아있음을 밝혔다. [9]와는 달리, 본 논문은 데이터 병렬성을 활용하여 재구축 작업 성능을 향상시킴과 동시에 데

이터 유실 위험 문제를 해결한다. 또한, 기존 재구축 부하의 원인을 파악하고 개선하여, 데이터 처리 성능 및 운영 측면에서의 이점을 가진다.

최근에는 DRAM의 휘발성으로부터 데이터 손실을 방지하기 위해, NVM(Non-Volatile Memory)을 활용하여 지속성 작업의 부하를 줄이는 연구가 수행되고 있다[10-13]. NVM을 로그 버퍼로 사용한 PB-Redis[10]는 로그 레코드를 NVM에 저장하여 데이터 유실을 방지하지만, 데이터 적재가 계속되어 AOF 파일의 크기가 증가할 경우, 데이터 처리 성능은 급격히 저하하는 문제가 있다. 반면, LibreKV[11]는 NVM을 캐시(Cache) 구조로 사용하여, Redis, Memcached, LevelDB와 같은 데이터베이스보다 시스템 성능과 메모리 사용의 효율성을 높이고, 체크섬(Checksum)을 이용하여 데이터 일관성 및 지속성을 보장한다. [12]에서는 NVM에 최적화된 구조의 해쉬 테이블을 적용한 새로운 라이브러리를 개발하여, 키-값 데이터베이스에서 메모리를 효율적으로 할당하고, 데이터의 일관성을 유지한다. [13]은 플래시 메모리의 비휘발성과 DRAM의 빠른 데이터 전송 속도를 결합시킨 차세대 메모리인 SCM(Storage Class Memory)을 사용하여, Redis의 지속성을 보장하면서 성능을 개선하였다.

3. Parallel Append-Only File (PAOF)

본 논문에서는 AOF 파일 재구축 작업으로 인하여 발생하는 데이터 처리 성능 저하와 메모리 사용량 급증 현상을 개선하기 위한 Parallel Append-Only File(PAOF) 재구축 기법을 제안한다. 제안하는 기법은 데이터 병렬성을 활용하여 현재 데이터 셋에 대한 AOF 작성 작업을 병렬로 수행함으로써 작업 시간을 단축시킨다. 또한, 메모리 사용량 급증의 주요 원인인 AOF Rewrite 버퍼를 사용하지 않으면서도 AOF 재구축 작업 중에 요청된 명령어에 대한 데이터 지속성을 보장한다.



<그림 5> PAOF 동작 과정: (a) Temp AOF 파일 및 자식 프로세스 생성

3.1 PAOF 로깅

본 논문에서 제안하는 기법인 PAOF는 그림 5와 같이 동작한다. 기존의 AOF 파일 재구축 기법들과 마찬가지로, 메인 프로세스는 PAOF가 동작하기 전까지 AOF 로깅 작업을 수행한다. AOF 파일의 크기가 최초 동작의 임계값을 초과하면 PAOF 작업이 호출된다.

그림 5(a)에 나타나 있듯이, PAOF가 동작하면 메인 프로세스는 Temp AOF 파일을 생성한 다음, AOF 파일 서술자가 Temp AOF 파일을 가리키도록 변경한다. PAOF 작업 중에 Redis 서버로 요청된 명령어에 대한 로그 레코드는 Temp AOF 파일에 기록된다.

파일 서술자를 변경한 Redis의 메인 프로세스는 AOF 재구축 작업을 수행할 자식 프로세스를 생성한다. 생성된 자식 프로세스는 현재 데이터셋을 구축하기 위해 필요한 로그 레코드만을 남기는 작업을 병렬로 수행한다. 이때, 메인 프로세스는 사용자로부터 요청된 명령어를 수행한다.

로그 파일 재구축을 병렬로 수행하기 위한 자식 프로세스의 동작 과정은 알고리즘 1과 같다. 부모 프로세스로부터 생성된 자식 프로세스는 AOF 작성 작업을 병렬로 수행하기 위해, 백그라운드 스레드 풀을 할당한다. 그다음, 각 스레드마다 서로 다른 스레드 번호와 인덱스 범위를 부여하고, 스레드는 자신의 스레드 번호를 파일 이름에 포함시켜 Temp PAOF 파일을 생성한다. 각 스레드는 할당받은 인덱스 범위에 저장된 키-값 엔트리에 대한 SET 명령어 로그 레코드를 생성하여, 자신이 만든 Temp PAOF 파일에 기록한다. 이때, 각 스레드마다 서로 다른 스레드 번호와 인덱스 범위를 가지기 때문에, 스레드마다 생성한 로그 레코드들은 중복되지 않는다.

자식 프로세스가 작업을 수행하는 동안 메인 프로세스는 사용자가 요청한 작업을 수행한다. 만약 사용자로부터 요청된 작업이 데이터셋을 변경하는 작업일 경우, 수행한 명령어에 대한 로그 레코드를 생성한다. 생성된 로그 레코드는 AOF 버퍼에 저장되고, 추후 *fsync* 주기에 따라서 Temp AOF 파일에 기록된다.

그림 5(b)는 자식 프로세스가 작업을 모두 마친 후의 동작 과정을 나타낸다. 자식 프로세스가 SET 명령어 로그 레코드 생성 작업을 완료하면, 메인 프로세스에게 작업이 종료되었음을 알리는 신호를 보낸다. 종료 신호를 수신한 메인 프로세스는 자식 프로세스를 종료한다.

자식 프로세스가 종료된 후, 메인 프로세스는 Temp AOF 파일의 이름을 AOF 파일로 변경한다. Temp AOF 파일의 이름을 변경하는 작업이 완료되면, 메인 프로세스는 자식 프로세스의 스레드에 의해 생성된 Temp PAOF 파일들의 이름을 변경한다. Temp PAOF 파일의 이름을 변경할 때, 메인 프로세스는 Temp PAOF 파일의 이름에 포함된 파

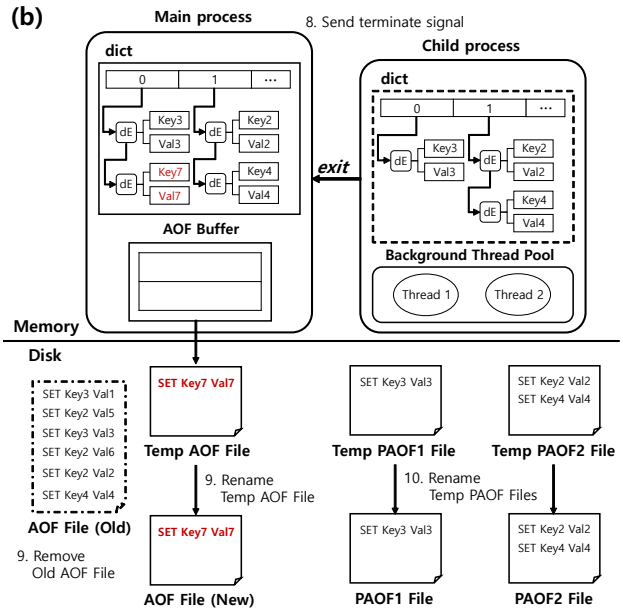
<알고리즘 1> 자식 프로세스의 AOF 파일 재구축 과정

Algorithm 1 PAOF of Child Process

Description

Create SET log records by allocating threads as many as the number of threads written in configuration file (max_thread_num)

```
1   i ← 0
    /* Create Thread pool */
2   ThreadPool threads[max_thread_num]
3   while i < max_thread_num do
    /* Assign Thread Number */
4     thread_num ← i + 1
    /* Allocate Index Range per thread */
5     min_index ← MinIndex(thread_num)
6     max_index ← MaxIndex(thread_num)
    /* Create Temp PAOF files */
7     TempPAOF[i] ← createTempPAOF(thread_num)
    /* Perform PAOF task */
8     pthread_create(threads[i], ReconstructionWithIndex(),
                     TempPAOF[i], min_index, max_index)
9     i ← i + 1
10  end while
```



<그림 5> PAOF 동작 과정: (b) 자식 프로세스의 작업 완료 후 디스크의 파일 이름 변경 과정

일 번호와 일치하도록 PAOF 파일 이름을 변경한다. Temp PAOF 파일의 이름 변경 작업이 완료되면 PAOF 동작이 정상 종료된다.

PAOF 작업이 완료된 이후, Redis는 이전과 같이 AOF 로깅을 수행하며, AOF 파일 크기가 이전 PAOF 작업 종료 시점 파일 크기의 2배가 되면, PAOF 작업이 다시 수행된다.

본 논문에서 제안하는 PAOF 기법은 데이터 병렬성을 활용하여 자식 프로세스의 작업 부하를 감소시킨다. 기존 AOF 재구축 기법은 하나의 스레드가 저장된 데이터셋을 하나씩 순회하며 키-값 엔트리에 대한 SET 명령어 로그 레코드를 생성한다. 기존 기법과는 다르게, PAOF 기법은 다수의 스레드를 할당하고 각 스레드마다 다른 인덱스 범위를 부여하여, SET 명령어 로그 레코드 생성 작업을 병렬 수행한다. 병렬 수행을 통해, PAOF의 자식 프로세스는 기존 재구축 기법들의 자식 프로세스보다 빠르게 작업을 완료할 수 있다.

Redis는 copy-on-write 기법을 활용하여 자식 프로세스를 생성한다. 생성된 자식 프로세스는 메인 프로세스와 현재 데이터셋에 대한 메모리 영역을 공유한다. 같은 메모리 공간을 공유하고 있기 때문에, 자식 프로세스가 작업을 수행하는 동안 데이터셋을 변경하는 명령이 요청되면, 메인 프로세스는 해당 데이터가 저장된 메모리 영역을 복사하고, 복사한 메모리 영역의 데이터셋을 변경한다. 이때, 메인 프로세스는 새로 복사된 메모리 영역을, 자식 프로세스는 원본 메모리 영역을 참조한다. 데이터 변경으로 인하여, 필요가 없어진 메모리 영역은 자식 프로세스의 작업이 끝날 때까지 유지된다. 따라서 자식 프로세스의 작업이 길수록 원본 데이터가 해당 메모리 영역을 점유하고 있는 시간이 증가한다. PAOF는 자식 프로세스 작업을 병렬로 수행하여 자식 프로세스 작업의 수행 시간을 감소시킴으로써, 원본 메모리 영역의 점유 시간을 줄일 수 있다.

기존 재구축 기법은 데이터 일관성 유지를 위한

AOF Rewrite 버퍼 사용이 요구되어, 메모리 사용량이 증가한다. 반면, PAOF 기법은 Temp AOF 파일을 별도로 생성하여 AOF 버퍼에 저장된 로그 레코드를 기록함으로써, AOF Rewrite 버퍼를 사용하지 않으면서도 데이터 일관성 유지가 가능하다. 로그 레코드의 중복 저장 문제를 해결함으로써, PAOF는 기존 AOF 파일 재구축 기법에서 AOF Rewrite 버퍼가 점유했던 만큼의 메모리 공간을 확보할 수 있다.

AOF Rewrite 버퍼 제거는 메모리 사용량 절감에 더하여 데이터 처리 성능 향상의 효과를 이끈다. 기존 재구축 기법들은 데이터 처리 성능 저하를 일으키는 *flush* 작업을 수행한다. 다량의 디스크 입출력을 유발하는 *flush* 작업은 작업이 완료되기까지 상당한 시간이 소요되며, 해당 작업 동안 요청된 명령어는 처리되지 못한다. 그러나 AOF Rewrite 버퍼를 사용하지 않는 PAOF 기법은 *flush* 작업이 요구되지 않기 때문에, 디스크 입출력 부하를 완화시켜 재구축 작업 중에도 데이터 처리 성능이 심각하게 저하되지 않는다. 또한, PAOF는 서버 자원 활용률을 극대화하는 이점을 가진다. 싱글 스레드 기반 프로그램인 Redis는 하나의 스레드만을 사용하기 때문에, 서버 자원을 최대로 활용하지 못한다는 단점이 존재한다. 반면, PAOF 기법은 자식 프로세스 작업 시 다수의 스레드를 사용하여, 시스템 자원을 효율적으로 활용할 수 있다.

3.2 PAOF 복구

하나의 파일만 사용하여 데이터셋을 메모리에 복구하는 기존 데이터 지속성 방법과 달리, PAOF 기법은 여러 개의 파일을 활용하여 데이터를 복구한다. PAOF 작업 중에는 최대 4가지 유형의 파일(AOF, Temp AOF, PAOF, Temp PAOF)이 존재하며, 각 단계마다 다른 유형의 파일이 생성된다.

<표 1> PAOF 동작 중 발생 가능한 비정상 종료 상황, 디스크 파일 종류, 복구 순서

오류 발생 시점	디스크 파일 종류	복구 순서
PAOF 동작 후 자식 프로세스 생성 전	AOF, Temp AOF, PAOF	1. PAOF 2. AOF 3. Temp AOF
자식 프로세스의 작업 완료 전	AOF, Temp AOF, PAOF, Temp PAOF	1. PAOF 2. AOF 3. Temp AOF
Temp AOF 파일 이름 변경 후	AOF, PAOF, Temp PAOF	1. Temp PAOF 2. AOF
Temp PAOF 파일 이름 변경 중	AOF, (Old) PAOF, Temp PAOF, (Renamed) PAOF	1. (Renamed) PAOF 2. Temp PAOF 3. AOF
Temp PAOF 파일 이름 변경 후	AOF, PAOF	1. PAOF 2. AOF

시스템 오류 혹은 충돌로 인하여 Redis가 다시 시작된 경우, 디스크에 저장된 파일의 종류와 개수를 검사하고, Redis가 비정상 종료된 시점을 유추하여 각 상황에 따라 다른 복구 절차를 진행한다.

데이터 병렬성을 활용하여 로깅 작업을 수행하는 PAOF의 자식 프로세스는 여러 개의 파일을 생성하여, 저장된 데이터셋에 대한 로그 레코드를 분산 저장한다. 그러므로 데이터 복구를 수행할 때 모든 PAOF 파일을 사용해야 한다. 표 1은 PAOF 로깅 동작 중에 발생할 수 있는 시스템 충돌로 인한 비정상 종료 상황, 디스크에 저장된 파일 종류, 그리고 데이터 복구 순서를 나타낸다.

3.2.1 PAOF 동작 후, 자식 프로세스를 생성하기 이전에 오류가 발생한 상황

AOF 파일을 재구축하기 위해 PAOF가 호출되면, Redis의 메인 프로세스는 Temp AOF 파일을 만들고 자식 프로세스를 생성한다. Redis가 다시 시작될 때, 디스크에 AOF 파일, Temp AOF 파일,

그리고 PAOF 파일만 있다면, PAOF가 동작한 이후 자식 프로세스를 생성하기 이전에 오류가 발생했음을 의미한다. 이 경우, Redis는 PAOF 파일, AOF 파일, Temp AOF 파일 순으로 파일을 읽어 데이터 복구를 진행한다. PAOF 파일을 AOF 파일보다 먼저 사용하는 이유는 다음과 같다. PAOF 파일에는 해당 파일이 생성된 시점까지 저장된 데이터셋에 대한 로그 레코드가 기록되어 있다. AOF 파일에는 이전 PAOF 작업이 호출된 이후부터 발생한 데이터 변경에 대한 로그 레코드가 기록되어 있다. PAOF 파일 이전에 AOF 파일을 복구에 사용하면, 데이터 일관성이 위반된다. 데이터 일관성과 지속성을 보장하기 위해, PAOF 파일을 읽어 전체 데이터셋을 구성한다. 그런 다음, AOF 파일과 Temp AOF 파일을 사용하여 데이터를 복구하고 갱신된 사항을 반영한다. Temp AOF 파일에는 가장 최근에 트리거 된 PAOF가 동작한 이후 요청된 명령어에 대한 로그 레코드가 포함되어 있기 때문에, Temp AOF 파일을 가장 마지막에 사용한다.

3.2.2 자식 프로세스의 작업이 완료되기 전에 오류가 발생한 상황

자식 프로세스가 작업하는 동안 시스템 충돌이 발생하면, 디스크에는 모든 유형의 파일이 존재한다. PAOF 로깅 작업에서는 자식 프로세스의 작업이 완료된 후에 Temp AOF 파일의 이름을 변경한다. 따라서 디스크에 Temp AOF 파일과 Temp PAOF 파일이 동시에 존재하면, 자식 프로세스의 작업이 완료되지 않았음을 나타낸다. 만약 자식 프로세스의 작업이 완료되자마자 오류가 발생했더라도 해당 작업이 완료되었음을 보장할 수 없다. 그러므로 위 상황에서 Temp PAOF 파일은 데이터 복구에 사용되지 않는다. Redis는 시스템 충돌이 발생하기 전까지 저장된 데이터를 복구하기 위해, PAOF 파일, AOF 파일, Temp AOF 파일을 순차적으로 읽으며 데이터 복구를 수행한다.

3.2.3 Temp AOF 파일의 이름을 변경한 후, 오류가 발생한 상황

디스크에 AOF 파일, PAOF 파일, 그리고 Temp PAOF 파일만 존재한다면, PAOF 파일의 개수와 Temp PAOF 파일의 개수를 통해 시스템 충돌이 발생한 시점을 유추할 수 있다. PAOF 파일과 Temp PAOF 파일의 수가 동일하다면, Redis가 Temp AOF 파일의 이름을 변경한 후에 시스템 오류가 발생했음을 의미한다. 본 논문에서 제안하는 기법에서, 메인 프로세스는 자식 프로세스의 작업이 완료되면 Temp AOF 파일의 이름을 변경한다. 이 때는 Temp PAOF 파일에 현재 데이터셋에 대한 모든 로그 레코드가 포함되어 있다는 것을 보장할 수 있다. 따라서 Redis는 기존의 PAOF 파일 대신에 Temp PAOF 파일을 사용하여 데이터셋을 구축한 후, 이름이 변경된 AOF 파일을 사용하여 갱신된 내용을 반영한다.

3.2.4 Temp PAOF 파일의 이름을 변경하는 중에 오류가 발생한 상황

3.2.3절과 동일한 유형의 로그 파일들이 존재하지만 PAOF 파일의 개수와 Temp PAOF 파일의 개수가 다르다면, Temp PAOF 파일의 이름을 변경하는 도중에 시스템 장애가 발생하였음을 의미한다. 그림 5(b)에 나타나 있듯이, 메인 프로세스는 Temp AOF 파일의 이름을 변경한 후, 모든 Temp PAOF 파일의 이름을 파일 번호 순서대로 하나씩 변경한다. 메인 프로세스가 하나의 Temp PAOF 파일의 이름을 PAOF 파일로 변경할 때마다 Temp PAOF 파일의 개수가 하나씩 감소한다. 따라서 디스크에 존재하는 파일의 유형이 3.2.3절과 동일하더라도 자

식 프로세스에서 생성한 파일들의 개수를 통해 시스템 장애 발생 시점을 식별할 수 있다. 위 경우에 대하여, Redis는 네 단계에 걸쳐 데이터 복구를 수행한다. 먼저, 데이터셋 복구를 위해 디스크에 존재하는 Temp PAOF 파일 중에 가장 작은 파일 번호를 가진 파일을 조회한다. 다음으로, 첫 번째 단계에서 찾은 Temp PAOF 파일보다 낮은 파일 번호를 가진 PAOF 파일들을 번호 순서대로 읽어 데이터 복구를 진행한다. 세 번째로, 디스크에 남아있는 모든 Temp PAOF 파일을 사용하여 데이터셋을 구축한다. 마지막으로, 이름이 변경된 AOF 파일을 사용하여 변경된 사항을 반영한다.

3.2.5 Temp PAOF 파일의 이름을 변경한 후, 오류가 발생한 상황

디스크에 하나의 AOF 파일과 여러 개의 PAOF 파일만 저장되어 있다면, Redis가 모든 Temp PAOF 파일의 이름을 변경한 후 또는 PAOF가 다시 호출되기 전에 시스템 충돌이 발생했음을 의미한다. PAOF 작업이 완료되어, 다시 동작할 때까지 디스크에 저장된 파일의 종류는 변경되지 않는다. 그러므로 두 경우에 대하여 동일한 데이터 복구 절차가 적용된다. 먼저, Redis는 모든 PAOF 파일을 사용하여 데이터셋을 재구성한다. 그다음, PAOF 작업 중에 변경된 내역을 반영하기 위해 AOF 파일에 기록된 로그 레코드를 수행하여 데이터 복구를 완료한다.

4. 실 험

본 실험에서는 PAOF가 데이터 병렬성을 활용함으로써 갖는 이점을 파악하기 위해, 로그 레코드 병렬 작성 시간과 복구 시간에 관한 실험을 진행하였다. 이를 바탕으로, 데이터 병렬성이 PAOF 기법에 미치는 영향을 분석하였다. 또한, 본 논문에서 제안하는 PAOF 기법과 기존 재구축 기법인 AOF Rewrite와 AOF Preamble 기법에 대한 재구축 부하를 비교하기 위한 실험을 수행하였다. 본 실험에서는 메모리 기반 데이터베이스 성능 측정에 가장 널리 사용되는 Memtier-benchmark[14]를 사용하여, 최대 메모리 사용량, 평균 메모리 사용량, 데이터 처리 성능, 데이터 복구 시간 및 생성된 로그 파일 크기를 측정하였다.

<표 2> 시스템 실험 환경

CPU	Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
DRAM	TEAMGROUP-UD4-2400 16GB * 4
HDD	ST3000DM008-2DM166 3TB
OS	CentOS Linux release 7.6.1810 (Core)
Redis version	5.0.7
Maxmemory	60 GB
appendfsync	everysec

4.1 실험 환경

본 논문에서 실험은 표 2와 같이 구성된 서버에서 진행하였다. Redis가 사용 가능한 메모리양을 나타내는 maxmemory는 서버 최대 가용 용량인 60 GB로 설정하였다. 또한, *fsync* 함수 호출 주기를 나타내는 appendfsync는 기존 Redis의 기본값인 everysec으로 설정하여, 매초마다 AOF 버퍼에 저장된 로그 레코드가 AOF 파일에 기록된다.

4.2 자식 프로세스의 재구축 작업 성능 평가

PAOF의 전체 성능을 평가하기에 앞서, 데이터 병렬성이 로그 레코드 생성에 미치는 영향을 파악하기 위해, PAOF의 자식 프로세스가 생성하는 스

레드 수를 2, 4, 8, 16개로 증가시키며 자식 프로세스의 로그 레코드 작성 시간을 측정하였다. 또한, 실험을 통해 생성된 파일을 사용한 데이터 복구 시간을 측정하여, 데이터 병렬성이 데이터 복구에 미치는 영향을 확인하였다.

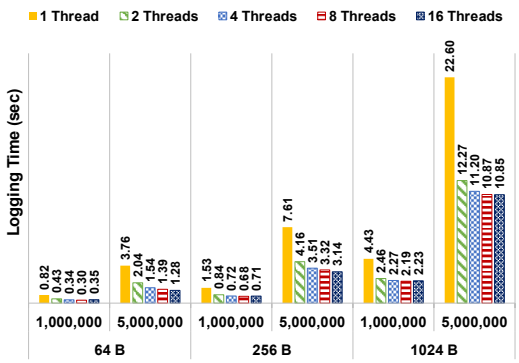
4.2.1 로그 레코드 작성 시간

본 실험에서는 스레드 개수 변경에 따른 자식 프로세스의 로그 레코드 작성 시간을 비교하기 위해, 1,000,000건과 5,000,000건의 키-값 엔트리에 대한 로그 작성 시간을 측정하였다. 적재되는 키의 크기는 16 B로 고정되며, 값 데이터는 64 B, 256 B, 1024 B로 점차 증가하는 크기를 가진다. 해당 실험에서 스레드 개수가 1인 경우는 AOF Rewrite의 자식 프로세스의 성능을, 2 이상인 경우에는 PAOF의 자식 프로세스의 성능을 나타낸다.

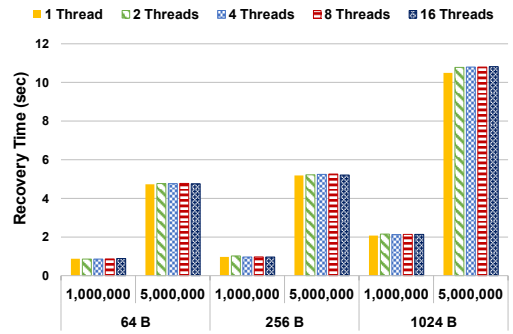
로그 레코드 작성 시간 실험 결과는 그림 6과 같다. 1,000,000건의 데이터에 대한 로그 작성 실험에서는, 사용되는 스레드의 개수가 1개에서 8개로 증가할수록, 로깅에 소요되는 시간이 점차 단축되었으나, 단축되는 감소폭은 점차 줄어들었다. 이와는 달리, 16개의 스레드를 사용하였을 때, 8개의 스레드를 사용했을 때보다 로깅 시간이 증가하였다.

자식 프로세스가 생성하는 스레드 수를 두 배로 증가시키면, 각 스레드가 생성 및 작성해야 할 로그 레코드 수가 절반으로 줄어, 각 스레드에 대한 로깅 부하가 감소된다. 그러나 이전보다 두 배 많은 파일이 생성되어, 이름 변경 작업을 수행해야 할 파일의 개수가 증가하므로, 이름 변경 작업에 대한 소요 시간이 증가한다.

1,000,000건의 데이터에 대한 실험에서는, Redis에 저장된 데이터의 개수가 충분하지 않았기 때문에, 로그 레코드 병렬 생성을 통한 시간 단축 효과가 다른 경우에 비해서 크지 않았다. 로그 레코드 병렬 생성을 통해 단축된 시간과 이름 변경 작업이



<그림 6> 스레드 개수에 따른
자식 프로세스의 로그 레코드 작성 시간



<그림 7> 생성된 파일을 사용한 데이터 복구 시간

완료되기까지 증가된 시간을 비교하였을 때, 시간 단축 효과가 크지 않으므로, 더 많은 스레드를 사용하더라도 오히려 시간이 증가하였다. 반면, 5,000,000건의 키-값 쌍에 대한 실험 결과에서는 적재된 키-값 쌍 데이터의 개수가 충분하여, 로그 레코드 작성에 사용되는 스레드의 개수가 증가할수록 로그 작성에 소요되는 시간이 감소하였다.

4.2.2 데이터 복구 시간

자식 프로세스의 로그 레코드 작성 실험을 통해 생성된 파일들을 사용하여 데이터 복구 시간을 측정하였다. 그림 7은 데이터 복구 작업이 완료되기까지 소요되는 시간을 나타내며, 각 스레드 수는 데이

터 복구 시 읽어야 할 파일 개수를 의미한다.

그림 7을 통해, 각 워크로드에 대하여 스레드 수가 증가하였음에도 불구하고, 데이터 복구 시간이 비슷하게 소요되었음을 알 수 있다. Redis는 싱글 스레드 기반이기 때문에 데이터 변경 작업은 메인 프로세스만이 수행한다. 데이터 복구 작업은 데이터 셋을 변경하는 작업이므로, Redis 서버 재시작 시 데이터 복구는 메인 프로세스가 진행한다. 본 실험을 통해, 데이터 복구 시간은 복구할 데이터의 개수에 의해 결정되는 것을 알 수 있다.

4.3 PAOF 스레드 수 변경 실험

본 실험에서는 데이터 병렬성이 PAOF의 전체 성능에 미치는 영향을 파악하기 위한 실험을 수행하였다. 16 B 크기의 키와 1 KB 크기의 값을 가진 엔트리에 대한 요청 수가 5,000,000건으로 구성된 워크로드에 대해 PAOF 작업 중에 사용되는 스레드 개수를 달리하여, 데이터 처리 성능, 실행 시간, 평균 메모리 사용량, 생성된 로그 파일 크기 및 복구 시간을 측정하였다.

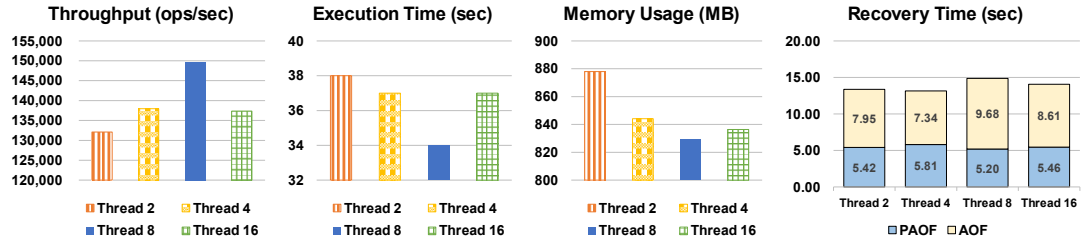
그림 8에서와 같이, 자식 프로세스가 생성하는 스레드 수가 2개에서 8개로 늘어날수록 초당 데이터 처리 성능이 증가하였으며, 이에 따라 워크로드 수행 시간이 단축되었다. 그러나 사용되는 스레드 수가 16개일 때는 8개인 경우보다 데이터 처리 성능이 감소하여, 워크로드 작업이 완료되기까지 소요되는 시간이 증가하였다. 워크로드 수행을 위해 스레드를 더 많이 사용했음에도 불구하고, 과도한 스레드 사용으로 인하여 오히려 데이터 처리 성능이 저하되었다.

메모리 사용량을 측정한 결과, 워크로드 수행 시간 측정 결과와 비슷한 양상을 보였다. 수행 시간이 단축될수록 평균적으로 적은 메모리를 사용하였다. 해당 실험을 통해, 과도한 스레드 사용은 데이터 처리 성능 저하와 함께 메모리 사용량 증가를 야기하는 것을 알 수 있다.

그림 8의 복구 시간 그래프와 표 3은 생성된 로그 파일의 크기에 의해 데이터 복구 시간이 달라진다는 것을 보여준다. 실험 결과를 통해, 전체 데이터 복구 시간은 생성된 AOF 파일 크기에 의해 결정되는 경향이 있음을 확인하였다. 대부분의 경우 PAOF 파일을 사용한 복구 시간은 약 5초 정도 유사하게 측정되었지만, AOF 파일에 대한 복구 시간은 차이가 존재하였다. 8개의 스레드를 사용한 PAOF에 의해 생성된 AOF 파일 크기는 약 1.47 GB이며, AOF 파일 복구 시간은 9.68초로 가장 오랜 시간이 소요되었다. 반대로, 4개의 스레드를 사

<표 3> PAOF에 의해 생성된 로그 파일 크기

Number of threads	AOF File size	PAOF file size
2	1.21 GB	409 MB per file
4	1.1 GB	210 MB per file
8	1.47 GB	95 MB per file
16	1.32 GB	49 MB per file



<그림 8> 스레드 개수에 따른 PAOF 성능

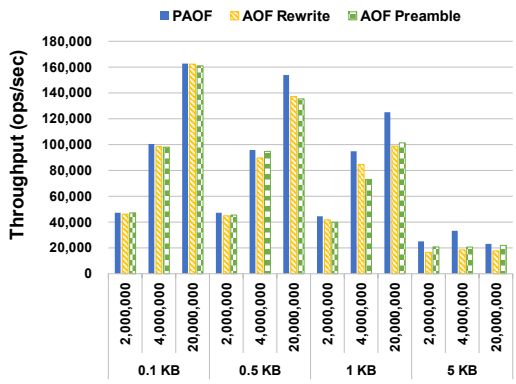
용했을 때 생성된 AOF 파일 크기는 약 1.1 GB이며, 가장 짧은 데이터 복구 시간인 7.34초로 측정되었다. 전체 데이터 복구 시간을 비교하면, 8개의 스레드를 사용했을 때는 14.88초로 가장 오랜 데이터 복구 시간이 기록되었으며, 스레드 수가 4개인 경우는 13.15초로 가장 짧은 데이터 복구 시간이 측정되었다. 이를 통해, 데이터 복구 시간은 생성되는 AOF 파일 크기에 의해 결정되는 것을 알 수 있다. PAOF의 자식 프로세스가 사용하는 스레드의 개수가 8개일 때, 데이터 처리 성능, 실행 시간, 메모리 사용량이 가장 이상적인 결과를 보였으므로, 이후 실험에서 PAOF의 자식 프로세스가 사용하는 스레드의 개수를 8개로 설정하여 진행하였다.

4.4 PAOF 성능 평가

본 논문에서 제안하는 PAOF 기법의 성능을 평가하기 위해, PAOF 기법과 기존 AOF 파일 재구축 기법과의 성능 비교 실험을 수행하였다. 본 실험에서 요청되는 데이터 키의 크기는 16 B로 고정된 크기를 가진다. 요청되는 개별 값의 크기를 0.1 KB, 0.5 KB, 1 KB, 5 KB로 증가시키고, 명령어 요청 횟수를 2,000,000건, 4,000,000건, 20,000,000건으로 증가시키며, 다양한 환경에서의 데이터 처리 성능, 최대 메모리 사용량, 평균 메모리 사용량, 그리고 데이터 복구 시간을 측정하였다.

4.4.1 데이터 처리 성능

그림 9는 각 워크로드에 따른 PAOF, AOF Rewrite, AOF Preamble의 데이터 처리 성능을 나타낸다. PAOF는 모든 경우에서 기존 기법보다 데이터 처리 성능이 높게 측정되었으나, 엔트리의 값의 크기가 0.1 KB인 경우는 데이터 처리 성능이 유사한 결과를 보였다. 엔트리의 값의 크기가 작을 때는 상대적으로 작업 소요 시간이 적기 때문에,



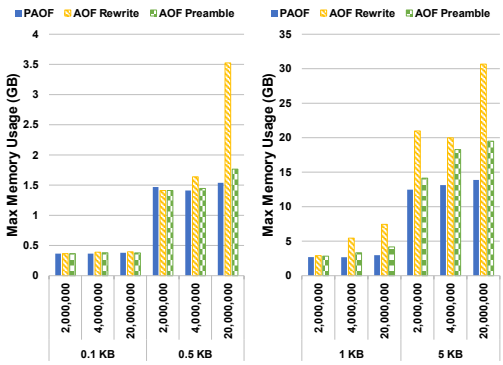
<그림 9> 워크로드 별 데이터 처리 성능

PAOF의 데이터 병렬성의 이점이 부각되지 않았다. 그러나 키-값 엔트리의 값의 크기가 증가할수록 PAOF와 기존 기법의 데이터 처리 성능의 차이가 점차 증가하였다. 특히, 엔트리의 값의 크기가 5 KB이고 요청하는 명령어가 4,000,000건으로 구성된 워크로드에 대한 데이터 처리 성능을 측정해본 결과, AOF Rewrite는 18,395 ops/sec가 측정되었지만, PAOF는 33,272 ops/sec로, PAOF의 데이터 처리 성능이 AOF Rewrite 보다 80.9% 향상되었다.

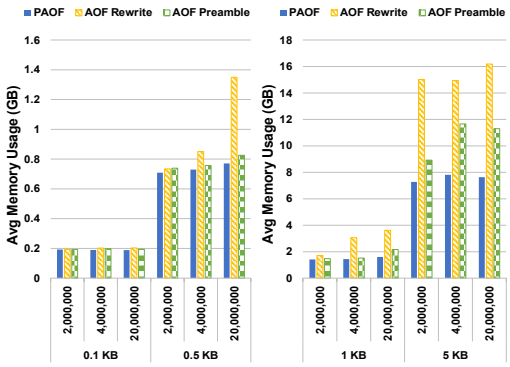
PAOF는 자식 프로세스 작업 시 데이터 병렬성을 활용하여, 자식 프로세스의 작업을 가속화한다. 기존 재구축 기법은 데이터 처리 지연의 원인이 되는 *flush* 작업을 수행하지만, PAOF 기법은 *flush* 작업을 수행하지 않으므로, 그림 9와 같이 기존 기법보다 높은 데이터 처리 성능을 이끌어낼 수 있다.

4.4.2 메모리 사용량

수많은 요청과 큰 값으로 구성된 데이터셋에 대한 데이터 지속성을 보장하기 위해서는 많은 양의 메모리가 요구된다. 인메모리 데이터베이스는 디스크에 비해 한정된 용량을 가진 메모리에 모든 데이터를 저장한다. 순간적인 메모리 사용량 급증은 시



<그림 10> 워크로드 별 최대 메모리 사용량



<그림 11> 워크로드 별 평균 메모리 사용량

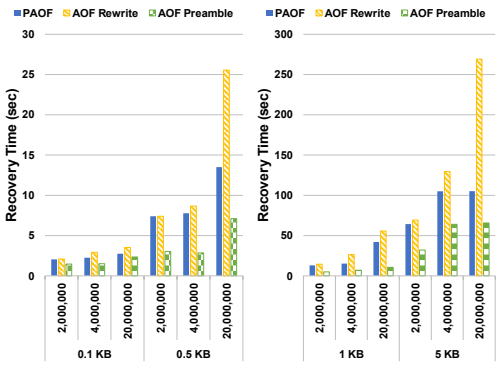
스템 비정상 종료를 발생시킬 위험이 있으며, 긴 시간 동안 메모리 사용량이 증가한 상태를 유지할 경우, 저장공간 부족으로 인한 시스템 정지를 초래한다. 그러므로 부가적으로 요구되는 메모리 소비를 절감하는 것이 매우 중요하다.

그림 10은 워크로드를 수행하는 동안 발생하는 재구축 작업으로 인해 AOF Rewrite 및 AOF Preamble의 최대 메모리 사용량이 크게 증가하는 것을 보여준다. 기존 방법은 로그 레코드를 중복 저장하여 메모리 사용량이 급격히 증가하지만, 로그 레코드를 중복 저장하지 않는 PAOF는 요청 횟수와 관계없이 최대 메모리 사용량이 거의 일정하다. 그림 10에 나타나 있듯이, 엔트리의 값의 크기가 1 KB이고 명령어 요청 횟수가 20,000,000건일 경우, PAOF의 최대 메모리 사용량은 약 2.96 GB, AOF Rewrite는 약 7.43 GB, AOF Preamble은 약 4.17 GB로 측정되어, PAOF의 최대 메모리 사용량은 AOF Rewrite에 비해 60.2%, AOF Preamble에 비해 29.0%까지 절감되었다. 실험 결과를 통해, PAOF가 기존 기법보다 메모리 부족으로부터 안전하다는 것을 시사한다.

앞서 언급한 본 논문에서의 접근방식과 기존 접근방식의 차이로 인하여, PAOF를 사용한 Redis는 최대 메모리 사용량뿐만 아니라 평균적으로도 가장 적은 메모리 사용량을 보였다. 그림 11은 대부분의 경우 값의 크기와 요청 횟수가 증가할수록 기존 방법의 평균 메모리 사용량과 제안하는 방법의 평균 메모리 사용량 간의 차이가 증가하는 것을 보여준다. 해당 실험에서, 엔트리의 값의 크기가 1 KB이고 요청되는 명령어가 20,000,000건인 경우, PAOF는 평균적으로 1.61 GB의 메모리를 사용한 반면, AOF Rewrite와 AOF Preamble은 각각 3.62 GB, 2.17 GB의 메모리를 사용하여, PAOF는 AOF Rewrite보다 55.5%, AOF Preamble보다 25.8%의 메모리 사용 절감 효과를 보였다. 4.4절의 실험 결과를 통해, PAOF는 데이터 처리 성능이 저하되지 않으면서, 메모리 사용량을 절감시키는 효과를 알 수 있다. 따라서 PAOF 방법은 다양한 상황에서 고가용성을 제공한다.

4.4.3 데이터 복구 시간

데이터 처리 성능 및 메모리 사용량 외에도, 데이터 복구에 관한 실험을 수행하였다. 데이터 복구 실험에서는 워크로드를 수행하여 생성된 로그 파일을 사용하였으며, 워크로드를 수행 후 Redis를 다시 시작하여 복구 시간을 측정하였다. 모든 상황에 대하여, 세 가지 지속성 방법 모두 데이터를 온전히 복구하였지만, 복구 시간에는 차이가 존재하였다.



<그림 12> 워크로드 별 데이터 복구 시간

값의 크기가 0.5 KB, 요청 수가 2,000,000건으로 구성된 워크로드를 수행하여 생성된 로그 파일을 사용한 데이터 복구에서는 PAOF와 AOF Rewrite 데이터 복구 시간이 동일하게 소요되었다. 두 기법 모두 워크로드가 종료하기 직전에 재구축 작업이 동작하여, 생성된 로그 파일의 크기가 동일하였다.

이를 제외한 모든 경우에서 PAOF 기법을 통해 생성된 로그 파일을 사용한 데이터 복구 시간이, AOF Rewrite 기법을 통해 생성된 로그 파일을 사용한 데이터 복구보다 짧게 소요되었다. 값의 크기가 5 KB이고, 요청된 명령어 개수가 20,000,000건인 경우, AOF Rewrite는 269.11초, PAOF는 105.11초의 데이터 복구 시간이 소요되어 가장 큰 차이를 보였다.

PAOF는 AOF Rewrite보다 빠른 속도로 재구축 작업을 완료하여, 작은 크기의 AOF 파일을 생성한다. 재구축 작업은 작업 종료 시점의 파일 크기에 의해 다음 작업의 시점이 결정된다. PAOF는 기존 재구축 기법보다 작은 크기의 파일을 생성하므로, 기존 재구축 기법보다 이른 시점에 호출된다. PAOF는 짧은 간격으로 AOF 파일에 존재하는 중복된 로그 레코드를 제거하기 때문에, 데이터 복구 시 수행해야 하는 복구 횟수를 감소시켜, AOF Rewrite보다 더 빠르게 데이터 복구를 완료한다.

데이터 복구 시간 실험의 모든 경우에서, AOF Preamble의 데이터 복구 시간이 가장 낮게 측정되었다. AOF Preamble에 의해 생성된 로그 파일을 사용하여 데이터 복구를 수행할 때, AOF 복구와 RDB 복구가 동시에 사용된다. AOF 복구와 달리, 복구할 데이터를 데이터 저장공간에 바로 저장하는 RDB 복구의 이점을 활용하여, AOF Preamble은 PAOF보다 데이터 복구가 완료되기까지 적은 시간이 소요되었다.

5. 결 론

저장된 데이터를 메모리 계층에 유지하는 메모리 기반 데이터베이스는 동작 속도가 우수한 DRAM의 이점을 활용하여, 신속한 데이터 처리가 가능하다는 장점을 지니고 있다. 모든 데이터를 메모리에 상주시킴으로써 빠른 데이터 처리 속도를 가지지만, DRAM의 휘발성으로 인하여 저장된 데이터가 유실될 위험이 크다. Redis는 데이터셋을 변경하는 작업에 대한 로그 레코드를 계속해서 덧붙이는 방법인 AOF 기법을 제공하여 데이터 유실을 방지한다. 하지만 로그 레코드가 지속적으로 기록됨에 따라, 로그 파일의 크기가 무분별하게 증가하는 문제를 야기한다. 이를 방지하기 위해, 로그 파일 재구축 기법을 제공하여 AOF 파일의 크기를 축소시키지만, 파일 재구축 작업을 위해 요구되는 부가적인 메모리 사용과 데이터 처리 지연으로 인하여, Redis의 메모리 사용량이 증가하고 데이터 처리 성능이 저하되는 문제가 발생한다.

본 논문에서는 기존 AOF 파일 재구축 기법이 가지는 재구축 부하를 개선하기 위해, PAOF 기법을 제안하였다. 데이터 병렬성을 활용하는 PAOF는 자식 프로세스의 작업 속도를 가속화하여, 재구축 작업에 소요되는 시간을 단축한다. 또한, 재구축 부하의 주요 원인이 되는 AOF Rewrite 버퍼를 사용하

지 않으면서도 데이터 재구축 작업을 정상적으로 수행하여, 데이터 처리 성능 향상과 동시에 메모리 사용량을 절감시키는 이점을 가진다.

PAOF의 성능을 평가하기 위해 다양한 환경에서 실험을 진행한 결과, PAOF는 AOF Rewrite보다 데이터 처리 성능이 최대 80.9% 향상되었다. 또한, 최대 메모리 사용량은 60.2%, 평균 메모리 사용량은 55.5%, 복구 시간은 최대 60.9%까지 감소되어, 재구축 부하가 크게 개선되었음을 확인하였다. AOF Preamble과의 비교 실험에서는, PAOF의 데이터 처리 성능이 최대 61.3% 향상되었으며, 최대 메모리 사용량은 29.0%, 평균 메모리 사용량은 33.0%까지 절감되었다. 그러나 AOF Preamble은 데이터 복구 시 RDB 복구와 AOF 복구를 함께 사용하여, RDB의 이점을 가지기 때문에 PAOF보다 데이터 복구 시간이 적게 소요되었다.

본 논문에서 제안하는 PAOF 기법은 재구축 부하를 완화시켰지만, AOF Preamble보다 데이터 복구 시 더 많은 시간이 요구되어 개선이 필요하다. 향후 연구에서는 데이터 복구 작업을 병렬적으로 처리하여, PAOF 기법의 복구 성능을 향상시키는 연구를 수행하고자 한다.

참고 문헌

[1] Srinivasan V, Bulkowski B, Chu WL, Sayyaparaju S, Gooding A, Iyer R, Shinde A, Lopatic T, “Aerospike: architecture of a real-time operational DBMS”, Proceedings of the VLDB Endowment, Vol.9, no.13, pp.1389-1400, 2016

[2] Sikka V, Färber F, Goel A, Lehner W, “SAP HANA: the evolution from a modern main-memory data platform to an enterprise application platform”, Proceedings of the VLDB Endowment, Vol.6, no.11, pp.1184-1185, 2013

[3] Memcached. https://memcached.org

[4] Redis. https://redis.io

[5] DB-Engine Ranking. https://db-engines.com/en/ranking/key-value+store

[6] Bao X, Liu L, Xiao N, Lu Y, Cao W, “Persistence and Recovery for In-Memory NoSQL Services: A Measurement Study”, IEEE International Conference on Web Services (ICWS), pp.530-537, 2016

[7] Cao W, Sahin S, Liu L, Bao X, “Evaluation and Analysis of In-Memory Key-Value Systems”, IEEE International Congress on Big Data, pp.26-33, 2016

[8] Jin MH, Park SH, “Overhead Analysis of AOF Rewrite Method in Redis”, The Journal of Korean Institute of Information Technology, Vol.15, no.3, pp.1-10, 2017

[9] Sung HS, Park SH, "Data Compression Storage and Parallel Snapshot Generation Technique in In-memory Key-Value Stores", Database Research, Vol.35, no.2, pp.33-53, 2019

[10] Kim DY, Choi WG, Sung HS, Lee JH, Park SH, "A Persistent Log Buffer Technique using Non-volatile Memory for In-Memory Key-Value Databases", The Journal of Korean Institute of Information Scientists and Engineers, Vol.45, no.11, pp.1193-1202, 2018

[11] Liu H, Huang L, Zhu Y, Shen Y, "LibreKV: A Persistent In-Memory Key-Value Store", IEEE Transactions on Emerging Topics in Computing, pp.1-12, 2017

[12] Huang K, Zhou J, Huang L, Shen Y, "NVHT: An efficient key-value storage library for non-volatile memory", Journal of Parallel and Distributed Computing, Vol.120, pp.339-354, 2018

[13] Giles E, Doshi K, Varman P, "Persisting In-Memory Databases Using SCM", IEEE International Conference on Big Data, pp.2981-2990, 2016

[14] Memtier-Benchmark. https://github.com/RedisLabs/memtier_benchmark

서 주 연

2020년 건국대학교 컴퓨터공학과 학사

2020년~현재 연세대학교 컴퓨터과학과 석사과정

관심분야 : 데이터베이스, 키-값 저장소

성 한 승

2017년 한국항공대학교 소프트웨어학과 학사

2020년 연세대학교 컴퓨터과학과 석사

2020년~현재 연세대학교 컴퓨터과학과 석사 후 연구원

관심분야 : 데이터베이스, 키-값 저장소, 리커버리

최 원 기

2014년 연세대학교 컴퓨터과학과 학사

2014년~현재 연세대학교 컴퓨터과학과 박사과정

관심분야 : 데이터베이스 시스템, 빅데이터, 분산처리 시스템

박 상 현

1989년 서울대학교 컴퓨터공학과 학사

1991년 서울대학교 대학원 컴퓨터공학과 공학석사

2001년 UCLA 대학원 컴퓨터과학과 공학박사

1991년~1996년 대우통신 연구원

2001년~2002년 IBM T. J. Watson Research Center Post-Doctoral Fellow

2002년~2003년 포항공과대학교 컴퓨터공학과 조교수

2003년~2006년 연세대학교 컴퓨터과학과 조교수

2006년~2011년 연세대학교 컴퓨터과학과 부교수

2011년~현재 연세대학교 컴퓨터과학과 교수

관심분야 : 데이터베이스, 데이터마이닝, 바이오 인포매틱스, 빅데이터 마이닝 & 기계학습