메모리 기반 키-값 저장소에서의 데이터 압축 저장과 병렬 스냅샷 생성 기법

Data Compression Storage and Parallel Snapshot Generation Technique in In-memory Key-Value Stores

성한승(Hanseung Sung)[1] 박상현(Sanghyun Park)[2]

요 약

실시간 데이터 처리와 저장이 요구되는 환경에서 메모리 기반 키-값 저장소인 Redis가 널리 사용되고 있다. 그러나 디스크보다 제한된 저장 용량을 가진 메모리가 주 저장 장치로 사용되기 때문에, 저장 가능한 데이터의 양이 다소 제한적이다. 또한, 시스템 장애가 발생할 경우 DRAM의 휘발성으로 인해 저장된 모든 데이터가 유실되는 문제가 있다. 이를 방지하기 위한 영속성 방법을 제공하지만, 영속성을 위한 작업으로 인해 데이터 처리 성능이 저하되고 메모리 사용량이 급증하는 문제가 있다. 본 논문에서는 LZF 알고리즘을 통해 데이터를 압축 저장하여 메모리의 용량 제한을 극복하는 데이터 압축 저장 기법과 스냅샷을 생성 시 발생하는 로깅 부하를 완화하기 위해 데이터 병렬성을 활용하는 병렬 스냅샷 생성 기법을 제안한다. 본 논문에서 제안하는 방법을 적용한 Redis는 기존 Redis보다 적은 메모리 사용량을 보였으며, 스냅샷 생성 및 데이터 복구에 소요되는 시간이 크게 단축되었음을 확인하였다.

주제어: 인-메모리 키-값 데이터베이스, 레디스, 데이터 압축, 스냅샷, 데이터 병렬성

1 연세대학교 컴퓨터과학과, 통합과정.

2 연세대학교 컴퓨터과학과, 교수, 교신저자.

+ 이 논문은 2019년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (IITP-2017-0-00477, (SW스타랩) IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발)

+ 논문접수: 2019년 07월 11일, 심사완료: 2019년 08월 01일, 게재승인: 2019년 08월 09일.

Abstract

Redis, a memory-based key-value store, is widely used in environments where real-time data processing and storage is required. However, because a memory with a limited storage capacity compared to disk is used as the primary storage device, the amount of data that can be stored is rather limited. In addition, in the event of system failure, there is a problem that all stored data are lost due to the volatility of the DRAM. Although it provides a persistence method to prevent this, there is a problem that data processing performance is degraded and memory usage is rapidly increased due to work for durability. In this paper, we propose a data compression storage technique to overcome the capacity limitation of memory by compressing and storing data through LZF algorithm and a parallel snapshot creation method that utilizes parallelism to alleviate the logging overhead that occurs when creating a snapshot. Redis using the proposed methods showed less memory usage than the existing Redis, and identified that the time required for snapshot creation and data recovery is also significantly reduced.

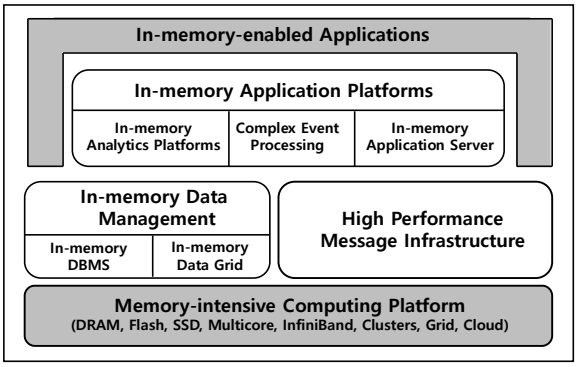
Keywords: In-memory Key-Value Database, Redis, Data Compression, Snapshot, Data Parallelism

1. 서 론

하드웨어 발전에 따른 메모리 가격 하락, 대용량 데이터의 빠른 양산 속도, 실시간 데이터 처리에 대한 수요가 증가함에 따라 인-메모리 컴퓨팅 기술이 부각되었다. 인-메모리 데이터 관리(In-memory Data Management) 기술 [1,2]은 빅데이터의 등장과 맞물려 그림 1과 같이 가트너가 분류한 인-메모리 컴퓨팅 기술 중에 가장 주목받는 기술로 자리매김하였다 [3].

메모리 기반 키-값 저장소 [4-10]는 인-메모리 데이터 관리 기술로서 DRAM(Dynamic Random-Access Memory)을 주 저장 장치로 사용하며, 데이터를 키-값 쌍의 형태로 저장하고 관리하는 데이터베이스를 일컫는다. 메모리 기반 키-값 저장소에서 모든 데이터 저장 및 조회는 메모리상에서만 이루어지기 때문에 데이터 처리 성능이 매우 빠르며, 다양한 자료구조를 통해 효율적인 데이터 관리가 가능하다. 하지만 메모리 기반 저장소는 디스크보다 상대적으로 저장용량이 작은 DRAM을 주 저장 장치로 사용하기 때문에, 저장 가능한 데이터 양이 다소 제한적이다. 또한 시스템 오류가 발생하여 종료되는 경우, DRAM의 특징인 휘발성으로 인하여 메모리에 저장된 모든 데이터가 손실되는 위험이 있다.

메모리 기반 키-값 저장소 중 하나인 Redis는 시스템 오류로 인한 데이터 유실을 방지하기 위해 RDB(Redis Database)와 AOF(Append-Only File)라 불리는 두 가지 데이터 영속성 방법을 제공한다 [11]. RDB는 특정 시점까지 저장된 데이터 세트를 압축된 바이너리 형식의 스냅샷으로 생성하는 방법이다. 또 다른 영속성 방법인 AOF는 데이터 삽입, 수정 또는 삭제에 대한 로그 레코드를 로그 파일에 덧붙이며 기록하는 방법이다. 이러한 방법을 통해 데이터 영속성을 제공할 수 있지만, 영속성을 위한 부가적인 작업으로 인하여 데이터 처리 성능이 저하되고 메모리 사용량이 급증하게 된다. 이러한 메모리 사용량의 증가는 가용 용량이 제한적인 메모리 기반 저장소에 부담을 가중한다.



[그림 1] 인-메모리 컴퓨팅 기술 분류

이와 같은 데이터 영속성 작업 부하를 줄이기 위해서 메모리 기반 저장소의 영속성 방법에 대한 연구가 지속해서 이루어지고 있다 [12-18]. Redis의 등장 이후, RDB 로그 파일과 AOF 로그 파일의 형식 및 특징에 대한 포렌식 관점에서의 분석 [12], Redis에서 제공하는 지속성 방법들 (RDB, AOF-Everysec, AOF-Always, AOF-No)의 복구 성능과 생성된 각 로그 파일의 크기에 대한 다양한 워크로드에서의 비교 분석 [13], 그리고 또 다른 메모리 기반 저장소인 Memcached와의 전반적인 성능과 CPU 및 메모리 사용량을 비교한 연구 [14]와 같이 Redis의 지속성 방법에 대한 특징 및 성능 분석과 관련된 연구가 주를 이루었다. [15]에서는 AOF 파일의 크기를 줄이기 위해 수행되는 AOF-Rewrite 중에 발생하는 메모리 증가 현상과 처리량 감소 현상에 대하여 분석하였다. 워크로드를 수행하는 동안 변화하는 메모리 사용량과 데이터 처리량을 측정하여, AOF-Rewrite 동작 중에 사용되는 AOF Rewrite 버퍼와 해당 버퍼의 내용을 파일에 기록하는 과도한 디스크 입출력이 원인임을 밝혔다. LESS [16]는, AOF-Rewrite 작업 중에 자식 프로세스에

서는 RDB를 수행하고 부모 프로세스에서는 AOF를 사용하는 방법을 제시하였다. 이를 통해 데이터 처리량이 향상되었지만, 빈번한 디스크 입출력으로 인하여 여전히 정지 상태(Block state)가 발생하였다. 최근에는 비휘발성 메모리가 상용화됨에 따라 메모리 기반 저장소에 데이터 영속성을 제공하고 빠른 처리 성능을 유지하기 위해 비휘발성 메모리를 활용하는 연구가 수행되고 있다 [17,18].

본 연구에서는 DRAM이 가진 용량 제한을 극복하고 보다 더 많은 데이터를 효율적으로 저장하기 위해 압축 알고리즘을 활용한 데이터 압축 저장 기법을 제안한다. 제안하는 방법은 한정적인 용량을 가진 메모리의 용량 제한을 극복하기 위해 일정 크기 이상의 데이터 삽입이 요청될 때, 고속 압축 알고리즘을 통해 데이터를 압축하고 저장하여 데이터 저장에 소모되는 메모리를 최소화한다. 또한, 저장된 데이터 양에 비례하여 증가하는 로깅 부하를 완화함으로써 고가용성 시스템을 구축하기 위해 데이터 병렬성을 활용하는 병렬 스냅샷 생성 기법을 제시한다. 스냅샷을 생성할 때, 데이터 세트를 분할하고 다수의 쓰레드를 통해 병렬 수행하여 빠르게 스냅샷을 생성함으로써 로깅 중 발생하는 부하를 감소시킨다.

본 논문에서 제시한 두 기법을 Redis에 적용하여 성능을 평가한 결과, 요청되는 데이터의 크기와 개수가 증가할수록 메모리 사용량이 크게 절감되었으며, 저장된 데이터의 크기와 개수가 증가할수록 스냅샷 생성 성능과 데이터 복구 성능이 크게 향상되는 것을 확인하였다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 본 연구의 이해를 돕기 위한 Redis 관련 배경지식을 소개하고, 3장에서는 Redis의 기존 저장 방식의 한계점 및 RDB 로깅 부하에 대하여 메모리 관점에서 자세히 분석한다. 4장에서는 본 논문에서 제안하는 기법인 데이터 압축 저장 기법과 병렬 스냅샷 생성 기법을 소개한다. 5장에서는 본 연구에서 제안하는 두 기법을 구현한 모델인 CP-Redis에 대해 Memtier-benchmark로 검증한 실험 결과를 분석하고, 마지막 6장에서 결론 및 향후 연구에 대하여 기술한다.

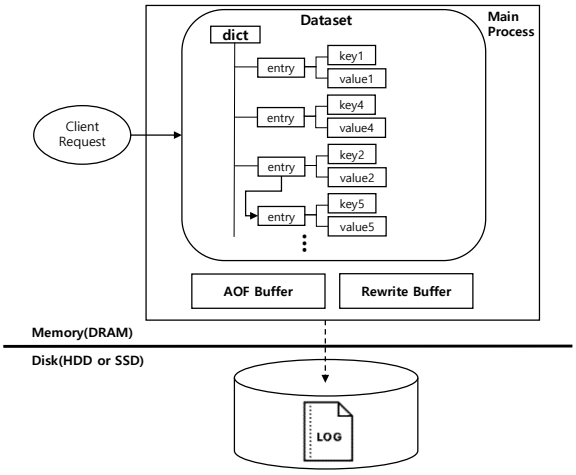
2. 배 경

2.1 Redis

Redis는 DRAM을 주 저장 장치로 사용하는 단일 쓰레드 기반의 메모리 기반 키-값 저장소이다. Redis는 효율적인 데이터 관리를 위해 string, hash, list, sorted set과 같은 다양한 자료구조를 제공하며 데이터 저장, 변경 및 삭제 외에도 약 160개의 다양한 기능을 제공한다. 분산 환경에서도 Redis를 사용할 수 있도록 클러스터 모드와 파티셔닝을 지원한다. 앞서 언급한 Redis의 장점을 넘어, Redis의 가장 큰 장점은 데이터 처리 성능이 매우 빠르다는 것이다. 또한 Redis는 시스템 장애로부터 저장된 데이터가 유실되는 것을 방지하기 위해 영속성 방법을 제공한다. Redis는 이러한 특징으로 인하여 실시간 데이터 처리가 필요한 환경에서 널리 사용되고 있다.

2.2 Redis의 데이터 저장 방법

Redis에서 저장된 데이터를 관리하는 구조인 dict는 그림 2와 같다. Redis에서 제공하는 자료구조 중에서 hash 구조의 메모리 사용량이 가장 적기 때문에 Redis는 기본 자료구조로 hash 구조를 사용한다. Redis의 dict구조에서는 요청된 키와 값에 대한 데이터를 하나의 엔트리(dictEntry)로 묶어서 저장한다. 엔트리에는 키에 대한 포인터, 값에 대한 Redis 객체 포인터, 다음 엔트리를 가리키는 엔트리 포인터가 포함되어 있다. 엔트리에 등록된 키는 문자열로, 값은 Redis 객체로 저장된다. Redis 객체는



[그림 2] Redis dict 구조

요청된 값 뿐만 아니라 데이터 타입, 참조 횟수, 시간, 값에 대한 포인터를 가지고 있다.

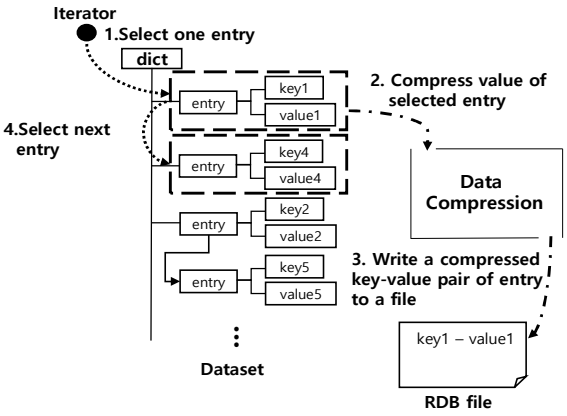
Redis 클라이언트 프로그램을 통해 Redis 서버에 키-값 쌍의 데이터 적재가 요청된 경우, Redis 서버는 요청된 파라미터에 대한 오류 검사 및 타입 분석을 수행한다. 요청된 파라미터에 이상이 없다면, 키에 대한 문자열의 hash 값을 계산하여 데이터가 저장될 dict의 위치를 결정한다. dict에 저장하고자 하는 키를 가진 엔트리가 없는 경우, 요청된 키와 값을 새로운 엔트리로 만들어서 dict에 저장한다. 만약 dict에 저장하고자 하는 키를 가진 엔트리가 이미 존재하는 경우, 별도의 엔트리를 할당하지 않고 해당 엔트리의 값만 요청된 값으로 변경한다. 마지막으로, 계산된 hash 값이 동일하지만 저장된 엔트리의 키가 다른 경우, 기존에 저장된 엔트리의 다음 포인터(Next pointer)가 새로 생성된 엔트리를 포인팅하여 요청된 데이터를 저장한다.

2.3 RDB (Redis Database)

Redis에서 제공하는 영속성 방법의 하나인 RDB는 메모리에 저장된 전체 데이터에 대하여 압축된 바이너리 형식으로 파일에 기록하는 방법이며, 주로 데이터베이스의 스냅샷을 생성하는 기법으로 알려져 있다. RDB는 일정 시간 안에 특정 횟수 이상의 데이터 삽입 및 변경이 발생하면 동작한다. 최초 RDB가 동작한 다음부터는 일정 주기마다 스냅샷을 생성한다.

RDB를 통해 생성된 로그 파일 상단에는 RDB 버전, Redis 버전, 생성 시간, 현재 메모리 사용량에 대한 정보가 기록된다. 시스템 정보가 기입된 다음에는 데이터베이스 번호와 함께 각 데이터베이스에 저장된 모든 키-값 쌍의 데이터가 기록된다. RDB 로그 파일의 마지막 부분에는 로그 파일의 무결성을 보호하기 위한 체크썸(checksum)이 기록된다.

2.3.1 RDB 로깅



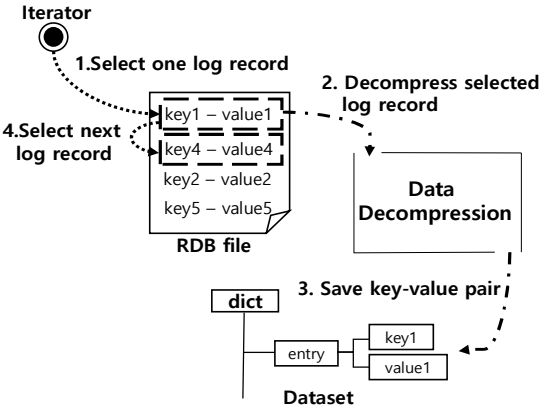
[그림 3] 자식 프로세스의 RDB 로깅 과정

RDB 로깅 방법은 일정한 간격마다 저장된 데이터 세트에 대한 스냅샷을 생성한다. 싱글 쓰레드 프로그램인 Redis에 여러 클라이언트가 동시에 다수의 명령을 요청하더라도 한 번에 하나의 명령만 실행할 수 있다. RDB 생성 작업이 메인 프로세스에서 수행되는 경우, 해당 작업이 완료될 때까지 사용자에 의한 요청은 지연될 수 있다. Redis는 RDB 생성으로 인한 성능 저하를 최소화하기 위해, RDB 파일을 생성하는 자식 프로세스를 생성한다. 자식 프로세스가 스냅샷을 생성하는 동안, 메인 프로세스는 사용자로

부터 요청된 명령어를 수행한다. 자식 프로세스의 RDB 로깅 과정은 그림 3과 같다. RDB 작업이 트리거 되면 Redis는 새로운 임시 RDB 로그 파일을 생성하고 RDB버전, Redis버전과 같은 Redis 정보를 기록한다. 그 다음, Redis는 데이터 세트에 저장된 엔트리 중 하나를 선택하여 데이터 압축 가능 여부를 검사한다. 압축으로 인한 부하를 최소화하기 위해, Redis는 LZF(Lempel-Ziv-Free) [19] 알고리즘을 사용하여 데이터를 압축한다. LZF는 압축할 문자열 데이터를 디코딩하여, 압축 조건을 만족하는 구간을 찾아 특정한 형식으로 데이터를 압축하는 라이브러리이다. 특히, DNA 염기서열 데이터와 같이 반복된 형태를 가진 데이터에 대하여 높은 압축 효율을 보인다. 압축 속도가 빠르고 압축 시 사용되는 메모리 양이 작기 때문에 Redis는 압축 알고리즘으로써 LZF를 사용한다. 선택된 엔트리의 값이 압축이 가능한 데이터인 경우, 값을 압축하여 임시 RDB 파일에 기록한다. 반대로, 압축이 불가능한 데이터인 경우에는 압축하지 않고 임시 RDB파일에 기록한다. Redis는 데이터 세트에 저장된 모든 키-값 쌍에 대하여 위의 과정을 반복한다. Redis에 저장된 모든 데이터가 로그 파일에 기록되면, Redis는 임시 로그 파일의 마지막에 체크썸을 기록한다. 마지막으로, 자식 프로세스에서 생성한 임시 RDB 파일의 이름을 현재의 RDB 로그 파일로 변경하면 RDB 생성 작업이 완료된다.

Redis에서 데이터 영속성을 제공하는 방법 중 나머지 하나인 AOF는 Redis가 수행한 명령어, 키, 값을 문자열 형식 그대로 기록하는 반면, RDB는 저장된 키와 값만 바이너리 형태로 기록한다. 압축이 가능한 데이터에 대해서는 압축하여 기록하기 때문에 작은 크기의 RDB 파일을 생성할 수 있다. 또한, AOF는 데이터에 대한 모든 변경 사항에 대하여 로그 레코드를 기록한다. 이로 인하여, 하나의 데이터에 대하여 다수의 로그 레코드가 존재할 수 있지만, RDB는 하나의 데이터에 대하여 오직 하나의 로그 레코드만 존재한다. 따라서 RDB에 의해 생성된 파일의 크기는 동일한 데이터 세트에 대해 AOF에 의해 생성된 파일보다 크기가 작다.

2.3.2 RDB 복구



[그림 4] RDB 복구 과정

시스템 오류가 발생하여 Redis가 재시작될 때, Redis는 로그 파일을 사용하여 데이터를 복구하기 위한 작업을 수행한다. 먼저, Redis는 복구에 사용하고자 하는 RDB 로그 파일에 기록된 체크썸, Redis 버전, RDB 버전을 확인하여 로그 파일의 무결성을 검사한다. 로그 파일의 무결성에 이상이 없다면 해당 로그 파일에 기록된 데이터를 복구하는 작업을 수행하며, 무결성에 이상이 있다면 Redis가 종료된다. RDB 로그 파일을 사용한 데이터 복구는 RDB 파일에 기록된 모든 데이터에 대하여 순차적으로 진행되며, 각각의 데이터에 대한 복구는 그림 4와 같이 진행된다. Redis는 RDB 로그 파일에 기록된 로그 레코드를 읽고, 해당 로그 레코드의 값에 대하여 압축 여부를 확인한다. 해당 로그 레코드의 값이 압축되어 있는 경우, 압축해제 함수를 통해 압축을 해제한 다음, dict에 키-값 데이터를 저장한다. 이와 반대로, 압축되지 않은 데이터는 dict에 바로 저장된

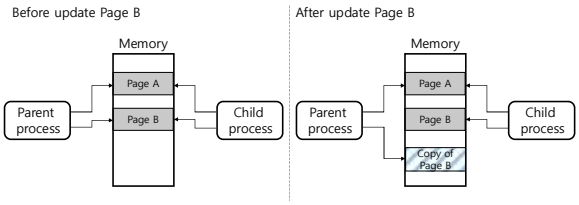
다. 데이터에 대한 저장이 완료된 이후, Redis는 다음 로그 레코드를 읽고 RDB 로그 파일에 기록된 모든 로그 레코드에 대하여 순차적으로 위의 과정을 반복한다.

다른 영속성 방법인 AOF와 비교했을 때, 데이터 복구 속도는 RDB가 AOF보다 빠르다. 수행했던 명령어를 재실행하여 초기 단계를 수행해야 하는 AOF 복구 방법과 달리, RDB 복구 방법에서는 데이터를 dict에 바로 저장한다. 또한 AOF에서는 하나의 키에 대하여 여러 번의 복구 작업을 수행하게 될 수 있지만, RDB는 하나의 키에 대하여 한 번의 복구 작업만 수행하여 데이터를 복구한다. 단순한 데이터 복구 절차와 적은 수행 횟수의 이점을 통해, RDB는 AOF보다 빠른 복구 성능을 보인다.

3. 연구 동기

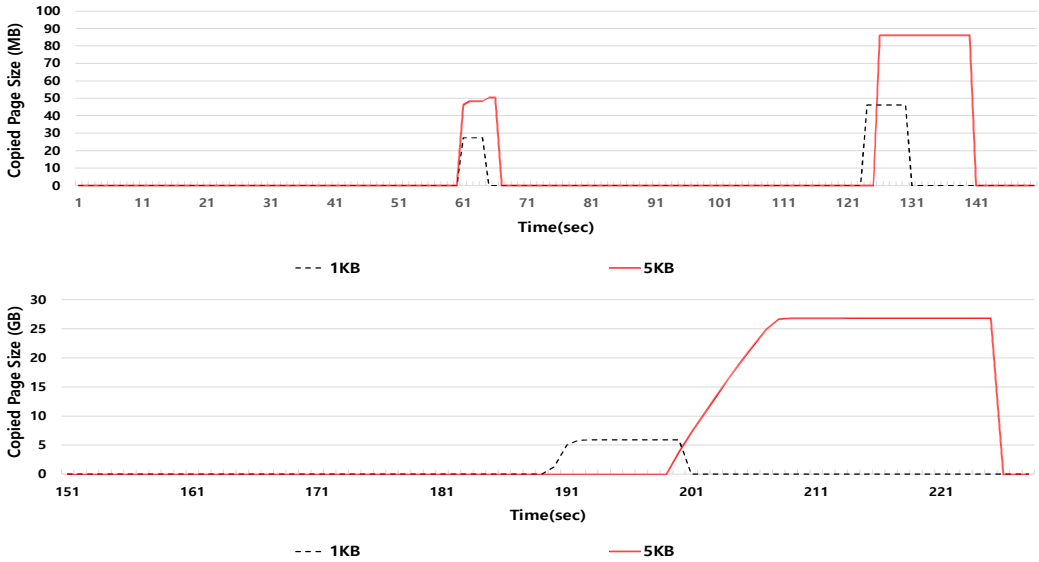
메모리 기반 저장소는 데이터 처리 성능과 더불어 저장 가능한 데이터의 개수와 전체 크기도 중요하다. 메모리 기반 저장소에서 주 저장소로 사용되는 DRAM은 HDD나 SSD와 같은 디스크에 비해 용량 대비 가격이 비싸고 저장 용량이 낮다. 메모리 기반 저장소는 데이터 저장을 위한 공간 외에도 인덱스, 로그 버퍼와 같이 시스템 운용을 위한 메모리 공간도 요구된다. 메모리 기반 저장소인 Redis는 데이터 저장 공간이 다소 한정적임에도 불구하고 빠르게 데이터를 처리하기 위해, 요청된 데이터를 원본 그대로 저장한다. 데이터를 원본 그대로 저장하는 것은 데이터 처리 성능 측면에서는 이점을 가지지만, 메모리 사용량 관점에서는 데이터 저장에 대한 부담을 가중시키는 문제가 있다. 최근에는 RedisLabs에서 개발한 ReJSON [20]과 같이 데이터 크기가 KB 이상인 데이터를 메모리 기반 저장소에 저장한다. JSON 데이터와 같이 큰 규모의 데이터인 뉴스, 블로그, XML, DNA 염기서열 데이터에 대한 저장이 Redis에 요청될 경우, 저장을 위해 많은 양의 메모리가 소모되며 메모리에 모든 데이터를 보관하기 때문에 메모리 부족 현상이 촉발된다.

만약 데이터 저장을 위한 Redis의 메모리가 부족하게 되는 경우, 사용자가 지정한 설정에 따라 동작을 달리한다. 사용자가 메모리 swap이 가능하도록 설정한 경우, Redis는 메모리의 일부 내용을 디스크 상에 존재하는 swap 영역으로 이동시켜 메모리 공간을 확보한 다음 요청된 데이터를 기록한다. 이때, 디스크 사용으로 인한 심각한 성능 저하가 발생하며, 서버 부하로 인하여 커넥션에도 영향을 미친다. 이러한 이유로, Redis에서는 메모리 swap 설정을 사용하지 않는 것을 권장한다. 하지만, 메모리 swap이 불가능하도록 설정한 경우, Redis는 사용가능한 메모리 공간이 확보될 때까지 요청된 명령어를 수행하지 못한다.



[그림 5] Copy-on-write 기법

현재 Redis의 RDB 기법은 부가적인 메모리 사용이 필연적이다. Redis의 메인 프로세스는 UNIX 시스템 콜인 fork()를 호출하여 RDB 로깅을 수행하는 자식 프로세스를 생성한다. 생성된 자식 프로세스는 copy-on-write 기법 [21,22]을 활용하여 현재 데이터 세트에 대한 스냅샷을 생성한다. Copy-on-write 기법은 그림 5에 나타나 있다. 자식 프로세스가 생성된 직후, 메인 프로세스와 자식 프로세스는 동일한 메모리 공간을 공유한다. 메인 프로세스에서 공유된 메모리 공간에 요청된 데이터를 삽입, 수정, 삭제하는 경우, 두 프로세스는 같은 메모리 공간을 공유할 수 없게 된다. 따라서 메인 프로세스는



[그림 6] RDB 작업 중 발생하는 copy-on-write 오버헤드 측정
(x축: 워크로드 수행 시간, y축: 복사된 페이지 크기)

변경하려는 데이터의 페이지를 복사하고 복사된 페이지를 수정한다. Copy-on-write 기법을 통해 Redis에 요청된 명령어를 중단 없이 처리할 수 있지만, 시스템 메모리 부족의 원인이 된다.

Copy-on-write로 인하여 복사된 페이지의 원본 페이지는 RDB 생성이 완료될 때까지 삭제되지 않고 유지된다. 저장된 데이터가 많을수록 스냅샷 생성에 더 오랜 시간이 소요되기 때문에, 원본 페이지들이 존속된다. 장기간 지속하는 원본 페이지의 메모리 점유는 메모리 부족을 촉진하여 시스템 장애 발생 위험을 증가시킨다.

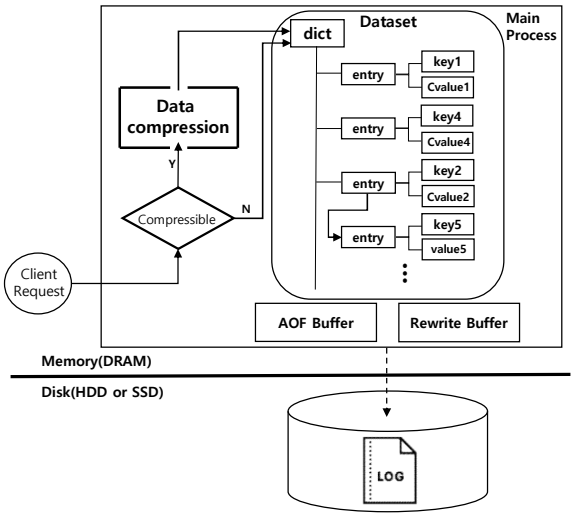
스냅샷을 생성하는 동안 copy-on-write로 인하여 발생하는 부하를 파악하기 위해, 두 개의 워크로드를 수행하는 동안 발생한 RDB 작업에서 복사되는 페이지의 크기와 원본 페이지의 존속 시간을 측정하였다. 본 실험에 사용된 두 개의 워크로드 모두 요청되는 16 Byte 크기의 키와 5,000,000건의 SET 명령으로 구성되었고, 요청되는 개별 값의 크기는 각각 1 KB와 5 KB로 구성하였다. 워크로드를 수행하는 동안 측정된 RDB 로깅 부하는 그림 6과 같다.

두 워크로드를 수행하는 동안 RDB 작업은 60초 간격으로 각각 세 차례 발생하였으며 비슷한 경향을 보였다. Redis에 적재된 데이터의 양이 증가함에 따라 RDB 작업 중에 복사되는 페이지의 크기와 원본 페이지가 유지되는 시간이 증가하였다. 또한, 요청되는 값의 크기가 증가할 경우, 복사된 페이지의 크기 또한 증가하였으며 RDB 작업 시간 역시 증가함에 따라 더 오랜 시간동안 원본 페이지가 유지된 것을 실험 결과를 통해 알 수 있다. 특히, 요청되는 개별 값의 크기가 5KB로 구성된 워크로드를 수행하며 발생한 세 번째 RDB 작업에서는 약 25.6 GB의 페이지가 복사되었으며, 해당 페이지가 26.76초 동안 유지되었다. 이는 삭제되어야할 많은 양의 원본 페이지가 장기간 동안 유지되었음을 의미한다.

해당 실험 결과는 더 많은 양의 데이터가 지속적으로 요청되는 프로덕션 환경에서의 Redis에 상당한 부하가 가중될 수 있음을 시사한다. 앞서 언급한 문제점을 해결하여 Redis의 고가용성을 제공하기 위해, 기존 데이터 저장 방법과 RDB 로깅 방법에 대한 개선이 필요하다.

4. 디자인

4.1 데이터 압축 저장



[그림 7] 데이터 압축 저장 기법

본 논문에서 제안하는 저장 방법은, 데이터 저장소로 사용되는 Redis에 더 많은 데이터를 저장하여 Redis의 효용 가치를 높이기 위해, 요청되는 키-값 데이터 중 값을 압축하여 저장한다. 이때, 압축으로 인한 데이터 처리 성능 감소를 최소화하기 위해 적은 양의 메모리를 사용하면서도 빠른 압축 속도를 보이는 LZF 압축 방식을 사용한다.

본 논문에서 제시하는 데이터 압축 저장 기법은 그림 7과 같이 설계되어 있다. 사용자로부터 데이터 삽입 또는 변경이 요청되는 경우, 요청된 파라미터에 대한 오류 검사 및 타입 분석을 수행한다. 요청된 파라미터에 이상이 없다면, 값에 대한 파라미터의 크기를 기준으로 압축 진행 여부를 결정한다. LZF 방식은 압축 속도, 압축률, 압축해제 속도를 고려하여 크기가 5 Byte보다 작은 데이터는 압축하지 않는다. 따라서 요청된 값 파라미터의 크기가 5 Byte 미만인 경우, Redis는 값 파라미터를 압축하지 않고 원본 그대로 키-값 데이터를 저장한다. 이와 반대로, 값 파라미터의 크기가 5 Byte 이상인 경우, LZF 방식을 사용하여 값 파라미터를 압축하고 키-압축된 값 형태로 데이터를 저장한다.

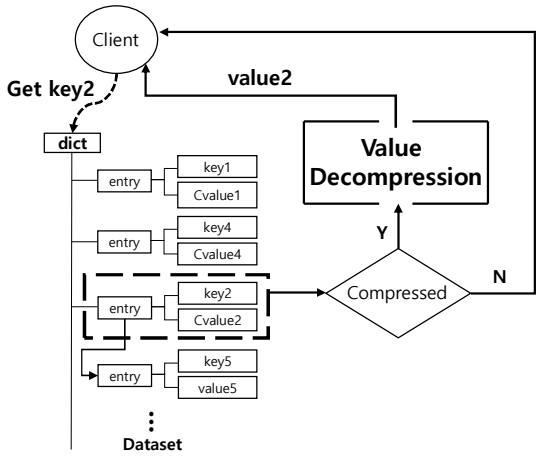
데이터 압축으로 인한 성능 저하를 최소화하여 Redis의 빠른 데이터 처리 성능을 유지하기 위해, 고속 압축 방식을 사용함과 동시에 키에 대해서는 압축을 수행하지 않고 원본 그대로 저장한다. 키를 통해서만 데이터 접근이 가능한 Redis에서는 요청된 명령을 수행하기 위해, 요청된 키와 저장된 키를 비교하는 빈도가 매우 잦다. 키를 압축하여 저장할 경우, 비교 연산을 위해 빈번하게 수행되는 저장된 키에 대한 압축해제 작업으로 인하여, 데이터 처리 성능이 심각하게 저하된다. 반면 값에 대해서만 압축을 하는 경우, 단 한 번의 압축 또는 압축해제 연산으로 요청된 명령을 수행할 수 있게 되어, 데이터 처리 성능 감소를 최소화할 수 있다. 이러한 이유로, 본 논문에서 설계한 데이터 압축 저장 방법에서는 값에 대해서만 압축하여 저장한다.

데이터 저장을 위해 소모되는 메모리 양은 요청된 키와 값의 크기에 비례한다. 데이터 압축은 요청된 값의 크기를 감소시키며, 이에 조응하여 데이터 저장에 요구되는 메모리 양이 감소하는 효과를 이끈다. 데이터 저장에 소모되는 메모리 사용량의 감소는 더 많은 양의 데이터 저장이 가능함을 의미한다.

4.2 압축된 데이터 조회

본 논문에서 제안하는 데이터 압축 저장 기법을 사용할 경우, 값 데이터는 압축된 형태로 저장된다. 따라서 사용자로부터 데이터 조회 연산이 요청되는 경우, 압축된 값 데이터에 대해서는 압축해제 연산이 요구된다. 데이터 조회 연산은 그림 8과 같이 수행된다.

데이터 조회 요청 시, 사용자는 조회 명령어와 함께 조회할 키를 서버로 전송한다. 해당 파라미터를 정상적으로 수신한 서버는 사용자로부터 요청된 키



[그림 8] 압축된 데이터 조회

와 동일한 키를 가진 엔트리가 dict에 있는지 조회한다. 이때, dict에 저장된 키는 압축되어있지 않고 원본 그대로 저장되어있기 때문에 부가적인 작업 없이 빠른 비교가 가능하다. 만약 요청된 키와 동일한 키를 가진 엔트리가 dict에 존재한다면, 엔트리의 값의 압축 여부를 검사한다. 해당 값 데이터가 압축되어 있다면, 압축해제 연산 후에 요청된 키에 대한 값 데이터를 전송한다. 반대로, 엔트리의 값 데이터가 압축되어있지 않은 경우, 압축해제 연산을 수행하지 않고 값 데이터를 전송한다.

압축된 데이터에 대한 조회 요청 시, 데이터 압축 해제 연산이 수행됨에 따라 데이터 조회 성능 감소가 불가피하다. 하지만, 키-값 데이터 중에 값 데이터만 압축되어있기 때문에 데이터 조회 명령 당 압축해제 연산을 최대 한 번만 수행하는 것을 보장할 수 있으며, LZF의 빠른 압축해제 성능을 통해 데이터 조회 성능 저하 감소를 최소화할 수 있다.

4.3 Parallel RDB 로깅

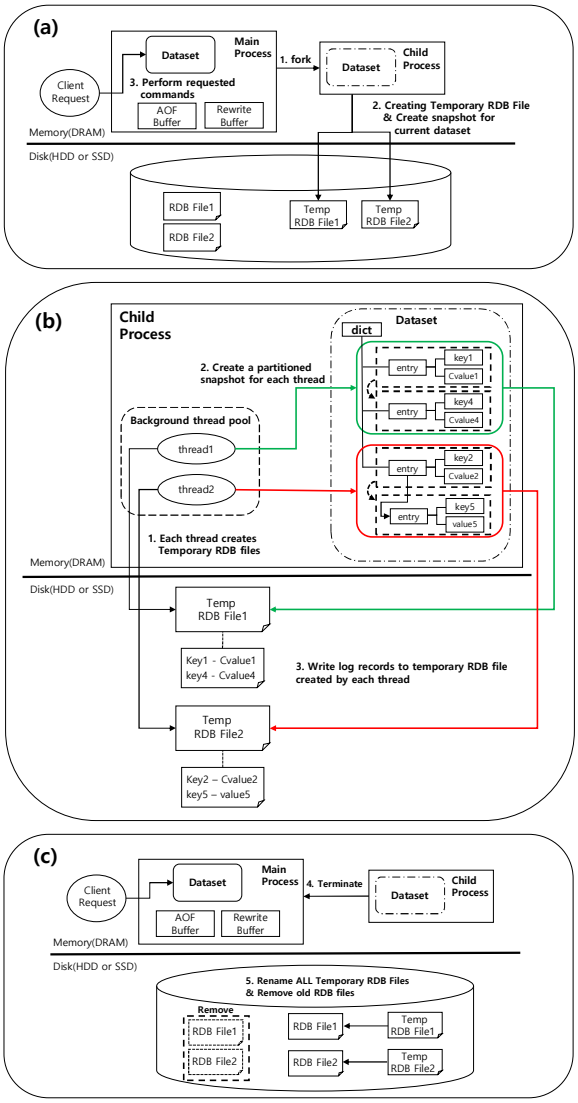
데이터 압축 저장 방법을 통해 데이터 저장소에 더 많은 데이터가 저장될 수 있는 이점을 얻었지만, 저장 가능한 데이터 크기와 개수의 증가로 인하여 RDB 로깅 시 발생하는 부하가 증가하는 문제가 발생한다. 따라서 다량의 데이터에 대한 스냅샷 생성 작업에도 낮은 로깅 부하를 발생시켜 고가용성을 제공하기 위한 새로운 영속성 방법인 PRDB(Parallel RDB)를 제시한다.

PRDB는 로깅 부하를 줄이기 위해 병렬성을 활용하여 스냅샷을 생성하는 영속성 방법이다. PRDB가 최초로 동작하기 위한 조건은 RDB의 동작 조건과 동일하다. 즉, 일정 기간 내에 특정 횟수 이상의 데이터 삽입 또는 변경 작업이 수행되면 PRDB가 동작한다. PRDB가 정상적으로 수행된 이후부터는 일정한 간격마다 동작한다.

그림 9는 PRDB의 로깅 과정을 나타내고 있다. PRDB 동작 조건이 충족되면, Redis의 메인 프로세스는 fork() 시스템 콜을 호출하여 자식 프로세스를 생성한다. 생성된 자식 프로세스는 copy-on-write 기법을 활용하여 현재 데이터 세트에 대한 스냅샷을 생성 작업을 수행하며, 메인 프로세스는 사용자로부터 요청되는 명령을 수행한다 (그림 9a).

자식 프로세스는 스냅샷 생성을 병렬로 수행하기 위해, 여러 개의 쓰레드를 생성한다. 생성된 쓰레드는 서로 다른 쓰레드 번호와 인덱스 범위를 할당받는다. 모든 쓰레드는 자신이 부여받은 쓰레드 번호를 파일 이름에 포함시켜 임시 RDB 파일을 생성하고, 주어진 인덱스 엔트리에 대한 로그 레코드를 생성한다. 생성된 로그 레코드는 해당 쓰레드가 생성한 임시 RDB 파일에만 기록된다. 임시 RDB 파일에 로그 레코드 기록을 완료한 쓰레드는 다음 엔트리를 선택하고 자신에게 할당된 인덱스 범위에 있는 모든 엔트리에 대해 위의 과정을 반복한다 (그림 9b).

할당된 인덱스 범위에 대한 스냅샷 생성을 완료한 쓰레드는 생성 결과를 자식 프로세스에 전송한다. 자식 프로세스는 모든 쓰레드가 작업을 완료할 때까지 대기한다. 모든 쓰레드가 로깅 작업을 완료하면,



[그림 9] PRDB 로깅 과정: (a) PRDB 시작 및 자식 프로세스 생성; (b) 자식 프로세스의 PRDB 동작 과정; (c) 병렬 RDB 생성이 완료된 후 분할된 RDB 파일들의 이름을 변경

자식 프로세스는 메인 프로세스로 작업이 완료되었다는 신호를 전송한다. 반대로, 작업에 실패한 쓰레드가 있는 경우 주 프로세스에 오류 신호를 전송한다. 결과를 수신한 부모 프로세스는 자식 프로세스를 종료시킨다. 자식 프로세스가 종료된 후, 메인 프로세스는 자식 프로세스에서 생성한 여러 개의 임시 RDB 파일들의 이름을 변경한다. 파일 이름변경 작업은 가장 작은 파일 번호를 가진 임시 RDB 파일부터 순차적으로 수행되며, 파일에 포함된 번호와 동일한 번호를 가진 RDB 파일의 이름으로 변경된다. 이때, 이전에 생성되었던 RDB 파일들이 삭제된다 (그림 9c).

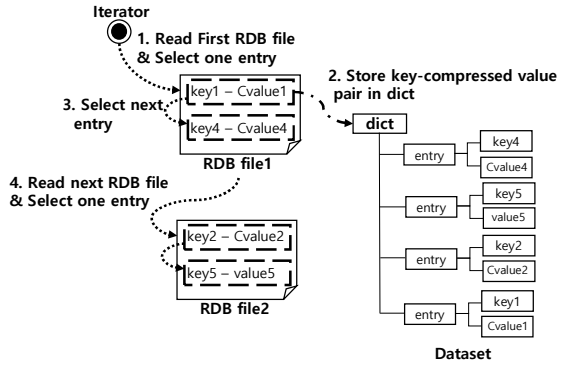
기존 RDB 방식은 하나의 프로세스가 저장된 모든 엔트리를 순차적으로 순회하며 스냅샷을 생성한다. 이와는 달리, PRDB는 스냅샷 생성에 사용할 쓰레드 개수만큼 전체 데이터 세트를 분할하고, 쓰레드마다 서로 다른 영역을 배정하여 스냅샷 생성을 동시에 수행한다. 따라서 동일한 데이터 세트에 대해서 PRDB는 RDB 보다 이른 시간 안에 스냅샷 생성을 완료할 수 있다. 또한, PRDB는 스냅샷 생성 시간 단축과 더불어 서버에 존재하는 CPU 이용률을 최대화 할 수 있는 이점이 있다. 기존 RDB 방식에서는 다수의 CPU가 있음에도 불구하고 하나의 CPU만을 사용하여 스냅샷을 생성한다. 이는 하나의 CPU에 작업이 집중되어 비효율적으로 작업이 수행됨을 의미한다. 이와는 다르게 PRDB는 여러 개의 CPU를 사용하여 스냅샷을 생성한다. 이를 통해 하나의 CPU에 작업이 집중되는 것을 방지함과 동시에 서버 자원을 최대한 활용하여 스냅샷 생성 작업을 효율적으로 수행할 수 있다.

RDB는 로그 레코드를 생성하기 전에 압축 가능한 데이터에 대해서는 압축하고 로그 레코드를 생성한다. 반면에 PRDB에서는 압축 가능한 데이터는 데이터 압축 기법을 통해 압축하여 저장되기 때문에, 곧바로 로그 레코드를 생성한다. PRDB에서는 로그 레코드를 생성하기 전에 저장된 엔트리에 대해 압축을 하지 않기 때문에 스냅샷 생성에 소요되는 시간이 단축된다.

3 장에서 언급한 바와 같이, 스냅샷 생성에 오랜 시간이 소요되는 경우 스냅샷 생성이 완료될 때까지 사라지지 않고 남아있는 다수의 원본 페이지로 인하

여 시스템 장애가 발생할 수 있다. 다량의 데이터가 적재된 상황에서도, PRDB는 빠르게 스냅샷 생성 작업을 완료하여 원본 페이지가 존속하는 것을 방지한다. 이를 통해 메모리 부족으로 인한 시스템 오류 발생 위험을 감소시켜, 서버의 안정성을 높일 수 있다.

4.3 PRDB 복구



[그림 10] prdbLoad 함수

시스템 오류가 발생하여 재시작된 Redis는 데이터를 복구하기 위한 복구 절차를 수행한다. Redis에서는 메인 프로세스만이 데이터 저장 및 변경을 할 수 있기 때문에, 데이터 복구 작업은 메인 프로세스에서만 진행된다. PRDB는 전체 데이터 세트에 대한 로그 레코드들을 각 파일에 분할하여 저장하므로, 전체 데이터 세트를 복구하기 위해서는 다수의 PRDB 파일이 복구에 사용되어야 한다. PRDB 로깅 중에는 두 가지 타입의 파일 (PRDB 파일, 임시 PRDB 파일)이 생성되며, 이름 변경 작업을 통해 파일의 개수가 변경된다. 디스크에 남아있는 파일의 타입과 개수를 통해 언제 시스템 충돌이 발생했는지 추론할 수 있다.

PRDB 복구 과정을 설명하기에 앞서, 데이터 복구 시 사용하는 PRDB 복구 함수인 prdbLoad에 대해 소개한다. prdbLoad는 인자로 전달받은 파일 타입과 파일 개수를 바탕으로 데이터를 복구한다. prdbLoad의 동작 과정은 그림 10과 같다. 먼저, prdbLoad는 인자로 전달받은 파일 타입과 동일한 타입을 가진 파일 중에서 파일 번호가 가장 작은 파일을 디스크에서 읽는다. 그다음, 파일에 기록된 로그 레코드를 읽고 dict에 저장하여 데이터를 복구한다. 이때, 로그 레코드가 압축되어 있더라도 압축을 해제하지 않고 바로 dict에 저장한다. 해당 파일에 대한 데이터 복구를 완료하면, 다음으로 큰 파일 번호를 가진 파일을 읽고 데이터를 복구한다. 인자로 전달받은 파일 개수만큼 파일을 읽어 데이터 복구가 완료될 때까지 위 과정을 반복한다.

PRDB는 빠른 로깅 성능과 더불어 빠른 데이터 복구 성능을 제공한다. 기존 Redis의 저장 방식에서는 요청된 데이터를 원본 그대로 저장하기 때문에, 압축된 데이터에 대해 RDB 복구를 할 때 압축해제 연산이 요구된다. 반면, 본 논문에서 제안하는 방법에서는 데이터를 압축하여 저장하기 때문에, PRDB 복구에서는 압축된 데이터를 압축된 형태 그대로 복구한다. PRDB에서는 압축해제를 수행하지 않기 때문에, RDB보다 더 빠르게 데이터를 복구할 수 있다.

알고리즘 1에서는 PRDB 로깅 중에 발생하는 시스템 오류 사례와 각 상황에서 데이터를 복구하는 절차를 의사 코드(Pseudo-code)로 설명한다. 시스

Algorithm 1 *Recovery of PRDB (status, num_of_PRDB, num_of_TempPRDB)*

```
   Input:
   status: The point at which the system crash occurred during the
   PRDB operation
   num_of_PRDB: Number of PRDB files to use for recovery
   num_of_TempPRDB: Number of TempPRDB files to use for
   recovery
   Output: none

   /* Start Recovery mechanism of PRDB */
   /* (Step 1 in Fig 9a to Step 4 in Fig 9c) */
1  if Status == Crash occurred before complete temporaryPRDB
   creation then
2     prdbLoad(PRDB, num_of_PRDB) /* Function to read all
                                     partitioned RDB files
                                     and recover dataset */
   /* (Step 5 in Fig 9c) */
3  else if Crash occurred during temporary PRDB rename then
4     prdbLoad(PRDB, num_of_PRDB)
5     prdbLoad(TempPRDB, num_of_TempPRDB)

   /* General case. The files on disk are PRDB */
6  else
7     prdbLoad(PRDB, num_of_PRDB)
8  End
```

템 오류 발생 이후, 재시작된 Redis는 디스크에 존재하는 파일과 그 개수를 파악하여 시스템 오류가 발생한 시기를 추론한다.

PRDB를 수행하는 도중에 오류가 발생하는 경우, 디스크에는 두 가지 타입의 파일이 존재하게 된다. PRDB 로깅 과정에서는 스냅샷 생성이 완료된 후 임시 PRDB 파일의 이름을 변경한다. 따라서 디스크에 동일한 개수의 PRDB파일과 임시 PRDB 파일이 존재하는 경우, PRDB 로깅이 완료되기 전에 시스템 오류가 발생한 것을 의미한다. 디스크에 존재하는 임시 PRDB 파일이 전체 데이터 세트에 대한 내용을 포함하고 있다는 것을 보장할 수 없기 때문에, 디스크에 존재하는 모든 PRDB 파일만을 사용하여 데이터 복구를 수행한다 (알고리즘1 1~2).

디스크에 PRDB 파일과 임시 PRDB 파일이 존재하지만 두 파일의 개수가 동일하지 않은 경우, 이는 임시 PRDB 파일의 이름을 변경하는 도중에 오류가 발생했음을 의미한다. 위와 같은 상황에서는 스냅샷 생성이 완료되었다는 것이 보장되므로, 데이터 복구 시 디스크에 있는 두 타입의 파일을 사용한다. PRDB의 이름 변경 작업은 파일 번호 순서대로 순차적으로 수행되기 때문에, 디스크에 존재하는 임시 PRDB 파일 중에서 가장 작은 파일 번호를 찾아 몇 개의 파일의 이름이 변경되었는지 파악할 수 있다. 위 상황에서는 디스크에 있는 임시 PRDB 파일 중에서 가장 작은 파일 번호를 가진 임시 PRDB 파일보다 작은 파일 번호를 가진 PRDB 파일과 모든 임시 PRDB 파일을 사용하여 전체 데이터 세트를 복구한다 (알고리즘1 3~5).

디스크에 PRDB 파일만 있는 경우, PRDB 정상적으로 수행되었거나 PRDB가 다시 수행되기 전에 시스템 오류가 발생했음을 의미한다. 두 상황에 대하여 모두 동일하게, 디스크에 존재하는 모든 PRDB를 사용하여 데이터를 복구한다 (알고리즘1 7).

5. 실험 및 결과

본 논문에서 제안하는 두 가지 방법을 적용한 CP-Redis(Compression storage and Parallel RDB-Redis)의 성능을 평가하기 위해, Redis의 기존 방법에 대한 비교 실험을 수행하였다. 모든 실험은 메모리 기반 저장소 성능 평가 벤치마크인 Memtier-Benchmark [23]를 사용하여, 데이터 적재 성능, 데이터 조회 성능, 메모리 사용량, 스냅샷 생성 시간, 데이터 복구 시간을 측정했다. 또한, 병렬로 스냅샷을 생성했을 때 발생하는 이점을 확인하기 위해, PRDB에 사용되는 쓰레드 개수를 증가시키며 로깅 시간과 복구 시간을 측정했다.

5.1 실험 환경

[표 1] 실험 환경 및 설정

Hardware Setup	
CPU	Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
RAM	DDR3 64 GB
Disk(SSD)	Crucial_CT250MX200 SSD1 250 GB * 3
Software Setup	
OS	Cent OS 7.3.1611 (Core)
OS Kernel	3.10.0-514.26.2.el7.x86_64
Redis version	4.0.10
Maxmemory option	60 GB
RDB option	save 60 10000
Memtier-Benchmark Version	1.2.13

실험을 위한 하드웨어 환경 및 소프트웨어 설정은 표 1과 같다. Redis에서 사용가능한 최대 메모리는 60 GB로 설정하였으며, RDB와 PRDB의 최초 동작 조건 및 생성 주기는 Redis에서 제공하는 기본값으로 설정했다. 60초 동안 10,000번 이상의 데이터 삽입 또는 변경이 발생하는 경우, 60초 간격으로 스냅샷을 생성하게 된다.

5.2 Memtier-Benchmark

RedisLabs에서 개발한 Memtier-Benchmark는 메모리 기반 저장소의 성능을 측정하기 위해 널리 사용되는 벤치마크이다. 해당 벤치마크의 특징 중 하나는 데이터 저장 명령어인 SET과 데이터 조회 명령어인 GET의 비율을 변경하여 다양한 워크로드를 생성할 수 있다는 것이다. 또한, 요청 수, 클라이언트 수, 키와 값의 범위 및 크기, 실험 시간과 같은 매개 변수를 제어할 수 있다. 실험 후에는 평균 쓰기 처리량 및 읽기 처리량에 대한 정보가 수집된다.

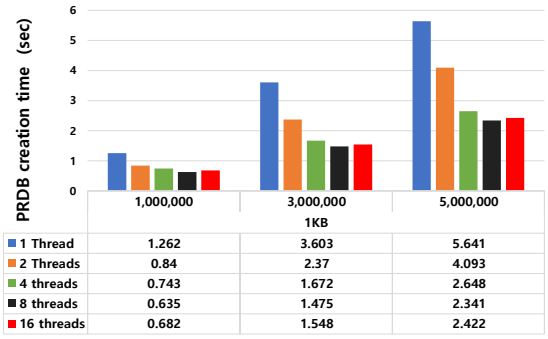
5.3 쓰레드 개수에 따른 PRDB 성능 측정

스냅샷 생성을 수행하기 위해 생성되는 적절한 수의 쓰레드는 스냅샷 생성 시간 단축 및 자원 절약의 효과를 이끈다. 반면에 필요 이상으로 쓰레드를 생성하면 교착상태에 빠져 성능이 저하되고 불필요한 자원 사용이 발생한다. 따라서 스냅샷 생성을 위해 적절한 수의 쓰레드를 생성하는 것이 중요하다.

PRDB에 사용되는 쓰레드 개수 변화에 따른 PRDB의 성능 변화를 비교하기 위해, 스냅샷 생성에 사용되는 쓰레드의 개수를 변경해가며 1,000,000, 3,000,000, 5,000,000건의 키-값 데이터 (키의 크기: 16 B, 값의 크기: 1 KB)에 대한 PRDB 생성 시간을 측정하였다. 또한, PRDB를 통해 생성된 파일을 사용한 복구 시간을 측정하여, 병렬성이 로깅과 복구에 미치는 영향을 파악했다.

5.3.1 PRDB 생성 시간

그림 11은 쓰레드 수가 증가함에 따라 PRDB 생성을 완료하기까지 소요되는 시간이 줄어든다는 것을 나타낸다. 쓰레드의 개수가 증가할수록 PRDB 작업을 완료하기까지 소요되는 시간이 줄어들었지만, 감소하는 시간 폭이 점차 줄어들었다. 모든 경우에서

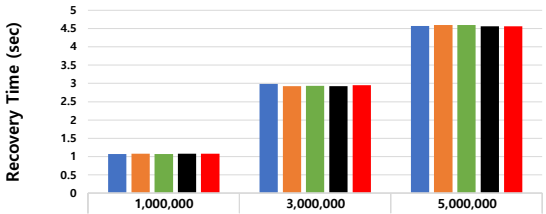


	1,000,000	3,000,000	5,000,000
1 Thread	1.262	3.603	5.641
2 Threads	0.84	2.37	4.093
4 threads	0.743	1.672	2.648
8 threads	0.635	1.475	2.341
16 threads	0.682	1.548	2.422

[그림 11] 사용되는 쓰레드 개수에 따른 PRDB 생성 시간 (x축: 키-값 엔트리 개수(상) / 값 크기(하), y축: 시간)

쓰레드의 개수가 1개에서 8개로 증가할 때, PRDB 작업 시간이 단축되었다. 그러나 16개의 쓰레드를 사용하여 PRDB를 생성했을 때, 8개의 쓰레드로 PRDB를 생성했을 때보다 더 많은 시간이 소요되었다. 이는 필요 이상으로 쓰레드가 사용되었다는 것을 의미한다. 본 실험 결과에서 8개의 쓰레드를 사용했을 때 가장 빠른 PRDB 생성 속도를 보이기 때문에, 이후 실험에서 CP-Redis의 자식 프로세스가 생성하는 쓰레드 개수를 8개로 설정하였다.

5.3.2 PRDB파일을 사용한 복구 시간



	1,000,000	3,000,000	5,000,000
1 Thread	1.073	2.987	4.569
2 Threads	1.077	2.921	4.599
4 Threads	1.072	2.934	4.596
8 Threads	1.076	2.927	4.564
16 Threads	1.078	2.947	4.564

[그림 12] 생성된 PRDB 파일을 사용한 복구 시간 (x축: 키-값 엔트리 개수(상) / 값 크기(하), y축: 시간)

5.3.1 절에서 수행된 실험을 통해 생성된 PRDB 파일을 사용하여 데이터 복구에 소요되는 시간을 측

정하였다. PRDB로 인하여 생성된 다수의 파일을 사용한 데이터 복구 시간은 그림 12와 같다. PRDB 생성에 사용되는 쓰레드 수가 증가할수록, 더 많은 분할된 PRDB 파일이 생성된다. 데이터 복구에 사용되는 파일이 증가했음에도 불구하고, 데이터를 복구하는데 소요된 시간은 각각의 경우마다 비슷하게 측정되었다. 이는 데이터 복구에 소요되는 시간은 복구에 사용되는 파일의 개수가 아닌, 복구하는 데이터 양에 의해 결정된다는 것을 의미한다. 본 실험을 통해, PRDB에 사용되는 쓰레드 개수의 증가는 데이터 복구에 큰 영향을 끼치지 않는다는 것을 알 수 있다.

5.4 워크로드에 따른 CP-Redis 성능 평가

다양한 크기의 값과 명령이 요청되는 환경에서의 CP-Redis의 성능을 평가하기 위해, 기존 데이터 저장 방법과 RDB를 사용하는 Redis에 대한 비교 실험을 진행하였다. 본 실험에서는 다양한 환경을 모사하고자 명령어 요청 수와 개별 값 데이터 크기를 변경하여 여러 개의 워크로드를 구성하였으며, 각 워크로드에 대한 자세한 구성은 표 2, 표 3과 같다. 본 실험에서는 각 워크로드를 수행한 다음 측정된 메모리 사용량, 데이터 적재 성능, 데이터 조회 성능, 스냅샷 생성 시간 및 데이터 복구 시간을 측정하였다.

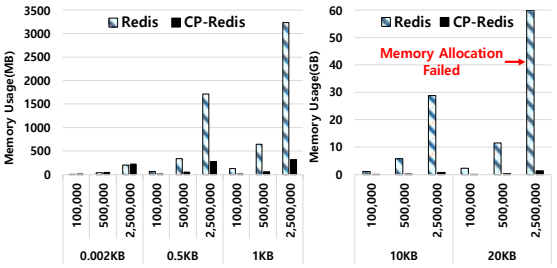
5.4.1 메모리 사용량

데이터 압축 저장 기법을 통한 메모리 사용량 절감 효과를 파악하기 위해, 표 2와 같이 구성된 각각의 워크로드를 수행한 다음 측정된 메모리 사용량을 비교하였다. 메모리 사용량은 워크로드 수행 후 저장소에 저장된 데이터 크기를 의미한다.

그림 13은 요청되는 SET 명령의 수와 값의 크기가 증가할수록, Redis와 CP-Redis의 메모리 사용량 차이가 점차 증가하는 것을 보여준다. 개별 데이터 크기가 0.002 KB로 설정된 워크로드에서는 압축

[표 2] 데이터 적재 워크로드 구성 표

Workload Name	Key Size (B)	Number of SET Requests	Value size (KB)	Total Data Size
W1	16	100,000	0.002	1.72 MB
W2			0.5	50.35 MB
W3			1	99.18 MB
W4			10	978.09 MB
W5			20	1.91 GB
W6		500,000	0.002	8.58 MB
W7			0.5	251.77 MB
W8			1	495.91 MB
W9			10	4.78 GB
W10			20	9.54 GB
W11		2,500,000	0.002	42.92 MB
W12			0.5	1.23 GB
W13			1	2.42 GB
W14			10	23.88 GB
W15			20	47.72 GB



[그림 13] 다양한 크기의 값 및 요청 수에 따른 메모리 사용량 (x축: SET 명령어 개수(상) 및 개별 값 데이터 크기(하), y축: 메모리 사용량)

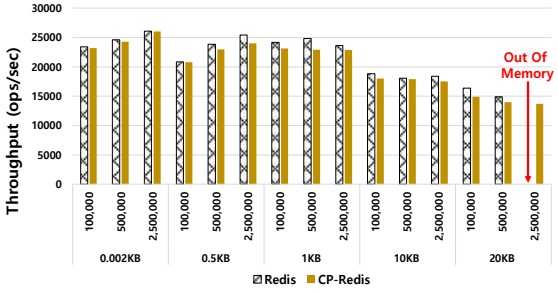
작업이 발생하지 않았기 때문에, Redis와 CP-Redis의 메모리 사용량이 비슷하게 측정되었다.

개별 데이터 크기가 20 KB인 값 데이터를 2,500,000건 적재하는 워크로드 W15를 수행한 결과, CP-Redis는 정상적으로 해당 워크로드를 수행한 것과 달리 Redis에서는 메모리 부족 현상(Out Of Memory)이 발생하여 워크로드를 정상적으로 수행하지 못하였다. Redis는 데이터를 저장할 때, 저장할 데이터 외에도 저장될 데이터의 타입, 인코딩, 시간, 참조 횟수, 엔트리와 같은 데이터 관리를 위한 추가적인 메모리가 요구된다. 이로 인하여, 사용자가 설정한 최대 메모리 용량보다 적은 양의 데이터를 적재할 수 있다. 표 2를 통해 해당 워크로드의 전체 데

이터 세트 크기는 47.72 GB로 본 실험에 사용된 Redis의 최대 메모리인 60 GB 보다 그 크기가 작은 것을 알 수 있다. CP-Redis도 기존 Redis와 마찬가지로 데이터 적재 시 데이터 관리를 위한 부가적인 메모리가 요구되지만, 요청된 값 데이터를 압축하여 저장하기 때문에 데이터 저장 및 영속성 작업을 위한 메모리 공간이 충분히 확보되어 해당 워크로드를 정상 수행한 반면, Redis는 값 데이터를 원본 그대로 저장하기 때문에 데이터 저장과 RDB 작업을 위한 메모리 공간이 부족하게 되어 해당 워크로드를 정상 수행하지 못하고 비정상 종료되었다.

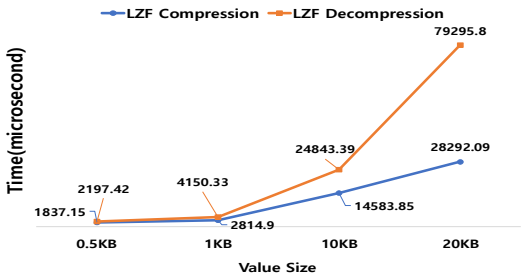
CP-Redis와 Redis 모두 워크로드를 정상적으로 수행한 경우에 대하여 메모리 사용량을 비교한 결과, 요청되는 개별 값의 크기가 0.5 KB, 1 KB, 10 KB, 20 KB 로 증가할수록, 데이터 저장에 사용되는 메모리 사용량이 Redis에 비해 평균적으로 83.5%, 90.1%, 97.2%, 98.0% 감소하였다. 이는 데이터 크기가 커질수록 CP-Redis는 Redis 보다 더 많은 양의 데이터를 저장할 수 있다는 것을 의미한다.

5.4.2 데이터 적재 성능



[그림 14] 다양한 요청 수 및 값의 크기에 따른 데이터 적재 성능 (x축: SET 명령어 개수(상) 및 개별 값 데이터 크기(하), y축: 처리량)

5.4.1 절에서 수행한 각 워크로드에 대한 데이터 적재 성능 비교 결과는 그림 14와 같다. 워크로드 W15에서는 워크로드를 수행하는 도중에 메모리 부족 현상이 발생하여 Redis가 비정상 종료되었기 때문에 데이터 처리량을 측정할 수 없었다.



[그림 15] LZF 압축 및 압축해제 소요시간

그림 14를 통해 데이터 적재 시 수행되는 압축으로 인하여 CP-Redis의 데이터 처리량이 Redis의 데이터 처리량보다 낮은 것을 확인할 수 있다. 데이터 압축 작업에 사용되는 LZF 방법에서는 5 Byte 미만의 데이터에 대해서는 압축을 하지 않기 때문에, 개별 값의 크기가 0.002 KB로 구성된 워크로드에서는 데이터가 압축되지 않고 원본 형태 그대로 저장된다. 압축에 소요되는 시간이 없음에도 불구하고 데이터의 압축 가능 여부를 검사하는 작업으로 인해 CP-Redis의 데이터 처리량이 Redis보다 0.73% 정도 감소하였다. 이러한 수치는 무시할 수 있는 수준이다.

압축이 가능한 0.5 KB, 1 KB, 10 KB, 20 KB 크기의 개별 값 데이터에 대해서는 압축 작업이 빈번히 발생했기 때문에, 압축을 수행하지 않았을 때보다 데이터 처리량이 감소되었다. 워크로드를 수행하는 동안 집계된 LZF의 데이터크기 별 평균 압축 시간은 그림 15와 같다. 요청되는 개별 값 데이터의 크기가 증가할수록 데이터 압축에 소요되는 시간이 증가함에 따라 데이터 처리량도 점차 감소하였다. 워크로드를 수행하는 동안 측정된 CP-Redis의 데이터 처리량은 Redis 보다 평균적으로 약 5.0% 정도 감소하였다. 비록 모든 워크로드에서 CP-Redis의 데이터 처리량이 감소하였지만, 데이터 저장에 사용되는 메모리 사용량이 크게 절감되므로 압축을 적용하는 것이 더 효과적인 것으로 보인다.

5.4.3 데이터 조회 성능

CP-Redis에서는 사용자로부터 조회가 요청된 키에 대한 값 데이터가 압축 되어있는 경우, 값 데이터를 원본 형태로 반환하기 위해 압축해제 연산을 수행한다. 압축해제 연산이 데이터 조회 성능에 미치는 영향을 파악하기 위해, 표 3과 같이 구성된 각 워크로드에 대한 CP-Redis와 Redis의 데이터 조회 성능을 측정하였다.

실험에 사용된 15개의 데이터 조회 워크로드는 다음과 같이 동작한다. 워크로드가 시작된 경우, 16 B로 고정된 크기의 키와 각 워크로드마다 다르게 설정된 크기의 값 데이터 쌍 1,500,000개를 적재한다. 이때, 적재되는 모든 키는 중복되지 않고 서로 다른 문자열을 가진다. 데이터 적재가 완료된 이후, 각 워크로드에 설정된 GET 명령어 개수만큼 저장된 데이터에 대한 데이터 조회 연산을 수행한다.

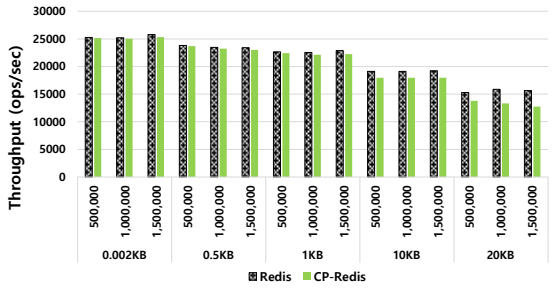
데이터 조회 성능 실험 결과는 그림 16과 같다. CP-Redis와 Redis 모두 조회하는 데이터의 크기가 증가할수록 데이터 처리량이 감소하였다. 이는 그림 15에 나타나 있듯이 압축된 원본 데이터의 크기가 증가할수록 압축해제에 소요되는 시간이 증가했기 때문이다. 조회하는 데이터의 크기가 0.5 KB, 1 KB, 10 KB, 20 KB로 증가할수록 CP-Redis의 데이터 조회 성능은 Redis에 비해 평균적으로 1.0%, 1.8%, 6.2%, 14.8% 정도 감소하였다.

워크로드 R13, R14, R15와 같이 개별 값 데이터 크기가 20 KB로 구성된 워크로드의 경우, 데이터 조회 요청 횟수가 증가할수록 CP-Redis의 데이터 처리량이 감소되었다. 원본 데이터의 크기가 20 KB인 압축된 데이터를 압축 해제하는 것이 상당히 오랜 시간을 요구하기 때문에(그림 15), 압축해제 연산을 동반하는 데이터 조회 요청 횟수의 증가가 데이터 처리량 감소의 원인이 된 것으로 보인다.

반면에 개별 값 데이터 크기가 20 KB로 구성된

[표 3] 데이터 조회 워크로드 구성 표

Workload Name	Key Size (B)	Number of SET Requests	Value size (KB)	Number of GET Requests
R1	16	1,500,000	0.002	500,000
R2				1,000,000
R3				1,500,000
R4			0.5	500,000
R5				1,000,000
R6				1,500,000
R7			1	500,000
R8				1,000,000
R9				1,500,000
R10			10	500,000
R11				1,000,000
R12				1,500,000
R13			20	500,000
R14				1,000,000
R15				1,500,000



[그림 16] 다양한 요청 수 및 값의 크기에 따른 데이터 조회 성능 (x축: GET 명령어 개수(상) 및 적재된 개별 값 데이터 크기(하), y축: 처리량)

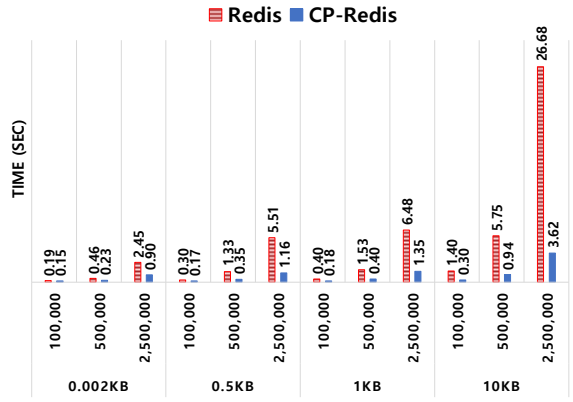
워크로드를 제외한 다른 워크로드(R1~R12)에서는, 워크로드에 구성된 개별 값 데이터 크기가 동일한 각 사례마다 데이터 조회 요청 횟수가 증가하더라도 Redis와 CP-Redis 모두 데이터 처리량이 비슷한 수준으로 측정되었다. 이러한 결과를 통해 상대적으로 작은 크기의 데이터를 저장 및 조회하는 메모리 기반 저장소에서 유의한 성능을 보여줌을 확인할 수 있었다.

5.4.4 스냅샷 생성 시간

본 실험에서는 표 2에 구성된 각 워크로드를 수행하여 저장된 각각의 데이터 세트에 대한 Redis의

RDB 작업 시간과 CP-Redis의 PRDB 작업 시간을 측정하였다. 워크로드를 수행하는 동안 스냅샷 생성 작업이 발생한 경우, 해당 작업의 수행 시간과 복사된 페이지의 크기를 측정하여 Redis와 CP-Redis의 메모리 오버헤드(Memory Overhead)를 비교하였다. Redis는 요청되는 개별 값의 크기가 20KB로, 명령어 요청 수는 2,500,000건으로 구성된 워크로드를 정상 수행하지 못하였으므로 CP-Redis와의 정확한 성능 비교가 불가능하다. 따라서 RDB 성능 관련 실험에서는 개별 값 데이터 크기가 20KB로 구성된 워크로드에 대한 결과는 포함하지 않았다.

그림 17을 통해 각각의 경우에서 CP-Redis의 스냅샷 생성 시간이 Redis보다 단축된 것을 알 수 있다. 저장된 키-값 데이터 개수와 값의 크기가 증가함에 따라 CP-Redis와 Redis 모두 스냅샷 생성에 소요되는 시간이 증가하였다. 또한 값의 크기와 데이터 개수가 증가할수록 CP-Redis의 스냅샷 생성 시간과 Redis의 스냅샷 생성 시간 차이도 점차 증가하였다. 특히, 값의 크기가 10 KB인 2,500,000건의 키-값 데이터에 대한 스냅샷 생성 작업을 수행했을 때, CP-Redis의 스냅샷 생성 시간이 Redis의 스냅샷 생성 시간보다 최대 86.43% 단축되었다.



[그림 17] 다양한 크기의 값 및 키-값 데이터 개수에 대한 스냅샷 생성 시간 (x축: 키-값 엔트리 개수(상) 및 개별 값 데이터 크기(하), y축: 시간)

[표 4] 스냅샷 생성으로 인한 메모리 오버헤드

Number of Requests	Value size (KB)	Redis		CP-Redis	
		Copied page (MB)	Time (sec)	Copied page (MB)	Time (sec)
2,500,000	0.002	28	1.10	22	0.44
	0.5	30	2.80	36	0.66
	1	36	3.48	36	0.79
	10	94	12.09	28	1.59
		26	26.72	38	2.97

저장된 개별 값의 크기가 0.002 KB일 때, Redis의 RDB 동작 시 데이터 압축 작업이 수행되지 않았음에도 불구하고 CP-Redis보다 낮은 스냅샷 생성 성능을 보였다. CP-Redis에서는 저장된 데이터 세트를 분할하여 스냅샷 생성을 병렬로 수행하기 때문에 Redis보다 더 빠른 스냅샷 생성 성능을 보였다. 따라서 데이터 병렬성으로 인하여 저장된 데이터의 개수가 많아질수록 Redis와 CP-Redis의 스냅샷 생성 시간 차이가 증가하는 것을 알 수 있다.

저장된 개별 값의 크기가 각각 0.5 KB, 1 KB, 10 KB인 경우에는 Redis에서 스냅샷을 생성할 때 데이터 압축 작업이 수행되었다. 이로 인하여 스냅샷 생성 작업을 완료하기까지 더 많은 시간이 소요되었다. 적재된 데이터 개수와 값의 크기가 증가함에 따라 더 많은 압축 작업을 수행하게 되어, 스냅샷 생성에 소요되는 시간이 크게 증가하는 것을 그림 17을 통해 확인할 수 있다. Redis와 달리, CP-Redis의 스냅샷 생성 작업에서는 데이터 압축이 수행되지 않는다. 따라서 값의 크기와 저장된 키-값 데이터 개수가 크게 증가하더라도, CP-Redis의 스냅샷 생성 시간은 Redis에 비해 상대적으로 작은 비율로 증가한다.

2,500,000건의 데이터를 적재하는 워크로드는 다른 워크로드들에 비해 작업이 완료되기까지 긴 시간이 소요된다. 이로 인하여 해당 워크로드를 수행하는 도중에 스냅샷 생성 작업이 발생하였다. Redis와 CP-Redis 모두 자식 프로세스에서 스냅샷을 생성하는 동안에 지속적인 데이터 삽입으로 인하여 여러

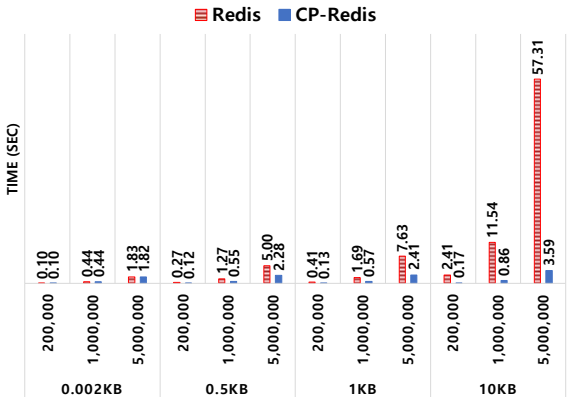
페이지가 복사되었다. 이때 복사된 페이지의 크기와 자식 프로세스의 작업 시간은 표 4와 같다. 복사된 페이지의 크기는 삭제되어야 할 원본 페이지의 크기와 동일하다.

값의 크기가 1 KB, SET 명령의 수가 2,500,000건으로 구성된 워크로드를 수행하는 동안 Redis와 CP-Redis에서 동일한 양의 페이지(36 MB)가 복사되었다. 삭제되어야 할 원본 페이지가 Redis에서는 3.484초 동안 유지되었지만, CP-Redis에서는 0.786초로, Redis에 비해 약 4.4배 정도 단축되었다. 모든 경우에서도 동일하게, 워크로드 중에 수행된 스냅샷 생성 작업에 대해 CP-Redis가 Redis보다 빠르게 완료하였다.

값의 크기가 1 KB, SET 명령의 수가 2,500,000건으로 구성된 워크로드에서는 Redis와 CP-Redis 모두 두 번의 스냅샷 생성 작업이 수행되었다. Redis의 첫 번째 스냅샷 생성 작업의 경우, CP-Redis에 비해 작업 시간이 오래 소요되었으며 지속적으로 데이터 삽입이 수행됨에 따라 비교적 많은 양의 페이지가 복사되었다. 두 번째 스냅샷 생성 작업은 워크로드가 끝날 때쯤 수행되어, 복사되는 페이지의 크기는 작았지만 스냅샷 생성 작업이 오랜 시간 수행됨에 따라 삭제되어야 할 원본 페이지가 작업 시간에 비례하여 존속하였다. 반면 CP-Redis는 빠르게 스냅샷 생성 작업을 완료하여 복사되는 페이지의 크기가 비교적 작았으며, CP-Redis를 통해 단축된 스냅샷 생성 시간만큼, 원본 페이지가 빠르게 삭제되어 메모리 오버헤드가 감소하는 것을 알 수 있다.

5.4.5 데이터 복구 시간

5.4.4 절에서 수행한 스냅샷 생성 작업을 통해 생성된 파일을 사용하여 CP-Redis와 Redis의 데이터 복구 시간을 비교하였다. CP-Redis는 Redis와 비교하여, 스냅샷 생성 성능이 향상됨과 더불어 데이터 복구 성능도 향상된 것을 그림 18을 통해 알 수 있다.



[그림 18] 다양한 크기의 값 및 요청 수에 대한 복구 시간 (x축: 키-값 엔트리 개수(상) 및 개별 값 데이터 크기(하), y축: 시간)

복구되는 개별 값의 크기가 0.5 KB, 1 KB, 10 KB로 증가할수록, 데이터 복구가 완료되기까지 소요되는 시간이 평균적으로 55.55%, 67.66%, 93.08%로 점차 단축되었다. 복구하는 개별 값의 크기가 0.002 KB인 경우, CP-Redis와 Redis 모두 데이터 압축해제 연산이 발생하지 않았기 때문에 데이터 복구 시간이 비슷하게 소요되었다. 반면에 압축해제가 필요한 데이터를 복구할 때는 Redis에서 빈번한 데이터 압축해제 연산이 수행했기 때문에 데이터 복구 시간에 차이가 발생하였다.

이러한 결과는 CP-Redis에서 시스템 오류가 발생하더라도 데이터 복구 작업을 빠르게 완료하여 Redis 보다 신속하게 정상 동작 될 수 있음을 암시한다.

6. 결론 및 향후연구

본 연구에서는 DRAM을 주 저장 장치로 사용하는 메모리 기반 저장소에 더 많은 데이터를 효율적으로 저장하기 위한 데이터 압축 저장 기법을 제안하였다. 또한, 적재 가능한 데이터 양이 증가함에 따라

증가하는 로깅 부하를 완화하기 위해, 데이터 병렬성을 활용한 스냅샷 생성 기법인 PRDB를 제안하였다. 본 논문에서 제안하는 방법의 효과를 검증하기 위해 엔터프라이즈 수준의 메모리 기반 저장소인 Redis에 두 기법을 구현하였으며, 각기 다른 데이터 크기와 명령어 개수로 구성된 다양한 워크로드를 사용하여 Redis와 비교하였다. 실험을 통해 저장되는 개별 데이터의 크기와 개수가 증가할수록 메모리 사용량, 스냅샷 생성 성능, 데이터 복구 성능에 큰 향상이 있는 것을 확인하였다. 특히, 데이터 크기가 10 KB로 설정된 워크로드 실험 결과에 따르면, Redis에 비해 평균적으로 메모리 사용량은 97.2%, 스냅샷 생성에 소요되는 시간은 82.88%, 데이터 복구에 소요되는 시간은 93.08%까지 단축되었다.

스냅샷 방식은 우수한 백업 및 복구 성능의 장점을 가졌지만, 일정한 주기마다 동작함에 따라 데이터가 유실될 위험이 있다. 즉, 스냅샷 파일이 생성된 후 재생성 될 때까지 삽입되거나 변경된 데이터에 대한 지속성을 보장하지 못한다. 데이터 유실을 최소화하고자 스냅샷 생성 주기를 단축할 경우, 데이터 지속성과 데이터 처리 성능 및 시스템 자원 간에 트레이드오프가 발생한다. 향후에는 비휘발성의 특징을 가진 NVRAM(Non-Volatile Random-Access Memory)을 활용하여 데이터 처리 성능 및 시스템 자원과 데이터 지속성의 상충관계를 극복하는 새로운 데이터 지속성 방법에 대한 연구를 진행하고자 한다.

참고 문헌

[1] Zhang H, Tubor BM, Chen G, Ooi BC, “Efficient In-memory Data Management: An Analysis”, Proceedings of the VLDB Endowment, Vol.7, no.10, pp 833-836, 2014

[2] Zhang H, Chen G, Ooi BC, Tan KL, Zhang M, “In-Memory Big Data Management and Processing: A Survey”, IEEE Transactions on Knowledge and Data Engineering, Vol.27, no.7, pp.1920-1948, 2015

[3] Pezzini M, “Innovation Insight: Invest in In-Memory Computing for Breakthrough Competitive Advantage”, Gartner, 2011

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

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

[6] GridGain. https://www.gridgain.com

[7] 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

[8] Lim H, Han D, Anderson DG, Kaminsky M, “{MICA}: A Holistic Approach to Fast In-Memory Key-Value Storage”, 11th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI 2014), pp. 429-444, 2014

[9] Ousterhout J, Gopalan A, Gupta A, Kejriwal A, Lee C, Montazeri B, Ongaro D, Park SJ, Qin H, Rosenblum M, Rumble S, Stutsman R, Yang S, “The RAMCloud storage System”, ACM Transactions on Computer Systems (TOCS), Vol.33, no.3, pp.7:1-7:55, 2015

[10] 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

[11] Redis Persistence Method. https://redis.io/topics/persistence

[12] Xu M, Xu X, Xu J, Ren Y, Zhang H, Zheng N, "A forensic analysis method for Redis database based on RDB and AOF File", Journal of Computers, Vol.9, no.11, pp.2538-2544, 2014

[13] 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

[14] 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

[15] 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

[16] Sung H, Jin M, Shin M, Roh H, Choi W, Park S, "LESS: Logging Exploiting SnapShot", IEEE International Conference on Big Data and Smart Computing (BigComp), pp. 1-4, 2019

[17] Zhang M, Yao X, Wang CL, "NVCL: Exploiting NVRAM in Cache-Line Granularity Differential Logging", IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), pp. 37-42, 2018

[18] Kim D, Choi WG, Sung H, Park S, "A Scalable and Persistent Key-Value Store Using Non-Volatile Memory", In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 464-467, 2019

[19] LZF Algorithm. http://oldhome.schmorp.de/marc/liblzf.html

[20] ReJSON. https://redislabs.com/blog/redis-as-a-json-store/

[21] Smith JM, Maguire Jr GQ, "Effects of copy-on-write memory management on the response time of UNIX fork operations", Computing Systems, Vol.1, no.3, pp.255-278, 1988

[22] Peterson ZNJ, "Data placement for copy-on-write using virtual contiguity", Diss. University of California, Santa Cruz, 2002

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

성 한 승

2017년 항공대학교 소프트웨어학과 학사

2017년~현재 연세대학교 컴퓨터과학과 석박사통합과정

관심분야 : 데이터베이스, 키-값 저장소, 리커버리

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, 바이오인포매틱스, 빅데이터 마이닝 & 기계학습