블록체인 워크로드에 최적화된 ZNS SSD 기반 LevelDB 설계 및 구현

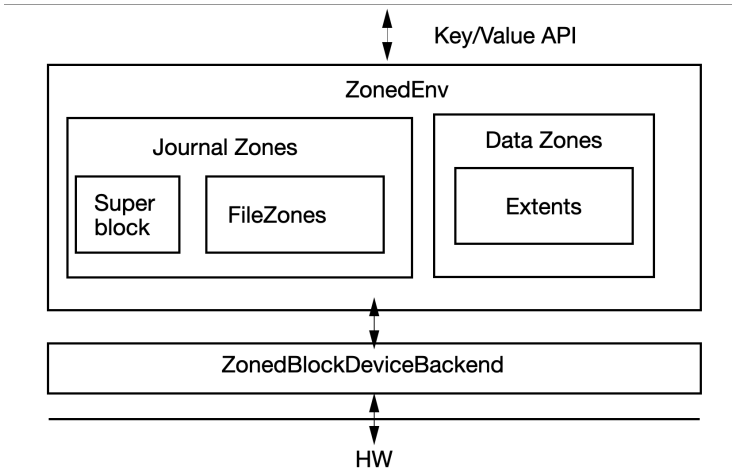
1. introduction

LSM-Tree 기반의 데이터베이스는 데이터를 기록하거나 수정할 때 덮어쓰기를 하지 않고, 항상 새로운 위치에 데이터를 추가하는 방식을 취합니다. 이후 rolling merge 과정을 통해 기존 파일들을 병합하고 불필요해진 과거 파일들을 일괄 삭제합니다. 특히, 데이터 복구를 위해 임시로 기록되는 WAL은 특정 체크포인트를 지나면 즉시 삭제되는 단기 수명을 가지는 반면, 상태 데이터가 담긴 SSTable은 상대적으로 장기 보존되는 특성이 있습니다.
본 프로젝트는 이러한 LSM-Tree의 논리적 동작 방식이 물리적으로 Sequential Write만을 허용하고 Zone 단위로 공간을 관리하는 ZNS SSD의 하드웨어적 특성과 완벽하게 부합한다는 점에 착안하였습니다. ZNS 환경에서는 데이터의 Lifetime에 따라 WAL과 SSTable을 서로 다른 존에 분리하여 저장할 수 있습니다. 이를 통해 Garbage Collection 발생 시 디스크 내부에서 유효 데이터를 옮기는 작업을 최소화하여 WAF (Write Amplification Factor)을 획기적으로 감소시킬 수 있습니다.
따라서 본 리포트에서는 대표적인 LSM-Tree 데이터베이스인 LevelDB에 ZNS를 연동하는 스토리지 엔진을 직접 구현합니다. 이 보고서에서는 구현 과정에서 직면한 기술적 난제들과 그 해결 과정을 상세히 기술하며, 나아가 방대한 트랜잭션과 상태 업데이트가 발생하는 블록체인 노드 환경의 워크로드 특성에 맞추어 시스템을 어떻게 최적화했는지 논의합니다. 마지막으로 기존 리눅스 표준 파일 시스템과 ZNS 환경에서의 LevelDB 성능(처리량 및 지연시간 등)을 비교 분석하여 본 아키텍처의 효용성을 입증합니다.

2. Architecture

본 프로젝트는 rocksdb에 zns를 연결한 코드를 참고했습니다.

1) Architecture



1-1) LevelDB Env 추상화
기존의 levelDB는 OS와의 의존성을 분리하기 위해 Env라는 interface를 사용합니다. 기존의 PosixEnv는 범용 파일 시스템 위에서 동작하도록 설계되어 있어, ZNS를 사용하기 어렵습니다. 따라서 ZNS를 직접 제어하고 관리하기 위해 EnvWrapper를 상속받은 ZonedEnv를 구현하였습니다.

1-2) Superblock과 FileZones
superblock은 초기화와 recover할 때 시작점을 나타내는 data입니다. FileZones는 해당 파일이 어느 zone의 어느 지점에 있는지 mapping해주는 data입니다. 기존 파일 시스템의 inode라고 볼 수 있습니다.

1-3) Extents
extent는 파일 시스템이 데이터를 zone에 배치할 때 사용하는 연속된 선형 공간의 block입니다. extent의 크기는 가변적이라서, 파일의 크기가 zone과 맞지 않더라도 internal fragmentation를 최소화하여 저장 공간을 효율적으로 사용할 수 있도록 합니다.

1-4) ZonedBlockDeviceBackend
ZonedBlockDeviceBackend는 하드웨어 인터페이스와 직접 통신하며 Zone 내 데이터 기록을 전담하는 추상화 계층입니다. 하드웨어 종속적인 제어 로직을 별도의 객체로 캡슐화하여 시스템의 논리적 구조와 분리함으로써, 다양한 하드웨어 환경에 대한 유연한 대응과 코드의 유지보수성을 극대화하였습니다.

2) Data I/O 흐름
데이터 쓰기와 읽기 흐름은 다음과 같다.

2-1) 쓰기 흐름
1) 쓰기 요청 수신: LevelDB가 ZonedWritableFile::Append를 호출하여 데이터를 전달한다. 이 때 데이터는 사용자의 설정에 따라 buffering 여부가 결정된다.
2) Buffering: buffered 옵션이 활성화된 경우, 데이터는 바로 디스크로 향하지 않고, posix_memalign으로 할당된 내부 버퍼에 먼저 저장되어 I/O 효율을 높인다.

3) zone 할당: 현재 파일에 할당된 zone이 없거나 기존 zone이 가득 찬 경우, ZoneFile::AllocateZone이 호출된다. 이 함수는 ZonedBlockDevice에 새로운 물리적 zone 할당을 요청하며, 이 때 파일의 수명 주기가 함께 전달된다.
4) 순차 기록: 할당된 zone의 Write Pointer 위치에 데이터가 순차적으로 기록된다.
5) extent 생성 및 매핑: 기록이 완료되면, ZoneFile::PushExtent를 통해 해당 데이터에 대한 extent를 생성하고, ZoneFile 내부의 매칭 리스트에 추가하여 논리적 주소와 물리적 주소를 연결한다.

```
Status ZoneFile::Append(void* data, int data_size) {
  uint32_t left = data_size;
  uint32_t wr_size, offset = 0;
  Statu
  // zo
  if (!active_zone_) {
    s = AllocateNewZone();
    if(!s.ok()) return s;
  }

  while (left) {
    // 다 썼으니깐
    // 이때까지 쓴 데이터 저장하고
    // 새로운 존을 만든다.
    if (active_zone_->capacity_ == 0) {
      PushExtent();

      s = CloseActiveZone();
      if (!s.ok()) {
        return s;
      }

      s = AllocateNewZone();
      if(!s.ok()) return s;
    }

    wr_size = left;
    if (wr_size > active_zone_->capacity_) wr_size = active_zone_->capacity_;
    // 실제 zone에 추가한다.
    s = active_zone_->Append((char*)data+offset, wr_size);
    if (!s.ok()) return s;

    file_size_ += wr_size;
    left -= wr_size;
    offset += wr_size;
  }
  return Status::OK();
}
```

leveldb::Zone *leveldb::ZoneFile::active_zone_

2-2) 읽기 흐름
1) 읽기 요청 수신: leveldb가 ZonedRandomAccessFile 또는 ZonedSequentialFile을 통해 특정 위치의 데이터를 요청한다.
2) 주소 번역: ZoneFile::PositionedRead 내부에서 GetExtent 함수를 호출한다. 이 함수는 파일이 소유한 extent list를 순회하며, 요청된 논리 주소가 실제 어느 물리적 zone에 위치하는지 계산한다.
3) 읽기: 번역한 주소를 바탕으로, 데이터를 읽어온다.

3. TroubleShooting

본 프로젝트 수행 과정에서 발생한 주요 문제점들과 이를 해결하기 위한 과정을 기술하겠습니다.

1) metadata 및 파일 삭제 로직 수정
문제 상황: rolling merge 완료 후 불필요해진 과거 SSTable 또는 checkpoint를 통과한 WAL 파일을 삭제하는 과정에서, 시스템이 삭제 대상 파일을 정상적으로 인식하지 못하거나 읽어오지 못하는 결함이 발생하였다.
원인 분석: 기존의 GetChildrenNoLock 함수는 OS의 기본 파일 시스템을 먼저 탐색하고, 이후 ZNS의 metadata(GetTaejukChildrenNoLock)를 탐색하여 두 목록을 합치도록 설계되어 있었습니다. 그러나 OS 파일 시스템 탐색 시 대상 디렉터리가 비어있거나 찾을 수 없어 Status::NotFound 에러가 발생하면, ZNS 파일 목록을 성공적으로 불러왔음에도 불구하고 함수 마지막에 이전의 에러 상태를 그대로 반환해 버리는 치명적인 논리적 오류가 있었습니다. 이로 인해 LevelDB 엔진은 디렉터리 읽기가 실패했다고 판단하여, ZNS 상에 존재하는 삭제 대상 파일들을 전혀 인식하지 못했습니다.

해결 방안: GetChildrenNoLock 로직을 수정하였다. 만약 파일 시스템 탐색에서 NotFound가 발생하면 바로 return하는 것이 아닌, ZNS의 metadata를 추가로 탐색해 파일 목록을 가져올 수 있도록 했습니다. 또한 return할 때, NotFound가 아닌 OK를 return해 기존 파일 시스템 에러를 무시할 수 있도록 했습니다.

2) Zone 고갈 에러 및 할당 전략 최적화
문제 상황: 전체 저장 용량이 충분함에도 불구하고, 데이터 기록이 zone allocation error가 빈번하게 발생하여 시스템이 중단되는 문제가 확인되었습니다.

원인 분석: 기존 로직은 새로운 ZoneFile이 생성될 때마다 매번 새로운 zone을 할당받았는데, 크기가 작은 file이 많이 생성되면서 하드웨어의 zone 한계를 빠르게 초과한 것이 원이었습니다.

해결 방안: 현재 쓸 수 있는 zone을 재사용해 매칭하는 전략을 사용하였습니다. AllocateIOZone 호출 시 GetBestOpenZoneMatch를 통해 이미 열려 있는 zone중 동일한 수명 주기를 가진 유효한 공간이 있는지 먼저 탐색하도록 수정하였습니다. 이를 통해 하드웨어 자원 점유를 최소화하면서도 zone 활용도를 극대화하였습니다.

4. Improvements

Blockchain 환경에서 쓸 수 있도록 다음과 같이 수정하였습니다.

1) Garbage collection 호출
기존 시스템의 가비지 컬렉션(GC)은 백그라운드 스레드에서 단순히 10초 주기로 폴링하여 실행되도록 구현되어 있었습니다. 그러나 블록체인 노드의 워크로드 특성상, 새로운 블록을 수신하고 상태를 동기화할 때 방대한 양의 데이터가 순간적으로 폭주하며 한꺼번에 기록됩니다. 이러한 환경에서는 10초라는 고정된 시간 내에 디스크의 가용 공간이 급격히 고갈될 수 있으며, GC 스레드가 미처 깨어나기도 전에 쓰기 작업이 실패하고 '디스크 공간 부족' 에러를 반환하며 데이터베이스가 중단되는 치명적인 한계가 있었습니다.
이러한 블록체인 환경의 특수성을 해결하기 위해, 본 프로젝트에서는 기존의 타이머 기반 GC를 'Event-driven 기반 하이브리드 GC'로 전면 개편하였습니다. 구체적으로 C++의

condition variable과 mutex를 도입하여 메인 I/O 스레드와 백그라운드 GC 스레드 간의 즉각적인 통신 채널을 구축했습니다.
데이터베이스 엔진이 새로운 데이터를 쓰기 위해 파일이나 Zone을 여는 시점에 실시간으로 디스크의 잔여 용량을 평가합니다. 만약 가용 용량이 임계치 이하로 떨어지는 위험 상황이 감지되면, 메인 스레드는 즉각 notify_one()시그널을 발생시킵니다. 이 시그널을 받은 GC 스레드는 남은 대기 시간과 무관하게 즉시 wake하여 존의 자원 회수(Reclaim) 작업을 시작합니다.
결과적으로 평상시에는 10초 주기의 유휴 상태를 유지하여 CPU 리소스 낭비를 막고, 블록 생성 시점에 대규모 트랜잭션이 몰릴 때는 GC가 실시간으로 개입하는 유연한 아키텍처를 완성하였습니다. 이를 통해 I/O 병목 및 공간 부족 에러를 원천 차단하고 블록체인 노드의 무결성과 연속성을 확보할 수 있었습니다.

```
void ZonedEnv::GCWorker() {
  while (run_gc_worker_) {

    {
      std::unique_lock<std::mutex> lock(gc_mtx_);
      gc_cv_.wait_for(lock, std::chrono::seconds(10));
    }
```

```
if(FreePercent() <= GC_START_LEVEL) {
  gc_cv_.notify_one();
}
```

5. Evaluation

본 프로젝트에서 구현한 ZNS 기반 스토리지 엔진의 성능을 검증하기 위해, 기존 리눅스 표준 파일 시스템(ext4) 기반 leveldb와의 비교 벤치마크를 수행하였습니다. 실험 결과는 다음과 같습니다.

워크로드 (500MB)	지표	기존 LevelDB (ext4)	제안 시스템 (ZNS LevelDB)	향상률
무작위 쓰기 (fillrandom)	**처리량 (MB/s)**	135.2 MB/s	**166.8 MB/s**	**+ 23.3% 향상**
	지연 시간 (*μ*s/op)	7.336 *μ*s	**5.945 *μ*s**	**- 18.9% 감소**
덮어쓰기 (overwrite)	**처리량 (MB/s)**	114.3 MB/s	**166.6 MB/s**	**+ 45.7% 향상**
	지연 시간 (*μ*s/op)	8.681 *μ*s	**5.953 *μ*s**	**- 31.4% 감소**
종합 지표	**Device WAF**	측정 불가	**1.001**	**하드웨어 수준 쓰기 증폭 완벽 억제**

1. 순수 I/O 처리량 극대화
무작위 쓰기 환경에서 제안된 ZNS LevelDB는 기존 대비 약 23.3% 향상된 166.8 MB/s의 처리량을 기록하였다. 이는 ZNS 엔진이 OS 계층을 우회하고 Direct I/O를 통해 Zone에 데이터를 순차적으로 기록함으로써 성능을 향상시킨 것을 알수 있다.

2. 덮어쓰기에서 압도적인 성능 유지
블록체인 노드의 State Tree 업데이트를 모사한 덮어쓰기에서 ZNS 아키텍처의 진가가 가장 뚜렷하게 나타났다. 기존 ext4 환경은 데이터 무효화 및 GC 부하로 인해 처리량이 15%하락한 반면, ZNS LevelDB는 동일한 성능을 유지하였다. 이는 새롭게 도입한 event-driven GC와 수명 주기 기반 zone 할당(WAL/SST 분리) 전략이 성능을 유지시키는데 도움이 되었음을 입증했다.

6. Future Work

현재의 순차 기록 방식은 Zone의 Write Point(WP)를 소프트웨어 레벨에서 관리해야 하므로, 멀티스레드 환경에서 I/O 병목이 발생할 여지가 있습니다. 향후 하드웨어 컨트롤러가 WP를 직접 관리해주는 Zone Append 명령어를 이용하고자 합니다. 이는 CPU 레벨에서의 오버헤드를 제거하고 여러 스레드가 동시에 하나의 Zone에 쓸 수 있게 해 처리량을 향상시킬 수 있습니다.