47

데이터 무결성과 보호

우리가 지금까지 살펴본 파일 시스템의 기본적인 기술들 이외에도 몇 가지 더 다루어야 할 주제들이 있다. 이번 장에서는 신뢰성을 다시 한 번 다루도록 하겠다(RAID를 설명한 장에서 저장 시스템의 신뢰성에 대해서 학습을 했었다). 이 장에서는 구체적으로 저장 장치가 기본적으로 신뢰할 수 없다는 전제 하에 파일 시스템이 데이터의 안전성을 보장하는 방법에 대해 살펴볼 것이다.

이 주제가 다루는 부분을 **데이터 무결성** 또는 **데이터 보호**라고 부른다. 이번 장에서 살펴볼 기술은 저장 시스템에서 데이터를 읽었을 때 그 데이터가 처음에 썼던 것과 동일하다는 것을 보장하는 기술이다.

핵심 질문: 데이터 무결성을 어떻게 보장하는가
저장 장치에 쓴 데이터가 보호되고 있다는 것을 시스템이 어떻게 보장할까? 어떤 기술들이 필요할까? 어떻게 하면 그 기술들을 공간과 시간 오버헤드가 적으면서 효율적으로 만들 수 있을까?

47.1 디스크 오류 모델

RAID를 다뤘던 장에서 배웠듯이, 디스크는 완전하지 않으며 (때에 따라) 오류가 발생할 수 있다. 초기 RAID 시스템에서는 오류 모델이 간단한 편이었다. 디스크 전체가 동작하던가, 또는 완전히 불능이었기 때문에 그러한 오류를 단순하게 판단할 수 있었다. RAID의 구현이 어렵지 않았던 것은 디스크의 **실패-시-멈춤**(**fail-stop**) 모델 때문이었다 [Sch90].

당신이 아직 배우지 못한 것이 있다면 현대의 디스크에서 나타나는 다른 종류의 오류 모델들이다. 구체적으로 Bairavasundaram 등이 자세히 연구한 내용에 따르면 [Bai+07; Bai+08] 현대의 디스크들은 정상적으로 동작하는 것처럼 보이지만 블럭들을 읽는 데 실패하는 경우가 있다고 한다. 구체적으로 빈번하게 발생하는 오류 중에 우리가 살펴볼 만한 것은 두 가지 종류의 단일-블럭 오류가 있다. 각각은 **숨어있는 섹터 에러**(**Latent**

sector error, LSE)와 **블럭 손상**(**block corruption**)이다. 각각에 대해서 좀 더 자세히 살펴보도록 하겠다.

LSE는 디스크 섹터(또는 섹터 그룹)가 어떤 이유로든 손상되었을 때 발생한다. 예를 들어, 디스크 헤드가 표면에 어떤 이유로 닿았다면(**헤드 크래시**(**head crash**), 일반적인 상황에서는 일어나면 안 됨) 표면을 망가뜨릴 것이고 비트들을 읽을 수 없게 만든다. 강한 방사선도 역시 비트를 반전시켜서 내용을 부정확하게 만들 수 있다. 다행스러운 것은 디스크 내의 **에러 정정 코드**(**error correcting codes, ECC**)를 사용하여 디스크 상에 있는 블럭의 비트가 괜찮은지 판단하고 어떤 경우에는 고치기도 한다. 어떤 비트의 상태가 좋지 않은데 그 부분의 에러를 고칠 충분한 정보를 갖고 있지 않다고 하자. 그때 해당 비트를 읽는 요청을 받으면 디스크는 에러를 리턴한다.

디스크가 손상 여부를 인식할 수 없게 내용이 **손상**(**corrupt**)된 경우이다. 이 경우 해당 블럭의 내용은 읽혀진다. 예를 들어, 버그가 있는 디스크 펌웨어 때문에 잘못된 위치에 쓰기를 했을 수 있다. 그 경우 디스크 ECC는 디스크의 내용이 정상이라고 표시하지만 사용자가 추후 해당 블럭을 읽었을 때, 잘못된 블럭이 리턴된다. 이전에 쓴 자료는 엉뚱한 곳에 기록되었기 때문이다. 마찬가지로 전송 버스 상에 오류로 인해서 호스트에서 디스크로 전송되는 도중에 블럭이 손상될 수도 있다. 손상된 데이터가 디스크에 저장이 된다. 사용자가 원하는 결과가 아니다. 이러한 종류의 오류들은 **조용한 오류**(**silent fault**)이기 때문에 더 심각하다. 오류가 있는 데이터를 리턴함에도 불구하고, 디스크는 문제를 전혀 알리지 않는다.

Prabhakaran 등은 이런 현대적 관점의 디스크 오류를 **부분-실패**(**fail-partial**) 디스크 오류 모델이라고 불렀다 [Pra+05]. 이와 같은 관점에서 디스크는 전체적으로는 불량으로 볼 수 있지만(고전적인 fail-stop 모델과 동일하게) 한편으로는 몇 개의 블럭들이 접근이 안 되거나(즉, LSE) 잘못된 내용(즉, 손상)을 갖고 있는 것을 제외하면 디스크가 동작하는 것처럼 보인다는 것이다. 그러므로 동작하는 것처럼 보이는 디스크를 사용할 때는, 가끔 주어진 블럭을 읽거나 쓸 때 에러를 리턴할 수 있으며(조용하지 않은 부분 오류) 가끔 잘못된 데이터를 리턴할 수 있다(조용한 부분 오류).

	저가	**고가**
LSE	0.094	0.014
블럭 손상	0.005	0.0005

〈그림 47.1〉 **패리티가 있는 RAID-4**

이 두 종류의 오류는 정확히 얼마나 드물게 발생할까? 그림 47.1은 두 편의 Bairavasundaram의 연구에서 발견한 결과를 정리하였다 [Bai+07; Bai+08].

관측 기간 동안 최소한 하나의 LSE 또는 블럭 손상이 보인 드라이브들의 백분율을 나타낸다(3년 동안 1.5백만 개 이상의 디스크 드라이브들). 디스크의 가격대에 따라 "저가"(일반적으로 SATA 드라이브) 그리고 "고가"(대부분 SCSI 또는 광섬유채널 방식)으로 나누어 표현하였다. 더 좋은 드라이브를 구매하는 것이 두 문제의 발생 빈도를

낮춘다는 것을 알 수 있다(약 열 배 정도). 저장 시스템에서 이러한 경우를 다루는 방법을 생각해야 할 만큼 충문히 많이 발생하고 있다는 것을 보여준다.

LSE에 대해 다음과 같은 현상이 발견되었다.

- 하나 이상의 LSE를 갖고 있는 고가의 드라이브들은 저가의 드라이브들처럼 추가적인 에러를 만들어 낸다.
- 대부분의 드라이브들의 연간 에러율은 둘째 해에 증가한다.
- LSE는 디스크의 크기에 비례하여 증가한다.
- LSE를 갖고 있는 대부분의 디스크는 50개 미만의 LSE를 갖고 있다.
- LSE를 갖고 있는 디스크들은 추가적인 LSE를 만들어 낼 확률이 있다.
- 공간과 시간 지역성이 상당히 두드러진다.
- 디스크를 지우는 것이 유용하다(이것으로 대부분의 LSE를 발견할 수 있다).

"섹터 손상"에 대해서 다음과 같은 특성이 관찰되었다.

- 같은 급의 드라이브들이라고 하더라도 모델에 따라서 손상 확률의 차이가 크다.
- 사용 연한에 따른 영향은 모델에 따라 다르다.
- 워크로드와 디스크의 크기는 섹터 손상 빈도에 큰 영향이 없다.
- 섹터 손상이 발생한 대부분의 디스크는 단지 몇 개의 손상된 부분을 갖고 있다.
- RAID로 묶인 디스크들 간이나 하나의 디스크에 있는 손상은 독립적이지 않다.
- LSE와는 연관성이 약하다.

이 오류들에 대해서 더 자세히 알고 싶다면 해당 논문을 읽어보기 바란다 [Bai+07; Bai+08]. 만약 신뢰할 수 있는 저장 시스템을 만들고 싶다면, LSE와 블럭 손상을 검출하고 복구할 수 있는 기술을 포함해야 한다.

47.2 숨어있는 섹터 에러(Latent Sector Error)

디스크의 부분 오류(partial failure) 두 가지에 대해 알았으니 대처 방법을 알아보자. 먼저 둘 중에 더 쉬운 주제인 숨어있는 섹터 에러를 해결해 보자.

핵심 질문: 숨어있는 섹터 에러를 어떻게 처리할까

저장 시스템이 숨어있는 섹터 에러들을 어떻게 처리해야 할까? 이런 형태의 부분 오류를 처리하기 위해서는 어떤 추가적인 기술을 필요로 할까?

숨어있는 섹터 에러는 쉽게 발견할 수 있기 때문에 해결이 어렵지 않다. 저장 시스템의 임의의 블럭을 접근할 때 디스크가 에러를 리턴한다면 저장 시스템은 어떤 식으로든 중복 정보를 저장하여 제대로 된 데이터를 리턴해야 한다. 미러링 RAID의 경우에서는 사본을 보관한다. 패리티를 기반으로 하는 RAID-4와 RAID-5 시스템의 경우에는 패리티 그룹 내의 다른 블럭들을 활용하여 해당 블럭을 재생성한다. 결론적으로 LSE처럼 쉽게 발견 가능한 문제에서는 잘 알려진 중복 정보를 이용한 복구 기법들이 존재하며, 이들 중복 기법들을 통해서 손쉽게 복구 할 수가 있다.

LSE가 점차 빈번해 지면서 수년 간 RAID의 설계에 변화가 왔다. RAID-4/5 시스템에서 발생할 수 있는 한 가지 흥미로운 문제는 하나의 디스크가 완전히 고장이 나면서 동시에 다른 디스크에서 LSE가 발생하는 경우이다. 어떤 디스크가 완전히 고장이 나면 RAID는 패리티 그룹 내의 모든 다른 디스크들을 읽어서 빠진 값을 다시 계산하여 디스크 **재구성(reconstruct)**을 시도한다(긴급 스페어에). 만약에 재구성하는 도중에 다른 디스크들 중 하나에서 LSE를 만나면, 문제가 발생한다. 재구성이 실패하게 된다.

이 문제를 해결하기 위해서 어떤 시스템들은 추가적인 기법을 도입했다. 예를 들면, NetApp의 **RAID-DP**는 두 개의 패리티 디스크를 사용한다 [Cor+04]. 재구성 도중에 LSE가 발생하면 추가적인 패리티가 블럭을 재구성하는 데 도움을 준다. 항상 그렇듯이 추가 비용이 발생한다. 각 스트라이프에 두 개의 패리티 블럭을 유지한다는 것은 큰 비용이다. 하지만 로그 기반의 구조를 갖는 NetApp의 **WAFL** 파일 시스템은 그러한 비용을 여러 측면으로 완화시켰다 [HLM94]. 마지막으로 남은 비용은 공간이다. 추가적인 패리티 블럭을 구성하기 위해 추가 디스크가 필요하다.

47.3 손상 검출: 체크섬

손상된 블럭을 찾아내는 것은 훨씬 어려운 일이다. 사용자가 원하지 않았던 값이 저장되었지만, 저장 장치 입장에서 보면 "제대로" 기록을 했기 때문이다.

핵심 질문: 어떻게 데이터가 손상이 되어도 무결성을 유지할까

오류의 침묵적인 특성을 고려한다면 손상이 일어났다는 것을 저장 시스템이 검출하기 위해서는 무엇을 해야 할까? 어떤 기술들이 필요한가? 어떻게 하면 기술들을 효율적으로 구현할 수 있을까?

손상을 *발견*하는 것이 핵심 문제이다. 블럭이 잘못되었다는 것을 사용자가 어떻게 알 수 있을까? 특정 블럭이 잘못되었다는 것을 알았다면 *복구*는 이전의 방법을 동일하게 적용할 수 있다. 블럭의 사본이 어딘가에 있어야 한다(그리고 손상되지 않았기를 바라야 한다). 우리는 여기서 검출 기술들에 집중하도록 하겠다.

현대의 저장 시스템이 데이터 무결성을 유지하기 위해서 사용하는 주요된 기술은 **체크섬(checksum)**이라고 불린다. 체크섬은 간단하게 말해서 데이터 청크(4 KB 블럭이라 하자)를 입력으로 하여 함수 값을 계산하는데, 이 결과는 데이터 내용에 대한 작은 요약

팁: 세상엔 공짜가 없다

세상엔 공짜는 없다. 미국의 숙어로서 There's No Such Things As A Free Lunch 라고 하고 약어로는 TNSTAAFL이라고 한다. 이것이 의미하는 바는 어떤 것을 공짜로 얻은 것처럼 보일 때, 사실은 어떤 식으로든 그에 대한 댓가를 지불할 가능성이 높다는 것이다. 이것은 식당에서 고객들에게 공짜 점심을 제공한다고 광고를 하여 손님을 유치했던 전략에서 유례되었다. 실제로 식당에 들어가서 "공짜" 점심을 먹으려면 주류를 주문해야 한다는 것을 깨닫게 된다. 물론, 당신이 열정적인 애주가라면 이것이 문제가 안될 수도 있다(또는 평범한 대학생들이라면).

정보이다(4 또는 8바이트). 체크섬의 목적은 데이터의 손상이나 변경 여부를 시스템이 판단할 수 있도록 하는 것이다. 체크섬을 데이터와 함께 저장하여 저장된 데이터로부터 계산한 현재의 체크섬이 저장 장치에 기록되어 있는 체크섬 값과 같은지 확인한다.

널리 사용되는 체크섬 함수

체크섬을 계산하기 위한 많은 함수들이 존재하며 각각의 강도(즉, 데이터 무결성을 보호하는 정도)와 연산 속도(즉, 함수의 계산 처리 속도)가 상이하다. 흔히 발생하는 절충이 필요한 상황이 여기서도 존재한다. 일반적으로 보호의 강도가 강할수록 비용이 더 비싸진다. 세상엔 공짜가 없다.

간단한 체크섬 함수로 XOR 연산이 있다. XOR 기반 체크섬은 체크섬을 구하고자 하는 각 데이터 블럭의 청크를 XOR 연산하여 최종적으로 전체 블럭을 대표하는 XOR 값을 생성한다.

16 바이트 크기의 블럭에 대해서 4 바이트 체크섬을 계산해 보자(이 블럭은 실제 디스크의 섹터나 블럭보다는 훨씬 작지만 이해를 돕기에는 충분하다). 16진수로 표현된 16 바이트의 데이터는 다음과 같다.

```
365e c4cd ba14 8a92 ecef 2c3a 40be f666
```

바이너리로 재표현하면 다음과 같이 나타난다.

```
0011 0110 0101 1110     1100 0100 1100 1101
1011 1010 0001 0100     1000 1010 1001 0010
1110 1100 1110 1111     0010 1100 0011 1010
0100 0000 1011 1110     1111 0110 0110 0110
```

각 줄에 4 바이트씩 데이터를 묶어서 나열하였기 때문에 체크섬의 결과를 확인하는 것이 쉽다. 각 열끼리 XOR 하여 얻은 체크섬의 값은 다음과 같다.

```
0010 0000 0001 1011     1001 0100 0000 0011
```

16진수로 표현하면 0x201b9403이다.

XOR 방식은 합리적이기는 하지만 제한이 있다. 체크섬을 계산하는 각 열에서 두 개의 값이 변하면 체크섬은 손상을 검출할 수 없다. 이 이유로 사람들은 다른 체크섬 함수를 연구하였다.

다른 기본적인 체크섬 함수는 덧셈이다. 이 방법의 장점은 빠르다는 것이다. 체크섬을 계산하려면 각 데이터의 청크에 대해서 2의 보수 덧셈을 하고 만약 오버플로우가 발생하면 무시한다. 이 기법은 데이터의 많은 부분이 변경되었어도 변경된 경우 이를 발견할 수 있기는 하지만, 데이터가 쉬프트된 경우에는 발견 못 할 수도 있다.

좀 더 복잡한 알고리즘은 **Fletcher checksum**으로서 John G. Fletcher라는 사람이 만들었다 [Fle82]. 두 개의 체크 바이트, $s1$과 $s2$의 계산이 필요한 비교적 간단한 방법이다. 구체적으로 다음과 같다. 블럭 D가 $d_1, \ldots, d_n$ 바이트로 이루어졌다고 가정하면 $s1$은 다음과 같이 정의된다. $s1 = s1 + d_i \; mod \; 255$(모든 d_i에 대해서 연산) 이고 $s2$의 계산은 $s2 = s2 + s1 \; mod \; 255$(이때도 모든 d_i에 대해서 연산)으로 한다 [Fen04]. 플렛처 체크섬은 CRC(다음 문단에서 소개)와 견줄 정도로 강력하다고 알려져 있으며 모든 한 비트, 두 비트 에러, 그리고 많은 경우의 동시다발적 에러들을 검출할 수 있다.

마지막으로 소개하는 또 하나의 흔히 사용되는 체크섬은 **Cyclic Redundancy Check**(**CRC**)이다. 복잡하게 들릴 수도 있지만, 기본 개념은 상당히 간단하다. 데이터 블럭 D에 대해서 체크섬을 계산한다고 가정해 보자. 이때에 할 일은 주어진 D를 큰 이진수(사실 비트의 나열일 뿐이다)로 여기고 사전에 합의한 값(k)으로 나누는 것이다. 계산의 나머지 값이 CRC 값이 된다. 이진 나머지 연산(modulo operation)은 상당히 효율적으로 처리를 할 수 있기 때문에 CRC는 네트워크 분야에서도 인기가 있다. 자세한 내용은 참고 문헌을 활용하자 [M13].

어떤 방식이 사용되든 완벽한 체크섬은 없다. 어떤 두 개의 전혀 다른 내용을 갖고 있는 블럭이 동일한 체크섬을 가질 수 있다. 이와 같은 경우를 **충돌**(**collision**)이라고 부른다. 이 사실은 직관적으로 알 수 있다. 결국 체크섬을 계산한다는 것은 어떤 큰 것(예, 4 KB)을 사용하여 작은 크기(예, 4 또는 8 bytes)의 요약 정보를 만들어 내는 것이기 때문이다. 좋은 체크섬 함수를 선택한다는 것은 충돌의 확률을 최소화하면서 계산은 간단하게 만드는 것을 의미한다.

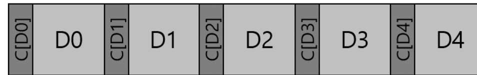
체크섬의 배치도

이제 체크섬의 계산 방식을 이해했으니, 저장 시스템에서 체크섬을 사용하는 방법을 분석해 보자. 첫 번째 질문은 체크섬의 배치 방법이다. 즉, 디스크 어디에 체크섬을 저장하는가를 질문해야 한다.

가장 기본적인 방식은 간단하게 각 디스크 섹터(또는 블럭)에 체크섬을 같이 저장하는 것이다. 데이터 블럭 D가 주어졌을 때 그 데이터의 체크섬을 C(D)라고 하자. 체크섬이 없을 때의 디스크 배치도는 다음과 같다.

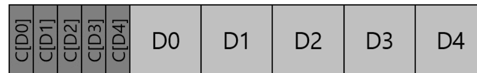
D0	D1	D2	D3	D4	D5	D6

체크섬이 있을 때는 다음과 같이 각 블럭에 하나의 체크섬이 추가가 된다.



체크섬은 보통 작다(예, 8 바이트). 디스크는 섹터 크기의 청크로(512 바이트) 또는 그 배수로만 쓸 수 있기 때문에 문제가 발생할 수 있다. 위와 같은 구조를 표현하는 방법에 문제가 생긴다. 드라이브 제조사가 채용한 해법 중 하나는 드라이브의 섹터를 520 byte 크기로 포맷을 하는 것이다. 그리고 섹터의 크기 중 8 byte를 체크섬 저장에 사용하는 것이다.

그러한 기능이 없는 디스크들의 경우에는 파일 시스템이 체크섬을 512 바이트 블럭에 저장할 수 있는 방법을 고안해야 한다. 가능한 방법 중의 하나는 다음과 같다.



이 방법에서는 n개의 체크섬들을 모아서 한 섹터에 저장하고 그 뒤를 이어서 해당하는 n개의 블럭들을 배치하였다. 이 방식은 모든 디스크들에 적용이 가능하다는 장점이 있지만 상대적으로 덜 효율적일 수 있다. 만약에 파일 시스템이, 예를 들어, 블럭 D1을 갱신하고자 한다면 C(D1)를 포함하는 체크섬 섹터를 읽은 후에 C(D1)를 새로 계산해야 할 뿐만 아니라 체크섬 섹터와 새로운 데이터 블럭 D1을 써야 한다(그러므로 하나의 읽기와 두 개의 쓰기가 필요하다). 그 전의 기법은(섹터당 하나의 체크섬) 변경을 반영하기 위해 한 번의 쓰기만 필요하다.

47.4 체크섬의 활용

체크섬의 배치 방법이 결정되었으니 이제 체크섬을 실제로 활용하는 방법에 대해서 이해해 보도록 하자. 블럭 D를 읽을 때면 사용자는(즉, 파일 시스템 또는 저장 장치 컨트롤러) 디스크에서 체크섬 $C_s(D)$를 읽는다(첨자 C_s가 있는 것에 유의하자). 이 체크섬을 **저장된 체크섬**(**stored checksum**)이라고 부른다. 사용자는 그 후에 해당 블럭 D로부터 체크섬 값을 계산해 내고, 이를 **계산된 체크섬**(**computed checksum**)($C_c(D)$)이라고 부른다. 이 시점에서 사용자는 저장된 체크섬과 계산된 체크섬을 비교한다. 만약 둘이 동일하다면(즉, $C_s(D) == C_c(D)$), 데이터는 손상이 없는 것이고 사용자에게 안전하게 리턴될 수 있다. 하지만 그 둘이 다르다면 (즉, $C_s(D) != C_c(D)$), 데이터가 처음 저장된 이후 변경되었다는 것을 의미한다(저장된 체크섬은 해당 시점의 체크섬을 나타내기 때문이다). 이 경우에는 손상된 것이고, 우리가 사용한 체크섬이 그 사실을 검출할 수 있도록 도와주었다.

블럭이 손상되었다면 자연스럽게 대처 방안을 질문하게 된다. 만약 복사본이 있다면 해답은 간단하다. 복사본을 대신 사용하면 된다. 저장 시스템에 복사본이 없다면 에러를 리턴하는 것 외에는 방법이 없다. 여하간에, 손상을 검출했다는 것만으로는 모든 것이 해결되지 않는다. 손상되지 않은 데이터를 얻을 길이 전혀 없다면 운이 다한 것이라고 생각해야 한다.

47.5 새로운 문제: 잘못된 위치에 기록

이제까지 다룬 LSE나 데이터 손상에 관련된 오류는 매우 기본적인 오류들이다. 하지만 현대의 디스크에서는 "황당한" 오류들이 발생할 수 있기 때문에 그에 대한 해법도 필요하다.

첫 번째 오류는 **잘못된 위치에 기록(misdirected write)** 이라고 부른다. 디스크와 RAID 컨트롤러에서 발생하는 현상으로 디스크에 데이터를 올바르게 썼지만 *잘못된* 위치에 쓰는 현상이다. 단일 디스크 시스템에서 이 현상이 나타났다는 것은 주소 x(의도한 주소)에 기록되어야 할 블럭 D_x 가 주소 y 에 기록이 되었다는 것을 말한다(그러므로 "손상된" D_y 를 얻는다). 추가적으로 RAID에서는 컨트롤러가 디스크 i 의 주소 x 에 $D_{i,x}$ 를 쓰는 것이 아니라 다른 디스크 j 에 기록하는 것을 뜻한다. 질문은 다음과 같다.

> **핵심 질문: 잘못된 위치에 기록하는 문제를 어떻게 해결할까**
> 저장 시스템이나 디스크 컨트롤러가 엉뚱한 곳에 쓰기를 어떻게 검출해야 할까? 체크섬에 어떤 기능이 추가되어야 할까?

그 해답은 간단하다. 각 체크섬에 추가적인 정보를 더하면 된다. 이 경우에는 **물리적 식별자(physical identifier** 또는 **물리적 ID)** 를 추가하는 것만으로 큰 도움이 된다. 예를 들어 체크섬 $C(D)$ 과 디스크 번호와 섹터 번호를 포함하여 저장한다. 이 추가 정보를 활용하여 사용자는 해당 블럭에 정확한 정보가 저장이 되었는지 판단할 수 있다. 구체적으로, 만약 사용자가 디스크 10번에서 4번 블럭($D_{10,4}$)을 읽는다고 하면 저장된 정보는 디스크 번호와 섹터 오프셋을 다음 그림처럼 포함해야 한다. 만약 해당 정보가 일치하지 않는다면, 엉뚱한 곳에 쓰기를 한 것이고, 손상되었음을 알 수 있다. 디스크 두 개를 사용하는 시스템에서 추가된 정보의 배치 방법은 다음의 그림과 같다. 이 그림 역시 전에 사용한 것들과 마찬가지로 크게 비율에 맞지 않는 것을 이해하기 바란다. 체크섬은 대체적으로 작으며(예, 8 byte) 블럭들은 훨씬 크다(예, 4 KB 또는 그 이상).

Disk 1	C[D0]	disk=1	block=0	D0	C[D1]	disk=1	block=1	D1	C[D2]	disk=1	block=2	D2
Disk 2	C[D0]	disk=2	block=0	D0	C[D1]	disk=2	block=1	D1	C[D2]	disk=2	block=2	D2

디스크 상의 자료 구조에서 볼 수 있는 것처럼 디스크의 추가 정보가 상당히 많아졌다. 디스크 번호가 각 블럭마다 반복되고 블럭의 오프셋도 해당 블럭 옆에 따라 붙어 있는 것을 볼 수 있다. 추가 정보의 존재는 전혀 놀라울 것이 없다. 에러 검출에 있어서(이 경우에는) 추가 정보의 존재는 핵심이며 복구에서도(다른 경우들) 중요하다. 완벽한 디스크들에서는 전혀 필요 없는 작은 추가 정보이지만 문제가 발생되면 이 정보가 그 상황을 해결하는 데 큰 도움이 된다.

47.6 마지막 문제: 기록 작업의 손실

불행하게도 엉뚱한 곳에 쓰기는 우리가 다뤄야 할 마지막 문제가 아니다. 어떤 현대의 저장 장치들은 **기록 작업의 손실**(**lost write**)이라는 문제도 갖고 있다. 이 문제는 상위 계층에게는 쓰기가 완료되었다고 알리지만 실제로는 저장되지 않은 경우를 나타낸다. 그렇기 때문에 디스크의 블럭은 새로운 내용으로 갱신되지 않고 예전의 블럭 내용이 남겨져 있는 상황이 된다.

여기서 질문은 이제까지 다룬 체크섬 전략들을 이러한 "기록 작업의 손실"에 문제의 해결에 사용할 수 있는가 하는 것이다. 불행하게도 이전의 방법들은 사용할 수 없다. 저장된 블럭은 제대로 된 체크섬을 갖고 있을 것이고 앞서 사용한 물리적 ID(디스크 번호와 블럭 오프셋)도 정확하기 때문이다. 마지막 질문은 다음과 같다.

핵심 질문: 기록 작업의 손실 문제를 어떻게 다룰까

저장 시스템 또는 디스크 컨트롤러는 기록 작업이 제대로 수행되지 않았을 경우 이를 어떻게 발견할 수 있을까? 체크섬 외에 어떤 다른 기능이 필요할까?

도움이 될 만한 가능한 해법들이 여럿 있다 [Kri+08]. 고전적인 방법들 중 한 가지 방법 [BS04]은 **쓰기 검증**(**write verify**) 또는 **쓰기 후 읽기**(**read-after-write**)라는 것을 수행하는 것이다. 쓰기를 수행한 후 즉시 그 값을 다시 읽는 것을 통해 시스템은 데이터가 디스크 표면에 잘 도착했다는 것을 알 수 있다. 하지만 이 방법은 그렇지 않아도 느린 쓰기 동작을 완료하기 위해 I/O의 수를 두 배로 늘린다.

어떤 시스템은 체크섬을 시스템의 다른 위치에 기록하여 잃어버린 쓰기를 검출해 내기도 한다. 예를 들어 Sun의 **Zettabyte 파일 시스템**(**ZFS**)의 경우 파일 블럭들의 체크섬을 아이노드와 간접 블럭에 저장한다. 그러므로 데이터 블럭에 대한 쓰기는 손실되었더라도 아이노드 내의 체크섬은 갱신되었을 것이므로, 예전의 데이터와 해당 블럭에 대한 체크섬(아이노드에 존재하는)이 일치하지 않게 된다. 아이노드와 데이터의 쓰기가 둘 다 손실되었다면 이 접근법이 실패하게 될 것이지만 그런 경우는 발생 확률이 적을 것이다(하지만 불행하게도 가능하다!).

47.7 Scrubbing

한 가지 의문이 떠올랐을 수 있다. 이 체크섬들은 언제 검사할까 하는 것이다. 물론 응용 프로그램이 데이터를 읽을 때 검사하기는 하지만 대부분의 데이터는 거의 접근이 안되기 때문에 검사가 안 된 채로 남아있게 된다. 검사가 안 된 데이터는 신뢰성 있는 저장 시스템을 만드는 데 문제가 될 수 있다. 오류가 있는 비트가 다른 데이터 영향을 미칠 수 있기 때문이다.

이 문제를 해결하기 위해서 많은 시스템들은 **디스크 다시 읽기(disk scrubbing)** 류의 기법을 사용한다 [Kri+08]. 주기적으로 시스템의 모든 블럭들을 읽어서 체크섬이 여전히 유효한지를 검사하여 디스크 시스템은 특정 데이터의 모든 사본이 모두 손상되는 확률을 줄인다. 일반적인 시스템들은 매일 밤 혹은 매주 데이터를 검사한다.

47.8 체크섬 오버헤드

마무리 하기 전에 데이터 보호를 위해서 체크섬을 사용하는 데 드는 비용을 알아 보기로 하자. 컴퓨터 시스템에서 흔한 공간과 시간이라는 두 가지 비용이 존재한다.

공간 오버헤드는 다시 두 가지 형태로 나눌 수 있다. 하나는 디스크 자체(또는 다른 저장 매체)의 오버헤드로서 디스크 상에 체크섬을 저장하기 위한 공간이 필요하며 그에 따라 사용자가 저장할 수 있는 공간이 줄어들게 된다. 일반적으로 4KB 데이터 블럭당 8 바이트이기 때문에 디스크 상에서 약 0.19%를 사용한다.

두 번째 종류의 공간 오버헤드는 시스템 메모리로 인한 것이다. 데이터를 접근할 때 메모리에 데이터와 체크섬을 읽어들일 수 있는 공간이 필요하다. 시스템이 체크섬을 확인하고 즉시 제거할 경우, 단시간 동안의 메모리만 필요하게 되어 문제가 되지 않는다. 하지만 체크섬이 메모리에 유지되어야 할 경우(메모리 손상에 대비하여 추가적인 보호 장치가 필요하기 때문에 [Zha+13]), 그때에는 비록 작은 용량이지만 메모리 필요량이 증가하게 된다.

공간 오버헤드는 그렇게 크지 않은 반면에 체크섬으로 인한 시간 오버헤드는 뚜렷하게 나타난다. 최소한 CPU는 데이터를 저장할 때(체크섬을 연산해야 함)와 접근할 때(체크섬을 다시 계산하여 저장된 체크섬과 비교해야 함) 각 블럭의 체크섬을 연산해야 한다. 체크섬을 사용하는 많은 시스템(네트워크 계층을 포함)에서 CPU 오버헤드를 줄이기 위해 사용하는 한 가지 방법은 어차피 데이터를 복사하는 것은 불가피하기 때문에 체크섬 연산과 데이터 복사(예, 커널의 페이지 캐시에서 사용자 버퍼로 데이터를 복사하기)를 하나의 연속적인 작업으로 처리하는 것이다. 복사하기와 체크섬 연산을 한 번에 처리하는 것은 상당히 효율적이다.

CPU 오버헤드 외에도 체크섬 기법들에 따라서 추가적인 I/O를 유발할 수 있다. 특히, 체크섬이 데이터와 구분되어 저장된 경우(그러므로 해당 블럭을 접근하기 위한 추가 I/O가 필요함)와 백그라운드에서 다시 읽기(scrubbing)를 수행하는 경우 I/O가 발생한다. 전자의 경우 설계를 통해 감소시킬 수가 있으며, 후자의 경우 다시 읽기 수행

시점을 적절히 조절하여 영향을 줄일 수 있다. 다시 읽기를 실행하기 제일 좋은 시점은 대부분의(전부는 아님!) 개발자들이 잠든 한밤 중일 것이다.

47.9 요약

우리는 체크섬의 구현과 사용을 중심으로 저장 장치에서 데이터를 보호하는 기법을 소개하였다. 체크섬은 종류에 따라 서로 다른 종류의 오류를 해결한다. 저장 장치가 발전하면서 여지없이 새로운 오류의 종류가 생겨날 것이다. 그러한 변화에 대응하기 위하여 기업과 연구자들은 기존 기법들을 근본적으로 재설계하거나 전혀 새로운 기법들을 개발해야 할지도 모른다. 기다려보면 알게 될 것이다. 시간은 그런 면에서 참 흥미롭다.

참고 문헌

[Bai+07] **"An Analysis of Latent Sector Errors in Disk Drives"**
Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler
SIGMETRICS '07, San Diego, California, June 2007
잠재적인 섹터 에러를 상세히 연구한 첫 논문이다. 다음 참고 문헌[Bai+08]에서도 설명한 것처럼 Wisconsin과 NetApp의 공동연구이다. 이 논문으로 Kenneth C. Sevcik이 우수 학생 논문상을 받았다. Sevcik은 너무 일찍 타계한 훌륭한 연구자이자 멋진 사람이다. 한 번은 저자들에게 미국에서 캐나다로 이주하고도 행복할 수 있다는 것을 보여주기 위해서 식당 중앙에 서서 캐나다의 국가를 불러주기도 했다.

[Bai+08] **"An Analysis of Data Corruption in the Storage Stack"**
Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008
디스크 손상에 대해서 정말 자세히 연구했다고 할 만한 첫 논문으로서 백오십만 개의 드라이브를 삼년 동안 관찰하여 얼마나 자주 손상되는지를 분석하는 것에 집중하였다. Lakshmi가 Wisconsin에서 저자들에게 지도받는 대학원생이었을 때 이 연구를 진행하였다. 그 당시 NetApp에서 여러 방학 기간 동안 인턴을 하면서 알게 된 동료들과 공동 연구로 진행하였다. 산업계와 같이 작업하는 것이 매우 흥미로울 뿐만 아니라 유용한 연구 결과도 만들어 낸다는 것에 대한 좋은 예시라고 할 수 있다.

[BS04] **"Commercial Fault Tolerance: A Tale of Two Systems"**
Wendy Bartlett and Lisa Spainhower
IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January 2004
결함 허용 시스템 구현에 대한 고전으로 IBM과 Tandem의 최고 기술에 대한 탁월한 개론이다. 이 분야에 관심이 있는 사람이라면 꼭 읽어야 하는 글이다.

[Cor+04] **"Row-Diagonal Parity for Double Disk Failure Correction"**
P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar
FAST '04, San Jose, CA, February 2004
중복을 추가하는 것이 전체 디스크 오류/부분적 디스크 오류가 결합된 문제를 해결하는 데 도움이 된다는 것을 보여준 초기 논문이다. 이론과 실제 연구 간의 균형에 대한 좋은 예이기도 하다.

[Fen04] **"Checksums and Error Control"**
Peter M. Fenwick
FAST '08, San Jose, CA, February 2008
URL: `www.cs.auckland.ac.nz/compsci314s2c/resources/Checksums.pdf`
체크섬에 대한 간단하고 훌륭한 사용 지침이다. 심지어 공짜로 사용할 수 있다.

[Fle82] **"An Arithmetic Checksum for Serial Transmissions"**
John G. Fletcher
IEEE Transactions on Communication, Vol. 30, No. 1, January 1982
Fletcher 기법의 원조 논문으로 저자의 이름과 동명의 체크섬 기법을 소개하였다. 물론, Fletcher 체크섬이라고 처음부터 부르지는 않았지만 그렇다고 이름을 붙이지도 않았다. 그렇기 때문에 개발자의 이름을 자연스레 따서 부르게 되었다. 그러니 허풍을 떨었다고 Fletcher를 탓하지 말자. 이와 유사 일화는 Rubik과 그의 정육면체를 들 수 있다. Rubik은 절대로 "루빅스 큐브"라고 부르지 않았다. 다만 그는 "내 큐브"라고 불렀다.

[HLM94] **"File System Design for an NFS File Server Appliance"**
Dave Hitz, James Lau, and Michael Malcolm

USENIX Spring '94
선구자적 논문으로 NetApp의 중요한 핵심 제품들과 개념들을 소개하였다. 이 시스템을 기반으로 하여 NetApp은 수십억 달러 가치의 저장 장치 기업으로 성장하였다. 기업의 설립 과정에 대해서 궁금하다면 Hitz의 자서전인 "How to Castrate a Bull: Unexpected Lessons on Risk, Growth, and Success in Business"(농담아니고 '황소를 거세하는 방법'이 실제 제목이다). 컴퓨터 과학을 전공했다고 황소 거세하기 따위는 피할 수 있으리라 생각했을 것이다.

[Kri+08] **"Parity Lost and Parity Regained"**
Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008
NetApp의 동료들과 공동으로 작업한 우리의 연구로서 여러 다른 체크섬 기법들이 데이터를 보호하기 위해 어떻게 동작하는지를 살펴보았다. 현재의 보호 전략들에 있는 여러 흥미로운 결함들을 밝혀냈으며 일부는 상용 제품들을 수정하도록 까지 만들었다.

[M13] **"Cyclic Redundancy Checks"**
URL: `http://www.mathpages.com/home/kmath458.htm`
누가 쓴 것인지 정확하지 않지만 CRC에 대한 정말 명확하고 간결한 설명이 정리되어 있다. 역시 인터넷은 정보로 가득 차있다.

[Pra+05] **"IRON File Systems"**
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005
디스크에 부분 실패 모드가 있다는 것을 소개한 우리의 논문이다. Linux의 ext3나 NTFS와 같은 파일 시스템이 이러한 실패에 어떻게 반응하는지를 상세하게 분석하였다. 결과적으로 아주 형편없이 처리하고 있었다! 수많은 버그와 설계 결함 그리고 이상한 점들을 발견해냈다. 그 중 일부분은 Linux 진영에 반영되어 당신의 데이터를 저장하는 새롭고 더욱 강인한 파일 시스템을 만드는 데 공헌하였다.

[RO91] **"Design and Implementation of the Log-structured File System"**
Mendel Rosenblum and John Ousterhout
SOSP '91, Pacific Grove, CA, October 1991
파일 시스템의 쓰기 성능을 개선하는 방법을 다룬 또 하나의 획기적인 논문이다.

[Sch90] **"Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial"**
Fred B. Schneider
ACM Surveys, Vol. 22, No. 4, December 1990
결함 허용 서비스 개발 방법을 전반적으로 다룬 고전 논문으로 기본이 되는 용어들의 정의가 다수 포함되어 있다. 분산 시스템을 개발하는 사람들은 꼭 읽어야 하는 글이다.

[Zha+13] **"Zettabyte Reliability with Flexible End-to-End Data Integrity"**
Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
MSST '13, Long Beach, California, May 2013
시스템의 페이지 캐시에 데이터 보호를 추가한 우리의 논문으로 메모리 손상뿐만 아니라 디스크 상의 손상에서도 보호할 수 있도록 하였다.