45

Data Integrity and Protection (Veri Bütünlüğü ve Koruma)

Şimdiye kadar incelediğimiz dosya sistemlerinde bulunan temel gelişmelerin ötesinde, bir dizi özellik üzerinde çalışmaya değer. Bu bölümde, bir kez daha güvenilirliğe odaklanıyoruz (daha önce RAID bölümünde depolama sistemi güvenilirliğini incelemiştik).Özellikle, modern depolama cihazlarının güvenilmez doğası göz önüne alındığında, bir dosya sistemi veya depolama sistemi, verilerin güvenli olmasını nasıl sağlamalıdır ?

Bu genel alana **veri bütünlüğü (data integrity)** veya **veri koruması (data protection)** denir. Bu nedenle ,şimdi depolama sisteminize koyduğunuz verilerin, depolama sistemi size geri döndüğünde aynı olmasını sağlamak için kullanılan teknikleri inceleyeceğiz.

CRUX: VERİ BÜTÜNLÜĞÜ NASIL SAĞLANIR?

Sistemler,belleğe yazılan verilerin korunmasını nasıl sağlamalıdır? Hangi teknikler gereklidir? Bu tür teknikler, hem düşük alan hem de zaman giderleriyle nasıl verimli hale getirilebilr?

45.1 Disk Hatası Modları

RAID ile ilgili bölümde öğrendiğiniz gibi, diskler mükemmel değildir ve (arada sırada) arızalanabilir. İlk RAID sistemlerinde, arıza modeli oldukça basitti: ya tüm disk çalışıyor ya da tamamen arızalanıyor ve böyle bir arızanın tespiti basittir. Disk arızasının bu **arıza durdurma(fail-stop)** modeli , RAID oluşturmayı nispeten basit hale getirir [S90].

Öğrenmediğiniz şey, modern disklerin sergilediği diğer tüm hata modu türleri hakkındadır. Özellikle, Bairavasundaram ve diğerleri. Çok ayrıntılı olarak incelendiğinde [B + 07, B + 08], modern diskler bazen çoğunlukla çalışıyor gibi görünecek, ancak bir veya daha fazla bloğa başarılı bir şekilde erişmekte zorlanacaktır. Özellikle, iki tür tek bloklu hata yaygındır ve dikkate değerdir: **gizli sektör hataları (laten-sector errors)** (**LSE'ler**) ve **blok bozulması (block corruption)**. Şimdi her birini daha ayrıntılı olarak tartışacağız.

	Ucuz	Maliyetli
LSE'ler	9.40 %	1.40%
Bozulma	0.50 %	0.05%

Şekil 45.1: **LSE'lerin Sıklığı ve Blok Bozulması**

LSE'ler, bir disk sektörü (veya sektörler grubu) bir şekilde zarar gördüğünde ortaya çıkar. Örneğin, disk kafası herhangi bir nedenle yüzeye dokunursa (bir **kafa çarpması(head crash)**, normal çalışma sırasında olmaması gereken bir şey), yüzeye zarar verebilir ve bitleri okunamaz hale getirebilir. Kozmik ışınlar da bitleri çevirebilir ve yanlış içeriklere yol açabilir. Neyse ki, disk içi **hata düzeltme kodları (error correcting codes)** (**ECC**), sürücü tarafından bir bloktaki disk bitlerinin iyi olup olmadığını belirlemek ve bazı durumlarda bunları düzeltmek için kullanılır ; iyi değillerse ve sürücü hatayı düzeltmek için yeterli bilgiye sahip değilse, bunları okumak için bir istek gönderildiğinde disk bir hata döndürür.

Ayrıca bir disk bloğunun diskin kendisi tarafından algılanamayacak şekilde **bozulduğu(corrupt)** durumlar da vardır. Örneğin, buggy disk üretici yazılımı yanlış konuma bir blok yazabilir ; Böyle bir durumda, disk ECC blok içeriğinin iyi olduğunu gösterir , ancak istemcinin bakış açısından daha sonra erişildiğinde yanlış blok döndürülür . Benzer şekilde, bir blok hatalı bir veri yolu üzerinden ana bilgisayardan diske aktarıldığında kopabilir; ortaya çıkan bozuk veriler disk tarafından depolanır, ancak istemcinin istediği şey bu değildir. Bu tür hatalar özellikle sinsidir çünkü **sessiz hatalardır(silent fauls)**; disk, hatalı verileri döndürürken sorunun hiçbir belirtisini vermez.

Prabhakaran ve diğerleri disk hatasının bu daha modern görünümünü , **başarısız kısmi(fail partial)** disk hatası modeli [P+05] olarak açıklar. Bu görüşe göre, diskler yine de bütünüyle başarısız olabilir (geleneksel hata durdurma modelinde olduğu gibi); ancak, diskler görünüşte çalışıyor olabilir ve bir veya daha fazla bloğun erişilemez hale gelmesine (örneğin, LSE'ler) veya yanlış içeriği tutmasına (yani, bozulma) neden olabilir. Bu nedenle, görünüşte çalışan bir diske erişirken, arada bir belirli bir bloğu okumaya veya yazmaya çalışırken bir hata döndürebilir (sessiz olmayan kısmi bir hata) ve arada bir yanlış verileri (sessiz kısmi bir hata) döndürebilir.

Bu tür hataların her ikisi de biraz nadirdir, ancak ne kadar nadirdir? Şekil 45.1, iki Bairavasundaram çalışmasından elde edilen bulguların bazılarını özetlemektedir [B+07,B+08].

Şekil, çalışma boyunca en az bir LSE sergileyen veya bozulmayı engelleyen sürücülerin yüzdesini göstermektedir (yaklaşık 3 yıl, 1,5 milyon disk sürücüsü). Şekil ayrıca sonuçları "ucuz" sürücülere (genellikle SATA sürücüler) ve "maliyetli" sürücülere (genellikle SCSI veya Fiber Kanal) ayırır . Gördüğünüz gibi, daha iyi sürücüler satın almak her iki tür sorunun sıklığını azaltırken (yaklaşık bir büyüklük sırasına göre), yine de depolama sisteminizde bunları nasıl ele alacağınız konusunda dikkatlice düşünmeniz gereken kadar sık görülürler.

LSE'lerle ilgili bazı ek bulgular şunlardır:

- Birden fazla LSE'ye sahip maliyetli sürücülerin, daha ucuz sürücüler kadar ek hatalar geliştirme olasılığı yüksektir
- Çoğu sürücü için, yıllık hata oranı ikinci yılda artar
- Disk boyutuyla birlikte LSE sayısı artar
- LSE'li disklerin çoğunda 50'den az disk bulunur
- LSE'li disklerin ek LSE'ler geliştirme olasılığı daha yüksektir
- Önemli miktarda mekansal ve zamansal yerellik vardır.
- Disk temizleme yararlıdır

Yolsuzlukla ilgili bazı bulgular:

- Bozulma olasılığı, aynı sürücü sınıfındaki farklı sürücü modellerine göre büyük farklılıklar gösterir
- Yaş efektleri modeller arasında farklılık gösterir
- İş yükü ve disk boyutunun bozulma üzerinde çok az etkisi vardır
- Bozuk olan disklerin çoğunda yalnızca birkaç bozulma vardır
- Bozulma bir disk içinde veya RAID'deki diskler arasında bağımsız değildir
- Mekansal yerellik ve bazı zamansal yerellikler vardır.
- LSE'lerle zayıf bir korelasyon vardır

Bu başarısızlıklar hakkında daha fazla bilgi edinmek için, muhtemelen orijinal makaleleri [B+07,B+08] okumalısınız. Ancak umarım ana nokta açık olmalıdır: gerçekten güvenilir bir depolama sistemi kurmak istiyorsanız, hem LSE'leri tespit etmek hem de yolsuzluğu önlemek için makine dahil etmelisiniz.

45.2 Gizli Sektör Hatalarını İşleme

Bu iki yeni kısmi disk arızası modu göz önüne alındığında , şimdi onlar hakkında ne yapabileceğimizi görmeye çalışmalıyız. Önce ikisinin daha kolayını, yani gizli sektör hatalarını ele alalım.

CRUX: GİZLİ SEKTÖR HATALARI NASIL ELE ALINIR

Bir depolama Sistemleri gizli sektör hatalarını nasıl işlemelidir? Bu kısmi arıza biçimiyle başa çıkmak için değil kadar fazladan makineye ihtiyaç vardır?

Görünüşe göre, gizli sektör hataları, (tanım gereği) kolayca tespit edildikleri için ele alınması oldukça basittir. Bir depolama sistemi bir bloğa erişmeye çalıştığında ve disk bir hata döndürdüğünde, depolama sistemi basitçe doğru verileri döndürmek için sahip olduğu yedekleme mekanizmasını kullanır. Örneğin, yansıtılmış bir RAID'de, sistem alternatif kopyaya erişmelidir; eşliğe dayalı bir RAID-4 veya RAID-5 sisteminde, sistem bloğu eşlik grubundaki diğer bloklardan yeniden oluşturmalıdır. Böylece, LSE'ler gibi kolayca tespit edilen sorunlar, standart artıklık mekanizmaları aracılığıyla kolayca giderilir.

LSE'lerin artan yaygınlığı, RAID tasarımlarını yıllar içinde etkilemiştir . Özellikle ilginç bir sorun, hem tam disk hataları hem de LSE'ler birlikte ortaya çıktığında RAID-4/5 sistemlerinde ortaya çıkar. Özellikle, bir diskin tamamı arızalandığında, RAID eşlik grubundaki diğer tüm diskleri okuyarak ve eksik değerleri yeniden hesaplayarak diski **yeniden yapılandırmaya(reconstruct)** çalışır (örneğin, etkin yedekte). Yeniden yapılandırma sırasında, diğer disklerden herhangi birinde bir LSE ile karşılaşılırsa, bir sorunumuz var demektir: yeniden yapılandırma başarıyla tamamlanamaz.

Bu sorunla mücadele etmek için, bazı sistemler fazladan bir yedeklilik derecesi ekler. Örneğin, NetApp'in **RAID-DP'si** bir [C+04] yerine iki eşlik diskine eşdeğerdir. Yeniden yapılanma sırasında bir LSE keşfedildiğinde, ekstra eşlik, eksik bloğun yeniden yapılandırılmasına yardımcı olur. Her zaman olduğu gibi, her şerit için iki parite bloğunun korunmasının daha maliyetli olması nedeniyle bir maliyet vardır; ancak, NetApp **WAFL** dosya sisteminin günlük yapılandırılmış doğası birçok durumda bu maliyeti azaltır [HLM94]. Kalan maliyet, ikinci eşlik bloğu için ekstra bir disk şeklinde alandır.

45.3 Bozulmayı Algılama: Sağlama Toplamı

Şimdi daha zorlu bir sorunun, veri bozulması yoluyla sessiz başarısızlıkların üstesinden gelelim. Bozulma ortaya çıktığında kullanıcıların kötü veri almasını nasıl önleyebiliriz ve böylece disklerin kötü veri döndürmesine neden olabiliriz?

CRUX: BOZULMAYA RAĞMEN VERİ BÜTÜNLÜĞÜ NASIL KORUNUR?

Bu kapı arızaların sessiz doğası göz önüne alındığında , bir depolama Sistemleri bozulmanın değil zaman ortaya çıktığını tespit etmek için ne yapabilir? Hangi tekniklere ihtiyaç var? Bunları verimli bir şekilde nasıl uygulayabilirsiniz ?

Gizli sektör hatalarının aksine, bozulmanın *tespiti* önemli bir sorundur. Bir müşteri bir bloğun kötüye gittiğini nasıl söyleyebilir? Belirli bir bloğun kötü olduğu bilindikten sonra, *kurtarma* öncekiyle aynıdır: bloğun başka bir kopyasına sahip olmanız gerekir (ve umarım bozulmamış bir tane!). Bu nedenle, burada tespit tekniklerine odaklanıyoruz .

Modern depolama sistemleri tarafından veri bütünlüğünü korumak için kullanılan birincil mekanizmaya **sağlama toplamı (checksum)** denir. Bir sağlama toplamı, bir veri yığınını (örneğin 4 KB'lık bir blok) giriş olarak alan ve söz konusu veriler üzerinden bir işlevi hesaplayan ve verilerin içeriğinin küçük bir özetini (örneğin 4 veya 8 bayt) üreten bir işlevin sonucudur. Bu özet, sağlama toplamı olarak adlandırılır. Böyle bir hesaplamanın amacı, sağlama toplamını verilerle birlikte depolayarak ve daha sonra verinin mevcut sağlama toplamını orijinal depolama değeriyle eşleştiğini daha sonraki erişimde onaylayarak bir sistemin verilen bir şekilde bozulup bozulmadığını veya değiştirilip değiştirilmediğini algılamasını sağlamaktır.

İPUCU: ÜCRETSİZ ÖĞLE YEMEĞİ YOK

Ücretsiz Öğle Yemeği Gibi Bir Şey Yoktur veya kısaca TNSTAAFL, görünüşte ücretsiz bir şey aldığınızda, gerçekte bunun için bir miktar maliyet ödediğinizi ima eden eski bir Amerikan deyimidir . Yemek yiyenlerin müşterileri için ücretsiz bir öğle yemeği reklamı yaptıkları, onları içeri çekmeyi umdukları eski günlerden geliyor; Sadece içeri girdiğinizde, "ücretsiz" öğle yemeğini almak için bir veya daha fazla alkollü içecek satın almanız gerektiğini fark ettiniz. Tabii ki, bu aslında bir sorun olmayabilir, kısmen hevesli bir alkolikseniz (veya tipik bir lisans öğrencisiyseniz).

Ortak Sağlama Toplamı İşlevleri(Common Checksum Functions)

Sağlama toplamlarını hesaplamak için bir dizi farklı işlev kullanılır ve güç (yani, veri bütünlüğünü korumada ne kadar iyi oldukları) ve hız (yani, ne kadar hızlı hesaplanabilirler) bakımından farklılık gösterir. Sistemlerde ortak olan bir ödünleşim burada ortaya çıkar: genellikle, ne kadar çok koruma alırsanız, o kadar pahalı olur. Ücretsiz öğle yemeği diye bir şey yoktur.

Bazı kullanımların özel veya (XOR) temel aldığı basit bir sağlama toplamı işlevi. XOR tabanlı sağlama toplamlarında, sağlama toplamı, sağlama toplamı veri bloğunun her bir öbeğinin XOR'u ile hesaplanır, böylece tüm bloğun XOR'unu temsil eden tek bir değer üretilir.

Bunu daha somut hale getirmek için, 16 baytlık bir blok üzerinden 4 baytlık bir kontrol toplamı hesapladığımızı hayal edin (bu blok elbette gerçekten bir disk sektörü veya blok olamayacak kadar küçüktür, ancak örnek için hizmet edecektir). Onaltılık olarak 16 veri baytı şöyle görünür:

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

Bunları ikili olarak görüntülersek, aşağıdakileri elde ederiz:

```
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
```

Verileri satır başına 4 baytlık gruplar halinde sıraladığımızdan, sonuçta elde edilen sağlama toplamının ne olacağını görmek kolaydır: son sağlama toplamı değerini elde etmek için her sütun üzerinde bir XOR gerçekleştirin:

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

Sonuç, onaltılıkta, 0x201b9403.

XOR makul bir sağlama toplamıdır, ancak sınırlamaları vardır. Örneğin, her bir sağlama toplamı birimi içinde aynı konumda bulunan iki bit değişirse, sağlama toplamı bozulmayı algılamaz. Bu nedenle, insanlar diğer sağlama toplamı işlevlerini araştırmışlardır

Bir diğer temel sağlama toplamı işlevi eklemedir. Bu yaklaşımın hızlı olma avantajı vardır; hesaplamak, taşmayı göz ardı ederek yalnızca verilerin her bir parçası üzerinde 2'nin tamamlayıcı eklemesini gerçekleştirmeyi gerektirir. Verilerdeki birçok değişikliği algılayabilir, ancak örneğin veriler kaydırılırsa iyi değildir.

Biraz daha karmaşık bir algoritma, mucit John G. Fletcher [F82] için (tahmin edebileceğiniz gibi) adlandırılan **Fletcher sağlama toplamı(Fletcher checksum)** olarak bilinir. Hesaplanması oldukça basittir ve iki kontrol baytının, s 1 ve *s 2'nin* hesaplanmasını içerir. Özellikle, bir D bloğunun *d*1...dn bayttan oluştuğunu varsayalım; s 1 aşağıdaki gibi tanımlanır: s 1 = (*s*1 + d i) *mod* 255 (tüm *d*i üzerinden hesaplanır); s 2 sırasıyla: s 2 = (s2 + *s* 1) *mod* 255 (yine tüm *d*i üzerinde) [F04]. Fletcher sağlama toplamı neredeyse CRC kadar güçlüdür (aşağıya bakın), tüm tek bit, çift bit hataları ve birçok seri çekim hatasını algılar [F04].

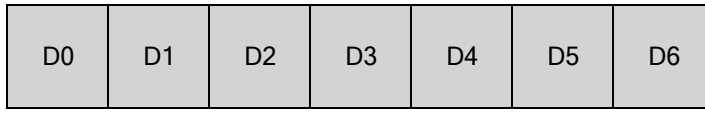
Yaygın olarak kullanılan son bir sağlama toplamı, **döngüsel artıklık denetimi(cyclic redunancy check)(CRC**) olarak bilinir. Bir D veri bloğu üzerinden sağlama toplamını hesaplamak istediğinizi varsayalım.Tek yapmanız gereken, *D'ye* büyük bir ikili sayıymış gibi davranmak (sonuçta sadece bir bit dizisidir) ve üzerinde anlaşmaya varılan bir değere (*k) bölmektir*. Bu bölümün geri kalanı ÇHS'nin değeridir. Görünüşe göre, bu ikili modulo işlemi oldukça verimli bir şekilde uygulayabilir ve bu nedenle CRC'nin ağdaki popülaritesi de uygulanabilir. Daha fazla ayrıntı için başka bir yere bakın [M13].

Kullanılan yöntem ne olursa olsun, per-fect sağlama toplamı olmadığı açık olmalıdır: aynı olmayan içeriğe sahip iki veri bloğunun aynı sağlama toplamlarına sahip olması mümkündür,buna **çarpışma(collesion)** denir.Bu gerçek sezgisel olmalıdır: sonuçta, bir sağlama toplamının hesaplanması büyük bir şey (örneğin, 4KB) alıyor ve çok daha küçük bir özet üretiyor (örneğin, 4 veya 8 bayt). İyi bir sağlama toplamı işlevi seçerken, hesaplaması kolay kalırken çarpışma olasılığını en aza indiren bir işlev bulmaya çalışıyoruz .

Sağlama Toplamı Düzeni (Checksum Layout)

Artık bir sağlama toplamının nasıl hesaplanacağı hakkında biraz bilgi sahibi olduğunuza göre, bir sonraki adımda sağlama toplamlarının bir depolama sisteminde nasıl kullanılacağını analiz edelim. Ele almamız gereken ilk soru, sağlama toplamının düzenidir, yani sağlama toplamları diskte nasıl depolanmalıdır ?

En temel yaklaşım, her disk sektörü (veya bloğu) ile bir sağlama toplamı depolar. Bir veri bloğu D verildiğinde, bu veri üzerindeki sağlama toplamını *C*(*D*) olarak adlandıralım. Böylece, sağlama toplamları olmadan, disk düzeni şöyle görünür:



Sağlama toplamlarıyla, düzen her blok için tek bir sağlama toplamı ekler:

Sağlama toplamları genellikle küçük olduğundan (örneğin, 8 bayt) ve diskler yalnızca sektör boyutundaki parçalar (512 bayt) veya bunların katları halinde yazabildiğinden, ortaya çıkan bir sorun, yukarıdaki düzenin nasıl elde edileceğidir. Sürücü üreticileri tarafından kullanılan bir çözüm , sürücüyü 520 baytlık sektörlerle biçimlendirmektir; sağlama toplamını depolamak için sektör başına fazladan 8 bayt kullanılabilir.

Böyle bir işlevselliğe sahip olmayan disklerde, dosya sistemi 512 baytlık bloklar halinde paketlenmiş sağlama toplamlarını depolamanın bir yolunu bulmalıdır. Böyle bir olasılık aşağıdaki gibidir:

Bu şemada, n sağlama toplamı bir sektörde birlikte depolanır, bunu n veri bloğu izler, ardından sonraki *n* blok için başka bir sağlama toplamı sektörü izler ve benzeri. Bu yaklaşım tüm disklerde çalışma avantajına sahiptir, ancak daha az verimli olabilir; örneğin, dosya sistemi D 1 bloğunun üzerine yazmak istiyorsa, C (D 1) içeren sağlama toplamı sektöründe okumalı, içindeki *C* (*D* 1) 'i güncellemeli ve ardından sağlama toplamı sektörünü ve yeni veri bloğu *D* 1'i yazmalıdır. (böylece, bir okuma ve iki yazma). Önceki yaklaşım (sektör başına bir sağlama toplamı) yalnızca tek bir yazma işlemi gerçekleştirir .

45.4 Sağlama toplamlarını kullanma

Bir sağlama toplamı düzenine karar verildiğinde , artık sağlama toplamlarının nasıl *kullanılacağını* fiilen anlamaya devam edebiliriz. Bir D bloğunu okurken, istemci (yani, dosya sistemi veya depolama denetleyicisi) sağlama toplamını da $C_s(D)$ diskinden okur, buna **depolanan sağlama toplamı(stored checksum)** (dolayısıyla alt simge) adını veririz. C_s). İstemci daha sonra sağlama toplamını, **hesaplanan sağlama toplamı(computed checksum)** $C_c(D)$ olarak adlandırdığımız, alınan D bloğu üzerinden hesaplar. Bu noktada, istemci depolanan ve hesaplanan sağlama toplamlarını karşılaştırır; eğer eşitlerse (yani, $C_s(D) == C_c(D)$, veriler büyük olasılıkla bozulmamıştır ve bu nedenle kullanıcıya güvenli bir şekilde iade edilebilir. Eğer *eşleşmiyorlarsa* (yani, $C_s(D) != C_c(D)$), bu, verilerin depolandığı zamandan bu yana değiştiğini gösterir (depolanan sağlama toplamı o sırada verilerin değerini yansıttığından). Bu durumda, sağlama toplamımızın tespit etmemize yardımcı olduğu bir bozulmamız var.

Bir bozulma göz önüne alındığında, doğal soru şu ki, bu konuda ne yapmalıyız? Depolama sisteminde yedek bir kopya varsa, cevap kolaydır: bunun yerine kullanmayı deneyin. Depolama sisteminde böyle bir kopya yoksa, olası cevap şudur:

tıklayarak bir hata döndürebilirsiniz. Her iki durumda da, bozulma tespitinin sihirli bir mermi olmadığını anlayın; Bozulmamış verileri elde etmenin başka bir yolu yoksa, şansınız kalmaz.

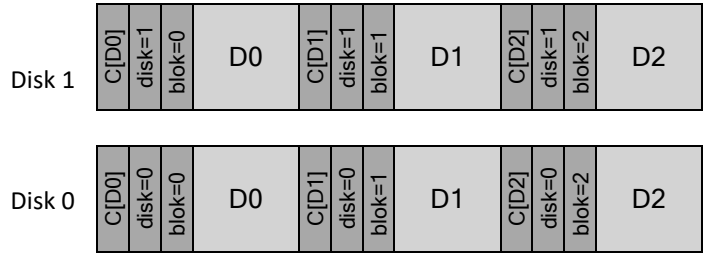
45.5 Yeni Bir Sorun: Yanlış Yönlendirilmiş Yazmalar

Yukarıda açıklanan temel şema, bozuk blokların genel durumunda iyi çalışır . Bununla birlikte, modern disklerin farklı çözümler gerektiren birkaç olağandışı arıza modu vardır.

İlgilenilen ilk başarısızlık moduna **yanlış yönlendirilmiş yazma(misdirected write)** denir. Bu *yanlış* konum dışında, verileri diske doğru şekilde yazan disk ve RAID denetleyicilerinde ortaya çıkar. Tek diskli bir sistemde, bu, diskin D x bloğunu x'e (istenildiği gibi) hitap etmek için değil, y'ye hitap etmek için (böylece D y'yi "bozmak") yazdığı anlamına gelir; ek olarak, çok diskli bir disk içinde. sistem, denetleyici ayrıca *D* i, x disk i'nin *x'ini* değil, başka bir disk j'ye adreslemek için yazabilir. Dolayısıyla sorumuz şu:

CRUX: YANLIŞ YÖNLENDİRİLMİŞ YAZIMLAR NASIL İŞLENİR?
Bir depolama Sistemleri veya disk denetleyicisi yanlış yönlendirilmiş yazmaları nasıl algılamalıdır ? Sağlama toplamından hangi ek özellikler gereklidir?

Cevap, şaşırtıcı olmayan bir şekilde, basittir:her bir sağlma toplamına biraz daha fazla bilgi ekleyin. Bu durumda, **fiziksel tanımlayıcı (physical identifier)** (**fiziksel kimlik**) eklemek oldukça faydalıdır. Örneğin, saklanan bilgiler şimdi sağlama toplamı C(D)'yi ve bloğun hem disk hem de sektör numaralarını içeriyorsa, istemcinin belirli bir yerel ayarda doğru bilginin bulunup bulunmadığını belirlemesi kolaydır. Spesifik olarak, müşteri disk 10'daki (D10,4) blok 4'ü okuyorsa, saklanan bilgiler aşağıda gösterildiği gibi o disk numarasını ve sektör ofsetini içermelidir. Bilgiler eşleşmezse, yanlış yönlendirilmiş bir yazma gerçekleşmiş ve artık bir bozulma algılanmıştır. İşte bu eklenen bilgilerin iki diskli bir sistemde nasıl görüneceğine dair bir örnek. Sağlama toplamları genellikle küçük (ör. 8 bayt) ve bloklar çok daha büyük (ör. 4 KB veya daha büyük) olduğundan, bu rakamın kendinden önceki diğerleri gibi ölçekli olmadığına dikkat edin:



Disk üzerindeki formattan, diskte artık oldukça fazla yedeklilik olduğunu görebilirsiniz: her blok için, disk numarası her blok içinde tekrarlanır ve söz konusu bloğun ofseti de bloğun yanında tutulur . Bununla birlikte, gereksiz bilgilerin varlığı sürpriz olmamalıdır; yedeklilik hata tespiti (bu durumda) ve kurtarmanın (diğerlerinde) anahtarıdır. Mükemmel disklerle kesinlikle gerekli olmasa da, biraz ekstra bilgi, ortaya çıkmaları durumunda sorunlu konumların tespit edilmesine yardımcı olmak için uzun bir yol kat edebilir .

45.6 Son Bir Sorun: Kayıp Yazma İşlemleri

Ne yazık ki, yanlış yönlendirilmiş yazılar ele alacağımız son sorun değildir. Özellikle, bazı modern depolama aygıtlarında, aygıt katman başına yukarı doğru bir yazmanın tamamlandığını bildirdiğinde ortaya çıkan, ancak aslında hiçbir zaman kalıcı olmayan **kayıp yazma(lost write)** olarak bilinen bir sorun da vardır; bu nedenle, geriye kalan güncellenmiş yeni içeriklerden ziyade bloğun eski içeriğidir.

Buradaki bariz soru şudur: Yukarıdan sağlama toplama stratejilerimizden herhangi biri (örneğin, temel sağlama toplamları veya fiziksel kimlik) kayıp yazmaları tespit etmeye yardımcı olur mu? Ne yazık ki, cevap hayır: eski blok muhtemelen eşleşen bir sağlama toplamına sahiptir ve yukarıda kullanılan fiziksel kimlik (disk numarası ve blok ofseti) de doğru olacaktır. Dolayısıyla son sorunumuz:

CRUX: KAYIP YAZILAR NASIL İŞLENİR?
Bir depolama sistemleri veya disk denetleyicisi kayıp yazmaları nasıl algılamalıdır? Sağlama toplamından hangi ek özellikler gereklidir?

[K+08] öğesine yardımcı olabilecek bir dizi olası çözüm vardır. Klasik bir yaklaşım [BS04], **yazma doğrulaması(write verify)** veya **yazdıktan sonra okuma(read-after-write)** gerçekleştirmektir; Bir sistem, bir yazma işleminden hemen sonra verileri tekrar okuyarak, verilerin gerçekten disk yüzeyine ulaşmasını sağlayabilir. Ancak, bu yaklaşım oldukça yavaştır ve bir yazmayı tamamlamak için gereken G/Ç sayısını iki katına çıkarır.

Bazı sistemler, kayıp yazmaları algılamak için sistemin başka bir yerine bir sağlama toplamı ekler. Örneğin, Sun'ın **Zettabyte Dosya Sistemi(Zettabyte File System) (ZFS),** her dosya sisteminde bir sağlama toplamı ve bir dosyaya dahil edilen her blok için dolaylı blok içerir. Bu nedenle, bir veri bloğuna yazma işlemi kaybolsa bile, inode içindeki sağlama toplamı eski verilerle eşleşmeyecektir. Sadece hem inode hem de verilere yazılanlar aynı anda kaybolursa, böyle bir şema başarısız olur, bu pek olası olmayan (ama ne yazık ki, mümkün!)bir durumdur.

45.7 Temizleme

Tüm bu tartışmalar göz önüne alındığında , merak ediyor olabilirsiniz: Bu sağlama toplamları gerçekte ne zaman kontrol edilir?Tabii ki, uygulamalar tarafından verilere erişildiğinde bir miktar kontrol yapılır,

Ancak çoğu veriye nadiren erişilir ve bu nedenle işaretlenmemiş olarak kalır. Kontrol edilmeyen veriler, güvenilir bir depolama sistemi için probatiktir, çünkü bit çürümesi sonunda belirli bir veri parçasının tüm kopyalarını etkileyebilir .

Bu sorunu çözmek için, birçok sistem çeşitli biçimlerde **disk temizleyici (disk scrubbing)** kullanır [K + 08]. Disk sistemi, sistemin *her* bloğunu periyodik olarak okuyarak ve sağlama toplamlarının hala geçerli olup olmadığını kontrol ederek, belirli bir veri öğesinin tüm kopyalarının kopma olasılığını azaltabilir . Tipik sistemler taramaları gecelik veya haftalık olarak planlar.

45.8 Sağlama Toplamının Genel Giderleri

Bitirmeden önce, şimdi veri koruması için sağlama toplamlarını kullanmanın bazı ek yüklerini tartışacağız.Bilgisayar sistemlerinde yaygın olduğu gibi iki farklı ek yük türü vardır : alan ve zaman.

Alan giderleri iki şekilde gelir. Birincisi, diskin (veya başka bir depolama ortamının) kendisindedir; depolanan her sağlama toplamı diskte yer kaplar ve artık kullanıcı verileri için kullanılamaz. Tipik bir oran, disk üzerinde %0,19 ek yük için 4 KB veri bloğu başına 8 baytlık bir sağlama toplamı olabilir .

İkinci tür alan yükü, sistemin belleğinde gelir. Verilere erişirken, artık verilerin yanı sıra sağlama toplamları için de bellekte yer olmalıdır. Bununla birlikte, sistem sağlama toplamını kontrol eder ve daha sonra bir kez atarsa, bu ek yük kısa ömürlüdür ve çok fazla endişe verici değildir. Yalnızca sağlama toplamları bellekte tutulursa (bellek bozulmasına karşı ek bir koruma düzeyi için [Z+13]) bu küçük ek yük gözlemlenebilir.

Alan genel giderleri küçük olsa da, sağlama toplamanın neden olduğu zaman ek yükleri oldukça belirgin olabilir. En azından, CPU'nun hem veriler depolandığında (depolanan sağlama toplamının değerini belirlemek için) hem de erişildiğinde (sağlama toplamını yeniden hesaplamak ve depolanan sağlama toplamıyla karşılaştırmak için) her blok üzerindeki sağlama toplamını hesaplaması gerekir. Sağlama toplamları (ağ yığınları dahil) kullanan birçok sistem tarafından kullanılan CPU ek yüklerini azaltmanın bir avantajı , veri kopyalama ve sağlama toplamını tek bir kolaylaştırılmış etkinlikte birleştirmektir; kopyaya her şekilde ihtiyaç duyulduğundan (örneğin, verileri çekirdek sayfası önbelleğinden bir kullanıcı arabelleğine kopyalamak için), birleşik kopyalama / sağlama toplamı oldukça etkili olabilir.

CPU ek yüklerinin ötesinde, bazı sağlama toplama şemaları, özellikle sağlama toplamları verilerden ayrı olarak depolandığında (bu nedenle bunlara erişmek için fazladan G /Ç gerektirdiğinde) ekstra G/Ç ek yüklerine neden olabilir. ve arka plan fırçalama için gereken ekstra G/Ç için. İlki tasarım gereği azaltılabilir; ikincisi ayarlanabilir ve bu nedenle etkisi, belki de böyle bir fırçalama faaliyeti gerçekleştiğinde kontrol edilerek sınırlandırılabilir. Gecenin yarısı, üretken işçilerin çoğunun (hepsi değil!) yatağa gittiği zaman, bu tür fırçalama faaliyetlerini gerçekleştirmek ve depolama sisteminin sağlamlığını artırmak için iyi bir zaman olabilir .

45.9 Özet

Modern depolama sistemlerinde veri korumasını tartıştık, sağlama toplamı uygulamasına ve kullanımına odaklandık. Farklı sağlama toplamları farklı arıza türlerine karşı koruma sağlar; depolama cihazları geliştikçe, şüphesiz yeni arıza modları ortaya çıkacaktır. Belki de böyle bir değişiklik, yeniden arama topluluğunu ve endüstriyi bu temel yaklaşımlardan bazılarını yeniden gözden geçirmeye veya tamamen yeni yaklaşımlar icat etmeye zorlayacaktır . Bunu zaman gösterecek. Ya da göstermeyecek .Zaman bu şekilde eğlencelidir.

References

[B+07] "An Analysis of Latent Sector Errors in Disk Drives" by L. Bairavasundaram, G. Good- son, S. Pasupathy, J. Schindler. SIGMETRICS '07, San Diego, CA. The first paper to study latent sector errors in detail. The paper also won the Kenneth C. Sevcik Outstanding Student Paper award, named after a brilliant researcher and wonderful guy who passed away too soon. To show the OSTEP authors it was possible to move from the U.S. to Canada, Ken once sang us the Canadian national anthem, standing up in the middle of a restaurant to do so. We chose the U.S., but got this memory.

[B+08] "An Analysis of Data Corruption in the Storage Stack" by Lakshmi N. Bairavasun- daram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci- Dusseau. FAST '08, San Jose, CA, February 2008. The first paper to truly study disk corruption in great detail, focusing on how often such corruption occurs over three years for over 1.5 million drives.

[BS04] "Commercial Fault Tolerance: A Tale of Two Systems" by Wendy Bartlett, Lisa Spainhower. IEEE Transactions on Dependable and Secure Computing, Vol. 1:1, January 2004. This classic in building fault tolerant systems is an excellent overview of the state of the art from both IBM and Tandem. Another must read for those interested in the area.

[C+04] "Row-Diagonal Parity for Double Disk Failure Correction" by P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar. FAST '04, San Jose, CA, February 2004. An early paper on how extra redundancy helps to solve the combined full-disk-failure/partial-disk-failure problem. Also a nice example of how to mix more theoretical work with practical.

[F04] "Checksums and Error Control" by Peter M. Fenwick. Copy available online here: http://www.ostep.org/Citations/checksums-03.pdf. A great simple tutorial on check- sums, available to you for the amazing cost of free.[F82] "An Arithmetic Checksum for Serial Transmissions" by John G. Fletcher. IEEE Trans- actions on Communication, Vol. 30:1, January 1982. Fletcher's original work on his eponymous checksum. He didn't call it the Fletcher checksum, rather he just didn't call it anything; later, others named it after him. So don't blame old Fletch for this seeming act of braggadocio. This anecdote might remind you of Rubik; Rubik never called it "Rubik's cube"; rather, he just called it "my cube."

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Spring '94. The pioneering paper that describes the ideas and product at the heart of NetApp's core. Based on this system, NetApp has grown into a multi-billion dollar storage company. To learn more about NetApp, read Hitz's autobiography "How to Castrate a Bull" (which is the actual title, no joking). And you thought you could avoid bull castration by going into CS.

[K+08] "Parity Lost and Parity Regained" by Andrew Krioukov, Lakshmi N. Bairavasun- daram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '08, San Jose, CA, February 2008. This work explores how different checksum schemes work (or don't work) in protecting data. We reveal a number of interesting flaws in current protection strategies.[M13] "Cyclic Redundancy Checks" by unknown. Available: http://www.mathpages.com/ home/kmath458.htm. A super clear and concise description of CRCs. The internet is full of infor- mation, as it turns out.[P+05] "IRON File Systems" by V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi,A. Arpaci-Dusseau, R. Arpaci-Dusseau. SOSP '05, Brighton, England. Our paper on how disks have partial failure modes, and a detailed study of how modern file systems react to such failures. As it turns out, rather poorly! We found numerous bugs, design flaws, and other oddities in this work. Some of this has fed back into the Linux community, thus improving file system reliability. You're welcome![RO91] "Design and Implementation of the Log- structured File System" by Mendel Rosen- blum and John Ousterhout. SOSP '91, Pacific Grove, CA, October 1991. So cool we cite it again.[S90] "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial" by Fred B. Schneider. ACM Surveys, Vol. 22, No. 4, December 1990. How to build fault tolerant services. A must read for those building distributed systems.[Z+13] "Zettabyte Reliability with Flexible End-to-end Data Integrity" by Y. Zhang, D. Myers,A. Arpaci-Dusseau, R. Arpaci-Dusseau. MSST '13, Long Beach, California, May 2013. How to add data protection to the page cache of a system. Out of space, otherwise we would write something...

Ödev(Simülasyon)

Bu ödevde, çeşitli sağlama toplamlarını araştırmak için checksum.py kullanacaksınız.

Soru

1. İlk önce checksum.py'yi hiçbir argüman olmadan çalıştırın. Ek, XOR tabanlı ve Fletcher sağlama toplamlarını hesaplayın. Cevaplarınızı kontrol etmek için -c' yi kullanın.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data

Decimal:          216        194        107         66
Hex:             0xd8       0xc2       0x6b       0x42
Bin:       0b11011000 0b11000010 0b01101011 0b01000010

Add:      ?
Xor:      ?
Fletcher: ?
```

2. Şimdi aynısını yapın, ancak çekirdeği (-s) farklı değerlere değiştirin.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -s 2

OPTIONS seed 2
OPTIONS data_size 4
OPTIONS data

Decimal:          244        242         14         21
Hex:             0xf4       0xf2       0x0e       0x15
Bin:       0b11110100 0b11110010 0b00001110 0b00010101

Add:      ?
Xor:      ?
Fletcher: ?
```

3. Bazen toplam ve XOR tabanlı sağlama toplamları aynı sağlama toplamını üretir (ör. veri değerinin tümü sıfırsa). Yalnızca sıfır içermeyen ve toplam ve XOR tabanlı sağlama toplamının aynı değere sahip olmasına yol açan 4 baytlık bir veri değerini (-D bayrağını kullanarak, örneğin -D a,b,c,d kullanarak) iletebilir misiniz? Genel olarak, bu ne zaman meydana gelir? Doğru olup olmadığınızı -c bayrağıyla kontrol edin.

   Toplam ve XOR tabanlı sağlama toplamları, yalnızca sağlama toplamı eşit sayıda sıfır ve bir içeriyorsa aynı sağlama toplamını üretecektir. Bunun nedeni, toplam sağlama toplamının basitçe veri bitlerinin toplamı olması ve XOR tabanlı sağlama toplamının, tüm veri bitlerinde bit düzeyinde bir XOR işlemi gerçekleştirmenin sonucu olmasıdır. Eşit sayıda sıfır ve birin hem toplamı hem de XOR değeri sıfır olacağından, bu durum sağlama toplamı aynı olacakt

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 1,2,3,4 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,2,3,4

Decimal:           1          2          3          4
Hex:            0x01       0x02       0x03       0x04
Bin:       0b00000001 0b00000010 0b00000011 0b00000100

Add:               10       (0b00001010)
Xor:                4       (0b00000100)
Fletcher(a,b):   10, 20   (0b00001010,0b00010100)
```

4. Şimdi, toplam ve XOR için farklı sağlama toplamı değerleri üreteceğini bildiğiniz 4 baytlık bir değer girin. Genel olarak, bu ne zaman olur?

   Genel olarak, 4 baytlık bir değer, verilerdeki sıfır ve birlerin sayısı eşit değilse, toplama ve XOR tabanlı sağlama toplamları için farklı sağlama toplamı değerleri üretecektir. Örneğin, 4 baytlık değer 0101 0101 0101 0101 ise, toplam sağlama toplamı sıfır olan tüm bitlerin toplamı olurken, XOR tabanlı sağlama toplamı, tüm bitlerde bit düzeyinde bir XOR işlemi gerçekleştirmenin sonucu olacaktır. yine sıfır olan bitler. Bu durumda, sağlama toplamı değerleri farklı olacaktır.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py 0101 0101 0101 0101

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data

Decimal:         216        194        107         66
Hex:            0xd8       0xc2       0x6b       0x42
Bin:       0b11011000 0b11000010 0b01101011 0b01000010

Add:      ?
Xor:      ?
Fletcher: ?
```

5. Sağlama toplamlarını iki kez hesaplamak için simülatörü kullanın (her biri farklı bir sayı grubu için bir kez). İki sayı dizisi farklı olmalıdır (örneğin, ilk kez -D a1,b1,c1,d1 ve ikinci kez -D a2,b2,c2,d2) ancak aynı toplam sağlama toplamını üretmelidir. Genel olarak, veri değerleri farklı olsa bile toplam sağlama toplamı ne zaman aynı olur? Özel cevabınızı -c bayrağıyla kontrol edin.

   Toplam sağlama toplamı, yalnızca kontrol edilen veri değerleri tam olarak aynı olduğunda aynı olacaktır. Başka bir deyişle, toplamı sağlama toplamı, bir veri kümesinin bütünlüğünü doğrulamanın bir yoludur ve bu veri kümeleri aynıysa, yalnızca iki farklı veri kümesi için aynı olacaktır.

Toplam sağlama toplamı, iki veri kümesindeki değerlerin toplamı aynı olduğunda aynı olacaktır. Örneğin, ilk veri kümesi [1, 2, 3, 4] değerlerini içeriyorsa ve ikinci küme [4, 2, -3, 7] değerlerini içeriyorsa, her iki küme için toplam sağlama toplamı 10 olacaktır.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 1,2,3,4 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,2,3,4

Decimal:            1          2          3          4
Hex:             0x01       0x02       0x03       0x04
Bin:       0b00000001 0b00000010 0b00000011 0b00000100

Add:             10       (0b00001010)
Xor:              4       (0b00000100)
Fletcher(a,b):   10, 20   (0b00001010,0b00010100)

p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 4,2,-3,7 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 4,2,-3,7

Decimal:            4          2         -3          7
Hex:             0x04       0x02       0x0-3       0x07
Bin:       0b00000100 0b00000010 0b00000b11 0b00000111

Add:             10       (0b00001010)
Xor:             -4       (0b0000b100)
Fletcher(a,b):   10, 23   (0b00001010,0b00010111)
```

6. Şimdi aynısını XOR sağlama toplamı için yapın.

İki dizinin XOR sağlama toplamı, diziler aynı sayıda öğeye sahip olduğunda ve her bir dizindeki öğeler aynı veya birbirinin XOR'u olduğunda aynı olacaktır.

Örneğin:

array1 = [1, 2, 3, 4]

array2 = [1, 2, 3, 4]

O zaman bu dizilerin XOR sağlama toplamı aynı olacaktır çünkü her dizideki değerler aynıdır. XOR sağlama toplamı, iki dizideki karşılık gelen her bir öğenin XOR'u (özel OR) alınarak hesaplanır. Örneğin, dizi1 ve dizi2'nin XOR sağlama toplamı şu şekilde hesaplanır:

checksum = array1[0] XOR array2[0] XOR array1[1] XOR array2[1] XOR array1[2] XOR array2[2] XOR array1[3] XOR array2[3]

Bu durumda, XOR sağlama toplamı 0 olacaktır çünkü iki dizideki karşılık gelen her eleman aynıdır ve herhangi bir sayının kendisi ile XOR'u 0'dır.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 1,2,3,4 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,2,3,4

Decimal:           1          2          3          4
Hex:            0x01       0x02       0x03       0x04
Bin:       0b00000001 0b00000010 0b00000011 0b00000100

Add:               10       (0b00001010)
Xor:                4       (0b00000100)
Fletcher(a,b):   10, 20   (0b00001010,0b00010100)

p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 1,2,3,4 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,2,3,4

Decimal:           1          2          3          4
Hex:            0x01       0x02       0x03       0x04
Bin:       0b00000001 0b00000010 0b00000011 0b00000100

Add:               10       (0b00001010)
Xor:                4       (0b00000100)
Fletcher(a,b):   10, 20   (0b00001010,0b00010100)
```

7. Şimdi belirli bir veri değerleri kümesine bakalım. Birincis: -D 1,2,3,4. Bu veriler için farklı sağlama toplamları (katkı maddesi, XOR, Fletcher) ne olacak? Şimdi bunu -D 4,3,2,1 üzerinden bu sağlama toplamlarını hesaplamakla karşılaştırın. Bu üç sağlama toplamı hakkında ne fark ettiniz? Fletcher diğer ikisiyle nasıl karşılaştırılır? Fletcher nasıl genel olarak basit toplamlı sağlama toplamı gibi bir şeyden "daha iyi"?

   Belirli bir veri değerleri kümesi için sağlama toplamı, verileri hata tespiti veya başka amaçlar için kullanılabilen sabit boyutlu bir değerde özetlemenin bir yoludur. Kullanılan belirli sağlama toplamı algoritması, verilerin nasıl özetlendiğini belirleyecektir.-D 1,2,3,4 veri değerleri durumunda, farklı sağlama toplamları aşağıdaki gibi olacaktır: toplam sağlama toplamı: Bu, tüm veri değerleri toplanarak hesaplanır, bu nedenle bu veriler için toplam sağlama toplamı -D 10 olur. XOR sağlama toplamı: Bu, tüm veri değerlerinde bit düzeyinde özel bir VEYA işlemi gerçekleştirilerek hesaplanır, dolayısıyla bu veriler için XOR sağlama toplamı -D 00 olur.

   Fletcher sağlama toplamı: Bu, verilerin bloklara bölünmesini, toplamın ve her blok için toplamların hesaplanmasını ve ardından nihai sağlama toplamını üretmek için bu değerlerin belirli bir şekilde birleştirilmesini içeren daha karmaşık bir sağlama toplamı algoritmasıdır. Bu veriler için Fletcher sağlama toplamının tam değeri, algoritmanın özel uygulamasına bağlı olacaktır. Bu sağlama toplamlarını -D 4,3,2,1 veri değerleri

üzerinden hesaplamak için karşılaştırırken, değerlerin farklı olduğunu görebiliriz. Bunun nedeni, sağlama toplamı algoritmalarının verileri farklı şekillerde özetlemesi ve verilerin belirli değerlerinin ve sırasının elde edilen sağlama toplamı değerini etkilemesidir.

Genel olarak, Fletcher sağlama toplamı, daha sağlam olduğu ve verilerdeki daha geniş bir hata aralığını algılayabildiği için basit bir toplam sağlama toplamından "daha iyi" kabul edilir. Bunun nedeni, Fletcher sağlama toplamının verileri bloklara ayırması ve her blok için are sağlama toplamları hesaplamasıdır, bu da hataların tespit edilmeme olasılığını azaltır. Ek olarak, Fletcher sağlama toplamı, sağlama toplamı değerlerini belirli hata türlerine karşı daha dayanıklı hale getiren belirli bir şekilde birleştirir.

```
p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 1,2,3,4 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 1,2,3,4

Decimal:            1          2          3          4
Hex:             0x01       0x02       0x03       0x04
Bin:       0b00000001 0b00000010 0b00000011 0b00000100

Add:               10       (0b00001010)
Xor:                4       (0b00000100)
Fletcher(a,b):   10, 20   (0b00001010,0b00010100)

p2075@p2075-HP-ProBook-430-G5 ~/Downloads/ostep-homework-master/file-integrity
 % ./checksum.py -D 4,3,2,1 -c

OPTIONS seed 0
OPTIONS data_size 4
OPTIONS data 4,3,2,1

Decimal:            4          3          2          1
Hex:             0x04       0x03       0x02       0x01
Bin:       0b00000100 0b00000011 0b00000010 0b00000001

Add:               10       (0b00001010)
Xor:                4       (0b00000100)
Fletcher(a,b):   10, 30   (0b00001010,0b00011110)
```

8. Hiçbir sağlama toplamı mükemmel değildir. Seçtiğiniz belirli bir girdi verildiğinde, aynı Fletcher sağlama toplamına götüren diğer veri değerlerini bulabilir misiniz? Genel olarak bu ne zaman olur? Basit bir veri dizisiyle başlayın (ör. -D 0,1,2,3) ve bu sayılardan birini değiştirip aynı Fletcher sağlama toplamı ile bitirip bitiremeyeceğinize bakın. Her zaman olduğu gibi, cevaplarınızı kontrol etmek için -c'yi kullanın.

Belirli bir girdi için aynı Fletcher sağlama toplamını üreten başka veri değerleri bulmak mümkündür. Ancak, bu tür veri değerlerinin bulunması özel bir yaklaşım gerektirecek ve Fletcher sağlama toplamı algoritmasının uygulanmasının ayrıntılarına bağlı olacaktır. Genel olarak, Fletcher sağlama toplamı, farklı veri kümeleri için aynı sağlama toplamı değerini üretme olasılığı düşük olacak şekilde tasarlanmıştır. Bunun nedeni, algoritmanın verileri bloklara ayırması, her blok için toplamları ve toplamları hesaplaması ve ardından nihai sağlama toplamını üretmek için bu değerleri belirli bir şekilde birleştirmesidir. Sonuç olarak, özel olarak oluşturulmadıkça iki farklı veri setinin aynı sağlama toplamı değerini üretmesi olası değildir. Belirli bir girdi için aynı Fletcher sağlama toplamını üreten veri değerlerini bulmanın bir yolu, genel veri değerlerini değiştirirken her blok için toplamları ve toplamları korumak için verileri belirli bir şekilde değiştirmek olacaktır. Ancak bu, Fletcher sağlama toplamı algoritmasının derinlemesine anlaşılmasını ve dikkatli deneyler yapılmasını gerektirecektir.

Ödev (Kod)

Ödevin bu bölümünde, çeşitli sağlama toplamlarını uygulamak için kendi kodunuzdan bazılarını yazacaksınız.

Soru

1. Bir giriş dosyası üzerinden XOR tabanlı bir sağlama toplamını hesaplayan ve sağlama toplamını çıktı olarak yazdıran kısa bir C programı (check-xor.c adı verilir) yazın. (Bir bayt) sağlama toplamını depolamak için 8 bitlik işarersiz karakter kullanın. Beklendiği gibi çalışıp çalışmadığını görmek için bazı test dosyaları oluşturun.

   Bu programı test etmek için, farklı içeriklere sahip bazı test dosyaları oluşturabilir ve beklenen sağlama toplamını üretip üretmediğini görmek için programı her biri üzerinde çalıştırabilirsiniz.

   ```
   $ echo "hello world" > test1.txt
   $ echo "goodbye world" > test2.txt
   $ ./check-xor test1.txt
   107
   $ ./check-xor test2.txt
   110
   ```

   Bu örnekte program, "test1.txt" girdi dosyası için 107'lik bir sağlama toplamı ve "test2.txt" girdi dosyası için 110'luk bir sağlama toplamı üretir. Bu değerler, giriş dosyalarındaki tek tek baytları XORlamanın sonucu oldukları için beklenen çıktıdır.

2. Şimdi bir giriş dosyası üzerinden Fletcher sağlama toplamını hesaplayan kısa bir C programı (check-fletcher.c olarak adlandırılır) yazın. Bir kez daha, çalışıp çalışmadığını görmek için programınızı test edin.
   Bu programı derlemek için aşağıdaki gibi bir komut kullanabilirsiniz:
   gcc -o check-fletcher check-fletcher.c
   Program derlendikten sonra aşağıdaki şekilde çalıştırabilirsiniz:
   ./check-fletcher <input_file>
   Bu, belirtilen giriş dosyasının Fletcher sağlama toplamını hesaplar ve onu onaltılık biçimde (iki sağlama toplamı baytının her biri için iki basamak kullanarak) stdout'a yazdırır.

3. Şimdi her ikisinin performansını karşılaştırın: biri diğerinden daha hızlı mı? Giriş dosyasının boyutu değiştikçe performans nasıl değişir? Programları zamanlamak için gettimeofday için dahili çağrıları

kullanın. Performansı önemsiyorsanız hangisini kullanmalısınız? Kontrol etme yeteneği hakkında?

Belirli uygulama ayrıntılarını ve girdi dosyasının boyutunu bilmeden hangi sağlama toplamı yönteminin daha hızlı olacağını söylemek zordur. Genel olarak, Fletcher sağlama toplamının, özellikle büyük girdi dosyaları için XOR tabanlı sağlama toplamından daha hızlı olduğu kabul edilir. Ancak, XOR tabanlı sağlama toplamının uygulanması genellikle daha kolaydır ve daha küçük girdi dosyaları için daha hızlı olabilir.

Performansı önemsiyorsanız, hangisinin daha hızlı olduğunu belirlemek için iki yöntemi kendi uygulamanız ve girdi verilerinizle test etmek en iyisidir. Hataları tespit etme yeteneğini önemsiyorsanız, Fletcher sağlama toplamının genellikle hataları tespit etmede XOR tabanlı sağlama toplamından daha etkili olduğu kabul edilir. Bununla birlikte, her iki yöntem de belirli hata türlerine karşı hassastır ve hiçbiri tam hata algılamayı garanti edemez.

4. 16 bit CRC hakkında bilgi edinin ve ardından uygulayın. Çalıştığından emin olmak için farklı girişlerden oluşan bir sayı üzerinde test edin. Basit XOR ve Fletcher ile karşılaştırıldığında performansı nasıl? Kontrol kabiliyetine ne dersiniz?

CRC (Cyclic Redundancy Check), veri iletimi veya depolanmasındaki hataları tespit etmek için yaygın olarak kullanılan bir sağlama toplamı türüdür. 16 bitlik bir CRC, bir veri girişinden bir sağlama toplamı oluşturmak için 16 bitlik bir polinom kullanır. Sağlama toplamı, veriler üzerinde bir dizi matematiksel işlem gerçekleştirilerek hesaplanır ve bu, iletim veya depolama sırasında meydana gelmiş olabilecek hataların tespit edilmesine yardımcı olur.16 bitlik bir CRC'yi test etmek için, farklı girdi verileri kullanarak bir dizi test durumu oluşturmanız ve ardından elde edilen sağlama toplamlarını beklenen sonuçlarla karşılaştırmanız gerekir. Bu, bir CRC hesap makinesi kullanılarak veya algoritmanın bir programlama dilinde uygulanmasıyla yapılabilir.

Performans açısından, 16 bitlik bir CRC, hataları tespit etmede genellikle basit bir XOR sağlama toplamından daha etkilidir. Benzer bir yaklaşım kullanan ancak farklı matematiksel işlemler kullanan Fletcher sağlama toplamından da daha etkilidir.16-bit CRC'nin bir avantajı, algılayabildiği hata türleri üzerinde bir dereceye kadar kontrol sağlamasıdır. Hesaplamada kullanılan polinomu dikkatlice seçerek, farklı senaryolar ve hata kalıpları için CRC'yi optimize etmek mümkündür. Bu, belirli türdeki hataların meydana gelme olasılığının daha yüksek olduğu uygulamalarda faydalı olabilir.

5. Şimdi bir dosyanın her 4 KB'lık bloğu için tek baytlık bir sağlama toplamı hesaplayan ve sonuçları bir çıktı dosyasına (komut satırında belirtilen) kaydeden bir araç (create-csum.c) oluşturun. Bir dosyayı okuyan, her blok üzerindeki sağlama toplamlarını hesaplayan ve sonuçları başka bir dosyada depolanan sağlama toplamlarıyla karşılaştıran ilgili bir araç (check-csum.c) oluşturun. Bir sorun varsa, program dosyanın bozuk olduğunu yazdırmalıdır. Dosyayı manuel olarak bozarak programı test edin.

Bu kodu (create-csum.c) derlemek için şöyle bir komut kullanın:
gcc -o create-csum create-csum.c
Ardından, aracı şu şekilde çalıştırabilirsiniz:
./create-csum <input_file> <output_file>
Bu, girdi dosyasının her 4KB bloğu için tek baytlık bir sağlama toplamı hesaplar ve sonuçları çıktı dosyasına yazar. Çıktı dosyası, son bloğun sağlama toplamı için sonuna eklenen ek bir bayt ile giriş dosyasıyla aynı boyutta olacaktır.

Bu programı (check-csum.c) test etmek için bir giriş dosyası oluşturabilir, create-csum programını kullanarak blokları için sağlama toplamlarını hesaplayabilir ve ardından doğrulamak için check-csum programını çalıştırabilirsiniz.