

Flash-based SSDs

Onlarca yıllık sabit disk sürücüsü hakimiyetinden sonra, yeni bir kalıcı depolama cihazı, kısa bir süre önce dünyada önem kazanmıştır. Genel olarak **katı hal depolama (solid-state storage)** olarak adlandırılan bu tür aygıtlarda sabit sürücüler gibi mekanik veya hareketli parçalar yoktur; basitçe inşa edilmişlerdir, daha ziyade transistörler, bellek ve işlemciler gibi. Ancak, tipik rastgele erişim belleğinin (örneğin DRAM) aksine, **katı hal depolama aygıtı** (bir **SSD**) güç kaybına rağmen bilgileri korur ve bu nedenle verilerin sürekli olarak depolanması için ideal bir adaydır.

Odaklanacağımız teknoloji, 1980'lerde Fujio Masuoka tarafından oluşturulan **flash** (daha özel olarak **NAND tabanlı flash**) olarak bilinir [M+14]. Flash'ın bazı benzersiz özellikleri vardır. Örneğin, belirli bir parça üzerine yazmak için (mesela bir **flaş sayfası**), önce daha büyük bir parça silmeniz gerekir (bir **flaş bloğu** gibi), bu da oldukça pahalı olabilir. Ayrıca, bir sayfaya çok sık yazmak, sayfanın yıpranmasına neden olur. Bu iki özellik, flash tabanlı bir SSD'nin yapımını ilginç bir zorluk haline getirir:

CRUX: FLASH TABANLI SSD NASIL YAPILIR

Flash tabanlı bir SSD'yi nasıl oluşturabiliriz? Silme işleminin pahalı doğasıyla nasıl başa çıkabiliriz? Tekrar tekrar üzerine yazmanın cihazın yıpranmasına neden olduğu göz önüne alındığında, uzun süre dayanan bir cihazı nasıl oluşturabiliriz? Teknolojideki ilerleme yürüyüşü hiç duracak mı? Veya şaşırtmayı sonlandıracak mı?

44.1 Bir Bit Saklama

Flash çipler, bir veya daha fazla biti tek bir transistörde depolamak için tasarlanmıştır; transistör içinde hapsolan yük seviyesi bir ikili (binary) değere eşlenir. Tek seviyeli bir hücre (SLC) flaşında, transistör içinde yalnızca tek bir bit depolanır (örn. 1 veya 0); **Çok seviyeli hücre (MLC)** flaşıyla, iki bit farklı şarj seviyelerine kodlanır; örneğin değerler 00, 01, 10, 11 olsun, bu değerler düşük, biraz düşük, biraz yüksek ve yüksek seviyelerle temsil edilir. Hücre başına 3 biti kodlayabilen **üç seviyeli hücre (TLC)** flaşı bile vardır. Genel olarak, SLC çipleri daha yüksek performans sağlar ve daha pahalıdır.

İPUCU: TERMİNOLOJİYE DİKKAT EDİN

Daha önce birçok kez kullandığımız bazı terimlerin (bloklar, sayfalar) bir flash bağlamında, ancak eskisinden biraz farklı şekillerde kullanıldığını fark etmiş olabilirsiniz. Yeni terimler hayatınızı zorlaştırmak için yaratılmadı (gerçi tam da bunu yapıyor olabilirler), terminoloji kararlarının verildiği merkezi bir otorite olmadığı için ortaya çıkıyor. Sizin için bir blok olan, duruma bağlı olarak başka biri için bir sayfa olabilir ve bunun tersi de geçerlidir. İşiniz basit: her alandaki uygun terimleri bilmek ve bunları, disiplinde bilgili insanların neden bahsettiğini anlayabileceği şekilde kullanmak. Bu, tek çözümün basit ama bazen acı verici olduğu anlardan biridir: hafızanızı kullanın.

Elbette, bu tür bit düzeyinde depolamanın tam olarak nasıl çalıştığına dair cihaz fiziği düzeyinde pek çok ayrıntı var. Bu kitabın kapsamı dışında olsa da, onun hakkında daha fazlasını kendi başınıza okuyabilirsiniz [10].

44.2 Bitlerden Bankalara/Düzlemlere

Antik Yunanistan'da da dedikleri gibi, tek bir tane (veya birkaç tane) depolamak depolama sistemi oluşturmuyor. Bu nedenle, flash çipler çok sayıda hücreden oluşan bankalar veya düzlemler halinde organize edilir.

Bir bankaya iki farklı boyutta olabilen birimle erişilir: Genellikle 128 KB veya 256 KB boyutunda bloklar (bazen **silme blokları** olarak adlandırılır) ve birkaç KB (örn. 4KB) boyutunda olan **sayfalar**. Her bir bankada çok sayıda blok vardır; her blokta çok sayıda sayfa vardır. Flash'ı düşünürken, disklerde ve RAID'lerde ifade ettiğimiz bloklardan ve sanal bellekte referans verdiğimiz sayfalardan farklı olan bu yeni terminolojiyi hatırlamanız gerekir.

Şekil 44.1'de bloklar ve sayfalar içeren bir flaş düzlemi örneği gösterilmektedir; bu basit örnekte her biri dört sayfa içeren üç blok vardır. Bloklar ve sayfalar arasında neden ayrım yaptığımızı aşağıda göreceğiz; bu ayrımın okuma ve yazma gibi flaş işlemleri ve hatta cihazın genel performansı için kritik olduğu ortaya çıktı. Öğreneceğiniz en önemli (ve tuhaf) şey, blok içindeki bir sayfaya yazmak için önce tüm bloğu silmeniz gerektiğidir; bu karmaşık ayrıntı, flash tabanlı bir SSD oluşturmanın ilginç ve değerli bir zorluk haline getiriyor ve bölümün ikinci yarısının konusunu oluşturuyor .

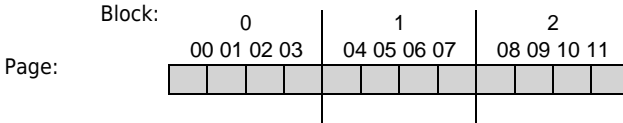


Figure 44.1: **A Simple Flash Chip: Pages Within Blocks**
(Şekil 44.1 : **Basit Bir Flash Yongası: Bloklar İçindeki Sayfalar**)

44.3 Temel Flash İşlemleri

Bu flash organizasyonu göz önüne alındığında, bir flash çipin desteklediği üç düşük seviyeli işlem vardır. **Oku (read)** komutu flaştan bir sayfa okumak için kullanılır; **sil (erase)** ve **program (program)** yazmak için art arda kullanılır. Detayları:

- **Oku (Read) (sayfa):** Flash çipin bir istemcisi, okuma komutunu ve aygıtta sayfa numarasını belirterek herhangi bir sayfayı (örneğin, 2KB veya 4KB) okuyabilir. Bu işlem tipik olarak çok hızlıdır, aygıttaki konumdan bağımsız olarak ve önceki (az veya çok) istegin konumundan bağımsız (bir diskin aksine) olarak 10 mikrosaniye kadar sürer. Herhangi bir yere aynı hızla erişebilmek, cihazın **rastgele erişimli (random access)** bir cihaz olduğu anlamına gelir.
- **Sil (Erase) (blok):** Flash içindeki bir *sayfa*'ya yazmadan önce, aygıtın doğası gereği önce sayfanın içinde bulunduğu *bloğun* tamamını **sil işlemine (erase)** tabi tutmanız gerekir. Sil işlemi, önemli olarak, (her biti 1 değerine ayarlayarak) bloğun içeriğini yok eder; Bu nedenle, silme işlemi gerçekleştirilmeden önce blokta önemsedığınız verilerin başka bir yere (belleğe veya belki başka bir flash bloğuna) kopyalandığından emin olmalısınız. Silme komutu oldukça pahalıdır ve tamamlanması birkaç milisaniye sürer. İşlem tamamlandıktan sonra tüm blok sıfırlanır ve her sayfa programlanmaya hazır olur.
- **Program (sayfa):** Bir blok silindikten sonra, program komutu bir sayfadaki 1'lerin bazılarını 0'lara değiştirmek ve bir sayfanın istenen içeriğini flash'a yazmak için kullanılabilir. Bir sayfayı programlamak, bir bloğu silmekten daha ucuzdur ancak bir sayfayı okumaktan daha maliyetlidir, genellikle modern flash çiplerde yaklaşık 100 mikrosaniye sürer.

Flaş çipleri hakkında düşünmenin bir yolu, her sayfanın kendisiyle ilişkili bir duruma sahip olmasıdır. Sayfalar **GEÇERSİZ** durumda başlar. Bir sayfanın içinde bulunduğu bloğu silerek, sayfanın durumunu (ve bu bloktaki tüm sayfaları) **SİLİNMİŞ** olarak ayarlarsanız blok içindeki her sayfanın içeriği sıfırlanır, ancak (daha da önemli) bunları programlanabilir hale getirir. Bir sayfayı programladığınızda, durumu **GEÇERLİ** olarak değişir, yani içeriği ayarlanmış ve okunabilir. Okumalar bu durumları etkilemez (ancak yalnızca programlanmış sayfalardan okumanız gerekir). Bir sayfa programlandıktan sonra içeriğini değiştirmenin tek yolu, sayfanın içinde bulunduğu bloğun tamamını silmektir. Aşağıda, 4 sayfalık bir blok içinde çeşitli silme ve program işlemlerinden sonra durum geçişlerine bir örnek verilmiştir:

	iiii	<i>Başlangıç: bloktaki sayfalar geçersiz (i)</i>
Erase()	→ EEEE	<i>Silinmiş (E) olarak ayarlanan bloktaki sayfaların durumu</i>
Program(0)	→ VEEE	<i>Program sayfası 0; durum geçerli (V) olarak ayarlandı</i>
Program(0)	→ error	<i>Sayfa programlandıktan sonra tekrar programlanamıyor</i>
Program(1)	→ VVEE	<i>Program sayfası 1</i>
Erase()	→ EEEE	<i>İçerik silindi; tüm sayfalar programlanabilir</i>

Ayrıntılı bir örnek

Yazma süreci (yani slime (erasing) ve programlama (programming)) çok sıra dışı olduğundan, mantıklı olduğundan emin olmak için ayrıntılı bir örnekten geçelim. Bu örnekte, 4 sayfalık bir blok içinde aşağıdaki dört tane 8 bitlik sayfaya sahip olduğumuzu hayal edin (her ikisi de gerçekçi olmayan küçük boyutlarda, ancak bu örnekte kullanışlıdır); Her sayfa daha önce programlandığı gibi GEÇERLİ'dir (VALID).

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Şimdi 0. Sayfaya yazmayı ve yeni içeriklerle doldurmayı istiyoruz diyelim. Herhangi bir sayfayı yazmak için önce tüm bloğu silmeliyiz. Bunu yaptığımızı varsayalım ve bloğu bu durumda bırakalım:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

İyi haber! Şimdi devam edip sayfa 0'ı örneğin içerik 00000011 ile programlayabilir, sayfa 0'ın eski durumunun (içerik 00011000) üzerine yazabiliriz. Bunu yaptıktan sonra bloğumuz şöyle görünür:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

Ve şimdi kötü haber: 1., 2. ve 3. sayfaların önceki içerikleri gitti! Bu nedenle, bir blok içindeki herhangi bir sayfanın üzerine yazmadan önce, önemsedığımız verileri önce başka bir konuma (ör. Bellek veya flash'ın başka bir yerine) taşımamız gerekir. Silme işleminin yapısı, yakında öğreneceğimiz gibi flash tabanlı SSD'leri nasıl tasarladığımız üzerinde güçlü bir etkiye sahip olacaktır.

Özet

Özetlemek gerekirse, bir sayfayı okumak kolaydır: Sadece sayfayı okuyun. Flash çipler bunu oldukça iyi ve hızlı bir şekilde yapar; performans açısından, mekanik arama ve döndürme maliyetleri nedeniyle yavaş olan modern disk sürücülerinin rastgele okuma performansını büyük ölçüde aşma potansiyeli sunar.

Bir sayfa yazmak daha karmaşık bir iştir; önce tüm blok silinmelidir (önce ilgilendiğimiz tüm verileri başka bir konuma taşımaya dikkat ederek) ve sonra istenen sayfa programlanmalıdır. Bu kadar pahalı olmakla kalmayıp, bu programlama/silme döngüsünün sık tekrarlanması flash çiplerin en büyük güvenilirlik sorununa yol açabilir: **Aşınma (wear out)**. Flash ile bir depolama sistemi tasarlarken, yazma performansı ve güvenilirliği merkezi bir odak noktadır. Yakında, modern SSD'lerin bu sorunlara nasıl saldırdığı ve bu sınırlamalara rağmen mükemmel performans ve güvenilirlik sağladığı hakkında daha fazla bilgi edineceğiz.

Devic e	Read (μ s)	Program (μ s)	Erase (μ s)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Şekil 44.2: **Raw Flash Performans Özellikleri**

44.4 Flash Performansı ve Güvenilirliği

Ham flash çiplerden bir depolama aygıtı oluşturmakla ilgilendiğimiz için, temel performans özelliklerini anlamak faydalı olacaktır. Şekil 44.2, popüler basında bulunan bazı sayıların kaba bir özetini sunmaktadır [V12]. Burada yazar, sırasıyla hücre başına 1, 2 ve 3 bit bilgi depolayan SLC, MLC ve TLC flaş boyunca okumaların, programların ve silmelerin temel işlem gecikmesini sunar.

Tablodan da görebileceğimiz gibi, okuma gecikmeleri oldukça iyidir ve tamamlanması yalnızca 10 mikrosaniye sürer. Program gecikmesi daha yüksek ve daha değişkendir, SLC için 200 mikrosaniye kadar düşüktür, ancak siz her hücreye daha fazla bit yükledikçe daha yüksektir; iyi bir yazma performansı elde etmek için paralel olarak birden fazla flaş çip kullanmanız gerekecektir. Son olarak, silme işlemleri oldukça pahalıdır ve tipik olarak birkaç milisaniye sürer. Bu maliyetle başa çıkmak, modern flaş depolama tasarımının merkezinde yer alır.

Şimdi de flash çiplerin güvenilirliğini düşünelim. Çok çeşitli nedenlerle (sürücü kafasının kayıt yüzeyiyle gerçekten temas ettiği korkunç ve fiziksel **kafa çarpması** dahil) arızalanabilen mekanik disklerin aksine, flash çipler saf silikondur ve bu açıdan endişelenilmesi gereken daha az güvenilirlik sorunu vardır. Birincil sorun aşınmasıdır; Şimdi flaş çiplerin güvenilirliğini ele alalım. Çok çeşitli nedenlerle (sürücü kafasının kayıt yüzeyiyle gerçekten temas ettiği korkunç ve oldukça fiziksel kafa çarpması dahil) arızalanabilen mekanik disklerin aksine, flaş çipleri saf silikondur ve bu anlamda daha az güvenilirlik sorunu vardır. endişelenmek Birincil endişe **aşınmadır**; bir flaş bloğu silinip programlandığında, yavaş yavaş biraz fazladan yük birikir. Zamanla, bu ekstra yük biriktikçe, 0 ile 1'i ayırt etmek giderek zorlaşır. İmkansız hale geldiği noktada, blok kullanılamaz hale gelir.

Bir bloğun tipik ömrü şu anda iyi bilinmemektedir. Üreticiler, MLC tabanlı blokları 10.000 P/E (Program/Silme) döngü ömrüne sahip olarak derecelendirir; yani, her blok arızalanmadan önce 10.000 kez silinebilir ve programlanabilir. SLC tabanlı yongalar, transistör başına yalnızca tek bir bit depoladıkları için, genellikle 100.000 P/E döngüsü olmak üzere daha uzun bir ömürle derecelendirilir. Bununla birlikte, son araştırmalar yaşam sürelerinin beklenenden çok daha uzun olduğunu göstermiştir [BD10].

Flaş çiplerdeki diğer bir güvenilirlik sorunu **bozulma/karışıklık (disturbance)** olarak bilinir. Bir flaş içinde belirli bir sayfaya erişirken, komşu sayfalarda bazı bitlerin ters çevrilmesi mümkündür; bu tür bit çevirmeleri, sırasıyla sayfanın okunmasına veya programlanmasına bağlı olarak **okuma kesintileri (read disturbs)** veya **program kesintileri (program disturbs)** olarak bilinir.

İPUCU: GERİYE DÖNÜK UYUMLULUĞUN ÖNEMİ

Geriye dönük uyumluluk, katmanlı sistemlerde her zaman bir endişe kaynağıdır. İki sistem arasında istikrarlı bir arayüz tanımlayarak, sürekli birlikte çalışabilirliği sağlarken arayüzün her iki tarafında yenilik sağlar. Bu tür bir yaklaşım birçok alanda oldukça başarılı olmuştur: işletim sistemleri, uygulamalar için nispeten kararlı API'lere sahiptir, diskler, dosya sistemlerine aynı blok tabanlı arabirimi sağlar ve IP ağ yığınındaki her katman, yukarıdaki katmana sabit, değişmeyen bir arabirim sağlar.

Bir nesilde tanımlanan arabirimler bir sonraki nesilde uygun olmayabileceği için, bu tür bir dayanıklılığın dezavantajı pek de şaşırtıcı değil. Bazı durumlarda, tüm sistemin yeniden tasarlanmasını en gereksiz şekilde düşünmek yararlı olabilir. Sun ZFS dosya sisteminde [B07] mükemmel bir örnek yer alıyor; dosya sistemleri ve RAID'in etkileşimini yeniden göz önünde bulundurarak ZFS'nin yaratıcıları daha etkili bir entegre tasarladılar (ve daha sonra gerçekleştirdiler).

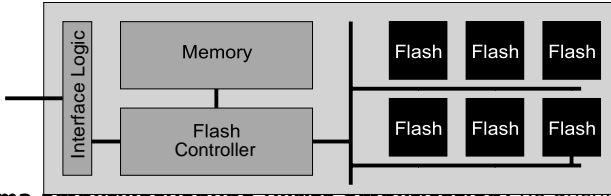
44.5 Raw (ham) Flash'tan Flash Tabanlı SSD'lere

Flash yongaları hakkındaki temel anlayışımız göz önüne alındığında, şimdi bir sonraki görevimizle karşı karşıyayız: temel bir flash yonga setini tipik bir depolama aygıtı gibi gören bir şeye nasıl dönüştüreceğiz. Standart depolama arabirimi, bir blok adresi verildiğinde 512 bayt (veya daha büyük) boyutunda blokların (sektörlerin) okunup yazılabileceği blok tabanlı basit bir arabirimdir. Flash tabanlı SSD'nin görevi, içindeki ham flash çiplerin üzerinde bu standart blok arabirimini sağlamaktır.

Dahili olarak, bir SSD belirli sayıda flash çipten oluşur (kalıcı depolama için). Bir SSD ayrıca bir miktar uçucu (yani kalıcı olmayan) bellek (ör. SRAM) içerir; bu tür bir bellek, verileri önbellege alma ve arabelleğe almanın yanı sıra aşağıda öğreneceğimiz tabloları eşleştirmek için kullanışlıdır. Son olarak, bir SSD, cihazın çalışmasını düzenlemek için kontrol mantığı içerir. (Bkz. Agrawal) Ayrıntılar için [A+08]; Şekil 44.3'te (sayfa 7) basitleştirilmiş bir blok diyagram görülmektedir.

Bu kontrol mantığının temel işlevlerinden biri, istemci okumalarını ve yazmalarını tatmin etmek ve bunları gerektiği gibi dahili flash işlemlerine dönüştürmektir. **Flash çeviri katmanı (Flash translation layer)** veya **FTL** tam olarak bu işlevi sağlar. FTL, mantıksal bloklardaki (cihaz arayüzünü oluşturan) okuma ve yazma isteklerini alır ve bunları temeldeki fiziksel bloklarda ve fiziksel sayfalarda (gerçek flash cihazını oluşturan) düşük seviyeli okuma, silme ve programlama komutlarına dönüştürür. FTL, mükemmel performans ve yüksek güvenilirlik sağlama hedefiyle bu görevi yerine getirmektedir.

Göreceğimiz gibi mükemmel performans, tekniklerin bir kombinasyonu ile gerçekleştirilebilir. Anahtarlardan biri, **paralel** olarak birden fazla flash çip kullanmak olacaktır; Bu tekniği daha fazla tartışmayacak olsak da, tüm modern SSD'lerin daha yüksek performans elde etmek için dahili olarak birden fazla çip kullandığını söylemekle yetinelim. Diğer bir performans hedefi, FTL tarafından flash çiplere verilen toplam yazma trafiğinin (bayt cinsinden) istemci tarafından SSD'ye verilen toplam yazma trafiğine (bayt cinsinden) bölünmesi olarak tanımlanan yazma amplifikasyonunu azaltmak olacaktır. Aşağıda göreceğimiz gibi, FTL oluşturmaya yönelik naif yaklaşımlar, yüksek



yazma ampirikasyonuna (write amplification) ve düşük performansa yol açacaktır.

Birkaç farklı yaklaşımın kombinasyonu ile yüksek güvenilirlik elde edilecektir. Yukarıda tartışıldığı gibi ana endişelerden biri **aşınmadır**. Tek bir blok çok sık silinir ve programlanırsa kullanılamaz hale gelir; sonuç olarak, FTL, aygıtın tüm bloklarının kabaca aynı anda aşınmasını sağlayarak, flash blokları boyunca yazma işlemlerini mümkün olduğu kadar eşit bir şekilde yaymaya çalışmalıdır; bunu yapmaya **aşınma seviyelendirme (wear leveling)** denir ve herhangi bir modern FTL'nin önemli bir parçasıdır.

Diğer bir güvenilirlik endişesi, program bozulmasıdır. Bu karışıklığı en aza indirmek için, FTL'ler genellikle silinmiş bir blok içindeki sayfaları düşük sayfadan yüksek sayfaya doğru programlayacaktır. Bu sıralı programlama yaklaşımı, karışıklığı en aza indirir ve yaygın olarak kullanılır.

44.6 FTL Organizasyonu: Kötü Bir Yaklaşım

Bir FTL'nin en basit organizasyonu, **doğrudan eşlenmiş (direct mapped)** olarak adlandırdığımız bir organizasyon olurdu. Bu yaklaşımda, mantıksal sayfa N'ye okuma, doğrudan fiziksel sayfa N'nin okunduğu bir sayfayla eşleştirilir. Mantıksal sayfa N'ye yazma daha karmaşıktır; FTL'nin ilk önce o sayfa N'nin içinde bulunduğu tüm bloku okuması gerekir; daha sonra bloku silmesi gerekir; son olarak, FTL eski sayfaları ve yenilerini programlar.

Muhtemelen tahmin edebileceğiniz gibi, doğrudan eşlemeli FTL'nin hem performans hem de güvenilirlik açısından birçok sorunu vardır. Performans sorunları her yazmada ortaya çıkar: aygıtın tüm bloğu okuması (maliyetli), silmesi (oldukça maliyetli) ve ardından programlaması (maliyetli) gerekir. Sonuç, ciddi yazma artışı (bir bloktaki sayfa sayısı ile orantılı) ve sonuç olarak, mekanik aramaları ve dönme gecikmeleri ile tipik sabit sürücülerden bile daha yavaş olan korkunç yazma performanslarıdır.

Daha da kötüsü, bu yaklaşımın güvenilirliği. Dosya sistemi meta verilerinin veya kullanıcı dosyası verilerinin üzerine tekrar tekrar yazılacak olursa, aynı blok silinir ve tekrar tekrar tekrar programlanabilir, hızla aşınır ve potansiyel olarak veri kaybı meydana gelebilir. Doğrudan eşlenmiş yaklaşım, istemci iş yüküne göre aşınma üzerinde çok fazla kontrol sağlar; iş yükü, yazma yükünü mantıksal bloklarına eşit olarak yaymazsa, popüler verileri içeren temel fiziksel bloklar hızla aşınır. Hem güvenilirlik hem de performans nedeniyle doğrudan eşlenmiş bir FTL kötü bir fikirdir.

44.7 Günlük (log) Yapılandırılmış FTL

Bu nedenlerden dolayı, günümüzde çoğu FTL hem depolama aygıtlarında (şimdi göreceğimiz gibi) hem de bunların üzerindeki dosya sistemlerinde (**günlük yapı dosya sistemleri** bölümünde göreceğimiz gibi) yararlı bir fikir olan **günlük yapılandırılmış (log structured)**. Mantıksal bloğa N yazıldığında, cihaz, yazmayı şu anda yazılmakta olan bloktan bir sonraki boş noktaya ekler; bu yazma stiline **günlük kaydı** diyoruz. N bloğunun sonraki okumalarına izin vermek için, cihaz bir **eşleşme tablosu** tutar (belleğinde ve bir şekilde cihazda kalıcıdır); bu tablo, sistemdeki her mantıksal bloğun fiziksel adresini saklar.

Temel günlük tabanlı yaklaşımın nasıl çalıştığını anladığımızdan emin olmak için bir örnek üzerinden gidelim. İstemciye göre cihaz, 512 baytlık sektörleri (veya sektör gruplarını) okuyup yazabildiği tipik bir disk gibi görünür. Basit olması için, istemcinin 4 KB boyutunda parçalar okuduğunu veya yazdığını varsayalım. Ayrıca, SSD'nin her biri dört adet 4 KB'lık sayfaya bölünmüş çok sayıda 16 KB boyutunda blok içerdiğini varsayalım; bu parametreler gerçekçi değildir (flash bloklar genellikle daha fazla sayfadan oluşur), ancak didaktik amaçlarımıza oldukça iyi hizmet edecektir.

İstemcinin aşağıdaki işlem sırasını verdiğini varsayalım:

- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2

Bu **mantıksal blok adresleri** (ör. 100), SSD istemcisi (ör. bir dosya sis-temi) tarafından bilgilerin nerede bulunduğunu hatırlamak için kullanılır.

Dahili olarak, cihazın bu blok yazma işlemlerini ham donanım tarafından desteklenen silme ve programlama işlemlerine dönüştürmesi ve SSD'nin **fiziksel sayfasının** verilerini depoladığı her bir mantıksal blok adresi için bir şekilde kaydetmesi gerekir. SSD'nin tüm bloklarının şu anda geçerli olmadığını ve herhangi bir sayfanın programlanabilmesi için silinmesi gerektiğini varsayalım. Burada, SSD'mizin ilk durumunu, tüm sayfaları GEÇERSİZ (i) olarak işaretlenmiş şekilde gösteriyoruz:

Block:	0	1	2
Page:	00 01 02 03	04 05 06 07	08 09 10 11
Content:			
State:	i i i i	i i i i	i i i i

İlk yazma SSD tarafından alındığında (mantıksal blok 100'e), FTL bunu dört fiziksel sayfa içeren 0 numaralı fiziksel bloğa yazmaya karar verir: 0, 1, 2 ve 3. Blok silinmediğinden, henüz yazamıyoruz; cihaz önce 0'ı engellemek için bir silme komutu vermelidir. Bunu yapmak aşağıdaki duruma yol açar:

Block:	0	1	2
Page:	00 01 02 03	04 05 06 07	08 09 10 11
Content:			
State:	E E E E	i i i i	i i i i

Blok 0 artık programlanmaya hazırdır. Çoğu SSD, sayfaları sırayla (yani düşükten yükseğe) yazacak ve **program bozulmasıyla (program disturbance)** ilgili güvenilirlik sorunlarını azaltacaktır. SSD daha sonra mantıksal blok 100'ün yazılmasını fiziksel sayfa 0'a yönlendirir:

Blok:	0	1	2
Sayfa:	00 01 02 03	04 05 06 07	08 09 10 11
İçerik:	a1		
Durum:	V E E E	i i i i	i i i i

Peki ya istemci mantıksal blok 100'ü okumak isterse? Nerede olduğunu nasıl bulabilir? SSD, mantıksal blok 100'e verilen bir okumayı fiziksel sayfa 0'ın okumasına dönüştürmelidir. Bu tür bir işlevselliğe uyum sağlamak için, FTL mantıksal blok 100'ü fiziksel sayfa 0'a yazdığında, bu gerçeği bir **bellek içi eşleme tablosuna (in-memory mapping table)** kaydeder. Bu eşleme tablosunun durumunu şemalarda da izleyeceğiz:

Tablo: 100 → 0

Bellek(Memory)

Blok:	0	1	2	
Sayfa:	00 01 02 03	04 05 06 07	08 09 10 11	Flash
İçerik:	a1			Chip
Durum:	V E E E	i i i i	i i i i	

Artık istemci SSD'ye yazdığında ne olduğunu görebilirsiniz. SSD, genellikle bir sonraki boş sayfayı seçerek yazma için bir konum bulur; daha sonra o sayfayı bloğun içeriğiyle programlar ve mantıksal-fiziksel eşlemeyi eşleme tablosuna kaydeder. Sonraki okumalar, istemci tarafından sunulan mantıksal blok adresini verileri okumak için gereken fiziksel sayfa numarasına çevirmek için tabloyu kullanır.

Şimdi örnek yazma akışımızdaki yazarların geri kalanını inceleyelim: 101, 2000 ve 2001. Bu blokları yazdıktan sonra, cihazın durumu:

Tablo: 100 → 0 101 → 1 2000 → 2 2001 → 3

Bellek

Blok:	0	1	2	
Sayfa:	00 01 02 03	04 05 06 07	08 09 10 11	Flash
İçerik:	a1 a2 b1 b2			Chip
Durum:	V V V V	i i i i	i i i i	

Günlük tabanlı yaklaşım, doğası gereği performansı artırır (yalnızca arada bir gerektiğinde siler ve doğrudan eşlemeli yaklaşımın maliyetli okuma-değiştirme-yazma işleminden tamamen kaçınılır) ve güvenilirliği büyük ölçüde artırır. FTL artık yazma işlemlerini tüm sayfalara yayabilir, **aşınma dengelemesi (wear leveling)** denilen işlemi gerçekleştirebilir ve cihazın kullanım ömrünü uzatabilir; aşınma dengelemesini aşağıda daha ayrıntılı olarak tartışacağız.

AYRINTI: FTL HARİTALAMA BİLGİSİ KALICILIĞI

Merak ediyor olabilirsiniz: Cihaz güç kaybederse ne olur? Bellekteki harita tablosu kaybolur mu? Açıkça, bu tür bilgiler gerçekten kaybedilemez, aksi takdirde cihaz bir kalıcı depolama cihazı olarak işlev yapamaz. Bir SSD'nin harita bilgisini geri yüklemeyi sağlayan bazı yöntemleri olmalıdır.

Yapılması gereken en basit şey, **bant dışı (out-of-band/OOB)** alan olarak adlandırılan her bir sayfayla bazı eşleştirme bilgilerini kaydetmektir. Cihaz güç kaybettiğinde ve yeniden başlatıldığında, OOB alanlarını tarayarak ve haritalama tablosunu hafıza muhafazasında yeniden yapılandırarak haritalama tablosunu yeniden oluşturması gerekir. Bu temel yaklaşımın sorunları vardır; tüm gerekli haritalama bilgilerini bulmak için büyük bir SSD'yi taramak yavaştır. Bu sınırlamanın üstesinden gelmek için, bazı üst uç aygıtlar kurtarma işlemini hızlandırmak için daha karmaşık **kayıt** ve **denetim** teknikleri kullanır; çökme tutarlılığı ve günlük yapıyı dosya sistemleri ile ilgili bölümleri okuyarak kayıt hakkında daha fazla bilgi edinin[AD14].

Maalesef, bu temel günlük yapılandırma yaklaşımının bazı dezavantajları var. İlki, lojik blokların üzerine yazılması, **çöp** olarak adlandırdığımız bir şeye yani sürücüdeki eski veri versiyonlarının ortaya çıkması ve yer kaplamasına neden olur. Cihaz, bu blokları bulmak ve gelecekteki yazılar için boş yer açmak için düzenli olarak **çöp toplama (garbage collection/GC)** işlemi yapmak zorundadır; aşırı çöp toplama, yazma artışını artırır ve performansı düşürür. İkincisi, bellekte harita tablosu maliyetinin yüksek olmasıdır; cihaz ne kadar büyükse, tablo için o kadar çok bellek gerekir. Şimdi her birini ayrıntılı olarak ele alacağız.

44.8 Çöp Toplama (Garbage Collection)

Bu gibi bir log yapılandırma yaklaşımının ilk maliyeti, çöp oluşturulması ve bu nedenle çöp toplama (yani ölü blok geri kazanımı) yapılması gerekir. Devam eden örneğimizi kullanarak bu konuyu anlamaya çalışalım. Mantıksal blok 100, 101, 2000 ve 2001'in cihaza yazıldığını hatırlayın.

Şimdi, blok 100 ve 101'in tekrar c1 ve c2 içeriğiyle yazıldığını varsayalım. Yazılan yazılar bir sonraki boş sayfalara (bu durumda fiziksel sayfalar 4 ve 5'e) yazılır ve harita tablosu bu doğrultuda güncellenir. Dikkat edin ki, bu programlama mümkün olabilmesi için cihazın önce blok 1'i silmesi gerekiyor.

Tablo:	100 → 4	101 → 5	2000 → 2	2001 → 3	Bellek
Blok:	0	1	2		
Sayfa:	00 01 02 03	04 05 06 07	08 09 10 11		
İçerik:	a1 a2 b1 b2	c1 c2			
Durum:	V V V V	V V E E	i i i i		

Flash
Chip

"Sorun şu anki durumda açıktır: fiziksel sayfalar 0 ve 1, GEÇERLİ olarak işaretli olmasına rağmen, içlerinde **çöp** bulunmaktadır, yani 100 ve 101 bloklarının eski sürümleri. Cihazın günlük yapılandırılmış doğası nedeniyle, üzerine yazma işlemleri çöp bloklar oluşturur, bu bloklar cihazın yeni yazma işlemleri için boş alan sağlaması için geri kazanılmalıdır.

Çöp bloklarının (açıkça **ölü bloklar (dead blocks)** olarak da adlandırılır) bulunması ve gelecekte kullanım için geri kazanılması işlemi, **çöp toplama** olarak adlandırılır ve modern bir SSD'nin önemli bir bileşenidir. Temel işlem basittir: bir blokta bir veya daha fazla çöp sayfası bulunan bir blok bulun, o bloktan canlı (çöp olmayan) sayfalar okunur, canlı sayfalar günlüğe yazılır ve (son olarak) blok yazma için kullanıma hazır hale getirilir.

Şimdi bir örnekle açıklayalım. Cihaz, bloğun 0. sırasındaki ölü sayfaları geri almak istediğini kararlaştırır. Blok 0'ın iki ölü bloğu (sayfa 0 ve 1) ve iki canlı bloğu (2000 ve 2001 numaralı bloğu içeren sayfa 2 ve 3) vardır. Bunu yapmak için cihaz:

- Blok 0'dan canlı verileri (sayfa 2 ve 3) okur
- Canlı verileri günlüğün sonuna yazar
- Blok 0'ı siler (daha sonra kullanım için boşaltır)

Garbage collector'ün çalışması için, her bloğun içinde bulunan verilerin canlı ya da ölü olduğunu belirlemeyi sağlayacak yeterli bilgiye sahip olması gerekir. Bu amaçla, her bloğun içinde bulunan sayfalardaki mantıksal bloğu belirten bilgilerin tutulması bir seçenektir. Cihaz bu eşleme tablosunu kullanarak, bloğun içindeki her sayfanın canlı veri içerip içermediğini belirleyebilir.

Örneğimizde (çöp toplama işlemi gerçekleşmeden önce), blok 0'ın içinde 100, 101, 2000 ve 2001 numaralı mantıksal bloklar bulunuyordu. Eşleme tablosunu (çöp toplama işleminden önce 100->4, 101->5, 2000->2, 2001->3 içeriyordu) kontrol ederek, cihaz kolayca her bir SSD bloğundaki sayfaların canlı bilgi içerip içermediğini belirleyebilir. Örneğin, 2000 ve 2001 hala eşleme tablosuna işaretlenmiştir; 100 ve 101 işaretlenmemiştir ve bu yüzden çöp toplama işlemi için adaydırlar.

Örneğimizde bu çöp toplama işlemi tamamlandığında cihazın durumu şu şekildedir:

Table:	100 → 4	101 → 5	2000 → 6	2001 → 7	Memory
Block:	0	1	2		
Page:	00 01 02 03	04 05 06 07	08 09 10 11		
Content:		c1 c2 b1 b2			
State:	E E E E	V V V V	i i i i		

Görüldüğü gibi, çöp toplama işlemi maliyetli olabilir ve canlı verilerin okunup yeniden yazılmasını gerektirir. Özelleştirme için en uygun aday, sadece ölü sayfalar içeren bir bloktur; bu durumda blok, pahalı veri geçişi olmadan hemen silinebilir ve yeni veriler için kullanılabilir.

AYRINTI: TRIM(KIRPMA) OLARAK BİLİNEREN YENİ BİR DEPOLAMA API'SI

Sabit diskleri düşündüğümüzde, genellikle sadece onları okuma ve yazma için en temel arayüzü düşünürüz: okuma ve yazma (genellikle yazmaların gerçekten kalıcı hale gelmesini sağlamak için bir tür **ön bellek temizleme (cache flush)** komutu da vardır ancak bazen bunu basitliği sağlamak için atlarız). Günlük yapı SSD'ler ve gerçekten de mantıksal-fiziksel bloklar arasındaki esnek ve değişen eşleştirme tutan herhangi bir cihaz için, **TRIM (KIRPMA)** işlemi olarak bilinen yeni bir arayüz kullanışlıdır.

TRIM işlemi, bir adres (ve muhtemelen bir uzunluk) alır ve cihaza adres (ve uzunluk) tarafından belirtilen bloğun/blokların silindiğini bildirir; böylece cihaz artık verilen adres aralığı ile ilgili herhangi bir bilgi takip etmek zorunda değildir. Standart bir sabit disk için, TRIM özellikle kullanışlı değildir, çünkü diskte blok adreslerinin belirli bir tabaka, parça ve sektörlerle sabit bir eşleştirme mevcuttur. Ancak bir günlük-yapısal SSD için, bir bloğun artık gerekli olmadığını bilmek çok yararlıdır, çünkü SSD daha sonra FTL'den bu bilgiyi kaldırabilir ve daha sonra boşaltma toplama sırasında fiziksel alanı geri alabilir.

Arayüz ve uygulamayı bazen ayrı varlıklar olarak düşüsek de, bu durumda uygulamanın arayüzü şekillendirdiğini görürüz. Karmaşık eşleştirmelere sahip olmak, hangi blokların artık gerekli olmadığını bilmek daha etkili bir uygulama için yararlıdır.

GC maliyetlerini azaltmak için bazı SSD'ler cihazın [A+08] **aşırı provizyonunu (overprovision)** sağlar; ekstra flaş kapasitesi ekleyerek, temizleme işlemi geciktirilebilir ve **arka plana (background)** itilebilir; bu da cihazın daha az meşgul olduğu bir zamanda yapılabilir. Daha fazla kapasite eklemek de iç bant genişliğini artırır ve bu da temizleme için kullanılabilir ve dolayısıyla istemcinin algılanan bant genişliğini zedeleyemez. Çok sayıda modern sürücü bu şekilde aşırı öngörü yapar, mükemmel genel performansı elde etmek için bir anahtardır.

44.9 Eşleme Tablosu Boyutu

Günlük yapılandırmanın ikinci maliyeti, aygıtın her 4 KB sayfası için bir giriş içeren son derece büyük eşleştirme tablolarının potansiyelidir. Örneğin, büyük bir 1 TB SSD ile, 4 KB sayfa başına tek bir 4 bayt giriş, bu eşleştirmeler için aygıtın ihtiyaç duyduğu 1 GB bellek ile sonuçlanır! Bu nedenle, bu **sayfa düzeyindeki (page-level)** FTL şeması pratik değildir.

Blok Tabanlı Eşleştirme

Eşleştirme maliyetlerini azaltma yaklaşımı, sayfa başına değil, cihazın her blokuna bir işaretçi tutmaktır, böylece eşleştirme bilgisinin miktarını

Blok Boyutu
Sayfa Boyutu

kadar azaltır. Bu **blok seviyesinde (block-level)** FTL, bir sanal bellek sisteminde daha büyük sayfa boyutlarına sahip olmakla benzerdir; bu durumda, VPN için daha az bit kullanırsınız ve her sanal adresin içinde daha büyük bir ofset elde edersiniz.



performans nedeniyle çok iyi çalışmaz. En büyük sorun, "küçük yazma" (yani fiziksel blok boyutundan daha küçük bir yazma) olurken ortaya çıkar. Bu durumda FTL, eski bloktan büyük miktarda canlı veriyi okumak ve yeni bir bloğa (küçük yazma verileriyle birlikte) kopyalamak zorundadır. Bu veri kopyalama, yazma genleştirmeyi çok artırır ve bu nedenle performansı azaltır.

Bu sorunu daha açık hale getirmek için, bir örnek inceleyelim. İstemcinin daha önce mantıksal blok 2000, 2001, 2002 ve 2003'ü (a, b, c, d içeriği ile) yazdığını varsayalım ve bunlar fiziksel blok 1'de fiziksel sayfalar 4, 5, 6 ve 7'de bulunuyor. Sayfa bazlı eşleştirme ile, çeviri tablosu bu mantıksal bloklar için dört eşleştirme kaydetmek zorundadır:

2000→4, 2001→5, 2002→6, 2003→7.

Eğer bunun yerine blok seviyesi eşlemeyi kullanırsak, FTL'nin sadece bu verinin tümü için bir adres çevirimini kaydetmesi gerekir. Ancak adres eşlemesi önceki örneklerimizden biraz farklıdır. Özellikle, cihazın mantıksal adres alanın flash içindeki fiziksel blokların boyutuna eşit olan parçalara bölünmüş olarak düşünülür. Bu nedenle, mantıksal blok adresi, bir parça numarası ve bir ofsetten oluşur. Dört mantıksal bloğun her bir fiziksel blok içinde sığdığını varsaydığımızdan, mantıksal adreslerin ofset bölümü 2 bit gerektirir; kalan (en anlamlı) bitler parça numarasını oluşturur.

2000, 2001, 2002 ve 2003 mantıksal blokları tümünün aynı parça numarası (500) vardır ve farklı ofsetlere sahiptir (sırasıyla 0, 1, 2 ve 3). Böylece, blok seviyesi eşlemesi ile, FTL, parça 500'ün blok 1'e (fiziksel sayfa 4'ten başlayarak) eşlediğini kaydeder, bu diyagramda gösterildiği gibi:

Table:	500	4		Memory
Block:	0	1	2	
Page:	00 01 02 03	04 05 06 07	08 09 10 11	Flash
Content:		a b c d		Chip
State:	i i i i	V V V V	i i i i	

Blok-tabanlı bir FTL'de okuma kolaydır. Önce FTL, istemci tarafından sunulan mantıksal blok adresinden en üst bitleri olarak parça numarasını çıkarır. Daha sonra FTL, parça numarasından fiziksel sayfaya geçişi tablo içinde araştırır. Son olarak FTL, mantıksal adresin ofsetini bloğun fiziksel adresine ekleyerek istenen flash sayfasının adresini hesaplar.

Örneğin, müşteri 2002 mantıksal adresine bir okuma işlemi yaparsa, cihaz mantıksal parça numarasını (500) çıkarır, eşleme tablosunda (4 bulma) çeviriyi araştırır ve mantıksal adresteki ofseti (2) çevirime (4) ekler. Sonuç olarak, fiziksel sayfa adresi (6) verinin bulunduğu yerdir; FTL, o fiziksel adrese okuma işlemini yaparak istenen veriyi elde edebilir (c).

Ancak müşteri 2002 mantıksal bloğuna (içeriği c' ile) yazarsa ne olur? Bu durumda, FTL 2000, 2001 ve 2003'ü okumak ve daha sonra tüm dört mantıksal bloku yeni bir konuma yazmak zorundadır ve eşleme tablosunu bu yönde günceller. Verilerin eskiden bulunduğu blok 1 (1) daha sonra silinebilir ve tekrar kullanılabilir, burada gösterildiği gibi:

Table: 500 → 8

Memory

Block:	0				1				2				Flash Chip
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:									a	b	c'	d	
State:	i	i	i	i	E	E	E	E	V	V	V	V	

Bu örnekten de anlaşılacağı gibi, blok seviyesi eşlemeler çevirimler için gerekli bellek miktarını büyük ölçüde azaltır, ancak yazmalar cihazın fiziksel blok boyutundan daha küçük olduğunda önemli performans sorunlarına neden olur; gerçek fiziksel bloklar 256 KB veya daha büyük olabilir, bu nedenle bu yazmalar olasıdır. Bu nedenle, daha iyi bir çözüme ihtiyaç vardır. Bu bölümün ne olduğunu anlatan bölüm olduğunu anlıyor musun? Daha da iyi, okumaya devam etmeden kendiniz çözebilir misiniz?

Karışık Eşleme

Esnek yazmayı etkinleştirip aynı zamanda eşleme maliyetlerini azaltmak için birçok modern FTL **karışık eşleme (hybrid mapping)** tekniğini kullanır. Bu yaklaşımda, FTL birkaç bloğu silinmiş tutar ve tüm yazıları onlara yönlendirir; bunlar **günlük blokları (log blocks)** olarak adlandırılır. FTL, bir salt blok tabanlı eşleme için gerekli tüm kopyalama olmaksızın bu günlük bloklarında herhangi bir sayfayı herhangi bir konuma yazmak istediğinden, bu günlük bloklar için *sayfa başına* eşleme tutar.

Bu nedenle FTL, belleğinde iki tür eşleme tablosuna sahiptir: *günlük tablosunda* küçük bir sayfa başına eşleme kümesi ve *veri tablosunda* daha büyük bir blok başına eşleme kümesi. Belirli bir mantıksal bloğu ararken, FTL ilk olarak günlük tablosuna bakacaktır; eğer mantıksal bloğun konumu orada bulunamazsa, FTL daha sonra veri tablosuna bakarak konumunu bulur ve istenen verilere erişir.

Hibrit haritalama stratejisinin anahtarı, günlük blok sayısını küçük tutmaktır. Günlük blok sayısını küçük tutmak için FTL, sayfa başına bir işaretçi olan günlük blokları düzenli olarak incelemek ve sadece tek bir blok gösterici tarafından işaret edilebilen bloklara *geçirmek* zorundadır. Bu değişim, bloğun içeriğine dayalı üç ana teknikten birini kullanarak gerçekleştirilir [KK+02].

Örneğin, FTL önceden 1000, 1001, 1002 ve 1003 sayılı mantıksal sayfaları yazmış ve onları 2. fiziksel blokta (8., 9., 10., 11. fiziksel sayfalar) bulundurmış olsun; 1000, 1001, 1002 ve 1003 sayılı yazılanların içeriğinin a, b, c ve d olduğunu varsayalım.

Log Table:

Data Table: 250 → 8

Memory

Block:	0				1				2				Flash Chip
Page:	00	01	02	03	04	05	06	07	08	09	10	11	
Content:									a	b	c	d	
State:	i	i	i	i	i	i	i	i	V	V	V	V	

OPERATING SYSTEMS Şimdi, istemcinin bu blokların her birinin üzerine (a', b', c' ve

d' verileriyle) tam olarak aynı sırada, şu anda mevcut olan günlük bloklarından birinde, örneğin fiziksel blok 0'da (fiziksel sayfa 0, 1, 2 ve 3) yazdığını varsayalım. Bu durumda, FTL aşağıdaki duruma sahip olacaktır:

Log Table: 1000→0 1001→1 1002→2 1003→3
 Data Table: 250 → 8 Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a'	b'	c'	d'					a	b	c	d	Chip
State:	V	V	V	V	i	i	i	i	V	V	V	V	

Bu bloklar öncekiyle tamamen aynı şekilde yazıldığı için, FTL **anahtarlama birleştirme (switch merge)** olarak bilinen bir işlemi gerçekleştirebilir. Bu durumda günlük blok (0), şimdi 0, 1, 2 ve 3 blokları için depolama konumu haline gelir ve tek bir blok gösterici tarafından işaret edilir; eski blok (2) şimdi silinir ve bir günlük blok olarak kullanılır. Bu en iyi durumda, gereken tüm sayfa başına işaretçilerin yerini tek bir blok işaretçi alır.

Log Table:
 Data Table: 250 → 0 Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a'	b'	c'	d'									Chip
State:	V	V	V	V	i	i	i	i	i	i	i	i	

Bu anahtarlama birleştirme, hibrit FTL için en iyi durumdur. Maalesef, bazen FTL bu kadar şanslı değildir. Aynı başlangıç koşullarını (1000 ... 1003 mantıksal bloklarının 2. fiziksel blokta depolandığını) düşünün, ancak istemci 1000 ve 1001 mantıksal bloklarını tekrar yazdı.

Bu durumda sizce ne olur? Başa çıkmak neden daha zor? (sonraki sayfadaki sonuçlara bakmadan düşünün)

Log Table:	1000→0	1001→1											
Data Table:	250 →8		Memory										
<hr/>													
Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash Chip
Content:	a'	b'							a	b	c	d	
State:	V	V	i	i	i	i	i	i	V	V	V	V	

Bu fiziksel bloğun diğer sayfalarını tekrar birleştirmek ve bu sayede tek bir blok gösterici tarafından işaret edilebilmeleri için FTL, **kısmi birleştirme (partial merge)** olarak adlandırılan bir işlem gerçekleştirir. Bu işlemde, 2. fiziksel blokta 1002 ve 1003 mantıksal blokları okunur ve sonra günlüğe eklenir. SSD'nin sonuç hali, yukarıdaki anahtarlar birleştirmeden farklı değildir, ancak bu durumda, FTL'nin hedeflerine ulaşmak için ekstra I/O işlemleri gerçekleştirmesi gerekir ve bu da yazma genişletme oranını artırır.

FTL tarafından karşılaşılan son durum, **tam birleştirme (full merge)** olarak adlandırılır ve daha fazla çalışma gerektirir. Bu durumda, FTL temizleme işlemini gerçekleştirmek için çok sayıda diğer bloktan sayfaları bir arada toplamalıdır. Örneğin, 0, 4, 8 ve 12 mantıksal bloklarının **A** günlük bloğuna yazıldığını düşünün. Bu günlük bloğunu blok haritalanmış bir sayfaya dönüştürmek için, FTL önce 0, 1, 2 ve 3 mantıksal bloklarını içeren bir veri bloğunu oluşturmaktadır ve bu nedenle FTL başka bir yerden 1, 2 ve 3'ü okumalı ve sonra 0, 1, 2 ve 3'ü birlikte yazmalıdır. Sonra, birleştirme aynı şekilde mantıksal blok 4 için yapılmalıdır, 5, 6 ve 7 bulunur ve tek bir fiziksel blokta birleştirilir. Aynı 8 ve 12 mantıksal blokları için de yapılmalıdır ve sonra (son olarak) **A** günlük bloğu serbest bırakılabilir. Sık sık tam birleşimlerin olması, şaşırtıcı olmadıkça, performansa ciddi zararlar verebilir ve bu nedenle mümkün olan her durumda kaçınılmalıdır [GY+09].

Sayfa Eşleştirme ve Önbelleğe Alma

Yukarıdaki hibrit yaklaşımın karmaşıklığı göz önüne alındığında, diğerleri sayfa eşlemeli FTL'lerin bellek yükünü azaltmak için daha basit yollar önermiştir. Muhtemelen en basiti, FTL'nin yalnızca aktif kısımlarını bellekte önbelleğe almaktır, böylece gereken bellek miktarını azaltır [GY+09].

Bu yaklaşım işe yarayabilir. Örneğin, belirli bir iş yükü yalnızca küçük bir sayfa kümesine erişirse, bu sayfaların çevirileri bellek içi FTL'de saklanır ve yüksek bellek maliyeti olmadan performans mükemmel olur. Elbette bu yaklaşım da kötü bir performans sergileyebilir. Hafıza muhafazasında gerekli çevirilerin **çalışma seti (working set)** buluna-mıyorsa, verilere erişebilmek için öncelikle eksik eşleştirmeyi getirmek için her erişim için minimum olarak ekstra bir flash okuma gerektirir. Daha da kötüsü, yeni haritalamaya yer açmak için FTL'nin eski bir harita yoklama işlemini **tahliye etmek (evict)** zorunda olması ve bu haritalama **kirliyse (dirty)** (yani, sürekli olarak flaş henüz yazılmamışsa) fazladan bir yazma da olacaktır. Ancak çoğu durumda iş yükü yerelliği gösterir ve bu önbelleğe alma yaklaşımı hem bellek yükünü azaltır hem de performansı yüksek tutar.

44.10 Aşınma Seviyelendirme

Son olarak, modern FTL'lerin gerçekleştirmesi gereken ilgili bir arka plan aktivitesi, yukarıda tanıtilan bir şekilde **aşınma düzeyinin dengelenmesidir (wear leveling)**. Temel fikir basittir: çok sayıda silme / programlama döngüsü bir flash bloğunu aşındıracağından, FTL tüm cihazın bloklarına düzenli olarak çalışmayı yaymaya çalışmalıdır. Bu şekilde, birkaç “popüler” blok hızla kullanılamaz hale gelse bile tüm bloklar kabaca aynı zamanda aşınır.

Temel günlük yapılandırma yaklaşımı, yazma yükünü yaymak için iyi bir başlangıç yapar ve çöp toplama da yardımcı olur. Ancak bazen bir blok, aşırı yazılmayan uzun ömürlü verilerle doldurulur; bu durumda çöp toplama hiçbir zaman bloku geri almayacaktır ve bu nedenle de yazma yükünün adil payını almaz.

Bu sorunu gidermek için FTL düzenli olarak bu bloklardan çıkan tüm canlı verileri okumalı ve başka bir yere tekrar yazmalıdır, böylece blok tekrar yazabilir hale gelmelidir. Bu aşınma düzeltme işlemi SSD'nin yazma güçlendirmesini artırır ve böylece tüm blokların kabaca aynı oranda aşınmasını sağlamak için ekstra I/O gerektirdiğinden performansı düşürür. Literatürde birçok farklı algoritma mevcuttur [A+08, M+14]; ilgileniyorsanız daha fazlasını okuyun.

44.11 SSD Performansı ve Maliyeti

Kapatmadan önce, kalıcı depolama sistemlerinde nasıl kullanılacaklarını daha iyi anlamak için modern SSD'lerin performansını ve maliyetini inceleyelim. Her iki durumda da klasik sabit disk sürücülerıyla (HDD) karşılaştırılacak ve ikisi arasındaki en büyük farkları vurgulayacağız.

Performans

Sabit disk sürücülerinden farklı olarak, flash tabanlı SSD'lerin mekanik bir kompozisyonları yoktur ve aslında “rastgele erişim” aygıtları olmaları bakımından DRAM'e çok daha benzer. Rastgele okuma ve yazma işlemleri gerçekleştirirken, disk sürücülerine göre performans açısından en büyük fark fark fark edilir; tipik bir disk sürücüsü saniyede yalnızca birkaç yüz rastgele I/O yürütebilirken, SSD'ler çok daha iyi iş yapabilir. Burada, SSD'lerin ne kadar daha iyi performans gösterdiğini görmek için modern SSD'lerden bazı veriler kullanıyoruz; özellikle FTL'lerin ham çiplerin performans sorunlarını ne kadar iyi gizlediğini merak ediyoruz.

Tablo 44.4, üç farklı SSD ve bir en üst seviye sabit sürücünün bazı performans verilerini göstermektedir; veriler birkaç farklı çevrimiçi kaynaktan [S13, T15] alınmıştır. İlk iki sütun rastgele I/O performansını, sağ iki sütun ise sıralı performansı göstermektedir; ilk üç satır üç farklı SSD (Samsung, Seagate ve Intel) için verileri gösterir ve son satır bir **sabit disk sürücüsü** (veya **HDD**) için performansı gösterir, bu durumda Seagate yüksek performanslı bir sürücüdür.

Tablodan birkaç ilginç gerçeği öğrenebiliriz. İlk ve en dramatik olanı, SSD'ler ve sabit sürücü arasındaki rastgele I/O performansındaki farktır.

Device	Random (rastgele)		Sequential (sıralı)	
	Okuma (MB/s)	Yazma (MB/s)	Okuma (MB/s)	Yazma (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Şekil 44.4: **SSD'ler Ve Hard Diskler: Performans Karşılaştırması**

SSD'ler rastgele I/O'larda onlarca hatta yüzlerce MB/s elde ederken, bu "yüksek performanslı" sabit sürücünün zirvesi sadece birkaç MB/sn'dir (aslında 2 MB/sn'ye yuvarladık). İkincisi, sıralı performans açısından çok daha az fark olduğunu görebilirsiniz; SSD'ler daha iyi performans gösterse de, ihtiyacınız olan tek şey sıralı performanssa, bir sabit sürücü yine de iyi bir seçimdir. Üçüncüsü, SSD rastgele okuma performansının SSD rastgele yazma performansına kadar iyi olmadığını görebilirsiniz. Rastgele yazma performansının bu kadar beklenmedik şekilde iyi olmasının nedeni, birçok SSD'nin rastgele yazmaları sıralı yazmalara dönüştüren ve performansı iyileştiren günlük yapıları tasarımından kaynaklanmaktadır. Son olarak, SSD'ler sıralı ve rastgele I/O'lar arasında bir miktar performans farkı sergiledikleri için, sabit diskler için dosya sistemlerinin nasıl oluşturulacağına ilişkin sonraki bölümlerde öğreneceğimiz tekniklerin çoğu SSD'ler için hala geçerlidir; sıralı ve rastgele I/O'lar arasındaki farkın büyüklüğü daha küçük olsa da, rastgele I/O'ları azaltmak için dosya sistemlerinin nasıl tasarlanacağını dikkatlice düşünmek için yeterince boşluk var.

Maliyet

Yukarıda gördüğümüz gibi, SSD'lerin performansı hatta ardışık I/O yaparken bile modern sabit disklerden çok daha iyi. Peki neden SSD'ler sabit diskler yerine depolama ortamı olarak tercih edilen bir seçenek haline gelemedi? Cevap basit: maliyet, ya da daha spesifik olarak kapasite birimi başına maliyet. Şu anda (A15), 250 GB'lık bir SSD 150 dolara mal oluyor; böyle bir SSD GB başına 60 cent mal ediyor. Tipik bir sabit disk 1 TB depolama için yaklaşık 50 dolara mal oluyor, bu da GB başına 5 cent ediyor. Bu iki depolama ortamı arasındaki fiyat farkı hala 10 katın üzerinde.

Bu performans ve maliyet farklılıkları, büyük ölçekli depolama sistemlerinin nasıl oluşturulacağını belirler. Eğer performans en önemli konuya, SSD'ler muhteşem bir seçimdir, özellikle rastgele okuma performansı önemliyse. Eğer diğer yandan büyük bir veri merkezi kuruyorsunuz ve mümkün olan en fazla bilgiyi depolamak istiyorsanız, büyük maliyet farkı sizi sabit disklerle yönlendirecektir. Tabii ki, bir hibrit yaklaşım mantıklı olabilir. Bazı depolama sistemleri hem SSD'ler hem de sabit diskler kullanılarak oluşturuluyor, daha popüler "sıcak" veri için daha küçük bir sayıda SSD kullanarak yüksek performans sağlarken, kalan "soğuk" (az kullanılan) veriyi sabit disklerde saklamak suretiyle maliyetleri azaltmak amacıyla. Fiyat farkının varlığı sürdüğü sürece, sabit diskler burada kalmaya devam edecektir.

44.12 Özet

Flash tabanlı SSD'ler, dünya ekonomisine güç sağlayan veri merkezlerindeki dizüstü bilgisayarlarda, masaüstlerinde ve sunucularda yaygın bir varlık haline geliyor. Bu nedenle, muhtemelen onlar hakkında bir şeyler bilmelisiniz, değil mi?

İşte kötü haber: Bu bölüm (bu kitaptaki pek çok bölüm gibi), teknolojinin son durumunu anlamanın yalnızca ilk adımı. Ham teknoloji hakkında daha fazla bilgi edinebileceğiniz bazı yerler arasında, gerçek cihaz performansı (Chen ve diğerleri [CK+09] ve Grupp ve diğerleri [GC+09] tarafından yapılanlar gibi), FTL tasarımındaki sorunlar (çalışmalar dahil) yer alır. Agrawal ve diğerleri [A+08], Gupta ve diğerleri [GY+09], Huang ve diğerleri [H+14], Kim ve diğerleri [KK+02], Lee ve diğerleri [L+07] ve Zhang ve diğerleri [Z+12]) ve hatta flaştan oluşan dağıtılmış sistemler (Gordon [CG+09] ve CORFU [B+12] dahil). Ve tabiri caizse, bir SSD'den yüksek performans elde etmek için yapmanız gereken her şeye gerçekten iyi bir genel bakış, "yazılı olmayan sözleşme" [HK+17] üzerine bir makalede bulunabilir.

Sadece akademik makaleler okumayın; ayrıca son gelişmeler hakkında popüler basından bilgi edinin (ör. [V12]). Burada, Samsung'un performansı (SLC yazmaları hızlı bir şekilde arabelleğe alabilir) ve kapasiteyi (TLC hücre başına daha fazla bit depolayabilir) en üst düzeye çıkarmak için aynı SSD içinde hem TLC hem de SLC hücrelerini kullanması gibi daha pratik (ancak yine de yararlı) bilgiler öğreneceksiniz.). Ve bu, dedikleri gibi, buzdağının sadece görünen kısmı. Dalın ve bu araştırma "buzdağı" hakkında kendi başınıza daha fazla bilgi edinin, belki de Ma ve diğerlerinin mükemmel (ve yakın tarihli) araştırmasından [A+14] başlayarak. Yine de dikkatli olun; buzdağları en güçlü gemileri bile batırabilir [W15].

AYRINTI: ANAHTAR SSD TERİMLERİ

- Bir **flaş çipler**, her biri **silme blokları (erase blocks)** (bazen sadece **bloklar (blocks)** olarak adlandırılır) halinde düzenlenmiş birçok bankadan oluşur. Her blok ayrıca belirli sayıda **sayfalara (pages)** bölünmüştür.
- Bloklar büyüktür (128KB - 2MB) ve nispeten küçük (1KB - 8KB) birçok sayfa içerir.
- Flash'tan okumak için adres ve uzunlukta bir okuma komutu verin; bu, bir istemcinin bir veya daha fazla sayfayı okumasına olanak tanır.
- Flaş yazma daha karmaşıktır. Öncelikle, istemcinin tüm bloğu **silmesi (erase)** gerekir (blok içindeki tüm bilgileri siler). Ardından istemci her sayfayı tam olarak bir kez **programlar (program)** ve böylece yazmayı tamamlayabilir.
- Yeni bir **kırpma işlemi (trim)**, belirli bir bloğa (veya blok aralığına) artık ihtiyaç duyulmadığında aygıtı haber vermek için yararlıdır.
- Flaş güvenilirliği çoğunlukla **yıpranma (wear out)** ile belirlenir; Bir blok çok sık silinir ve programlanırsa kullanılamaz hale gelir.
- Flash tabanlı bir **katı hal depolama aygıtı (SSD)** normal blok tabanlı bir okuma/yazma diski gibi davranır; **flash çeviri katmanı (FTL/flash translation layer)** kullanarak bir istemciden okuma ve yazma yazma biçimini okuma, silme ve programlara dönüştürerek alttaki flash yongalara dönüştürür.
- Çoğu FTL, silme / program döngülerini en aza indirerek yazma maliyetini azaltan **günlük yapıdadır (log-structured)**. Bellek içi çeviri katmanı, mantıksal yazmaların fiziksel ortam içinde bulunduğu yerleri izler.
- Günlük yapı, FTL'lerle ilgili önemli bir sorun, **yazma güçlendirmesi (write amplification)** sağlayan **çöp toplama (garbage collection)** maliyetidir.
- Başka bir sorun, oldukça büyük olabilen haritalama tablosunun boyutudur. **Hibrit haritalama (hybrid mapping)** kullanmak veya sadece ftl'nin sıcak parçalarını **ön belleğe (caching)** almak olası çözümlerdir.
- Son bir sorun **aşınma seviyelendirmesidir (wear leveling)**; FTL, söz konusu blokların silme / programlama yükünden paylarını almasını sağlamak için çoğunlukla okunan bloklardan zaman zaman veri taşımalıdır.

References

- [A+08] "Design Tradeoffs for SSD Performance" by N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy. USENIX '08, San Diego California, June 2008. *An excellent overview of what goes into SSD design.*
- [AD14] "Operating Systems: Three Easy Pieces" by Chapters: *Crash Consistency: FSCK and Journaling and Log-Structured File Systems.* Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *A lot more detail here about how logging can be used in file systems; some of the same ideas can be applied inside devices too as need be.*
- [A15] "Amazon Pricing Study" by Remzi Arpaci-Dusseau. February, 2015. *This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!*
- [B+12] "CORFU: A Shared Log Design for Flash Clusters" by M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis. NSDI '12, San Jose, California, April 2012. *A new way to think about designing a high-performance replicated log for clusters using Flash.*
- [BD10] "Write Endurance in Flash Drives: Measurements and Analysis" by Simona Boboila, Peter Desnoyers. FAST '10, San Jose, California, February 2010. *A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100x.*
- [B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available here: http://www.ostep.org/Citations/zfs_last.pdf. *Was this the last word in file systems? No, but maybe it's close.*
- [CG+09] "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications" by Adrian M. Caulfield, Laura M. Grupp, Steven Swanson. ASPLOS '09, Washington, D.C., March 2009. *Early research on assembling flash into larger-scale clusters; definitely worth a read.*
- [CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives" by Feng Chen, David A. Koufaty, and Xiaodong Zhang. SIGMETRICS/Performance '09, Seattle, Washington, June 2009. *An excellent overview of SSD performance problems circa 2009 (though now a little dated).*
- [G14] "The SSD Endurance Experiment" by Geoff Gasior. The Tech Report, September 19, 2014. Available: <http://techreport.com/review/27062>. *A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.*
- [GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications" by L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf. IEEE MICRO '09, New York, New York, December 2009. *Another excellent characterization of flash performance.*
- [GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings" by Aayush Gupta, Youngjae Kim, Bhuvan Urganekar. ASPLOS '09, Washington, D.C., March 2009. *This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.*
- [HK+17] "The Unwritten Contract of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, Belgrade, Serbia, April 2017. *Our own paper which lays out five rules clients should follow in order to get the best performance out of modern SSDs. The rules are request scale, locality, aligned sequentiality, grouping by death time, and uniform lifetime. Read the paper for details!*
- [H+14] "An Aggressive Worn-out Flash Block Management Scheme To Alleviate SSD Performance Degradation" by Ping Huang, Guanying Wu, Xubin He, Weijun Xiao. EuroSys '14, 2014. *Recent work showing how to really get the most out of worn-out flash blocks; neat!*

[J10] "Failure Mechanisms and Models for Semiconductor Devices" by Unknown author. Report JEP122F, November 2010. Available on the internet at this exciting so-called web site: <http://www.jedec.org/sites/default/files/docs/JEP122F.pdf>. *A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.*

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems" by Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho. IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002. *One of the earliest proposals to suggest hybrid mappings.*

[L+07] "A Log Buffer-Based Flash Translation Layer by Using Fully-Associative Sector Translation." Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song. ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007 *A terrific paper about how to build hybrid log/block mappings.*

[M+14] "A Survey of Address Translation Technologies for Flash Memories" by Dongzhe Ma, Jianhua Feng, Guoliang Li. ACM Computing Surveys, Volume 46, Number 3, January 2014. *Probably the best recent survey of flash and related technologies.*

[S13] "The Seagate 600 and 600 Pro SSD Review" by Anand Lal Shimpi. AnandTech, May 7, 2013. Available: <http://www.anandtech.com/show/6935/seagate-600-ssd-review>. *One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.*

[T15] "Performance Charts Hard Drives" by Tom's Hardware. January 2015. Available here: <http://www.tomshardware.com/charts/enterprise-hdd-charts>. *Yet another site with performance data, this time focusing on hard drives.*

[V12] "Understanding TLC Flash" by Kristian Vatto. AnandTech, September, 2012. Available: <http://www.anandtech.com/show/5067/understanding-tlc-nand>. *A short description about TLC flash and its characteristics.*

[W15] "List of Ships Sunk by Icebergs" by Many authors. Available at this location on the "web": <http://en.wikipedia.org/wiki/Listofshipssunkbyicebergs>. *Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.*

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes" by Yiyang Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.*

Homework (Simulation)

Bu bölümde, SSD'lerin nasıl çalıştığını daha iyi anlamak için kullanabileceğiniz basit bir SSD simülatörü olan `ssd.py` tanıtılmaktadır. Simülatörün nasıl çalıştırılacağına ilişkin ayrıntılar için README dosyasını okuyun. Uzun bir README dosyasıdır, bu yüzden bir fincan çay kaynatın (muhtemelen kafeinli gerekli olacaktır), okuma gözlüğünüzü takın, kedinin kucagina kıvrılmasına izin verin¹ ve işe koyulun.

Questions

- Ödev çoğunlukla “-T log” bayrağı ile simüle edilen günlük yapıllı SSD'ye odaklanır. Karşılaştırma için diğer SSD türlerini kullanacağız. Öncelikle bayraklarla **-T log-s 1 -n 10 -q** bayrağını çalıştırın. Hangi işlemlerin gerçekleşmiş olduğunu anlayabilir misiniz? Yanıtlarınızı kontrol etmek için **-c**'yi kullanın (veya **-q-c** yerine **-C**'yi kullanın). Farklı rastgele iş yükleri oluşturmak için farklı **-s** değerleri kullanın.

```
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -s 1 -n 10 -C
ARG seed 1
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

cmd 0:: write(12, u) -> success
cmd 1:: write(32, M) -> success
cmd 2:: read(32) -> M
cmd 3:: write(38, 0) -> success
cmd 4:: write(36, e) -> success
cmd 5:: trim(36) -> success
cmd 6:: read(32) -> M
cmd 7:: trim(32) -> success
cmd 8:: read(12) -> u
cmd 9:: read(12) -> u

FTL 12: 0 38: 2
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiii iiii iiii iiii iiii iiii
Data uM0e
Live + +
```

- Şimdi sadece komutları gösterin ve Flash'ın ara durumlarını çözüp çözemeyeceğinize bakın. Her komutu göstermek için

-T log -s 2 -n 10 -C bayraklarını çalıştırın. Şimdi, her komut arasında Flash'in durumunu belirleyin; durumları göstermek ve haklı olup olmadığınızı görmek için **-F**'yi kullanın. Gelişmekte olan uzmanlığınızı test etmek için farklı rastgele tohumlar kullanın.

```

kadrm@kadit-virtual-machine:~/Desktop/ostep/ostep-homework/ftle-ssd$ python3 ssd.py -f log -s 2 -n 10 -C
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

cmd 0:: write(36, F) -> success
cmd 1:: write(29, 9) -> success
cmd 2:: write(19, I) -> success
cmd 3:: trim(19) -> success
cmd 4:: write(22, g) -> success
cmd 5:: read(29) -> 9
cmd 6:: read(22) -> g
cmd 7:: write(28, e) -> success
cmd 8:: read(36) -> F
cmd 9:: write(49, F) -> success

FTL 22: 3 28: 4 29: 1 36: 0 49: 5
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEE iiii iiii iiii iiii iiii iiii
Data F9IgeF
Live ++ ++

```

-T log -s 2 -n 10 -C
Komutu çalıştırıldı


```
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -s 2 -n 10 -C -F
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state True
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

cmd 0:: write(36, F) -> success

FTL 36: 0
Block 0      1      2      3      4      5      6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vEEEEEEEE iiii iiii iiii iiii iiii iiii
Data F
Live +

cmd 1:: write(29, 9) -> success

FTL 29: 1 36: 0
Block 0      1      2      3      4      5      6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvEEEEEEEE iiii iiii iiii iiii iiii iiii
Data F9
```

-T log -s 2 -n 10 -C -F
Komutu çalıştırıldı

```

Data F9
Live ++

cmd 2:: write(19, I) -> success

FTL 19: 2 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data F9I
Live +++

cmd 3:: trim(19) -> success

FTL 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data F9I
Live ++

cmd 4:: write(22, g) -> success

FTL 22: 3 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data F9Ig
Live ++ +

cmd 5:: read(29) -> 9

FTL 22: 3 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data F9Ig
Live ++ +

cmd 6:: read(22) -> g

FTL 22: 3 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data F9Ig
Live ++ +

cmd 7:: write(28, e) -> success

```

```

cmd 7:: write(28, e) -> success

FTL 22: 3 28: 4 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEE iiii iiii iiii iiii iiii iiii
Data F9Ige
Live ++ ++

cmd 8:: read(36) -> F

FTL 22: 3 28: 4 29: 1 36: 0
Block 0 1 2 3 4 5 6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEE iiii iiii iiii iiii iiii iiii
Data F9Ige
Live ++ ++

cmd 9:: write(49, F) -> success

FTL 22: 3 28: 4 29: 1 36: 0 49: 5
Block 0 1 2 3 4 5 6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEE iiii iiii iiii iiii iiii iiii
Data F9IgeF
Live ++ +++

```

3. **-r 20** bayrağını ekleyerek bu sorunu daha da ilginç hale getirelim. Bu komutlarda ne gibi farklılıklara neden olur? Yanıtlarınızı kontrol etmek için **-c**'yi tekrar kullanın.

```
kadir@kadir-virtual-machine: ~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -r 20
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 20
ARG cmd_list
ARG ssd_type direct
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiivvvvvv iiiiivvvvvv iiiiivvvvvv iiiiivvvvvv iiiiivvvvvv iiiiivvvvvv iiiiivvvvvv
Data
Live

FTL 5: 5 14: 14 29: 29 37: 37 44: 44 45: 45
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State EEEEEvEEEE EEEvEEEEEE EEEEEEEEEv EEEEEEEEEv EEEEEvEEEE iiiiivvvvvv iiiiivvvvvv
Data q U K U q i GX
Live + + + + ++
```

4. Performans, silmelerin, programların ve okumanın sayısına göre belirlenir (burada kırpmanın maliyetsiz olduğunu varsayıyoruz). Yukarıdaki iş yükünü tekrar çalıştırın, ancak herhangi bir ara durum (örn. **-T log-s 1-n 10**) göstermeden. Bu iş yükünün ne kadar süreceğini tahmin edebilir misiniz? (Varsayılan silme süresi 1000 mikrosaniyedir, program süresi 40'dır ve okuma süresi 10'dur) yanıtınızı

kontrol etmek için **-S** bayrağını kullanın. Silme, programlama ve okuma sürelerini **-E**, **-W**, **-R** bayraklarıyla da değiştirebilirsiniz.

-T log-s 1-n 10 süreci çalıştırıldığında yapılan işlemlere göre (okuma, yazma, silme ve kırpma (kesme/trim)) hangi işlemten kaç defa yapıldığına göre (süre kısıtlaması getirilmeden) tanımlı süreler çarpılıp toplanarak total süre belirlenir.

```
kadir@kadir-virtual-machine: ~/Desktop/ostep
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/ftle-ssd$ python3 ssd.py -T log -s 2 -n 10 -S
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quit_cmds False
ARG show_stats True
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL 22: 3 28: 4 29: 1 36: 0 49: 5
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvEEEE iiii iiii iiii iiii iiii iiii
Data F9IgeF
Live ++ +++

Physical Operations Per Block
Erases 1 0 0 0 0 0 0 Sum: 1
Writes 6 0 0 0 0 0 0 Sum: 6
Reads 3 0 0 0 0 0 0 Sum: 3

Logical Operation Sums
Write count 6 (0 failed)
Read count 3 (0 failed)
Trim count 1 (0 failed)

Times
Erase time 1000.00
Write time 240.00
Read time 30.00
Total time 1270.00
```

5. Şimdi, günlük yapılı yaklaşımın performansını ve (çok kötü) doğrudan yaklaşımı (**-T log** yerine **-T direct**) karşılaştırm. İlk olarak, doğrudan yaklaşımın nasıl performans göstereceğini düşündüğünüzü tahmin edin, ardından **-S** bayrağıyla cevabınızı kontrol edin. Genel olarak, günlük yapılı yaklaşım doğrudan yaklaşımdan ne kadar daha iyi performans gösterir?

Günlük yaklaşımda sıralı olarak kaydedilen veri doğrudan adreslemede karışık olarak kaydedilir.

```

kadir@kadir-virtual-machine: ~/Desktop/ostep/ost
kadir@kadir-virtual-machine: ~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T direct -s 2 -n 10 -5
ARG seed 2
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type direct
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats True
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
    0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL 22: 22 28: 28 29: 29 36: 36 49: 49
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
    0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii EEEEEEEv EEvEEEEv EEEEEEEv EEEEEEEv iiii iiii
Data iiii iiii I  g  e9  F  F
Live      +  ++  +  +

Physical Operations Per Block
Erases  0      1      3      1      1      0      0      Sum: 6
Writes  0      1      6      1      1      0      0      Sum: 9
Reads   0      0      5      1      0      0      0      Sum: 6

Logical Operation Sums
Write count 6 (0 failed)
Read count 3 (0 failed)
Trin count 1 (0 failed)

Times
Erase time 6000.00
Write time 300.00
Read time 60.00
Total time 6320.00

```

6. Şimdi çöp toplayıcının (garbage collector) davranışını inceleyelim. Bunun için büyük (-G) ve küçük (-g) filigranları uygun şekilde ayarlamalıyız. Öncelikle günlük yapıli SSD'ye herhangi bir çöp toplama olmadan daha büyük bir iş yükü çalıştırdığınızda ne olduğunu gözlemleyelim. Bunu yapmak için **-T log -n 1000** bayraklarıyla çalıştırın (büyük filigran varsayılanı 10'dur, dolayısıyla GC bu yapılandırmada çalışmaz). Ne olacağını düşünüyorsunuz? Görmek için **-C** ve belki **-F** kullanın.

```
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -n 1000
ARG seed 0
ARG num_cmds 1000
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL      5: 63      9: 43      27: 52      28: 40      42: 57
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv
Data  q1UKUGqXg0 Pukzj6XrZ txTzZfd01z 1g3ljJlppr KjsZIdP7h5 wSWTxCUYJC uqmoUB2ISd
Live      + +      + +      +
```

7. Çöp toplayıcıyı açmak için daha düşük değerler kullanın. Büyük filigran (**-G N**), sisteme **N** blok kullanıldıktan sonra toplamaya başlamasını söyler; küçük filigran (**-G M**), kullanımda yalnızca **M** blok kaldığında sisteme toplamayı durdurmasını söyler. Çalışan bir sistem için hangi filigran değerlerinin işe yarayacağını düşünüyorsunuz? Komutları ve ara cihaz durumlarını göstermek için **-C** ve **-F** tuşlarını kullanın ve görün.

8. Bir diğer yararlı bayrak, toplayıcının çalıştığında ne yaptığını gösteren **-J**'dir. Hem komutları hem de GC davranışını görmek için **-T log -n 1000 -C -J** bayraklarıyla çalıştırın. GC hakkında ne fark ediyorsunuz? GC'nin nihai etkisi elbette performanstır. Nihai istatistiklere bakmak için **-S**'yi kullanın; çöp toplama nedeniyle kaç tane fazladan okuma ve yazma oluyor? Bunu ideal SSD (**-T ideal**) ile karşılaştırın; Flash'ın doğası gereği ne kadar fazladan okuma, yazma ve silme var? direct yaklaşımla da karşılaştırın; günlük yapılı yaklaşım hangi yönden (silme, okuma, programlama) üstündür?


```
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -n 10 -C -J
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc True
ARG show_state False
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

cmd 0:: write(37, q) -> success
cmd 1:: write(39, i) -> success
cmd 2:: write(29, U) -> success
cmd 3:: write(14, K) -> success
cmd 4:: write(12, U) -> success
cmd 5:: trim(12) -> success
cmd 6:: trim(39) -> success
cmd 7:: write(44, G) -> success
cmd 8:: write(5, q) -> success
cmd 9:: write(45, X) -> success

FTL 5: 6 14: 3 29: 2 37: 0 44: 5 45: 7
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvvEE iiii iiii iiii iiii iiii iiii
Data qUKUGqX
Live + ++ ++
```

-T log -n 1000 -C -J
Komutu çalıştırıldı

```

kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T ideal -n 10 -C
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type ideal
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc True
ARG show_state False
ARG show_cmds True
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiivviii iiiiivviii iiiiivviii iiiiivviii iiiiivviii iiiiivviii iiiiivviii
Data
Live

cmd 0:: write(37, q) -> success
cmd 1:: write(39, i) -> success
cmd 2:: write(29, U) -> success
cmd 3:: write(14, K) -> success
cmd 4:: write(12, U) -> success
cmd 5:: trim(12) -> success
cmd 6:: trim(39) -> success
cmd 7:: write(44, G) -> success
cmd 8:: write(5, q) -> success
cmd 9:: write(45, X) -> success

FTL      5:  5  14: 14  29: 29  37: 37  44: 44  45: 45
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiivviii iiiiivviii iiiiivviii iiiiivviii iiiiivviii iiiiivviii iiiiivviii
Data      q      U K      U      q i      GX
Live      +      +      +      +      ++

```

-T ideal -n 1000 -C -J
Komutu çalıştırıldı

9. Keşfedilecek son bir husus, **iş yükü kaymasıdır (workload skew)**. İş yükü değişikliklerine kayma eklemek, mantıksal blok alanının daha küçük bir kısmına daha fazla yazmanın gerçekleşmesini sağlayacak şekilde yazmaları değiştirir. Örneğin, **-K 80/20** ile çalıştırmak, yazma işlemlerinin %80'inin blokların %20'sine gitmesini sağlar. Farklı çarpıklıklar seçin ve çarpıklığı anlamak için önce **-T direct**'i ve ardından günlük yapıları bir cihaz üzerindeki etkiyi görmek için **-T**

log'u kullanarak rastgele seçilmiş birçok işlemi gerçekleştirin (örn. **-n 1000**). Ne olmasını bekliyorsun? Keşfedilecek diğer bir küçük çarpıklık kontrolü **-k 100**'dür; çarpık bir iş yüküne bu bayrağı ekleyerek, ilk 100 yazma çarpık(kaymış) olmaz. Fikir, önce çok fazla veri oluşturmak, ancak daha sonra yalnızca bir kısmını güncellemektir. Bunun bir çöp toplayıcı üzerinde ne gibi bir etkisi olabilir?

```
kadir@kadir-virtual-machine: ~/Desktop/ostep/ostep-homework/ftle-ssd$ python3 ssd.py -T log -n 10 -K 80/20
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew 80/20
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL 4: 4 7: 7 9: 5 13: 6 15: 3
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvvEE iiii iiii iiii iiii iiii iiii
Data givJ6XNo
Live +++++
```

-T log -n 10 -K 80/20
Komutu çalıştırıldı

```

kadir@kadir-virtual-machine: ~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -n 10
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL 5: 6 14: 3 29: 2 37: 0 44: 5 45: 7
Block 0 1 2 3 4 5 6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvvEE iiii iiii iiii iiii iiii iiii
Data q!UKUGqX
Live + ++ +++

```

-T log -n 10
Komutu çalıştırıldı

```
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T direct -n 10
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type direct
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL 5: 5 14: 14 29: 29 37: 37 44: 44 45: 45
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State EEEEEvEEEE EEvEvEEEE EEEEEEEEv EEEEEEEvEv EEEEvvEEEE iiii iiii
Data q U K U q i GX
Live + + + + ++
```

-T direct -n 10
Komutu çalıştırıldı

```
kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -n 100
ARG seed 0
ARG num_cmds 100
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 0
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL 0: 46 2: 39 4: 56 5: 31 6: 50 9: 43 12: 12 15: 45 16: 55 17: 51
    18: 54 19: 49 24: 41 25: 34 27: 52 28: 40 29: 18 30: 29 31: 36 32: 20
    33: 24 34: 28 35: 53 37: 0 40: 42 42: 57 43: 48 45: 7 46: 30 47: 23
    48: 32

Block 0
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvvv vvvvvvvvEE iiii iiii
Data qIUkUGqXg0 Pukzj6xXrz txTzZfd01z 1g3ljJlppr KjsZIdP7h5 wSWTxCUY
Live + + + + + ++ ++ +++ + + +++++ ++ +++++++
```

-T log -n 100
Komutu çalıştırıldı

10.

```

kadir@kadir-virtual-machine:~/desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -T log -n 10 -k 100
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 100
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
    0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL      5: 6 14: 3 29: 2 37: 0 44: 5 45: 7
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
    0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvvEE iiii iiii iiii iiii iiii iiii
Data q1UKUGqX
Live + ++ +++

```

-T log -n 10 -K 100
Komutu çalıştırıldı

-T log -n 10 -k 100
Komutu çalıştırıldı


```

kadir@kadir-virtual-machine:~/desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -l log -n 10 -k 100
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 100
ARG read_fail 0
ARG cmd_list
ARG ssd_type log
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
    0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL      5: 6 14: 3 29: 2 37: 0 44: 5 45: 7
Block 0      1      2      3      4      5      6
Page 000000000 111111111 222222222 333333333 444444444 555555555 666666666
    0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvvvvEE iiii iiii iiii iiii iiii iiii
Data q1UKUGqX
Live + ++ +++

```



```

kadir@kadir-virtual-machine:~/desktop/ostep/ostep-homework/file-ssd$ python3 ssd.py -l direct -n 10 -k 100
ARG seed 0
ARG num_cmds 10
ARG op_percentages 40/50/10
ARG skew
ARG skew_start 100
ARG read_fail 0
ARG cmd_list
ARG ssd_type direct
ARG num_logical_pages 50
ARG num_blocks 7
ARG pages_per_block 10
ARG high_water_mark 10
ARG low_water_mark 8
ARG erase_time 1000
ARG program_time 40
ARG read_time 10
ARG show_gc False
ARG show_state False
ARG show_cmds False
ARG quiz_cmds False
ARG show_stats False
ARG compute False

FTL (empty)
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiii iiii iiii iiii iiii iiii iiii
Data
Live

FTL      5: 5  14: 14  29: 29  37: 37  44: 44  45: 45
Block 0      1      2      3      4      5      6
Page 0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State EEEEEvEEEE EEvEvEEEE EEEEEEEvEv EEEEEvEEEE iiii iiii iiii
Data  q      U K      U      q i      GX
Live  +      +      +      +      ++

```

-T direct -n 10 -k 100
Komutu çalıştırıldı