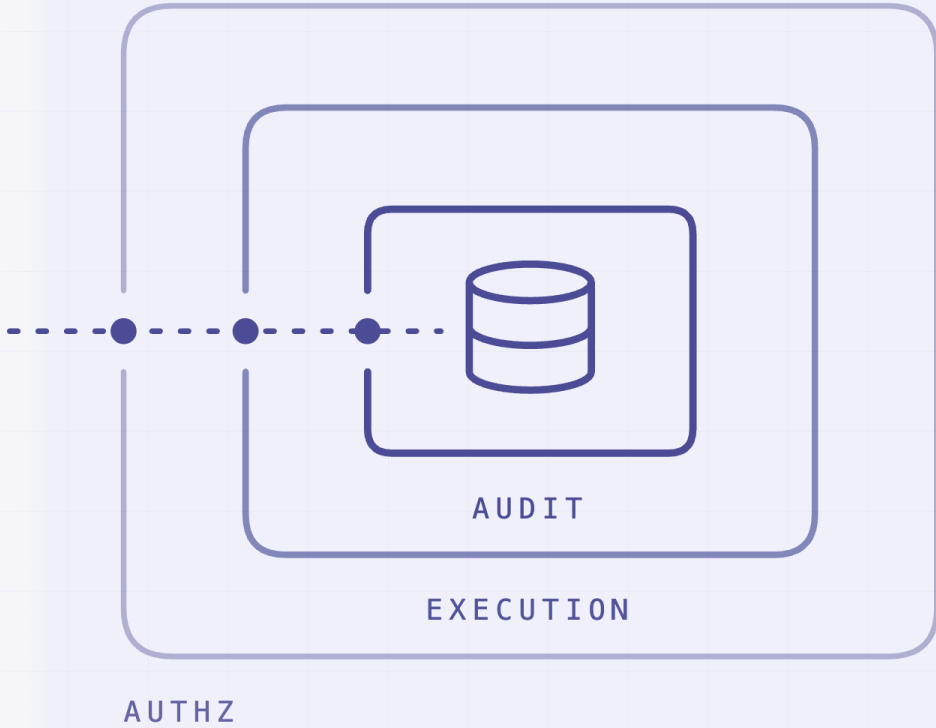




Production Veritabanı Erişim Rehberi

Production Veritabanı Erişim Rehberi

Kimlikler, credential'lar, just-in-time erişim, aracılı yürütme ve audit

Muhammet Şafak

İçindekiler

Önsöz	5
KISIM I — TEMELLER	12
Bölüm 1 — Kimsenin Tasarlamadığı Kapı	13
Bölüm 2 — Production Verisi İçin Bir Tehdit Modeli	28
Bölüm 3 — Kimlikler, Roller ve En Az Yetki	46
KISIM II — UYGULAMA ERİŞİMİ	69
Bölüm 4 — Credential'lar ve Secret'lar	70
Bölüm 5 — Politika Olarak Bağlantılar	89
Bölüm 6 — Okumalar, Replica'lar ve Veri Kopyaları	107
KISIM III — İNSAN ERİŞİMİ	120
Bölüm 7 — İçeri Giden Yollar: Bastion'lar, Proxy'ler ve Access Gateway'ler	121
Bölüm 8 — Just-in-Time ve Break-Glass Erişim	142
Bölüm 9 — Aracılı Yürütme: Sorgu Analizi ve Tehlikeli İşlemler	167
Bölüm 10 — Ad-hoc Sorgular ve Veri Düzeltmeleri	183
Bölüm 11 — Şema Değişiklikleri ve Migration Erişimi	206
KISIM IV — VERİYİ KORUMAK	227
Bölüm 12 — Hassas Veri: Sınıflandırma, Maskeleyme ve Minimizasyon	228
Bölüm 13 — Backup'lar, Export'lar ve Diğer Yan Kapılar	250
KISIM V — KANITLAMAK VE İŞLETMEK	271
Bölüm 14 — Audit ve Hesap Verebilirlik	272
Bölüm 15 — Olaylar Sırasında Erişim	290
Bölüm 16 — Bir Erişim Kontrol Listesi ve Kapanış Düşünceleri	308
Ek	322

Önsöz

Ledgerly'de bir Perşembe akşamını düşünün; Ledgerly, var olmayan bir fatura ve ödeme şirketi. Robin on-call. Bir müşterinin faturası öğleden beri "sending" durumunda takılı kalmış ve çözüm küçük görünüyor: tek satır, tek bir status kolonu. Ekibin masaüstü veritabanı client'ı paylaşılan admin parolasını hatırlıyor, primary'ye bağlanıyor ve bu tür client'ların sıkça yaptığı gibi otomatik commit ediyor. Tek satırlık bir düzeltme, her tabloyu değiştirebilen bir rolle bu client'tan geçiyor ve **WHERE** koşulu tek bir fatura yerine bir grup faturayı kapsıyor.

Robin akşamın geri kalanını onarımla geçiriyor. Daha uzun süren şey, onu izleyen soru: bu statement'ı kim çalıştırdı? Veritabanı log'u, her servisin ve, diyelim, yirmi kişinin kullandığı paylaşılan admin rolünü gösteriyor. Parola yöneticisi girdiyi kimin açtığını gösteriyor, kimin neyi çalıştırdığını değil. Ekip değişikliği uygulamanın yan etkilerinden yeniden kurabiliyor ve sonunda kuruyor. Ama bir gün kadar, tam olarak ne olduğunu, hangi satırlara olduğunu ve aynı kapıyı aynı anda başka bir şeyin kullanıp kullanmadığını kimse güvenle söyleyemiyor.

Bu hikâyedeki hiçbir şey tek başına makul olmayan bir şey değildi. Parola paylaşıldı, çünkü paylaşmak ilk servisi ayağa kaldırmanın en hızlı yoluydu. Client otomatik commit ediyordu, çünkü varsayılanı bu. Rol her tabloyu değiştirebiliyordu, çünkü işin başında her servisin hangi tablolara ihtiyacı olduğuna karar vermekten daha kolaydı. Her adım kendi gününde makuldü. Birlikte, kimsenin tasarlamadığı bir kapı inşa ettiler.

Bu kitap o kapı hakkında.

Bu kitap neden var

Production veritabanına erişim çoğunlukla tesisat işi gibi görülür. İlk servis deploy edildiğinde kurulum, sonra yalnızca bir şey bozulduğunda ya da biri ayrıldığında ele alınır. Bu arada veritabanı, şirketin işlettiği en değerli şeye

dönüşür: müşteriler, para, geçmiş. Şirketin ona ne kadar bağımlı olduğu ile erişimine ne kadar az tasarım emeği harcandığı arasındaki uçurum, bu kitabın konusudur.

Her bir parça hakkında hatırı sayılır miktarda şey yazıldı. Veritabanı dokümantasyonu rol ve grant'lerin nasıl oluşturulacağını anlatır. Secret araçları credential'ların nasıl verileceğini anlatır. Güvenlik rehberleri kalıcı erişimin neden risk olduğunu anlatır. Bulması daha zor olan, bir mühendisin gözünden bütün resimdir: parçaların birbirine nasıl oturduğu, her birinin sistemi kuran ve işleten insanlara neye mal olduğu ve sınırlı zamanı olan bir ekibin bunları hangi sırayla benimseyebileceği.

Bu kitap o resim olmaya çalışıyor ve tek bir iddiayla başlıyor. Production veritabanı erişimi bir credential dağıtım problemi değildir; bir yetkilendirme, yürütme ve hesap verebilirlik problemidir.

Varsayılan bakış, erişimi parolayı kimin tuttuğu ve parolanın nasıl saklandığı, rotate edildiği ve dağıtıldığı meselesi olarak görür. Yukarıdaki Perşembe akşamı bu bakışın nerede bittiğini gösteriyor. Robin meşru bir credential taşıyordu ve credential tam olarak verildiği işi yaptı. Başarısız olan, onun çevresindeki her şeydi: hiçbir şey bu kişinin o tabloya bir yazma çalıştırması gerektiğine karar vermedi, hiçbir şey statement'ı denetlemedi ya da commit'ten önce etkisini sınırlamadı ve hiçbir şey sonucu sonradan bir kişiye bağlamadı. Yetkilendirme neyin, nerede olabileceğine karar verir. Yürütme, her statement'ın veriye nasıl ulaştığıdır. Hesap verebilirlik, geriye kalan kanıttır. Her bölüm hizmet ettiği ayağı adlandırır ve Kısım I üçü için de sözlüğü kurar. Kitap her kontrolü bir bedeli olan bir karar olarak ele alır. Ödeme token'ları tablosu için doğru olan bir kontrol, fatura şablonları tablosu için anlamsız olabilir; önemli olduğunda kitap bunu söyleyecek.

Duruşum konusunda açık olmalıyım. Kitap savunanın tarafından yazıldı. Tehditler yalnızca bir kontrolün neden var olduğunu açıklayabildikleri ölçüde anlatılıyor; buradaki hiçbir şey içeri girmenin rehberi değil. Korku satmıyor ve ihlal istatistikleri aktarmıyor, çünkü sakın bir tasarım review'u alarm veren bir review'dan daha yararlıdır. Ve sürtünmeyi dürüstçe hesaba katıyor. Her kontrol mühendislerden ve on-call olandan bir şey ister; maruziyeti azaltmadan sürtünme ekleyen bir kontrol güvenlik tiyatrosudur. Popüler bir uygulamanın tiyatro olduğunu düşündüğümde bunu söyleyeceğim ve nedenini açıklayacağım.

Bu kitap kimin için

Kitap, production veritabanıyla konuşan servisler geliştiren **backend ve platform mühendisleri**, o veritabanını işleten **SRE ve DBA rolleri**, erişimin nasıl işlemesi gerektiğine karar vermesi gereken **tech lead'ler** ve hiç özel bir güvenlik işlevi olmamış bir ekipte **güvenlikten yeni sorumlu hale gelen mühendisler** için yazıldı. Paylaşılan parolanın hâlâ paylaşılan parola olup olmadığını hiç merak ettiyseniz, bir iş arkadaşınıza yazma erişimi vermeden önce tereddüt ettiyseniz ya da "bu tabloyu kim görebilir?" diye sorulup dürüst cevabın bir günlük kazı gerektirdiğini fark ettiyseniz, aklınızdaki okur sizsiniz.

Güvenlik geçmişine ihtiyacınız yok. Birkaç şeye aşina olmanız gerekiyor:

- İlişkisel veritabanları: tablolar, roller ve grant'ler, transaction'lar ve replica'nın ne olduğu.
- Bir ağ üzerinden veritabanına bağlanan servisler ve connection pool'ların ve timeout'ların sözlüğü.
- On-call işi: alarm almak, baskı altında bir şeyi onarmak ve sonrasında ne olduğunu yazıya dökmek.

Belirli bir veritabanı engine'ine aşinalık örneklerde işe yarar, ama gerekli değildir; bunu daha sonraki bir bölüm açıklıyor.

Bu kitap ne değildir

Bu bir veritabanı yönetimi kitabı değil. Size bir planner'ı ayarlamayı, bir buffer'ı boyutlandırmayı ya da bir replikasyon topolojisi seçmeyi öğretmeyecek. Yönetimin erişim için önem taşıdığı yerlerde, replica'lar ve migration'larda olduğu gibi, kitap o kısmı ele alıyor, gerisini veritabanınızın kendi dokümantasyonuna bırakıyor.

Bu bir uyum el kitabı değil. GDPR, KVKK, PCI DSS, SOC 2 ve HIPAA gibi rejimler, denetçilerin ve müşterilerin ne sorduğunu şekillendirdikleri için gündeme geliyor ve kitap onları yalnızca bu ruhla anlatıyor: bu türden rejimler kimin neyi görebildiğini, erişimin nasıl onaylandığını ve sonrasında hangi kaydın bulunduğunu sorma eğilimindedir. Hiçbir rejimi yorumlamıyor, maddelerine atıf yapmıyor ya da sizden ne istendiğini söylemiyor. Bu yorum hukuk ve uyum çalışma arkadaşlarınıza aittir ve kitabın amacı onlara üzerinde akıl yürütebilecekleri bir mimari vermektir.

Ve bu bir saldırı kitabı deęil. İeri girme teknikleri bulamayacaksınız. Bir blmn bir tehdide ihtiyacı olduęunda, riski bir cmleyle adlandırıyor, rneęin sızan bir credential, yanlış veritabanına dşen bir sorgu ya da açık bırakılmış bir vendor session'ı, ve kontrole geiyor. Tehdidi daha kesin anlatmak tasarımı deęiştirmeyecekse, ayrıntı dıřarıda bırakılıyor.

Kitap nasıl dzenlendi

Kitabın beř kısmı ve on altı blm var, ardından bir ek geliyor.

Kısım I, Temeller (Blm 1 ila 3) yukarıdaki tezi savunuyor ve geri kalan her řey iin szlę kuruyor. Blm 1, bir veritabanına aılan kapıların nasıl biriktięine bakıyor. Blm 2, ktye kullanımın yanı sıra hataları da ieren, production verisi iin bir tehdit modeli kuruyor. Blm 3, sonraki her blmn dayandıęı kimlikleri, rolleri ve en az yetkiyi ele alıyor.

Kısım II, Uygulama Eriřimi (Blm 4 ila 6) servisler hakkında. Blm 4, statik parolalardan dinamik credential'lara ve workload identity'ye kadar credential'ları ve secret'ları ele alıyor. Blm 5, baęlantıları politika olarak ele alıyor: pooling, limitler ve timeout'lar. Blm 6, okumaları, replica'ları ve production'ın sessizce yarattıęı dięer veri kopyalarını ele alıyor.

Kısım III, İnsan Eriřimi (Blm 7 ila 11) insanlar hakkında. Blm 7, bastion'ları, proxy'leri ve access gateway'leri karřılařtırıyor. Blm 8, just-in-time ve break-glass eriřimi ele alıyor. Blm 9, yrtmenin kendisini ele alıyor: bir gateway'in her statement'ı nasıl denetledięini, etkisini nasıl sınırladıęını ve nasıl commit ettięini. Blm 10, ad-hoc sorguları ve veri dzeltmelerini ele alıyor. Blm 11, řema deęiřikliklerini ve bunları kimin yapabileceęini ele alıyor.

Kısım IV, Veriyi Korumak (Blm 12 ve 13) kimin baęlanabildięinden ne grebildięine geiyor. Blm 12, sınıflandırmayı, maskeleymeyi ve minimizasyonu ele alıyor. Blm 13, řimdiye kadar kurulan btn kontrolleri atlayan backup'ları, export'ları ve dięer yan kapıları ele alıyor.

Kısım V, Kanıtlamak ve İřletmek (Blm 14 ila 16) bir ekibin kontrollerinin alıřtıęını nasıl bildięini ve onları nasıl alıřır tuttuęunu ele alıyor. Blm 14, audit log'larını ve eriřim review'larını ele alıyor. Blm

15, baskının en yüksek, kestirmelerin en cazip olduğu olaylar sırasındaki erişimi ele alıyor. Bölüm 16, bir kontrol listesi, çıkış testleri olan bir olgunluk merdiveni ve kültür üzerine bazı kapanış düşünceleri sunuyor.

Ek; bir sözlük, örüntülerin hızlı başvurusu, tehlikeli işlemlerin bir matrisi, bir uyum eşleştirmesi ve erişim review'ları ile olay sonrası review'lar için şablonlar içeriyor.

Her bölüm aynı biçimde. Kısa, açıkça kurgu olan bir senaryoyla açılıyor. Sonra problemi ve bariz yaklaşımın neden yetersiz kaldığını açıklıyor, kontrolü sunuyor, bedellerini ve ne zaman kullanılmaması gerektiğini söylüyor. Kişisel kusur olarak değil, tasarımdaki tuzaklar olarak ele alınan sık yapılan hataları sıralıyor ve beş ila yedi çıkarımla ve bir sonraki bölüme bir köprüyle bitiyor. Ortak biçim, bölümleri sırayla okuyabileceğiniz ya da bu haftanın problemine uyan bölüme atlayabileceğiniz ve yine de maliyet bölümünü beklediğiniz yerde bulabileceğiniz anlamına geliyor.

Ledgerly ile tanışın

Ledgerly, kurgusal bir işletmeden işletmeye fatura ve ödeme servisi. Şirketler onun üzerinden fatura kesiyor, ödeme topluyor ve nelerin el değiştirdiğinin bir defterini tutuyor. Beş servis çalıştırıyor: `billing-api`, `invoice-worker`, `payments-service`, `reporting-job` ve `admin-console`. Primary veritabanının iki replica'sı ve veri ekibi için bir analytics kopyası var. Müşteri iletişim bilgilerini, banka hesabı bilgilerini ve ödeme token'larını saklıyor; erişim tasarımının önemli olmasının nedeni de bu.

Etrafındaki insanlar tanıdık rolleri dolduruyor. On-call backend mühendisleri, bir destek ekibi, bir veri ekibi, veritabanılarını işleten küçük bir platform grubu ve export'ları alan harici bir muhasebe vendor'ı var.

Kitabın başında Ledgerly'nin paylaşılan bir admin parolası, backend ekibinin çoğu için kalıcı erişimi, laptop'larda masaüstü client'ları ve kimin ne yaptığına dair hiçbir kaydı var. Sonunda her servis için bir rolü, kısa ömürlü credential'ları, insanlar için bir gateway'i, veriyi düzeltmenin review edilmiş bir yolu, çalışmadan önce kontrol edilen ve sınırlanan statement'ları, maskelenmiş kopyaları ve sessizce yeniden yazılamayan bir log'u var. Oraya adım adım ulaşıyor ve her bölüm hangi adımı eklediğini ve o adımın ekibe neye mal olduğunu söylüyor.

Ledgerly uydurma ve başına gelen her şey de öyle. Her bölüm, kurgu olarak işaretlenmiş bir senaryoyla açılıyor ve içindeki her sayı varsayımsal. Robin, Sam, Jordan ve Alex gibi isimler, bir ekipteki rollerin yer tutucuları. Bir senaryo ters gittiğinde, ders onu ters götüren tasarım hakkındadır, hikâyedeki kişi hakkında asla.

Kod nasıl okunur

Bu kitaptaki hiçbir örnek gerçek bir programlama dilinde yazılmadı. Prosedürler sade pseudocode ile yazıldı; roller, grant'ler ve tablolar ise belirli bir engine'e ait olmayan, SQL benzeri bir notasyon kullanıyor. Kitaptaki hiçbir şeyin bir terminale yapıştırılması amaçlanmadı. Belirli bir proxy, vault ya da bulut ürünü için konfigürasyon sözdizimi bulamayacaksınız, çünkü o sözdizimi değiştir ve arkasındaki örüntü değişmez.

Diyagramlar Mermaid kullanıyor. Her birinden önce neye bakmanız gerektiğini söyleyen bir cümle var ve reddedilen ya da başarısız olan yollar etiketli. Örnekler hiçbir zaman gerçekçi secret'lar, hesap numaraları ya da e-posta adresleri içermez; onların yerine `<secret>` ve `IBAN-PLACEHOLDER` gibi yer tutucular kullanılır.

Neden İngilizce terimler, neden engine'den bağımsız

Kitap önce İngilizce yazıldı ve bu Türkçe sürümde de terimlerin çoğu İngilizce bırakıldı, çünkü break-glass'tan credential'a kadar bu alanın sözlüğünün çoğu Türkçe çalışan ekiplerde de İngilizce kullanılıyor ve terimlerin tanınabilir kalmasını istedim. Kitabın engine'den bağımsız olmasının nedeni de buna bağlı. Bir ekip PostgreSQL, MySQL ya da başka bir şey çalıştırıyor olabilir ve bir bulut kimlik servisi, bir secret yöneticisi ya da elle yazılmış bir betik kullanıyor olabilir. Örüntüler hepsinde aynı: sahibi kullanıcılardan ayır, her workload'a kendi kimliğini ver, credential'ların süresi dolsun, insanların önüne kimlik farkındalığı olan bir giriş koy, çalışmadan önce ne çalışacağını kontrol et, yazmaları onları reddedebilecek bir şeyin üzerinden çalıştır, olanları kaydet, review et. Ürünler ve sözdizimleri sık değişir. Gerçek bir ürün adlandırıldığında örnek olarak adlandırılıyor ve davranışı sürümler boyunca geçerli olması gereken terimlerle anlatılıyor.

Kesinlik üzerine bir not

Önerilerimden hangilerinin kesin, hangilerinin tercih olduğunu söylemeye çalıştım. Bazıları kesin, çünkü yanılmanın bedeli müşteri verisinin ifşası ya da defterin kaybı ve durum böyle olduğunda bunu söylüyorum. Çoğu bir muhakeme meselesi ve düşünceli bir ekip, boyutuna, düzenleyicilerine, müşterilerine ve ne sıklıkla alarm aldığına göre bunları farklı tartabilir. Bilmediğim yerde "bilmiyorum" diyeceğim ve soruyu çözecek değişkenleri adlandıracam.

Bir öneriye katılmıyorsanız, katılmadığınız öncülü bulabilmenizi isterim. Her bölümün gerekçesini ve bedellerini açıkça yazmasının nedeni bu. Anladığınız ve kendi kısıtlarınıza uyarladığınız bir tasarım, bir kitaptan kopyaladığınız bir tasarımdan daha değerlidir.

Bölüm 1 kapının kendisiyle başlıyor.

KISIM I

Temeller

Bölüm 1 — Kimsenin Tasarlamadığı Kapı

Ledgerly'yi ilk yıllarında düşünün. Olası bir müşteri bir güvenlik anketi gönderiyor ve bir satır sıradan görünüyor: "Production veritabanınıza bağlanabilen her kişiyi ve sistemi listeleyin." Veritabanlarına başka işlerin yanında bakan Alex, sakın bir öğleden sonra anketi eline alıyor.

Alex, bütün backend ekibinin bildiği tek bir admin girdisini barındıran parola yöneticisiyle başlıyor. Sonra aynı parolayı saklayan beş servis geliyor, ardından masaüstü veritabanı client'larının onu hatırladığı laptop'lar. Sonra analytics kopyası, yani maskelenmemiş bir dump'ın gecelik restore'u ve paylaşılan depolamada kimsenin oluşturduğunu hatırlamadığı birkaç eski dump. Alex'in baktığı her yer bir başkasına çıkıyor.

Anketin ikinci bir satırı var: "Ne yaptıklarını nasıl biliyorsunuz?" Alex veritabanı log'unu açıyor ve her satırda tek bir rol adı buluyor; statement ister bir servisten gelsin, ister bir satırı düzelten bir mühendisten, ister replica'yı okuması gereken bir rapordan. İlk sorunun dürüst cevabı bir yığın yer, ikincisinin dürüst cevabı ise Ledgerly'nin bilmediği.

Alex'in daha iyi bir parolaya ihtiyacı yok. Alex'in erişimin neyden oluştuğunu ve kısa, savunulabilir bir cevabın neye benzeyeceğini bilmesi gerekiyor. Bu bölümün sorusu bu: production veritabanına erişim neyden oluşur ve Ledgerly'nin bir haftalık kazı yapmadan o ankete cevap verebilmesi için ne gerekir?

Bu bölüm, Bölüm 2'nin yaptığı gibi Ledgerly'nin başlangıç durumunu anlatıyor ve henüz hedef durumdan bir adım eklemiyor. İş, sonraki adımlar için sözlüğü ve haritayı vermek.

Problem ve neden zor olduđu

Production veritabanına erişim nadiren tasarlanır. Birikir. İlk servisin içeri girecek bir yola ihtiyacı olur, biri bir admin rolü ve bir parola oluşturur. İkinci servis onu yeniden kullanır, çünkü ilki işe yaramıştır. Bir mühendisin bir satıra bakması gerekir, parola bir masaüstü client'ına girer. Zamanlanmış bir job'ın veriye ihtiyacı olur ve diğer her şeyle aynı connection string'i alır. Bu adımların hiçbirisi erişim hakkında bir karar değildir ve hepsinin toplamı, bölüm başlığının adını verdiği kapıdır.

Alex saymaya gittiğinde Ledgerly'nin kapısının nasıl görüldüğüne bakın. Paylaşılan admin parolası bir ekip parola yöneticisinde duruyor ve örnek olarak, onu yirmi kadar kişinin bildiğini varsayalım. Açtığı admin rolü her tablonun sahibi. Beş servisin hepsi bu rolle bağlanıyor, her servisin konfigürasyonunda saklanan aynı parolayı kullanıyor ve deployment pipeline'ı da migration'ları onunla çalıştırıyor. Mühendisler primary'ye laptop'larından şirket VPN'i üzerinden, parolayı hatırlayan ve her statement'ı çalışır çalışmaz commit eden client'larla ulaşıyor. Erişim kalıcı: bir yıl önce başka bir ekibe geçenler dahil, kimse onu hiç kaybetmiyor.

Replica'lar aynı admin login'ini kabul ediyor ve raporlama sorguları bazen primary üzerinde çalışıyor. Analytics kopyası maskelenmemiş bir dump'ın restore'u ve dump dosyaları orada kalıyor. Harici bir muhasebe vendor'ı hiçbir Ledgerly veritabanına bağlanmıyor, ama fatura ve defter verisinin export'larını alıyor ve `reporting-job`'ın export adımı bu dosyaları paylaşılan admin login'ile yazıyor. Veritabanı log'u tek bir rol gösteriyor. Bu düzende hangi kişinin hangi statement'ı çalıştırdığını kaydeden hiçbir şey yok.

Bunu bir disiplin problemi olarak görmek cazip: insanlara dikkatli olmalarını söyle, parolayı daha sık değiştir. Ben oradan başlamazdım. Yalnızca herkes hatırladığında işleyen bir kural, nöbetteki kişi ilk yorulduğunda çöker. Sistem her principal'a aynı yetenekleri ve aynı adı veriyor; dolayısıyla dikkatli olmak isteyen biri için dikkatli olunacak bir şey yok: seçilecek daha dar bir rol yok, veritabanına "salt okunur" demenin bir yolu yok, bir isim taşıyan bir session yok.

İnsanların ilk başvurduğu çare için de aynısı geçerli: parolayı daha iyi yönetmek. Onu bir vault'ta saklamak, rotate etmek, yalnızca isteyenlere vermek. Bu bir credential dağıtımıdır ve yapılmaya değer, ama tek bir

soruyu cevaplar: secret'ı kim tutuyor. Production veritabanı erişimi bir credential dağıtım problemi değildir; bir yetkilendirme, yürütme ve hesap verebilirlik problemidir. Yetkilendirme, her principal'ın neyi, hangi güzergâhtan yapabileceğini sorar. Yürütme, her statement'ın veriye nasıl ulaştığını, önce bir şeyin onu denetleyip denetlemediğini ve ne kadar uzağa erişebildiğini sorar. Hesap verebilirlik, kanıtın her statement'ı bir kişiye ve bir isteğe bağlayıp bağlamadığını sorar. Mükemmel saklanmış bir parola, secret'ı kimin tuttuğunu halleder ve başka hiçbir şeyi: onu tutan kişi yine de kimsenin denetlemediği bir statement çalıştırabilir ve log yine paylaşılan bir isim gösterebilir.

Bu yüzden bu, bir politika sorusu değil, bir mühendislik sorusudur. Bir politika ne olması gerektiğini söyler. Bir tasarım ne olabileceğini ve olduğunda neyin kayıt bıraktığını belirler. Bu bölümün geri kalanı bir tasarım probleminin biçimini alıyor: erişim neyden oluşur, onun ne yapmasını istiyoruz ve her parça bu kitapta nerede ele alınıyor.

Erişim beş parçadır

Herhangi bir tasarımı değerlendirmeden önce, tasarlanan şeyin kesin bir tanımının olması işe yarar. Bir production veritabanına erişimin beş parçası olduğunu söylemeyi yararlı buluyorum: bir kimlik, bir grant, bir yol, bir yürütme ve bir kayıt. Tablo, her parçanın hangi soruyu cevapladığını ve kitabın tezinin hangi ayağına ait olduğunu söylüyor.

Parça	Soru	Ayak
Kimlik	Kim ya da ne harekete geçiyor: bir kişi mi, bir workload instance'ı mı?	Hesap verebilirlik
Grant	O principal en baştan neyi yapabilir: roller, yetkiler, tavan?	Yetkilendirme
Yol	Bağlantı nasıl geliyor: ağ erişimi, gateway, pooler?	Yetkilendirme (nerede)
Yürütme	Bir statement veriye nasıl ulaşıyor? Onu kim ya da ne gönderiyor, çalışmadan önce denetleniyor mu, ne kadar uzağa erişebiliyor (transaction, row guard, timeout) ve nasıl commit ediliyor.	Yürütme
Kayıt	Hangi kanıt kalıyor ve bu kanıt statement'ı kişiye, isteğe ve sonuca bağlıyor mu?	Hesap verebilirlik

Üç ayak parçalara şöyle eşlenir. Yetkilendirme, istek başına karara bağlanan grant artı yoldur. Yürütme, her statement'a olan şeydir. Hesap verebilirlik, kimlik artı kayıttır. Bir parola bunların yalnızca ilkinde, o da yalnızca kapıda dokunur.

Kimlik, bir principal'ın söylediği kişi olduğuna dair doğrulanmış iddiadır.

Principal, kimlik doğrulayan her şeydir: bir kişi, bir servis, zamanlanmış bir job, bir vendor entegrasyonu. Principal, bir ifade insanlar ve workload'lar için aynı anda geçerli olduğunda kullanılacak sözcüktür ve bu kitap onu baştan sona bu anlamda kullanıyor. Rol bir kimlik değildir.

Rol, veritabanı içinde adlandırılmış bir yetki demetidir ve birçok principal tarafından paylaşılabilir; paylaşılan admin rolünün tam olarak yaptığı da bu.

Grant, principal'ın içeri girdikten sonra ne yapmasına izin verildiğidir.

Burada grant, bir yetkiyi bir role ya da bir rolü bir **login**'e vermenin eylemi ve ortaya çıkan kayıttır; Bölüm 3 tanımını genişletiyor. **Yetki**, bir tabloyu okumak gibi tek bir belirli şeydir ve login, bir principal'ın bağlandığı veritabanı düzeyindeki hesaptır. **Yol**, bir bağlantının veritabanına ulaşmak için izlediği güzergâhtır: bir ağ, bir proxy, bir gateway, bir laptop.

Yürütme, bir statement'ın veriye nasıl ulaştığıdır: onu kimin ya da neyin gönderdiği, çalışmadan önce bir şeyin onu denetleyip denetlemediği, etkisini neyin sınırladığı ve nasıl commit edildiği. Bir grant, bir rolün invoices'ı güncelleyebileceğini söyler. Bu güncellenmenin tek bir satıra dokunması gerektiğini, birinin ona önce baktığını ya da geri alınabilecek bir transaction içinde çalıştığını söylemez. Başlangıç durumunda hiçbir şey bu soruları cevaplamıyor: masaüstü client'ı yazılan her şeyi gönderiyor ve anında commit ediyor. Bölüm 9 bu parça hakkında.

Kayıt, sonradan elde kalan kanıttır, "kim ne yaptı ve ne zaman?" sorusunun ve bir yazma için "hangi satırlara, hangi istek altında?" sorusunun cevabıdır.

Her parça kendi başına başarısız olabilir, ama her biri yararlı olmak için ötekilere bağlıdır. Ayrı bir kimlik olmadan, kesin bir grant'in bağlanacağı kimse yoktur. Seçilmiş bir yol olmadan, bir kimlik hiç kimlik sormayan bir güzergâhla atlatılabilir. Kontrollü bir yürütme olmadan, dar bir grant yine de kimsenin istediğinden daha geniş bir statement'a izin verir. Bir kayıt

olmadan, diğer dördünün işe yarayıp yaramadığını söyleyemezsiniz. Bu bağımlılık, yalnızca bir parçayı, genellikle parolayı düzeltmenin problemin geri kalanını ayakta bırakmasının nedenidir.

Bir test: log soruyu cevaplayabiliyor mu?

Hangi parçaların eksik olduğunu bulmanın bir yolu, log'a bir soru sormak ve cevabın nerede durduğunu izlemektir. Aşağıdaki pseudocode tek bir statement'ı beş parçadan geçirerek izliyor ve yalnızca başlangıç durumunun kaydettiklerini kullanıyor.

```
function explainStatement(logLine):
  login = logLine.login
  // identity: does this login map to exactly one principal?
  principals = principalsUsing(login)
  if count(principals) != 1:
    return { answer: "unknown", stoppedAt: "identity" }
  // path: did the connection arrive by a route we chose?
  if not isKnownPath(logLine.source):
    return { answer: "unknown", stoppedAt: "path" }
  // grant: was this statement within what the principal was meant
  to do?
  if not withinIntendedGrant(principals[0], logLine.statement):
    return { answer: principals[0], stoppedAt: "grant" }
  // execution: was a write reviewed and bounded before it
  committed?
  if isWrite(logLine.statement):
    if not logLine.reviewedStatement or not
    logLine.countShownBeforeCommit:
      return { answer: principals[0], stoppedAt: "execution" }
    if logLine.rowsAffected is unknown or logLine.rowsAffected >
    logLine.expectedRows:
      return { answer: principals[0], stoppedAt: "execution" }
  // record: does the evidence tie the statement to the person, the
  request and the result?
  if logLine.correlationId is empty or logLine.result is unknown:
    return { answer: principals[0], stoppedAt: "record" }
  return { answer: principals[0], stoppedAt: null }
```

Ledgerly'nin başlangıç durumunda `principalsUsing("admin")` yirmi kadar kişiyi, beş servisi ve deployment pipeline'ını döndürüyor; dolayısıyla fonksiyon her statement için ilk kontrolde duruyor. Bu log'daki bir bug değil. Log gördüğü şey konusunda, yani tek bir login konusunda doğruyu söylüyor. Boşluk, birçok principal'ı tek bir login'in arkasına koyan tasarımda. Kimliği düzeltin, aynı fonksiyon bir sonraki adımda yürütme kontrolünde duracaktır. Kurucu olay, fonksiyonun tam da bunun için var olduğu durumdur: `WHERE` koşulu bir grup kaydı kapsayan tek satırlık bir

düzeltilmenin `rowsAffected` değeri `expectedRows`'un çok üzerindedir, hiç review edilmemiştir ve `COMMIT`'ten önce sayısını hiç göstermemiştir; dolayısıyla kişi bilinse ve grant buna izin verse bile "execution"da durur. O kontrolü geçen bir statement, hiçbir şey onu bir isteğe ve bir sonuca bağlamıyorsa yine de "record"da durabilir.

Uygulamalar ve insanlar, tek bir soru

İnsanlar "veritabanı erişimi" duyduğunda çoğunlukla sorgu araçlarıyla çalışan mühendisleri gözünün önüne getirir. Bu resim, önemli bir biçimde eksik. Ledgerly'nin primary'sine gelen trafiğin büyük kısmı yazılımdan geliyor: web uygulamasına hizmet veren `billing-api`, hatırlatma gönderen `invoice-worker`, ödeme sağlayıcılarıyla konuşan `payments-service`, geceleri çalışan `reporting-job` ve destek personeli adına hareket eden `admin-console`. Yazılım, herhangi bir insandan daha sık, daha uzun süre ve daha az gözetimle bağlanır.

Servisler ve insanlar farklı biçimlerde başarısız olur, bu yüzden Kısım II uygulamaları, Kısım III insanları ele alıyor. Karşıtlık somut:

	Servisler	İnsanlar
Sıklık	Sürekli, günün her saati	Ara sıra, seri hâlinde
Gözetim	Çalışırken yok	Sonucu bir kişi izliyor
Credential ömrü	Kısa ömürlü ve bir broker tarafından verilir (Bölüm 4)	Bir session ya da onaylanmış bir pencere (Bölüm 7 ve 8)
Tipik hata	Sızan bir secret, tükenen bir connection pool, yanlış veritabanını gösteren bir deploy	Baskı altında elle çalıştırılan, istenenden daha geniş bir statement

İkisi de Kısım I'deki temellere dayanır.

Gözden kaçırılması kolay bir principal türü daha var: bir kişi adına hareket eden bir servis. `admin-console`'u destek personeli kullanıyor, ama başlangıç durumunda her şey gibi o da paylaşılan admin login'ile bağlanıyor; dolayısıyla veritabanı konsolu görüyor, tıklayan kişiyi asla. Destekten Sam aynı anda tek bir müşteriye bakıyor ve tasarım, konsolun her müşterinin verisine kimsenin olmayan bir isim altında erişmesine izin

veriyor. Bölüm 3, konsolun kişinin kimliğini her eylemle birlikte nasıl iletildiğini, Bölüm 14 ise audit kayıtlarının nasıl birleştirildiğini gösteriyor; böylece kişi bu atlamadan sağ çıkıyor.

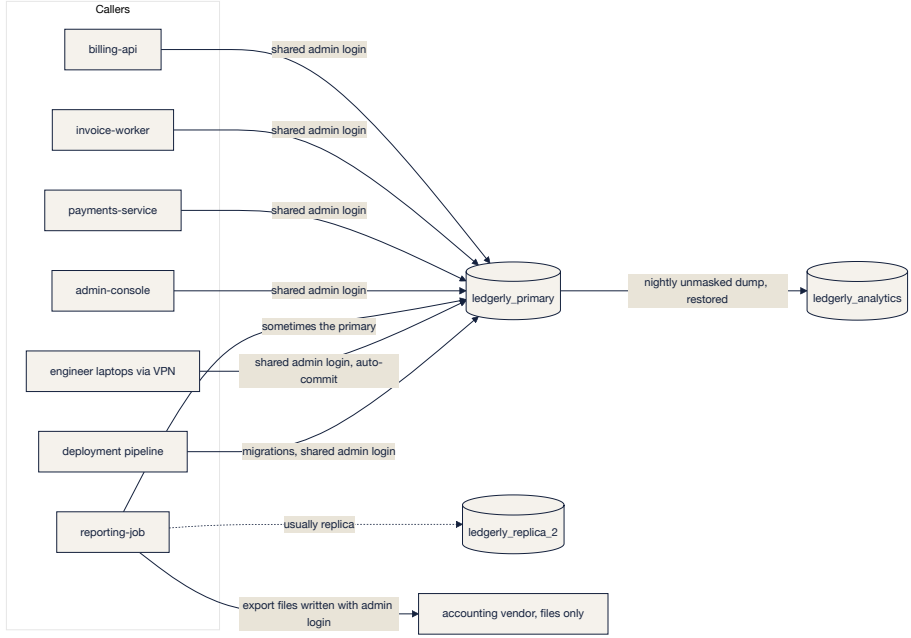
Güvenlik ve güvenilirlik aynı sorudur

Ekipler çoğu zaman iki liste tutar. Biri güvenlikle ilgili: kim veri çalabilir, kim erişimi kötüye kullanabilir. Diğer operasyonlarla ilgili: veritabanını neyin çökertebileceği ya da bozabileceği. Genellikle farklı kişilere aittirler ve farklı zamanlarda review edilirler. Erişim için ben bunları birleştirdim, çünkü sorular aynı.

Birbiriyle ilgisiz görünen üç olayı ele alın. Yanlış ya da eksik bir `WHERE` koşuluna sahip bir statement, tek bir fatura yerine her faturayı güncelliyor. Bir laptop client'ı bir bağlantıyı açık bırakıyor ve bütün akşam bir slotu tutuyor. Production veritabanının bir dump'ı paylaşılan depolamaya kopyalanıyor ve onu okumakla hiçbir ilgisi olmayan biri tarafından okunuyor. Birincisi bir veri bütünlüğü problemi, ikincisi bir availability problemi, üçüncüsü bir gizlilik problemi. Yine de her biri tasarımdan aynı şeyi istiyor: tek bir principal, daha dar bir güzergâh, bir limit ya da bir kayıt olmadan neden bu kadar geniş bir şey yapabildi?

Üç olay bir tasarım nedenini paylaşıyor: grant'in, ömrün ya da yolun daraltılmaması ve sonradan gösterecek hiçbir şeyin olmaması. Yanlış `WHERE` koşulu çoğunlukla bir grant ve yürütme boşluğu, açık bağlantı bir ömür ve limit boşluğu, kopyalanan dump ise bir yol boşluğu. Grant'i daraltmak, ömrü sınırlamak, yolu seçmek, statement'ı sınırlamak ve kaydı tutmak üç olaya birden yardım eder. Tek bir tasarımın iki listeye de hizmet etmesinin nedeni bu. `payment_tokens`'ı okuyamayan bir rol, müşterileri bir sızıntıdan korur; `ledger_entries`'e yazamayan bir rol ise defteri kötü bir toplu güncellemeden korur. İkisi de aynı dar grant verme alışkanlığından gelir.

Aşağıdaki diyagram Ledgerly'nin başlangıç durumunu gösteriyor. Kaç okun aynı düğümde bittiğine ve o düğümde analytics kopyasına giden tek oka bakın.



İki şey öne çıkıyor. Birincisi, primary'ye giden her yol aynı adı taşıyor; dolayısıyla veritabanı bir worker'ı bir insandan ayırt edemiyor. İkincisi, veri bir yan güzergâhtan da çıkıyor: gecelik restore, hiçbir analistin ihtiyaç duymadığı kolonlar dahil her şeyi, kendi kitlesi olan bir depoya kopyalıyor.

İçeri giden yolların yelpazesi

Paylaşılan parola ile tam kontrollü bir tasarım arasında bir örüntü yelpazesi var ve sonraki bölümler her birini bu yelpazede konumlandırıyor. Diyagram, her statement üzerindeki kontrolün en azdan en çoğa doğru ilerliyor. Her adımın ne eklediğine ve ilk statement düzeyindeki kontrolün nerede ortaya çıktığına bakın.



Sağa doğru her adım bir özellik ekler ve bir şeye mal olur. Bir bastion yolu daraltır, ama öteki taraftaki session hâlâ gözetimsizdir. Bir session proxy'si bir bağlantıyı bir kişiye bağlar ve kaydeder, ama grant'ın izin verdiği her şeyin yazılabileceği bir session teslim eder. Yürütmeye aracılık eden bir gateway, her statement'a çalışmadan önce bakar ve onu reddedebilir. Bir ürün yolu session'ı tamamen ortadan kaldırır: kişi "bu faturayı yeniden

gönder" gibi tanımlı bir işlem ister ve asla SQL yazmaz. Buradaki hiçbir şey en sağdaki örüntünün her zaman en iyisi olduğunu söylemiyor; aynı zamanda inşa etmesi en çok iş gerektiren ve değiştirmesi en yavaş olandır. Bölüm 7 ilk dördünü somut yollar üzerinde karşılaştırıyor, Bölüm 9 statement düzeyindeki kontrolleri ele alıyor ve kapanış bölümü örüntüleri ayaklara göre puanlıyor.

İyi erişim neye benzer

Herhangi bir araç seçmeden önce hedefi yazardım. Beş özellik kullanıyorum ve her birinin bir bedeli var.

En az yetki, yalnızca bir görevin ihtiyaç duyduğunu, ihtiyaç duyduğu süre boyunca vermektir. Bedeli, her görevin neye ihtiyaç duyduğunu bulmanın emeği ve bir görevin daha fazlasına ihtiyaç duyduğu ortaya çıktığında yaşanan sürtünmedir. Daha fazlasını istemenin hızlı bir yolu olmadan bunu benimseyen bir ekip, insanların onu dolaştığını görecektir.

Kısa ömür, erişimin kendiliğinden bitmesi demektir. Bir saat süren bir grant'i kimsenin hatırlaması gerekmez. Bunu, kullanılsın ya da kullanılmasın sürekli var olan **kalıcı erişim** ile karşılaştırın. Kalıcı bir grant, kimsenin bir daha istemek zorunda kalmayacağına dair bir sözdür ve ilk ay için rahattır, sonraki her ay için pahalıdır, çünkü kimse sözün neyi kapsadığını hatırlamaz. Kısa ömrün bedeli daha çok istemek ve bir onarımın ortasında gelebilecek bir süre dolumudur.

Gözlemlenebilirlik, erişimin her kullanımının principal'ı adlandıran bir kayıt bırakması ve kaydın principal'ın değiştiremeyeceği bir yerde durması demektir. Bedeli depolama ve, daha önemlisi, dikkattir: kimsenin okumadığı bir log güvenlik tiyatrosudur, çünkü depolamaya mal olur ve hiçbir koruma sağlamaz. Bölüm 14 buna geri dönüyor.

Geri döndürülebilirliğin iki anlamı var ve ikisi de önemli. Erişimin geri alınması kolay olmalı ve erişimle yapılan bir hatanın geri alınması ya da en azından sınırlandırılması mümkün olmalı. Revoke rutin olmalı ve bir hatayı geri almak, review edilmiş düzeltmeler ve restore edilebilen backup'lar gerektirir. Ucuz bir geri döndürülebilirlik biçimi, bir yazmayı açık bir transaction içinde çalıştırmak ve COMMIT'ten önce satır sayısını göstermektir: tek satıra dokunması gereken ve, diyelim, dört yüz bildiren

bir güncelleme, hâlâ rollback edilebilirken durdurulur. Bölüm 9, bir gateway'in bunu nasıl varsayılan yaptığını ele alıyor. Bedeli, provalara harcanan zamandır.

Düşük sürtünme, ekiplerin yazmayı unuttuğu özelliktir. Meşru yol yavaşsa insanlar gayrimeşru olanı bulur ve haklı da olurlar, çünkü sabaha karşı 2'deki olay beklemez. Düşük sürtünmenin bedeli, diğer dördünü kısıtlamasıdır: her kontrol, haftanın en kötü saatinde kullanılabilecek kadar ucuz olmalıdır.

Bu beşi birbirine karşı çeker. Kısa ömür sürtünme ekler. Gözlemlenebilirlik bir atlama ekler: kaydedilen bir session'ın onu kaydeden bir şeyden geçmesi gerekir ve o şey yavaş olabilir ya da çökmüş olabilir. En az yetki rolleri çoğaltır. Beşini birden en üst düzeye çıkaran bir ayar olduğunu düşünmüyorum. Yararlı soru "kimin erişimi olmalı?" değil, "hangi statement'lar, neyin üzerinden ve hangi kayıtla çalışabilir?" sorusudur. Sonraki her bölüm bunun bir dilimini cevaplıyor ve cevabın neye mal olduğunu söylüyor.

İlk bir envanter

Herhangi bir kontrolü benimsemeden önce, var olanı yazardım. Aşağıdaki taslak, Ledgerly'nin başlangıç durumu için bir envanterin biçimini gösteriyor. Amaç, kapı başına bir satır, adlandırılmış bir sahip ve bir gerekçe olması; böylece sahibi olmayan bir satır bir boşluk olarak görünür oluyor.

```
door:      desktop client on engineer laptops
principals: about twenty people (illustrative)
login:     shared admin
reaches:   ledgerly_primary, ledgerly_replica_1, ledgerly_replica_2
network source: company VPN, any laptop
grants:    owns every table
can write?: yes, auto-commit, no row limit
record:    database log shows the shared role only
owner:     nobody
reason:    "needed to fix rows during incidents"
ends:      never
```

Kapı başına bir blok doldurun: her servis, zamanlanmış job, deployment pipeline'ı, laptop'lar, analytics restore'u, paylaşılan depolamadaki dump'lar ve export dosyaları. Çoğu ekip owner ve ends satırlarının boş bırakılanlar

olduğunu görür. Böyle bir envanter bir proje planı değildir. "Primary'ye kim erişebilir?" sorusunu bir günlük kazıdan okuyabileceğiniz bir belgeye çeviren şeydir.

Bu kitabın haritası

Sonraki bölümler erişimin her parçasını ve yukarıdaki her hatayı sırayla ele alıyor. Tablo, bu bölümün ortaya attığı her soruyu kitabın cevapladığı yere bağlıyor.

Soru	Ayak	Nerede cevaplanıyor
Veriyi kimden ve neyden, hangi tür olaylardan koruyoruz?	Üçü de	Bölüm 2
Her principal'a nasıl ayrı bir kimlik ve işe yarayan en dar rolü veririz?	Yetkilendirme	Bölüm 3
Servisler, bir dosyada paylaşılan parola olmadan credential'ları nasıl alır?	Hesap verebilirlik (kimlik)	Bölüm 4
Bağlantı limitleri ve timeout'lar veritabanını koruyan bir politikaya nasıl dönüşür?	Yetkilendirme ve yürütme (sınırlar)	Bölüm 5
Okumalar replica'lara nasıl ulaşır ve verinin diğer kopyalarına ne olur?	Yetkilendirme	Bölüm 6
İnsanlar veritabanına hangi yollarla ulaşır ve her yolda ne durur?	Yetkilendirme (yol), hesap verebilirlik	Bölüm 7
Biri bir görev için yazma erişimini nasıl alır ve bu nasıl biter?	Yetkilendirme	Bölüm 8
Bir statement çalışmadan önce nasıl kontrol edilir ve nasıl commit edilir?	Yürütme	Bölüm 9
Ad-hoc sorgular ve veri düzeltmeleri nasıl review edilebilir kalır?	Yürütme	Bölüm 10
Şemayı kim değiştirir ve hangi erişimle?	Yürütme	Bölüm 11
Hangi veri ek koruma gerektirir ve nasıl maskelenir ya da esirgenir?	Yetkilendirme	Bölüm 12
Backup'lar, export'lar ve kontrolleri dolanan diğer güzergâhlar ne olacak?	Yetkilendirme (yol)	Bölüm 13
Ayakta kalan bir kaydı nasıl tutar ve review ederiz?	Hesap verebilirlik	Bölüm 14

Soru	Ayak	Nerede cevaplanıyor
Bir olay sırasında ne değişir?	Üçü de, baskı altında	Bölüm 15
Önce ne yapıyoruz ve bunu nasıl sürdürüyoruz?	Üçü de	Bölüm 16

Kitabın beş kısmı var: temeller, uygulama erişimi, insan erişimi, veriyi ve kopyalarını korumak, ve hepsini kanıtlamak ve işletmek. Her bölüm, açılışında hizmet ettiği ayağı adlandırıyor. Sırayla okumanız gerekmiyor. Acil probleminiz beş konfigürasyon dosyasındaki paylaşılan parolaysa, başlanacak yer Bölüm 4'tür, ama Bölüm 3 sizi bir sonraki paylaşılan rolü kazara yaratmaktan kurtarır.

Trade-off'lar ve maliyetler

İlk maliyet, **küçük bir gruptan zaman ve dikkat**. Birinin envanteri kurması, rolleri tanımlaması, credential'ları ayarlaması ve review'ları yürütmesi gerekiyor. Küçük bir ekipte bu çoğu zaman bir kişinin yarı zamanlı işidir ve ürün işiyle yarışır. İş hiç takvime alınmazsa, tasarım kâğıt üzerinde kalır. Küçük, bitmiş bir ilk adımı büyük bir plana tercih ederim.

İkinci maliyet, **on-call olan için sürtünme**. Her kontrol, yorgun bir kişi ile bir onarımın arasına bir adım koyar. Dar roller, ilk denemenin bazen en kötü anda bir yetki hatasıyla başarısız olması demektir. Kısa ömürlü erişim, tekrar istemek demektir. Hızlı yolun ne olduğuna önceden karar verin ve onu prova edin. Her olayda atlanan bir kontrol, kontrol değildir; atlamadan önceki bir gecikmedir.

Üçüncü maliyet, **ihtiyaç anında ayakta olması gereken yeni sistemler**. Bir gateway, bir secret deposu ve bir log deposu, her biri çökebilecek başka bir şeydir. Birini benimsemeden önce, o kesinti sırasında veritabanına ihtiyacı olan kişiye ne olacağını sorun ve elinizde bir cevap hazır tutun.

Tam tasarımın başlamak için yanlış yer olduğu durumlar da var. Müşteri verisi olmayan, iki kişinin işlettiği tek servisli bir prototipin bir gateway'e ya da bir review takvimine ihtiyacı yok. Ona, hiçbir kişininkinden ayrı kendi login'i gerekir, çünkü bu ucuzdur, ve ekibin soruyu yeniden ele almasına neyin yol açacağını söyleyen bir not gerekir: ilk müşteri, üçüncü mühendis ya da ilk harici entegrasyon. Şirketiniz için doğru anı

bilmiyorum. Duruma bağı ve neye bağı olduğı şu: verinin ne kadar hassas olduğı, ona kaç principal'ın dokunduğı ve kötü bir statement'ın geri alamayacağınız bir zarar verebilmesinin ne kadar hızlı olduğı.

Son olarak, iyi erişim özelliklerinin bir sınırı var. Blast radius'u azaltırlar ve davranışı görünür kılarlar. Bir veritabanını güvenli yapmazlar ve bu kitaptaki hiçbir bölüm aksini iddia etmiyor. **Blast radius**, bir principal, credential ya da eylem ters gittiğinde ne kadarının etkilenebileceğidir ve bu kitabın işi o yarıçapı sıfır yapmak değil, daha küçük ve daha görünür yapmaktır.

Sık yapılan hatalar

Bunlar kişisel kusurlar değil, tasarım tuzaklarıdır. Her biri bir nedenle caziptir ve her birinin, kimsenin daha dikkatli olmasına bağı olmayan bir çaresi vardır.

- **Parola değıikliğini çare saymak.** Yeni bir parola ilerleme gibi hissettirir ve eski bir iş arkadaşının bilgisini ortadan kaldırır. Ama yirmi kişi ve beş servis hâlâ, diyelim, onu paylaşıyorsa, tasarım değıismemiştir. Çözüm, kimsenin bir parolayı paylaşmasına gerek kalmayan bir tasarımdır ve Bölüm 3 ile 4'ün kurduğı şey budur.
- **Grant'i düzeltip yolu unutmak.** Dar roller yalnızca bağılantılar onları kullanmak zorundaysa işe yarar. Laptop'lar primary'ye hâlâ eski bir admin login'iyle ulaşabiliyorsa, dar roller isteğe bağıdır. Yalnızca her rolün değı, her yolun envanterini çıkarın.
- **Uygulamaları dışarıda bırakmak.** "Kimin veritabanı erişimi var" review'ları çoğı zaman yalnızca insanları listeler, çünkü insanlar bir organizasyon şemasında bulunur. Daha yetkili credential'ları servisler tutar ve onları daha çok kullanır. Her workload'u, her zamanlanmış job'ı ve her betiğı dahil edin.
- **Güvenli yolu güvensiz olandan daha yavaş yapmak.** Production'ı okumak bir bile ve bir gün alırken paylaşılan parola, diyelim, on saniye alıyorsa, paylaşılan parola her seferinde kazanır. Daha yavaş olmayan bir güvenli yolu hedefleyin.

Uyarı: Paylaşılan depolamadaki maskelenmemiş bir dump, başlangıç durumunun müşteri verisine giden en doğrudan yoludur ve klasörü okuyabilen herkese açıktır. Bu dosyaları bulmayı ve kaldırmayı, rollerin herhangi bir yeniden tasarımından önce, ilk envanterin parçası olarak ele alın.

Temel çıkarımlar

- Production veritabanı erişimi bir credential dağıtım problemi değildir; bir yetkilendirme, yürütme ve hesap verebilirlik problemidir ve parola yalnızca ilk soruyu, o da kapıda cevaplar. Erişim beş parçalı bir tasarımdır, yani Kimlik, Grant, Yol, Yürütme ve Kayıt, ve herhangi birindeki bir zayıflık ötekileri baltalar.
- Paylaşılan bir admin hesabı kimsenin verdiği bir karar değildir; birikir ve çaresi, kimsenin bir parolayı paylaşmasına gerek kalmayan bir tasarımdır.
- Uygulamalar insanlardan daha sık ve daha az gözetimle bağlanır; bu yüzden erişimin her ele alınışı, servisleri, job'ları ve araçları insanların yanında principal olarak kapsmalıdır.
- Güvenlik ve operasyonel güvenilirlik, erişim için tek bir sorudur, çünkü yanlış bir WHERE koşulu, sızan bir bağlantı ve kopyalanan bir dump bir tasarım nedenini paylaşır: grant'in, ömrün ya da yolun daraltılmaması ve hiçbir kaydın olmaması.
- İyi erişimin beş özelliği vardır: en az yetki, kısa ömür, gözlemlenebilirlik, geri döndürülebilirlik ve düşük sürtünme; bunlar birbirine karşı çeker, dolayısıyla her kontrol neye mal olduğuyula birlikte sunulmalıdır.
- İlk adım, servisleri, laptop'ları ve veri kopyalarını da kapsayan, kapı başına bir satırlık bir envanterdir; eksik bir sahip ya da eksik bir bitiş tarihi burada görünür bir boşluk olarak ortaya çıkar.
- Kontroller blast radius'u azaltır ve davranışı görünür kılar, ama bir veritabanını güvenli yapmazlar; dürüst hedef daha küçük bir yarıçap ve daha net bir görüştür.

Bir sonraki bölüm, erişimin biçiminden onu korumanın nedenlerine dönüyor. Bölüm 2, kontrollerin kimin ve neyin için olduğunu soruyor, ters gidebilecek olayları üç türe ayırıyor ve Ledgerly için kitabın geri kalanının işaret edebileceği bir tehdit modeli yazıyor.

Bölüm 2 — Production Verisi İçin Bir Tehdit Modeli

Ledgerly'nin güvenlik sorumlusunun, üç aylık bir review'dan önce platform ekibine kısa bir soru gönderdiğini düşünün: "Production veritabanını kimden koruyoruz?" Veritabanlarını işleten Alex bir dakika içinde cevap veriyor: çoğunlukla dışarıdakilerden. Sonra Alex duraksıyor, çünkü cevap ekibin geçen yıla ait kendi biletleriyle uyuşmuyor. Yanlış müşteriye dokunan bir destek isteği. Bir öğleden sonra boyunca fatura yazmalarını yavaşlatan bir raporlama sorgusu. Bir günlük hata ayıklama için bir laptop'ta restore edilen ve kimsenin silmesi hatırlatılmayan bir dump. Amaçlanandan bir kolon fazla çıkan bir vendor export'u.

Bunların hiçbirinde dışarıdan biri yoktu. Hiçbirinde kötü niyet de yoktu. Hepsinde makul bir görevi olan birine açık bir kapı ve görevin ihtiyaç duyduğundan fazlasına izin veren bir rol vardı. Alex'in ilk cevabı, güvenlik gibi duyulan cevaptı. İkinci liste, sistemi anlatan listedir.

Bu bölüm, Bölüm 1'in yaptığı gibi Ledgerly'yi başlangıç durumunda anlatıyor ve kendisi bir kontrol eklemiyor. Eklediği şey, kitaptaki her sonraki kontrolün karşısında seçildiği referans olan yazılı bir tehdit modeli. Tehditleri üç ayağın hepsine karşı çerçeveleyiyor: bir principal'ın ne yapmasına izin verildiği, her statement'ın veriye nasıl ulaştığı ve geriye hangi kanıtın kaldığı. Cevapladığı soru, Alex'in bir dakikada cevaplayamadığı soru: veriye kim ve ne zarar verebilir, hangi kapılardan ve hasar ne kadar uzağa yayılır?

Problem ve neden zor olduğu

Tehdit modeli olmayan erişim tasarımı, en gürültülü endişenin peşinden gitme eğilimindedir. Bu endişe dışarıdan bir saldırıansa, ekip bir çevre koruması kurar: bir VPN, bir firewall kuralı, bir ağ allowlist'i. Bunlar önemlidir. Ama bir çevre koruması "veritabanına kim yaklaşabilir?" sorusunu cevaplar ve bir kişinin ya da servisin içeri girdikten sonra ne

yapabileceği hakkında hiçbir şey söylemez. Ledgerly'nin başlangıç durumunda, paylaşılan parolaya sahip VPN'deki herkes admin rolünün yapabildiği her şeyi yapabilir, yani her şeyi.

Bariz alternatif, bir kontrol listesinden sırayla uygulanan bir kontroller listesidir. Bir kontrol listesi, hangi kapılarınızın en çok risk taşıdığını bilmez. Bir kesinti olmadan kimsenin rotate edemeyeceği bir parolanın rotate edilmesini ister ve her Restricted kolonu çok daha geniş bir kitlesi olan bir depoya kopyalayan gecelik restore hakkında hiçbir şey söylemez. Model olmadan seçilen kontroller, availability'ye ve modaya göre seçilir. Bazıları maruziyeti azaltır. Bazıları güvenlik tiyatrosudur, risk azalmadan sürtünmedir ve ekip hangisinin hangisi olduğunu söyleyemez, çünkü kontrolün ne için olduğunu hiç yazmamıştır.

Yazılı bir modele üç nedenle ihtiyaç var. Birincisi, kapılar teknik nesnelerdir: bir login, bir ağ yolu, paylaşılan depolamadaki bir dosya, bir pipeline değişkeni. Bir politika "erişim uygun olmalı" diyebilir, ama bir login'in neye ulaşabileceğine yalnızca bir tasarım karar verir. İkincisi, en büyük riskler, Önsöz'ün anlattığı gibi, sıradan seçimlerin etkileşiminden doğar: paylaşılan bir parola, otomatik commit eden bir client ve her tablonun sahibi olan bir rol. Hiçbir tek seçim yanlış görünmez, bu yüzden tek tek seçimlerin bir review'u problemi bulmaz. Üçüncüsü, modelin bakımı yeterince ucuz olmalıdır, yoksa bir kez yazılır ve kendi hâline bırakılır.

Burada kullanılan anlamda bir tehdit modeli küçüktür. Veri üzerinde kimin ve neyin hareket edebileceğini, hangi yoldan, neyin ters gidebileceğini, etkinin ne kadar uzağa yayıldığını ve hangi kontrolün buna cevap verdiğini listeler. Bir risk puanı ya da bir tahmin değildir. Her kapı hakkında aynı soruyu sormanın bir yoludur, böylece cevaplar karşılaştırılabilir.

Not: Kitap tehditleri yalnızca bir kontrolü haklı çıkardıkları ölçüde anlatıyor. Bu bölüm, sızan bir credential ya da yanlış veritabanına düşen bir sorgu gibi neyin ters gidebileceğini adlandırıyor ve kimsenin bunu nasıl gerçekleştireceğini anlatmıyor. Daha fazla ayrıntı tasarımı değiştirmeyecekse, dışarıda bırakılıyor.

Veritabanına kim ve ne ulaşır

Bölüm 1, bir production veritabanına erişimin beş parçası olduğunu söyledi: bir kimlik, bir grant, bir yol, bir yürütme ve bir kayıt. Bir tehdit modeli principal'ları listeleterek başlar ve her birine beş soruyu sorar. Ledgerly'de listelemeye değer dokuz tür aktör var. Bunlardan ikisi, vendor ve dışarıdaki saldırgan, kesin anlamda principal değil, listede olmalarının nedeni de bu: listeden çıkarılmaları kolay.

Servisler

Servisler (`billing-api`, `invoice-worker`, `payments-service` ve `admin-console`; `reporting-job` ise bir sonraki başlıkta job olarak ele alınıyor) veritabanının en yoğun kullanıcılarıdır. Gözetimsiz çalışırlar, credential'larını konfigürasyonda tutarlar ve yalnızca kodlarının doğru olduğu ölçüde doğru davranırlar. Bir servis, kimse yanlış bir şey yapmadan da kötü davranabilir: limite ulaşılan kadar bağlantı açan bir retry döngüsü, bir index silindiği için bir tabloyu tarayan bir sorgu, bir staging build'ini primary'ye yönlendiren bir deployment.

Başlangıç durumunda her servis aynı admin login'iyle bağlanıyor; dolayısıyla `billing-api` ve `invoice-worker`'ın her biri yalnızca bir avuç tabloya ihtiyaç duysa da, her servisin blast radius'u veritabanının tamamı. En küçük servisteki bir kusur, en büyüğünün erişim alanına sahip.

Batch job'lar

`reporting-job` ayrı bir satırı hak ediyor, çünkü bir job bir takvimle, çoğu zaman geceleri, kimse izlemeyenken çalışır. Çok veri okuma, uzun transaction'ları açık tutma ve dosya üretme eğilimindedir. Başlangıç durumunda bazen primary'yi okuyor ve export adımı vendor dosyalarını paylaşılan admin login'iyle yazıyor. Normal davranışı zaten toplu erişime benziyor, bu da anormal davranışını bir insanınkinden daha zor fark edilir kılıyor.

On-call mühendisler

On-call rotation'ı, geniş erişime sahip olma baskısını en çok hisseden gruptur ve bunun iyi bir nedeni var: gece gelen bir alarm, kimsenin uyanık olmadığı bir onayı beklemenin zamanı değildir. Başlangıç durumunda paylaşılan admin parolasını, onu hatırlayan ve her statement'ta commit

eden masaüstü client'ları üzerinden kullanıyorlar. Düzeltme aciliyetinin ve bir şeyi değiştirmenin önündeki sürtünmesizliğin birleşimi, bunu kazaların en olası olduğu grup ve tasarımın en iyi koruyabileceği grup yapıyor.

Destek

Destek personeli, aynı anda tek bir müşteriye bakmak için `admin-console`'u kullanıyor. Buradaki tehdit kapsamıdır, kötü niyet değil. Her müşteriye gösterebilen bir araç ve her kolonu okuyabilen bir rol tarafından desteklenmesi, tek faturalık bir soruyu her faturayı gösterebilecek bir sistemle cevaplamak demektir. Tasarım sorusu, yalnızca dar olana ihtiyaç duyulduğunda geniş yeteneği nasıl kullanılamaz kılacağımızdır.

Veri ekibi

Veri ekibi, başlangıç durumunda maskelenmemiş bir dump'ın gecelik restore'u olan `ledgerly_analytics` üzerinde çalışıyor: her Restricted ve Confidential kolon, işi veriye toplu olarak bakmak olan bir grup tarafından erişilebilir. Erişimleri meşru. Endişe, kopyanın primary'den daha geniş bir kitlesi, daha gevşek bir yönetimi ve daha az log'u olması; dolayısıyla primary üzerindeki bir kontrol, kopyada duranı korumuyor.

Vendor'lar ve entegrasyonlar

Harici muhasebe vendor'ı hiçbir Ledgerly veritabanına bağlanmıyor. Dosya alıyor. Bu, tehdidin biçimini değiştiriyor: maruziyet bir login değil bir export ve blast radius'u, export'un içerdiği kolonlar ve kopyanın vendor tarafında ne kadar süre var olduğu. İlgili sorular, dosyanın neyi içerdiği, sonrasında kime ait olduğu ve dosyaya bir kolon eklenirse bunu birinin fark edip etmeyeceği.

CI/CD

Deployment pipeline'ı da bir principal, bir kişi ya da servis değil altyapı olsa da. Başlangıç durumunda migration'ları aynı admin login'iyle çalıştırıyor, çünkü admin rolü her tablonun sahibi ve onları başka hiçbir şey değiştiremiyor, ve credential pipeline'ın konfigürasyonunda duruyor. Herhangi bir tabloyu değiştirebilen ve kalıcı bir credential tutan bir pipeline aynı anda bir ele geçirilme riski ve bir hata riskidir; çünkü kötü bir migration sahibin tüm yetkileriyle çalışır.

Platform ve DBA personeli

Küçük platform grubu veritabanılarını işletiyor ve kontrollerin kendisini değiştirebilen tek aktör o: roller, grant'ler, insanların geçtiği giriş, log. Başlangıç durumunda herkes gibi paylaşılan admin login'ini kullanıyor; dolayısıyla çalışmalarını kimsenin çalışmasından ayıran bir şey yok. Hedef tasarımı, üç principal'ın kendi login'leri üzerinden kullandığı, erişimi yönetmek için bir rolü, `ledgerly_access_admin`'i tutuyor. Bu tehdidi ortadan kaldırmıyor; yerini değiştiriyor. Bu role sahip biri tarafından yapılan yanlış bir grant ya da kapatılmış bir log ayarı, modeldeki diğer her satırın ne anlama geldiğini değiştirir.

Bu nedenle model **control plane**'i başlı başına bir varlık olarak listeliyor: access gateway, credential broker, `ledgerly_access_admin` ve audit store. Veri kontrollerle korunuyor ve kontroller çok daha küçük bir insan ve sistem kümesi tarafından korunuyor. Maruziyet hacim olarak küçük, etki olarak büyük ve sorular her kapıdakiyle aynı: ona kim ulaşabilir, neyi değiştirebilir ve kim fark eder.

Dışarıdaki saldırgan

Dışarıdaki bir saldırgan, bir credential ele geçirene kadar bir principal değildir; bu yüzden onu modellemenin yararlı yolu, ne yapabileceğine değil, neyi elde etmesi gerekeceğine göre modellemektir. Savunanın soruları şunlar: hangi kapılar sızabilecek bir credential kabul ediyor, bu neye ulaşırdı ve nasıl öğrenirdik? Cevaplar diğer sekiz aktörden gelir. Sızan bir credential, sahibinin blast radius'unu devralır.

Üç tehdit sınıfı

Aynı aktörler üç farklı biçimde zarar verebilir ve kontroller her biri için farklıdır. Sınıfları karıştırmak uyumsuz kontrollere yol açar: kötüye kullanım için tasarlanmış bir niyet onayı, bir kazaya karşı pek bir şey yapmaz, çünkü onaylayan görevin meşru olduğu konusunda hemfikiridir ve statement yine de ters gider.

Hatalar

Hata, doğru olması amaçlanan ve olmayan bir eylemdir: yanlış ortama karşı çalıştırılan bir statement, amaçlanandan fazla satırla eşleşen bir koşul, primary için fazla ağır bir sorgu, eski bir konfigürasyonla production'a yöneltilmiş bir betik. Aktör, işini yapan bir principal'dır.

Çoğu ekipte bu sınıfın en sık görüleni olmasını beklerdim, istatistiksel değil yapısal bir nedenle. Kötüye kullanım, bir saiki olan bir kişi gerektirir ve ele geçirilme, bir fırsatı olan bir saldırgan gerektirir, ama bir hata yalnızca bir görevi olan bir kişi gerektirir. Hataların ne kadar daha sık olduğuna dair bir sayım yok ve sunulsa da ona güvenmezdim. Tasarım için sonuç her iki durumda da geçerli: hatalara yönelik kontroller, öteki ikisiyle hiç karşılaşsın ya da karşılaşmasın her ekibin sahip olduğu sınıfa karşı korur.

Hatalar en iyi, tek bir statement'ın erişebildiği şeyi daraltarak ele alınır, çünkü insanlardan sonsuza dek dikkatli olmaları istenemez. İhtiyaç duymadıkları tablolara dokunamayan roller (Bölüm 3), başıboş bir sorguyu durduran timeout'lar (Bölüm 5), her statement'ın çalışmadan önce analizi (Bölüm 9), four-eyes review'lu yazmalar için ayrı bir yol (Bölüm 8 ve 10) ve sonradan veriyi onarmanın review edilmiş bir yolu (Bölüm 10) hep bu sınıfa aittir.

Four-eyes adımı burada yalnızca tek bir koşulla yardımcı olur: reviewer statement'ı ve beklenen etkisini görür. Niyeti onaylamak ("1042 numaralı faturayı düzelt") kötüye kullanım sınıfı için bir kontroldür ve hata sınıfına dokunmaz. Statement'ı ve değiştirmesi gereken satır sayısını onaylamak hataları ele alır.

Kötüye kullanım ve merak

Kötüye kullanım burada, meşru erişimi olan bir principal'ın onu görevin dışında bir şey için kullanması demektir: bir müşterinin kaydını merakından okumak, işin gerektirdiğinden fazlasını export etmek, iş arkadaşlarının verisine bakmak. Düpedüz merakı içerir; bu muhtemelen en yaygın biçimdir ve kötü eylem saymak en zor olanıdır. Nadir kasıtlı vakayı da içerir.

Tasarımın cevabı, ekipten şüphelenmek değil. Bir ekibin, kalıcı bir grant'in gezinmeyi mümkün kıldığını ve iz bırakmadığını fark etmek için kötü niyet varsaymasına gerek yok. Önemli olan kontroller erişimi bir göreve bağlı tutar ve görünür kılar: Restricted kolonları esirgeyen salt okunur roller, Confidential verinin maskelenmiş görünümüleri (Bölüm 12), süreli grant'ler (Bölüm 8) ve bir reviewer'ın gerçekten okuduğu bir audit log (Bölüm 14). Kanıt olmadan, "bu erişim uygun muydu?" sorusu, hiçbir şey yapmamış olanlar dahil kimse için cevaplanamaz.

Ele geçirilme

Ele geçirilme, bir credential'ın ya da bir makinenin amaçlanan principal olmayan birinin elinde olması demektir: bir konfigürasyon dosyasından sızan bir parola, kaybolan bir laptop, güvenmemesi gereken bir şey tarafından okunan bir pipeline değişkeni. Savunanın bunun nasıl olduğunu bilmesine gerek yok, olabileceği gerçeğine göre tasarım yapmak için. Tasarım, credential'ın ne yapabileceğini, ne kadar süreyle yapabileceğini ve kimsenin bunu bilip bilmeyeceğini kontrol eder.

Ele geçirilme, bir credential'ın özelliklerinin en çok önem taşıdığı yerdir. Paylaşılan, statik ve kısıtlanmamış bir credential, tek bir laptop'un kaybını her tablonun kaybına çevirir. Tek bir role kapsamlanmış, kısa ömürlü ve tek bir session'a verilmiş bir credential, aynı olayı sınırlı ve görünür bir olaya çevirir. Bölüm 4 ve 7, credential'ların nasıl verildiğini; Bölüm 3, ne kadar az taşımaları gerektiğini; Bölüm 14 ise nasıl fark edileceğini ele alıyor.

Not: Bir olayı sınıflandırmak çoğu zaman bir muhakeme meselesidir. Bir grup kaydı kapsayan tek satırlık bir düzeltme bir hatadır. Aynı statement, görevi olmayan biri tarafından bilerek çalıştırılırsa kötüye kullanımdır, bir laptop'tan alınan bir credential'la çalıştırılırsa ele geçirilmedir. Veritabanı her seferinde aynı statement'ı görür; bu, bir login'i adlandıran log'lar için ve niyeti tahmin etmeye bağlı olmayan kontroller için bir argümandır.

Neyi koruyoruz

Verinin korumaya değer üç özelliği var ve model her tehdidin hangisine dokunduğunu söylemeli. Bunlar tanıdık gizlilik, bütünlük ve availability'dir ve her birinin Ledgerly'ye özgü bir anlamı var.

Gizlilik, kimin neyi okuyabileceğiyle ilgilidir. Ledgerly için ağırlık eşit dağılmıyor. `iban`, `account_holder_name` ve `token_value` Restricted, dolayısıyla ifşaları geri alınması zor olur. `contact_name`, `contact_email`, `contact_phone` ve `billing_address` Confidential müşteri PII'sidir (kişisel olarak tanımlanabilir bilgi). `invoices` ve `payments` tutarları Internal. Internal veriye dokunan bir tehdit, Restricted veriye dokunandan daha az tasarım emeği hak eder ve model tüm kolonlara eşit davranmak yerine bu farkı göstermelidir. Bu türden veri koruma rejimleri hangi kişisel veriyi tuttuğunuzu ve onu kimin görebildiğini sorma eğilimindedir; bu da aynı farktır; yorum hukuk ve uyum çalışma arkadaşlarınıza aittir.

Bütünlük, verinin doğru kalıp kalmadığıyla ilgilidir. Çoğu tablo için bir hata sonraki bir yazmayla düzeltilebilir, ama `ledger_entries` ve `audit_events` append-only'dir ve defter için bütünlük gizlilikten daha önemlidir: yanlış bir kayıt, `payments-service` üzerinden yeni bir düzeltme kaydıyla giderilir, asla bir düzenlemeyle değil. Bu iki tabloya yönelik bir bütünlük tehdidi bu yüzden türce farklıdır. Aynı zamanda kanıtın en çok önem taşıdığı olandır, çünkü sessizce değiştirilebilen bir defter kimsenin güvenemeyeceği bir defterdir.

Availability, veritabanının ona bağlı servislere hizmet verip vermediğiyle ilgilidir. Güvenlik tartışmalarında en sık unutulmuş özelliktir ve kazaların baskın olduğu yerdir. Uzun bir lock, primary üzerinde büyük bir tabloyu tarayan bir sorgu ya da bir retry döngüsünden gelen bir bağlantı seli, faturalamayı ve ödeme tahsilatını bir saldırının durdurabileceği kadar tamamen durdurabilir. Availability modelde birinci sınıf satırlar alıyor ve bağlantı politikasına (Bölüm 5) ve okumaların nereye gidebileceğine (Bölüm 6) bağlıyor.

Bu üçünün yanında dördüncü bir özellik duruyor: **hesap verebilirlik**, yani kaydın hangi principal'ın ne yaptığını söyleyebilmesi. Bu, erişimin kayıt parçasıdır. Başlangıç durumunda yok, çünkü log yalnızca paylaşılan rolü gösteriyor ve bu yokluk her diğer tehdidin maliyetini artırıyor.

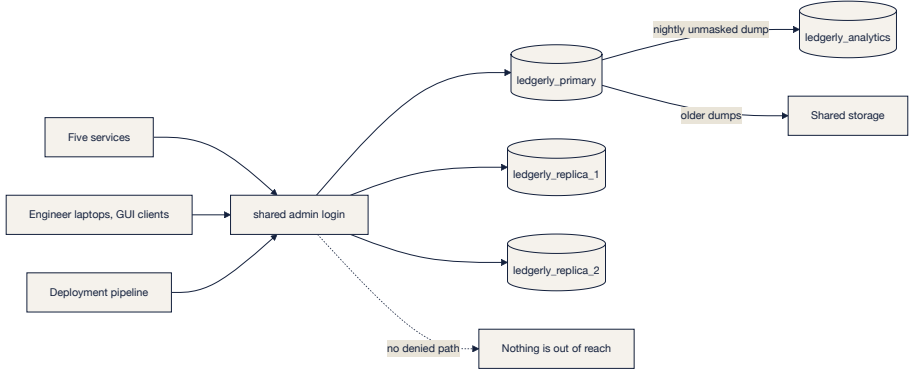
Blast radius

Modelin tehditleri karşılaştırmak için bir aracı daha gerekiyor: tek bir arızanın ne kadar uzağa gittiğini söylemenin bir yolu. **Blast radius**, bir principal, credential ya da eylem ters gittiğinde ne kadarının etkilenebileceğidir. Terim yalnızca belirli bir şey hakkında kullanıldığında yararlıdır, bu yüzden soru her zaman "bu login'in blast radius'u nedir?" olur, asla "bizim blast radius'umuz nedir?" değil.

Bir blast radius'un yedi boyutu vardır ve her birine ayrı ayrı bakmak yardımcı olur:

- **Veri kapsamı:** principal'ın hangi tablolara, kolonlara ve satırlara dokunabildiği.
- **Eylem kapsamı:** okuyabilip yazabildiği, silebildiği ya da şemayı değiştirebildiği.
- **Ortam kapsamı:** hangi veritabanlarına bağlanabildiği: primary, replica'lar, analytics kopyası.
- **Zaman kapsamı:** erişimin ne kadar süre geçerli kaldığı, bir session'dan sonsuza kadar.
- **Kopya kapsamı:** okuyabildiği verinin başka nerede çoğaltıldığı ve bu kopyalara kimin ulaşabildiği.
- **Statement kapsamı:** tek bir statement'ın, bir şey onu durdurmadan önce ne kadar şeyi değiştirebildiği: dokunduğu satırlar, rollback edilebilecek bir transaction içinde çalışıp çalışmadığı ve bir timeout'un onu sınırlayıp sınırlamadığı.
- **Ağ kaynağı:** hangi ağların ve makinelerin credential'ı hiç sunabildiği.

Ledgerly'nin başlangıç durumundaki paylaşılan admin login'i her boyutu sonuna kadar zorluyor. Tüm tablolara ve tüm kolonlara ulaşıyor, okuyabiliyor, yazabiliyor, silebiliyor ve değiştirebiliyor, primary'ye ve iki replica'ya bağlanıyor, süre dolumu yok ve verisinin primary dışında zaten kopyaları var. Her statement sınırsız ve anında commit ediyor ve VPN'deki her laptop credential'ı sunabiliyor. Diyagram nereye bakılacağını gösteriyor: soldaki her şey aynı credential'ı tutuyor, bu yüzden sağdaki her şey tek bir yarıçapın içinde.



Bunu hedef durumdaki `app_reporting_job` rolüyle karşılaştırın: salt okunur, bir replica üzerinde, Restricted kolonlara erişimi yok. Başlangıç durumunda Restricted veriyi ifşa edebilecek bir kusur, o rol için bir replica üzerinde ağır bir sorgudur. Kusur aynı; blast radius değişti.

Karşılaştırmayı dürüst tutmak için, yarıçapı login başına, bir puan olarak değil bir soru listesi olarak yazın ve iki login'i yan yana koyun. Aşağıdaki taslak örnek niteliğindedir; bir inceleme yardımcısıdır ve kitabın üzerine inşa ettiği bir araç değildir. Bilinmeyen bir cevap bilinmeyen olarak kaydedilir, çünkü bilinmeyenleri "sorun yok"a yuvarlayan bir model, var olma nedeni olan vakaları gizler.

dimension	shared admin login	<code>app_reporting_job</code>
data reach	every table	unknown until Chapter 3
lists the tables		
restricted cols	all	none
action reach	read, write, delete, DDL	read
environment	primary, both replicas	one replica
time reach	no expiry	unknown until Chapter 4
decides		
copy reach	unmasked restore, dumps	the export files it
writes		
statement reach	unbounded, commits at once	read-only transaction,
statement timeout		
network origin	company VPN, any laptop	unknown until Chapter 7
decides		

Karşılaştırma eşitsiz olduğu için yararlıdır. İkinci kolonun "unknown" dediği yerde, model tasarımının henüz cevaplamadığı bir soru bulmuştur ve bu bir bulgudur, tablodaki bir boşluk değil.

İki sınır geçerli. Bir yarıçap, bir login'in ne yapabileceğini anlatır, ne yapacağını değil; yani bir tahmin değil bir planlama sınırıdır. Ve her tehdidin ne kadar olası olduğu hakkında hiçbir şey söylemez. Gün boyu açık olan bir kapıdaki dar bir yarıçap, üç ayda bir açılan bir kapıdaki geniş bir yarıçaptan daha fazla dikkati hak edebilir; modelin yarıçapı maruziyetle birleştirmesi gerekmesinin nedeni de bu.

Ledgerly tehdit modeli

Aktörler, sınıflar, özellikler ve blast radius yerine oturunca, model tek bir tabloya sığıyor. Aşağıdaki tablo Ledgerly'nin başlangıç durumunu kapsıyor. Her satır bir aktör ve bir yol; o çift için en önemli tehdit sınıfı, yukarıdaki özellikler üzerindeki etki, kontrolü sağlayan bölümler ve **artık risk**: o kontroller yerindeyken hâlâ doğru olan şey. "Maruziyet", kapının ne sıklıkla kullanıldığının ve kötüye kullanılmasının ne kadar kolay olduğunun niteliksel bir yargısıdır (yüksek, orta ya da düşük) ve biri katılmayabilsin diye yazılır.

Aktör	Yol	Ana sınıf	Etki	Maruziyet	Ele alındığı yer	Artık risk
Servisler (beşi de)	Her servisin konfigürasyonundan paylaşılan admin login'i	Hata, ele geçirilme	Herhangi bir tablo, tüm kolonlar; bağlantılar tükenirse availability	Yüksek	Bölüm 3, 4, 5	Bir kusur hâlâ kendi rolünün erişim alanına sahip
reporting-job	Bazı çalıştırmalarda primary'yi okuyor	Hata	Ağır okumalar yazmalarla yarıyor; primary'nin availability'si	Orta	Bölüm 5, 6	Bir replica üzerindeki ağır bir okuma, diğer replica okuyucularını hâlâ yavaşlatır
On-call mühendisler	Laptop, GUI client'ı, paylaşılan admin parolası, auto-commit	Hata, ele geçirilme	Kapsamsız yazmalar; herhangi bir tablonun bütünlüğü	Yüksek	Bölüm 7, 8, 10	Verilmiş bir rol içinde elle yazılmış bir statement hâlâ yanlış olabilir

Aktör	Yol	Ana sınıf	Etki	Maruziyet	Ele alındığı yer	Artık risk
Destek	Paylaşılan login altında hareket eden <code>admin-console</code>	Kötüye kullanım, hata	Confidential müşteri verisinde gezinme; yanlış müşteri düzenlemeleri	Orta	Bölüm 3, 10, 12	Destek hâlâ bir müşterinin vakasının ihtiyaç duyduğu kolonları görür
Veri ekibi	Maskelenmemiş analytics restore'u ve eski dump'lar	Kötüye kullanım, ele geçirilme	Her Restricted ve Confidential kolonun gizliliği	Yüksek	Bölüm 12, 13	Maskelenmiş veri yine de çıkarım yoluyla değerleri ele verebilir
Muhasebe vendor'ı	Export edilen dosyalar	Hata, kötüye kullanım	Export edilen kolonların gizliliği ve ömürleri	Orta	Bölüm 12, 13	Vendor, teslimattan sonra dosyayı elinde tutar
Deployment pipeline'ı	Pipeline değişkeni olarak saklanan admin login'i	Hata, ele geçirilme	Şema ve veri bütünlüğü; tüm tablolar	Orta	Bölüm 4, 11	Review edilmiş bir migration hâlâ sahip yetkileriyle çalışır
Platform ve DBA personeli	Şimdilik paylaşılan admin login'i; hedef tasarımıda <code>ledgerly_access_admin</code>	Hata, kötüye kullanım, ele geçirilme	Grant'ler, kontroller ve kanıt değiştirilebilir	Orta	Bölüm 3, 7, 14	Rol var, dolayısıyla kötüye kullanımı mümkün; yalnızca tespit kalıyor
Control plane (varlık)	Gateway, credential broker, audit store	Hata, ele geçirilme	Diğer her satırın anlamı değişir	Orta	Bölüm 4, 7, 14	Tek bir bileşenin ele geçirilmesi, tuttuğu kanıt zayıflatır
Herhangi bir insan session'ı	Verilmiş bir rol içinde elle yazılmış bir statement	Hata	Sınırsız statement, anında commit edilir	Yüksek	Bölüm 9, 10	Bir statement geçerli, onaylanmış ve yine de etkisi yanlış olabilir
Herhangi bir ağ kaynağı	Şirket VPN'i primary'ye doğrudan ulaşıyor	Ele geçirilme	Sızan bir credential, VPN'deki her laptop'tan çalışır	Orta	Bölüm 7	İzin verilen bir kaynaktan çalınan bir credential orada hâlâ çalışır

Aktör	Yol	Ana sınıf	Etki	Maruziyet	Ele alındığı yer	Artık risk
Dışarıdaki saldırgan	Yukarıdaki satırlardan herhangi bir sızan credential	Ele geçirilme	Credential sahibinin yarıçapını devralır	Düşükten ortaya	Bölüm 4, 7, 14	Ömrü içinde çalınan bir credential, süresi dolana kadar çalışır
Her aktör	Veritabanı log'unda adlandırılmış login yok	Tüm sınıflar	Diğer her satırın hesap verebilirliği	Yüksek	Bölüm 7, 14	Kayıt, onu tutan depo kadar güvenilirdir

Tablodan dört gözlem çıkıyor ve onu kurmanın nedeni bunlar.

Birincisi, paylaşılan admin login'i satırların çoğunda görünüyor. Tablo, birçok yüzü olan tek bir problemi gösteriyor ve ilk kontrol, principal başına ayrı bir rol (Bölüm 3), onun büyük bir kısmını birden ele alıyor. Hangi sırayla çalışılacağını ortaya çıkaran bir model işini yapıyordur.

İkincisi, en yüksek maruziyetler arasında hiç kapı olması amaçlanmamış kapılar var: analytics restore'u ve eski dump'lar. Hiçbiri, kimsenin veritabanına dair zihnindeki diyagramda bir erişim yolu değil. Bölüm 6 ve 13'ün her veri kopyasını bir sahibi ve bir süre dolumu olan bir kapı olarak ele almasının nedeni bu.

Üçüncüsü, artık risk kolonu tablonun dürüst kısmıdır. Her kontrol bir şey bırakır: verilmiş bir rol hâlâ kötüye kullanılabilir, geçerli bir statement hâlâ yanlış olabilir, bir kayıta ancak deposu kadar güvenilebilir. Artık riski yazılmamış bir satır, kimsenin sormadığı bir satırdır.

Dördüncüsü, control plane, insan session'ı ve ağ kaynağı satırları bir aktörü değil bir varlığı ya da bir yolu adlandırıyor ve son satır da tek bir aktör adlandırmıyor. Eksik hesap verebilirlik bütün sistemin bir özelliğidir ve diğer her satırın maliyetini artırır, çünkü adlandırılmış bir login olmadan hiçbir olayın kapsamı belirlenemez. Bölüm 14 bu satırı kapatıyor. Önceki bölümler, onu kapatmayı mümkün kılan kimlikleri kuruyor.

Uyarı: Tamamlanmış bir tehdit modeli, Ledgerly'nin en zayıf olduğu yerlerin bir haritasıdır. Onu, tarif ettiği sistemlerle aynı erişim kısıtlamalarına sahip bir yerde tutun ve herkese açık bir bilete ya da geniş kitleli bir wiki'ye yapıştırmayın. Maskelenmemiş kopyaları ve paylaşılan credential'ları listeleyen bir belge, onu sürdürenlere yararlıdır ve başka kimseye değil.

Modeli canlı tutmak

Bir kez yazılan bir tehdit modeli bir anlık görüntüdür ve Ledgerly bir entegrasyon eklediği ya da veri ekibi yeni bir kopya aldığı gün yukarıdaki tablo eskimiş olur. Onu, bir runbook gibi, adlandırılmış bir sahibi olan yaşayan bir doküman olarak ele alırdım; tablo zayıflıkları listelediği için, tarif ettiği sistemlerle aynı okuma kısıtlamalarına sahip bir repository'de tutulur. Bunun bir maliyeti var. Erişim konfigürasyonu genellikle daha geniş okunabilen bir repository'de yaşar, bu yüzden orada yapılan bir değişiklik modelin review'unda kendiliğinden görünmez. Bu maliyeti kabul eder ve boşluğu bir kuralla kapatırdım: erişim konfigürasyonunu değiştiren bir pull request, modeldeki ilgili değişikliğe bağlantı verir ya da neden hiçbirine gerek olmadığını söyler.

Bazı olaylar, kimsenin hatırlaması gerekmeden bir güncellemeyi tetiklemeli:

- Production verisine ulaşan yeni bir servis, job, vendor ya da araç.
- Yeni bir veri kopyası: yeni bir replica, bir export, bir staging yenilemesi, tek seferlik bir analiz için bir dump.
- Kimin neye ulaşabildiğinde bir değişiklik: bir yeniden yapılanma, yeni bir on-call rotation'ı, sona eren bir vendor sözleşmesi.
- Hangi satırın yanlış olduğuna dair kanıt olan bir olay ya da ramak kala.
- Hangi varsayımın kaydığına dair kanıt olan bir erişim review'unun bulgusu (Bölüm 14).

Her satır, bir yabancıнын ona itiraz edebileceği kadar kayıt tutmalı: aktör, yol, sınıf, etki, maruziyet yargısı, kontrol ve sahibi, son review tarihi ve satırın dayandığı varsayım. Sonuncusu ekiplerin atladığı ve en yararlı olan

alandır: "vendor export'u yalnızca bu kolonları içerir" kontrol edilebilecek bir iddiadır ve kontrol edilmemiş bir iddiaya dayanan bir satır bir tahmindir.

Küçük bir kontrol, eskimiş satırları bulabilir. Hiçbir şeye karar vermez. Bir insanı yanlış olma olasılığı en yüksek satırlara yöneltir ve eksik bir review tarihini eskimiş sayar. Ayrıca yukarıda listelenen olayları girdi olarak alır: bir pipeline bir login, bir rol ya da bir veri kopyası oluşturduğunda bir olay kaydeder ve hiçbir satırın kapsamadığı bir olay bir review maddesi açar; böylece güncelleme kimsenin hatırlamasına bağlı olmaz.

```
function findStaleThreatRows(rows, maxAgeDays, accessEvents):
  stale = []
  // an event from the pipeline: a new login, role or data copy
  for each event in accessEvents:
    if not anyRowCovers(rows, event):
      stale.append(newReviewItem(event))
  for each row in rows:
    // a row with no review date has never been reviewed
    if row.last_reviewed is null:
      stale.append(row)
    else if now() - row.last_reviewed > maxAgeDays:
      stale.append(row)
    // a control with no owner is a control nobody maintains
    else if row.control_owner is null:
      stale.append(row)
  if count(stale) > 0:
    notify("platform", "threat model rows need review")
  return stale
```

Kadans bir trade-off'tur. Üç aylık bir geçiş, küçük bir ekibin sürdürebileceği bir şeydir ve yıllık bir geçiş nadiren yeterlidir, ama aralığı Ledgerly'nin kapılarının ne kadar hızlı değiştiğine göre seçerdim. Güvenlik ve uyum sorumluları review'un sorduğu soruları belirler ve platform ekibi onları sistem hakkındaki olgularla cevaplar.

Trade-off'lar ve maliyetler

Bir tehdit modelini yazmak zaman, sürdürmek daha fazla zaman alır. Küçük bir ekip için ilk sürüm makul olarak birkaç günlük dikkatli tartışma alır ve bakım, özellikle işiyle yarışan yinelenen bir maddedir. Kimse sahip olmazsa, maliyet bir kez ödenir ve fayda erir. Tek servisli ve tek veritabanlı

çok küçük bir sistemde tam tablo problemin ihtiyacından fazladır; kapıları ve her birinin en kötü durum yarıçapını listeleyen yarım bir sayfa yeterlidir.

Model ayrıca yanlış bir güven yaratabilir: derli toplu bir tablo her tehdidin üzerinde olduğunu düşündürür. Birinin aklına gelen tehditleri listeler ve listelenmemiş bir kapı ona görünmezdir. En iyi savunma yukarıdaki güncelleme tetikleyicileri ve her olaydan sonra şunu sorma alışkanlığıdır: "hangi satır bunu öngörürdü ve hiçbiri öngörmediyse, eksik satır nedir?"

Maruziyet yargıları öznelidir. İki mühendis aynı kapıyı farklı derecelendirir ve anlaşmazlık bilgilendiricidir, ama bir derecelendirme, kanıtın desteklediğinden daha fazla kesinlik ima eder. Onu üç seviyeyle sınırlardım ve puan hesaplamaya direnirdim.

İnsan gruplarını aktör olarak listelemenin insani bir maliyeti var. On-call mühendislerle ya da desteğe tehdit aktörü demek bir suçlama gibi okunabilir. Cevap çerçevelemede: model bireyleri değil yolları ve rolleri tarif eder ve girdileri tasarımın neye izin verdiğini söyler. Tablodaki ekipler, veriye ulaşmak için en meşru ihtiyaca sahip oldukları için orada.

Tam modelin ne zaman kullanılmayacağı: şu anda kimse "kimin erişimi var?" sorusunu cevaplayamıyorsa, tehditleri yazmadan önce login'lerin ve veri kopyalarının bir envanteriyle başlayın, çünkü yanlış bir envantere dayanan bir model var olmayan bir sistemi tarif eder. Ve onu bir kararın yerine kullanmayın. Sıralar ve açıklar; hangi kontrolleri önce kuracağınızı ve hangi riskleri yazılı olarak kabul edeceğinizi yine ekip seçer.

Sık yapılan hatalar

- **Yalnızca dışarıdaki saldırıya göre modellemek.** Güvenliğin yaygın resmine uyar ve kontroller tanıdaktır. Maliyeti, hataların ve merakın ele alınmadan kalmasıdır. Üç sınıfın her birine tabloda kendi satırlarını verin.
- **Tehditler yerine kontrolleri listelemek.** "Bir VPN'imiz var" soru olmadan bir cevabı kaydeder. Önce neyin ters gidebileceğini, sonra kontrolü yazın; böylece tehdidi artık geçerli olmadığında bir kontrol emekliye ayrılabilir.

- **İnsan olmayan aktörleri dışarıda bırakmak.** Pipeline'ların, job'ların ve export'ların gözden kaçırılması kolaydır çünkü kimse onlar olarak login olmaz ve en çok kalıcı yetkiyi tutma eğilimindedirler. Her principal'ı ve her export'u listeleyin.
- **Veriye tek biçimde hassas gibi davranmak.** Her kolon "kritik" ise, model nereden başlanacağına dair hiçbir rehberlik vermez. Dört veri sınıfını (Restricted, Confidential, Internal, Public) kullanın; böylece bir tehdidin ağırlığı verinin ağırlığını izler.
- **Availability'yi atlamak.** Başboş bir sorgu bir güvenlik olayına benzemez, bu yüzden güvenlik review'larından kaçır. Bir ödeme işi için bu, en az bir güvenlik olayı kadar önemli olabilir. Availability'ye kendi satırlarını verin.
- **Bir kez yazmak.** Model birkaç hafta doğrudur, sonra ağızdan ağıza aktarılan bir bilgiye dönüşür. Bir sahip atayın ve güncelleme tetikleyicileri ekleyin.

Temel çıkarımlar

- Bir tehdit modeli; aktörlerin, yolların, tehdit sınıflarının, etkilerin ve kontrollerin kısa yapılandırılmış bir listesidir ve işi, bir ekibin kapıları birbirine karşı karşılaştırmasını sağlamaktır.
- Hatalar, kötüye kullanım ve ele geçirilme farklı kontroller gerektirir ve yalnızca dışarıdakileri dikkate alan bir tasarım ilk ikisini ele almadan bırakır; niyeti onaylamak kötüye kullanımı ele alırken, statement'ı ve beklenen etkisini onaylamak hataları ele alır.
- Her aktör listede olmalıdır; pipeline'lar, batch job'lar, vendor export'ları ve kontrolleri işleten platform personeli dahil, çünkü kimsenin onlar olarak login olmadığı principal'lar çoğu zaman en çok kalıcı yetkiyi tutar ve control plane de bir varlıktır.
- Gizlilik, bütünlük ve availability'nin her biri kendi satırlarına ihtiyaç duyar ve hesap verebilirlik, yani kaydın hangi principal'ın ne yaptığını söyleyebilme yeteneği, eksik olduğunda diğer her tehdidin maliyetini artırır.
- Blast radius her zaman belirli bir login, credential ya da eylem hakkındadır ve yedi boyutla tarif edilir: veri, eylem, ortam, zaman, kopyalar, statement ve ağ kaynağı.

- Ledgerly'nin başlangıç durumunda tek bir paylaşılan admin login'i neredeyse her satırda görünüyor; dolayısıyla her principal için ayrı bir rol, tablonun en büyük kısmını birden küçülten kontroldür.
- Model ancak bakımı kadar iyidir: ona bir sahip verin, güncellemeleri yeni servisler ve yeni veri kopyaları gibi olaylara bağlayın ve onu yalnızca sistemi sürdüren kişilerin okuyabileceği bir yerde tutun.

Model size hangi kapıların en çok önem taşıdığını söylüyor ve Ledgerly'de neredeyse hepsi tek bir login'e çıkıyor. Bölüm 3 o login'i parçalamaya başlıyor: principal'ların kimler olduğunu, bir rolün neyi tutması gerektiğini ve her birinin ne kadar az şeye ihtiyaç duyduğunu.

Bölüm 3 — Kimlikler, Roller ve En Az Yetki

Ledgerly'de bir perşembe akşamını düşünün. `invoice_disputes` adlı yeni bir tablo salı günü yayına girdi ve güvenlik sorumlusu platform ekibine kısa, makul bir soru soruyor: bunu kim okuyabilir? Alex grant'lere bakarak başlıyor ve hiç grant bulamıyor, çünkü her çağırın `admin` olarak bağlanıyor; tablonun sahibi de bu, her şeyi okuyabiliyor. Alex bunun üzerine tabloyu gerçekte kimin okuduğunu arıyor. Veritabanı log'unda tabloya dair binlerce satır var ve hepsini aynı login yazmış: worker, gece raporu, laptop'larındaki iki mühendis ve destek aracı.

Yeni tabloyu kimse yerine oturtamıyor. "Bunu kim okuyabilir?" sorusunun dürüst cevabı "veritabanına ulaşabilen herkes", "Kim okudu?" sorusunun dürüst cevabı ise "biri". Kimse yanlış bir şey yapmadı. Tasarım, veritabanından bu çağırınları birbirinden ayırmasını ya da herhangi birini sınırlamasını hiç istemedi.

Bölüm 1, erişimi beş parçaya ayırdı: **Kimlik**, **Grant**, **Yol**, **Yürütme** ve **Kayıt**. Kitabın tezi, erişimin bir credential dağıtım problemi değil, bir yetkilendirme, yürütme ve hesap verebilirlik problemi olduğudur. Bu bölüm yetkilendirme ayağına hizmet eder. İlk iki parçayı, yani kimin istediğini (Kimlik) ve o principal'ın ne yapabileceğini (Grant) ele alır; grant'lerin nerede bittiğini de işaretler, çünkü orada Yürütme parçası devralır (Bölüm 9).

Kitabın yürüyen örneğinden ilk hedef durum adımlarını ekler. Her tablonun sahibi `ledgerly_owner` olur ve kimse owner olarak bağlanmaz. Her servis, işe yarayan en dar grant'lerle kendi application rolünü alır. Bölüm ayrıca tam rol şemasını, principal'ların rollere ulaşma biçimini, kolon düzeyindeki grant'leri ve just-in-time erişimin tavanını da tanıtır. Sonraki bölümler bu tanımların üzerine kurulur ve onları tekrar etmez.

Problem ve neden zor olduđu

Bir veritabanı her statement'ta iki soruya cevap verir: bu kim ve bu principal bunu yapabilir mi? Paylaşılan bir hesap birinci cevabı anlamsız kılar, geniş yetkili bir hesap ise ikinci cevabı her zaman "evet" yapar. İkisine de varmak kolaydır. Bir ekip ilk veritabanını oluşturduđu gün, tek login, tek parola ve tam yetki, işe yarayan en ucuz şeydir.

Bedel sonra, eksik kanıt olarak gelir. Admin login'ini paylaşan insanlar ve servisler (diyelim yirmi kişi, beş servis, deployment pipeline'ı ve export adımı) hepsi onu kullandığında, veritabanı log'u bir statement'ı hangisinin çalıştırdığını söyleyemez. Application log'u, uygulamanın ne yapmak istediğini anlatır; veritabanına gerçekte neyin ulaştığını değil. Bu bir disiplin problemi değil, bir tasarım problemidir; çünkü tasarımın attığı bilgiyi hiçbir bireysel özen geri getirmez.

Bariz çare bir politika yazmaktır: yalnızca gerekeni grant et. Bir politika burada işe yaramaz, çünkü veritabanı geniş seçeneği kolay seçenek yapar. Her şeyi grant etmek tek bir statement'tır ve asla başarısız olmaz. İşe yarayan en dar kümeyi grant etmek ise bir workload'un çalıştırdığı her sorguyu bilmeyi gerektirir; kimsenin hatırlamadığı aylık job ve yılda iki kez çalışan retry yolu da buna dahildir. Fazla dar bir grant production'ı en kötü anda bozar. Fazla geniş bir grant hiçbir şeyi bozmadı, dolayısıyla kimse fark etmez. Bu asimetri altında yetkiler yalnızca büyür.

Sahiplik ikinci bir tuzak ekler. Çoğu engine'de bir tablonun owner'ı yapısını değiştirebilir, tabloyu silebilir ve başkalarına erişim grant edebilir. Bir servis startup'ta tablo oluşturduđu için owner olarak bağlanıyorsa, servis veritabanının erişim modelini yeniden yazma gücünü elinde tutar. Owner bir insansa, model o insanın işte kalmasına bağlı olur. Sahiplik yapısal bir yetkidir ve trafiği işleyen hiçbir şeyde, bir kariyeri işleyen hiçbir şeyde de durmamalıdır.

Çare bir talimat değil, bir yapıdır: principal başına ayrı bir login, yetkileri taşıyan küçük bir rol kümesi ve ikisini de ekibin amaçladığıyla uyumlu tutmanın bir yolu.

Principal'lar, login'ler ve roller

Bu bölüm Kimlik ve Grant üzerinde çalışır; bir rolün nereden bağlanabileceğini söylediği yerde Yol'a da değinir. **Principal**, kimlik doğrulayan her şeydir. Ledgerly'nin dünyasında ayırmaya değer dört tür var.

- **İnsanlar.** On-call backend mühendisleri, destek çalışanları, veri ekibi ve platform ekibi. Gelip giderler, ekip değiştirirler ve farklı yerlerden çalışırlar.
- **Servisler.** `billing-api`, `invoice-worker` ve `payments-service`. Uzun süre çalışırlar, çok sayıda instance olarak koşarlar ve en yoğun trafiği taşırlar.
- **Job'lar.** `reporting-job` ve `export` adımı. Bir zamanlamaya göre çalışır ve biterler. Bir job genellikle bir servisten daha azına, daha kısa bir süre için ihtiyaç duyar.
- **Araçlar.** `admin-console` en açık örnektir. Bağlanış biçimiyle bir servistir, ama bir kişi adına hareket eder; bu yüzden her eylemde iki principal vardır: araç ve arkasındaki insan.

Harici muhasebe sağlayıcısı hiçbir Ledgerly veritabanına bağlanmaz; dosya alır ve bu yolu Bölüm 13 ele alır.

Kimlik, login ve rol üç ayrı şeydir

Kitap üç sözcüğü ayrı tutar, çünkü bunları karıştırmak erişim tasarımındaki karışıklığın çoğunun kaynağıdır.

Kimlik, bir principal'ın söylediği kişi olduğuna dair doğrulanmış iddiadır. **Login**, bir principal'ın bağlandığı veritabanı düzeyindeki hesaptır. **Rol**, adlandırılmış bir yetki demetidir. Bir principal'ın kimliği vardır, bir login üzerinden bağlanır ve login bir ya da daha fazla rolün üyesidir. Veritabanı log'unda görünen login'dir. Login'in ne yapabileceğine karar veren roldür.

Bu, modele hatırlamaya değer bir cümle kazandırır: kimse bir rol olarak login olmaz. Bir rol, login'lere grant edilmek ve yetki taşımak için vardır ve asla bir insan değildir. Log, bir insanın ya da workload instance'ının olması gereken yerde bir rol adı gösteriyorsa, model bir yerde atlatılmıştır.

İnsanlar için access gateway (Bölüm 7), kişinin kimliğini taşıyan, session başına bir login oluşturur ve login varsayılan olarak `human_readonly` alır. Servisler için her instance, kendi `app_*` rolünün üyesi olan bir login ile bağlanır. Hedef durumda bu login dinamik ve kısa ömürlüdür (Bölüm 4). Başlangıç durumunda ise paylaşılan admin login'idir. Merkezi soru açısından sonuç açıktır: log bir login'i adlandırır, bir login de tek bir kişiye ya da tek bir workload instance'ına karşılık gelir; dolayısıyla "o statement'ı kim çalıştırdı?" sorusunun bir cevabı vardır.

Paylaşılan hesap neden kanıtı yok eder

Bölüm 1 bunu tam olarak savundu; kısa hâli şu: paylaşılan bir hesap bir statement'ı kimseye atfedemez, tek bir kişi için revoke edilemez ve en geniş çağırının ihtiyacının altına daraltılamaz.

Bir yol dikkat ister: `app_admin_console` bir workload'dur, ama kullanıcıları insandır. Console, her eylemle birlikte insan principal'ın kimliğini iletir ve aşağıdaki taslak bunun nasıl yapıldığını gösterir.

```
function consoleAction(person, action):
    correlationId = newCorrelationId()
    outcome = "failed"
    // set on the session the console borrowed, cleared when it is
    returned
    setSessionContext({ actor: person.id, correlation: correlationId
    })
    try:
        run(action)
        outcome = "done"
    finally:
        clearSessionContext()
        // a failed or half-finished action is recorded too
        auditLogAppend({ actor: person.id, correlation: correlationId,
                        workload: "admin-console", action:
        action.name,
                        result: outcome })
```

Veritabanı login'i workload'u adlandırır; session context ve `audit_events` ise kişiyi ve correlation id'yi taşır. Böylece application log'u, audit trail ve statement'ın kendi kaydı tek bir anahtarı paylaşır. Kayıtların nasıl birleştirildiğini Bölüm 14 ele alır. Context'i console ayarlar; yani bu, veritabanının doğruladığı bir olgu değil, workload'un ileri sürdüğü bir iddiadır. Satır kuralı bölümündeki pooler uyarısı bunun için de geçerlidir.

Dört katmanlı bir rol hiyerarşisi

Rol şemasını dört katmanda kurardım: kimsenin kullanmadığından başlayıp insanların kullandıklarına doğru. Her katmanın tek bir işi var.

Birinci katman: owner

`ledgerly_owner` her şema nesnesinin sahibidir: şemalar, tablolar, view'lar ve sequence'ler. Onunla kimse login olmaz. Var olma nedeni, sahipliğin ne bir insana ne de bir servise bağlı olmasıdır.

```
CREATE ROLE ledgerly_owner;  
// no login is ever created for this role; objects are created  
// under it only by a migration that assumes it  
  
CREATE ROLE ledgerly_migrator;  
GRANT ledgerly_owner TO ledgerly_migrator;  
// membership grants the owner's privileges, not ownership: objects  
// belong to the active role, so the migration always assumes  
// ledgerly_owner explicitly, on every engine
```

İkinci katman: migrator

`ledgerly_migrator` şema değişikliklerini çalıştırır. Deployment pipeline'ı onun login'ini kullanır, insanlar kullanmaz ve kendi başına hiçbir veri yetkisi taşımaz. `ledgerly_owner`'ın üyesidir. Bazı engine'lerde bu üyelik, owner'ın yetkilerini herhangi bir açık adım olmadan aktarır, ama sahipliği aktarmaz: yeni bir nesne, oluşturulduğu anda hangi rol aktifse ona ait olur; bu yüzden migration her zaman `ledgerly_owner`'ı açıkça üstlenir.

Migrator kalıcı owner yetkileri taşır ve bunları kısa ömürlü bir credential üzerinden kullanır. Yetkiler asla sona ermez; yalnızca her bir credential sona erer. Üç şey onu sınırlar: credential yalnızca pipeline'da yaşar, kısa ömürlüdür (Bölüm 4 ve 11) ve pipeline prosedürü onu kullanmanın tek onaylı yoludur. Şema değişiklikleri, hiçbir insan ya da servis sahipliği elinde tutmadan mümkündür; bu sınırlar tutuyorsa owner'ın her kullanımı bir pipeline çalışmasına karşılık gelir. Migration'ların nasıl review edildiğini Bölüm 11 ele alır.

Bir şema değişikliğine ait backfill'ler migration'ın içinde

`ledgerly_migrator` olarak çalışır. Verideki diğer her toplu değişiklik bir veri düzeltmesidir ve Bölüm 10'u izler. Bu çizgiyi çekiyorum, çünkü

migrator'ın gücü büyüktür ve her sakıncalı değişikliğin güzergâhı hâline gelmemelidir.

Üçüncü katman: application rolleri

Her workload kendi rolünü alır. Servis başına bir rol, beş kod tabanıyla uyumlu tutulacak beş tanım demektir. Karşılığında her credential'ın blast radius'u o servisin işi kadardır: `invoice-worker` için olan sızarsa, worker'ın yapabildiği şeyi açığa çıkarır ve worker'ın `payment_tokens`'a hiç ihtiyacı olmadı.

Roller `app_billing_api`, `app_invoice_worker`, `app_payments_service`, `app_reporting_job` ve `app_admin_console`'dur. Önek onu kimin kullandığını söyler: workload'lar için `app_`, insanlar için `human_`. Roller kişi adı ya da ortam adı taşımaz.

Dördüncü katman: insan rolleri

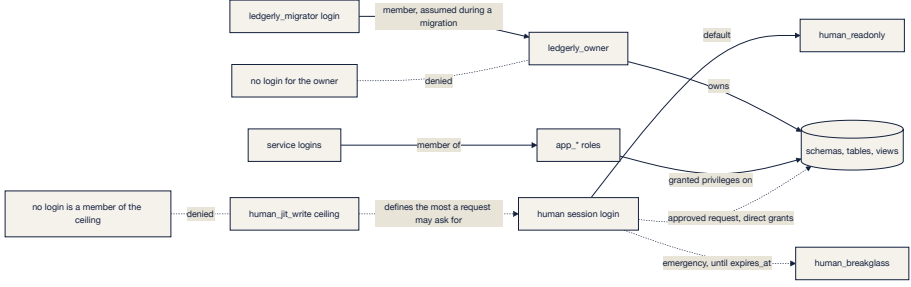
İnsanlar üç rol alır ve bunlar nasıl tutuldukları bakımından birbirinden farklıdır.

- `human_readonly` varsayılandır. İnceleme için okur, Restricted kolonları saklar ve Bölüm 12'nin söylediği yerlerde Confidential kolonları maskeler.
- `human_jit_write` bir tavandır. Hiçbir login'e grant edilmez. Onaylanan her istek, bunun bir alt kümesini, adı geçen tablolarla ve adı geçen kolonlarda, doğrudan session login'ine grant eder ve sonra revoke eder.
- `human_breakglass`, yoğun log'lanan geniş bir acil durum erişimidir (Bölüm 8 tanımlar) ve sonrasında zorunlu review gerektirir.

Üçünün içinde en az sezgisel olanı tavandır ve grant mekanizminden sonra ona döneceğim.

Diyagram

Aşağıdaki diyagram kimin neyin üyesi olduğunu ve kimin neye sahip olduğunu gösterir. Bağlı olmayan iki şeye bakın: `ledgerly_owner` için hiçbir login yoktur ve hiçbir login `human_jit_write`'in üyesi değildir.



Grant'ler: şema, tablo ve kolon

Grant, bir role bir yetki vermenin, bir login'e ya da başka bir role bir rol vermenin ya da yalnızca JIT erişim için bir session login'ine doğrudan bir yetki vermenin hem eylemi hem de ortaya çıkan kayıdır. **Yetki** (privilege), bir grant'in izin verdiği belirli şeydir; örneğin tek bir tabloda SELECT. Üç anlam sözlüğü kesin tutar ve bu bölümde hangisini kastettiğimi söyleyeceğim.

Hiçbir şeyden başlayın

Her rolün başlangıç noktası hiç yetki olmamasıdır. Workload çalışana kadar grant ekleyin ve durun. Engine'ler, yeni bir login'in varsayılan olarak ne yapabildiği bakımından farklıdır: bazılarında herhangi bir login nesne adlarını görebilir ya da varsayılan bir şemada nesne oluşturabilir. Her login'in otomatik olarak aldığı şeyleri listeleyin ve bir rolün ihtiyaç duymadıklarını kaldırın.

Yetkiler üç düzeyde uygulanır ve toplanırlar: bir rol, herhangi bir düzeyde kendisine grant edilen her şeyin birleşimini taşır ve dar bir grant, geniş bir grant'i asla ortadan kaldırmaz.

- **Şema düzeyi.** Bir rolün bir şemadaki nesneleri görüp göremeyeceği. Bunu gerektiren engine'lerde, tablo düzeyindeki grant'ler onsuz hiçbir işe yaramaz; bu da şema erişimini kullanışlı bir kaba anahtar yapar.
- **Tablo düzeyi.** Bütün bir tabloda SELECT, INSERT, UPDATE ve DELETE.
- **Kolon düzeyi.** Aynı yetkiler, yalnızca adı geçen kolonlarda.

Kolon düzeyindeki grant'ler önemlidir, çünkü hassas veri eşit dağılmamıştır: customers Confidential iletişim bilgilerini, bank_accounts Restricted iban ve account_holder_name'i, payment_tokens ise

Restricted token_value'yu tutar. Tablo düzeyinde bir grant, tablonun her kolonuna evet der. Yetkiler toplandığı için kolonları kısıtlamak, yalnızca kolon düzeyinde grant etmek ve rolün o tabloda, doğrudan ya da üyesi olduğu başka bir rol üzerinden, hiçbir tablo düzeyi grant'i taşıması demektir.

Application grant'leri

İlk iki application rolü, tablo düzeyinde:

```
GRANT SELECT, INSERT, UPDATE ON customers, invoices, invoice_lines TO
app_billing_api;
GRANT SELECT ON payments, ledger_entries TO app_billing_api;
GRANT INSERT ON audit_events TO app_billing_api;

GRANT SELECT, INSERT, UPDATE ON invoices, invoice_lines TO
app_invoice_worker;
GRANT SELECT ON customers TO app_invoice_worker;
GRANT INSERT ON audit_events TO app_invoice_worker;

// not granted to either role: bank_accounts, payment_tokens,
// UPDATE or DELETE on ledger_entries and audit_events
```

app_billing_api, ledger_entries ve payments'ı okur ama yazamaz; çünkü payments, bank_accounts ve payment_tokens'a yalnızca payments-service yazar ve ledger_entries'e yalnızca o insert eder. İki rol de hiçbir şeyde DELETE taşımaz; bir servisin buna ihtiyacı varsa grant, bir gerekçeyle o tablo için eklenir ve review bunu fark eder.

Diğer roller için kolon düzeyi grant'ler

Matriste hançer işareti olan roller kolon kısıtlıdır. Aşağıdaki kolon listeleri örnek amaçlıdır: hangi Confidential kolonların ve hangi maskelenmiş biçimde görüldüğü Bölüm 12'nin kararıdır.

```
// illustrative column lists
GRANT SELECT (id, created_at, status) ON customers TO
app_reporting_job;
GRANT SELECT (id, customer_id, amount_minor, currency, status)
    ON payments TO app_reporting_job;
GRANT SELECT ON invoices, ledger_entries TO app_reporting_job;

GRANT SELECT (id, created_at, status) ON customers TO
app_admin_console;
GRANT SELECT, UPDATE (status) ON invoices TO app_admin_console;
// UPDATE (status) is a column-level grant: the tool can change an
invoice's
// status and no other column of the row
```

Son grant en çok sevdiğimdir. `app_admin_console` bir invoice'ın `status`'unu değiştirebilir ve satırda başka hiçbir şeyi değiştiremez; grant niyeti ifade eder ve aracın kodu sonradan değişse bile veritabanı bunu uygular. `app_reporting_job` hiçbir ana tabloda yazma yetkisi taşımaz: sonuçlarını `ledgerly_analytics`'e yazar.

Not: Fiziksel olarak replike eden engine'lerde roller bütün cluster için bir kez tanımlanır; bu yüzden her rolün nereden bağlanabileceği bir bağlantı katmanı kuralıdır (Bölüm 5 ve 7), grant'lerin ifade ettiği bir şey değildir. Ayrıntıyı Bölüm 6 verir.

Referans olarak matris

Aşağıdaki tablo, ana tabloları okuyan ya da onlara yazan altı rol için hedef durum referansıdır. S SELECT, I INSERT, U UPDATE, D DELETE demektir; tire erişim yok demektir; hançer işareti (†) kolon kısıtlı demektir: Restricted kolonlar saklanır ve Confidential kolonlar Bölüm 12'nin söylediği yerlerde maskelenir.

Tablo	app_ billing_ api	app_ invoice_ worker	app_ payments_ service	app_ reporting_ job	app_admin_ console	human_ readonly
customers	S, I, U	S	S	S†	S†	S†
bank_ accounts	—	—	S, I, U	—	—	—
payment_ tokens	—	—	S, I, U, D	—	—	S†
payments	S	—	S, I, U	S†	S†	S†

Tablo	app_ billing_ api	app_ invoice_ worker	app_ payments_ service	app_ reporting_ job	app_admin_ console	human_ readonly
invoices	S, I, U	S, I, U	S	S	S, U (yalnızca status)	S
invoice_ lines	S, I, U	S, I, U	—	S†	S†	S†
ledger_ entries	S	—	S, I	S	—	S
audit_ events	I	I	I	—	I	S

Matrisi kolon kolon okurum. Her kolon, yetki olarak yazılmış bir workload'un iş tanımıdır ve bir reviewer bunun hâlâ doğru olup olmadığını sorabilir. Baştan sona bakıldığında, `bank_accounts`'a yalnızca `payments-service` ulaşır, `human_readonly`'nin kolon kısıtlı okuması dışında `payment_tokens`'a yalnızca `payments-service` dokunur ve hiçbir insan rolü varsayılan olarak yazmaz.

Append-only yapısaldır

Hiçbir application rolü, hiçbir insan rolü ve hiçbir JIT grant'i `ledger_entries` ya da `audit_events` üzerinde `UPDATE` veya `DELETE` taşımaz; buna `human_breakglass` da dahildir. Bir düzeltme yeni bir kayıttır, asla bir düzenleme değil. Ledger düzeltmeleri yalnızca `payments-service`'in düzeltme yolu üzerinden yapılır, asla bir insan login'ile yapılmaz.

Tek bir yapısal istisna vardır: `ledgerly_owner` ve dolayısıyla çalışan bir migration bu tabloları değiştirebilir, çünkü owner tabloyu değiştirebilir ya da silebilir ve kendine üzerinde her yetkiyi grant edebilir. Bölüm 11'in bunlara dokunan her migration'ı review etmesinin nedeni budur. Kontrol review'dadır, yetkide değil.

Varsayılanlar, sahiplik ve superuser

Default yetkiler

Birçok engine, bir grant'in sonradan oluşturulan tablolara da uygulanmasını sağlayabilir; ister bir owner'a bağlı bir default yetki olarak, ister şema genelinde bir grant olarak. Bu başlangıçta kullanışlıdır, sonradan bir tuzaktır; çünkü yeni bir tablo, grant ne diyorsa onu sessizce devralır.

Tercihim boş bir varsayılandır. Yeni bir tabloyu owner dışında kimse okuyamaz ve onu oluşturan migration her grant'i aynı dosyada belirtir. Bedel, migration başına birkaç satır fazladır; kazanç ise tek bir review'un bir değişikliğin bütün erişim etkisini göstermesidir.

Uyarı: Gelecekteki tablolarda `human_readonly` gibi geniş bir role `SELECT` grant eden bir default yetki, bir sonraki hassas tabloyu oluşturulduğu anda açığa çıkarır. Önce bu tür varsayılanları review edin ve gelecekteki tabloları kapsayanları kaldırın.

Gösterim örnek amaçlıdır, çünkü default yetkilerin tarafsız bir sözdizimi yoktur.

```
// illustrative notation: defaults for objects created by
ledgerly_owner
DEFAULT PRIVILEGES FOR ledgerly_owner ON TABLES: none

// each migration that creates a table states its own grants
CREATE TABLE <new_table> (...);
GRANT SELECT ON <new_table> TO app_billing_api;
```

Sahipliğin tehlikesi

Yaygın engine'lerde owner bir nesneyi silebilir, tanımını değiştirebilir, herkese yetki grant edebilir ve çoğu durumda başkaları için geçerli olan satır düzeyi kısıtlamaları yok sayabilir. Sahipliği elinde tutan, erişim modelinin kendisini değiştirebilir.

Bu yüzden `ledgerly_owner` etrafındaki kurallar sıkıdır ve bu sıkılık ucuzdur. Onun için hiçbir login yoktur. Application rolleri, kendi kullanımları için oluşturdukları tablolar dahil, asla tablo sahibi olmaz;

çünkü "servis tablosunu startup'ta oluşturuyor" bir workload'un şemanın bir kısmına sahip olmasının yoludur. Owner, sistemdeki en az kullanılan roldür ve doğru durum da budur.

Superuser

Çoğu engine'in yetki kontrollerini atlayan yerleşik bir yönetim rolü vardır; managed servisler bunu vermeyebilir. Burada buna superuser diyorum; engine ile birlikte geldiği için Ledgerly rol şemasının dışındadır.

Ben onu mühürlü bir alet olarak ele alırdım. Günlük işte hiçbir uygulama, job ya da insan session'ı onu kullanmaz. Credential'ı, platform ekibinin alabileceği bir yerde tutulur ve alınması bir kayıt bırakır; hiçbir rolün yapamadığı engine düzeyindeki işlemler içindir, örneğin replication'ı onarmak gibi. Dürüst sınır, bu bölümdeki her kontrolü geri alabilmesidir ve toparlanmayı imkânsız kılmadan bu gücü ortadan kaldıran bir tasarım bilmiyorum. Kullanımını nadir, görünür ve review edilen bir şey yapın; Bölüm 15 buna geri dönüyor. Rol tanımlarını uygulamak ya da login vermek gibi rutin yönetim bunun yerine `ledgerly_access_admin`'i kullanır. Superuser rutin işler için kullanılıyorsa, bir rolün olması gereken bir yetkisi eksiktir.

`ledgerly_access_admin`, üç principal tarafından, her biri kendi login'i üzerinden kullanılan tek bir roldür: erişim tanımını uygulayan pipeline adımı, credential broker (Bölüm 4) ve access gateway (Bölüm 7). Login oluşturur ve üyelikleri yönetir, hiçbir tabloya sahip değildir ve hiçbir veri yetkisi taşımaz; yine de bu geniş bir güçtür: onu elinde tutan, yönettiği roller içinde her şeyi grant edebilir. Ayrı login'ler üçünü log'da birbirinden ayırır, ama yapabilecekleri bakımından ayırmaz.

Control plane'in kendi gücünü sınırlayan şey bu yüzden rol değil, çevresidir. Üç login'in her birine yalnızca kendi bileşeninden ulaşılabilir. Her eylem audit log'a gider (Bölüm 14) ve aşağıdaki drift kontrolü sonucu kodla karşılaştırır. Erişim tanımındaki değişiklikler kendi küçük reviewer kümesinden geçer. Bu sınırlar kötüye kullanımı görünür kılar; imkânsız kılmaz ve bunu erişim review'unda söyledim.

Daha da daraltmak: view'lar, satır kuralları ve tenant'lar

View'lar

View, bir tablo gibi davranan saklı bir sorgudur. View'a erişim grant edilen bir rol, alttaki tabloda yetki taşımadan sorgunun açığa çıkardığını okur.

```
// illustrative: a narrowed projection owned by ledgerly_owner
CREATE VIEW customer_support_summary AS
  SELECT id, created_at, status FROM customers;
GRANT SELECT ON customer_support_summary TO app_admin_console;
```

Not: Birçok engine'de bir view, çağıranın değil owner'ının yetkileriyle çalışır. Daraltmayı işe yarar kılan budur; aynı zamanda fazla şey seçen bir view'un bunu view üzerinde grant almış her role sızdırmasının da nedenidir. Bazı engine'ler diğer davranışı seçmenize de izin verir. Sizininki varsayılan olarak ne yaptığını kontrol edin ve view'u, temel tabloda hiçbir yetkisi olmayan bir rolle test edin.

Engine destekliyorsa kolon düzeyi grant'leri tercih ederim, çünkü grant tablonun yanında yaşar. Biçimin kendisi amaç olduğunda view daha iyidir: bir join, hesaplanmış bir değer ya da değişen bir tablonun üzerinde kararlı bir arayüz. Kitap JIT erişim için istek başına bir view kullanmaz ve bunun iki nedeni vardır: bir view'un sahibi olmalıdır, dolayısıyla onu oluşturan gateway'in owner gücüne ihtiyacı olurdu ve her istek için yapılmış bir view sahiplik modelini bozar.

Satır kuralları ve tenant'lar: bir karar kuralı

Bir satır kuralı, matrisin bir hücresi içindeki satırları daraltır ve hücrenin söylediğini değiştirmez. Destek aynı anda tek bir müşteriye bakar; bu yüzden "app_admin_console yalnızca customer_id'si vakanın müşterisine eşit olan invoices satırlarını görür" gibi bir kural, kolon filtresinin uymadığı yere uyar. Engine'ler sözdiziminde anlaşılamaz; bu yüzden kitap hiçbir gösterim vermez.

Ledgerly'nin müşterileri işletmelerdir ve her biri bir başkasının invoice'larını görmemeyi bekler. Bu izolasyon çoğunlukla uygulamanın işidir: billing-api her sorguda müşteriye göre filtreler ve hiçbir rol bir

tenant'ı diğerinden ayırmaz; çünkü application rolü herhangi birine hizmet verebilmek için hepsini okuyabilmelidir.

Bir satır kuralını, destek için ya da ikinci bir tenant katmanı olarak, yalnızca iki koşul birlikte sağlandığında kullanın: yanlış bir satır ağır sonuç doğuruyor ve uygulama context değerini her session için güvenilir biçimde ayarlıyor. Bedel, her statement'ın predicate'i taşıması ve bir şeyin context'i session başına ayarlamasıdır. Bir pooler'ın arkasında, bir session'da ayarlanan bir değer aynı bağlantıyı bir sonraki ödünç alana ulaşabilir (Bölüm 5 buna dönüyor). Context eksik olduğunda kural hiç satır döndürmemelidir ve ilk test bu durumdur. Açık kalarak başarısız olan bir kural, hiç kuralın olmamasından kötüdür. Ayrı veritabanları çok daha yüksek bir operasyonel fiyata daha güçlü bir sınırdır; hangisinin uyduğu kaç tenant'ınız olduğuna ve onlara ne vaat ettiğinize bağlıdır.

Just-in-time erişimin tavanı

`human_jit_write` tavanı tanımlar: bir just-in-time (JIT) isteğinin isteyebileceği en fazla şey, adı geçen tablolar ve kolonlar üzerindeki yetkiler olarak ifade edilir. JIT erişim, belirli bir iş için ve sınırlı bir süre için verilen erişimdir; onayları ve ömürleri Bölüm 8 ele alır, bu bölüm yalnızca tavanı tanımlar.

Rol hiçbir login'e grant edilmez. Bir istek, tavanın içine sığan (yetki, tablo, kolonlar) üçlülerinin, istekle birlikte saklanan bir listesidir. Onayda tam olarak bu üçlüler doğrudan isteyen session login'ine grant edilir ve süre dolusunda tam olarak bu üçlüler revoke edilir. Paylaşılan bir rol istek başına kapsamlandırılmaz; bu yüzden istek başına kapsam grant'lerde yaşar ve rol dış sınırı çizer.

```

// the ceiling: what a JIT request may ask for, never granted to a
login
GRANT SELECT ON customers, payments, invoices, invoice_lines
    TO human_jit_write;
// id and created_at are assigned by the database, never changed or
supplied
// by a data fix, so the write privileges below name every other
column
GRANT UPDATE (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
GRANT INSERT (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
// DELETE only where a request names it, and never on a table with
// Restricted columns
GRANT DELETE ON customers, payments, invoices, invoice_lines
    TO human_jit_write;

// illustrative column lists; Restricted columns are not named
anywhere
GRANT SELECT (customer_id), UPDATE (customer_id)
    ON bank_accounts TO human_jit_write;
GRANT SELECT (customer_id, provider, last_four, expires_on),
    UPDATE (customer_id, provider, last_four, expires_on)
    ON payment_tokens TO human_jit_write;
// no table-level SELECT on either table: it would include the
Restricted columns
// no INSERT or DELETE on either: a new or removed row sets or
destroys
// Restricted values

```

Bloğu, dışarıda bıraktıkları için okuyun. `bank_accounts` ve `payment_tokens` yalnızca adı geçen birkaç Restricted olmayan kolon için açıktır ve satır tanımlayıcıları ya da zaman damgaları için asla açık değildir: `iban`, `account_holder_name` ve `token_value` bir JIT isteğinde asla yer alamaz. INSERT ilk dört tabloyla sınırlıdır ve `id` ya da `created_at`'i adlandıramaz; böylece geri yüklenen ya da yeni bir satır her zaman veritabanının atadığı değerleri alır. İki append-only tablo ve gateway'e ait erişim isteği kayıtları tavanın tamamen dışındadır. Bir DELETE dışındaki her üçlü kolonlarını adlandırır.

İki bedel burada bir kez söylenir. Bir satırı sahibine bağlayan kolonlar, örneğin `customer_id`, Restricted değildir; bu yüzden bir JIT UPDATE onları değiştirebilir. Böyle bir owner-link değişikliği tavanın içinde kalır, ama bir tehlikeli işlem sınıfıdır: yürütme politikası onu gate'ler ve ham SQL olarak çalıştırılmaz, ürün yoluna yönlendirilir (Bölüm 10). Kontrol,

yalnızca four-eyes onayı değil, gate ve ürün yoludur. Ve bir JIT SELECT, adını verdiği her Confidential kolonu maskelenmemiş gösterir; bu, onaylayanın kabul ettiği şeyin bir parçasıdır.

Tavanı dürüst tutan şey güven değil, bir kontroldür. Gateway, tavanın dışındaki her isteği reddeder (Bölüm 8 kontrolü gösterir) ve aşağıdaki drift testi hiçbir login'in human_jit_write'in üyesi olmadığını doğrular; bir tavanı kalıcı bir grant'e dönüştürecek hata budur.

Engine'de kolon düzeyi grant olmadığında

Her engine kolon düzeyi grant'leri desteklemez. Kitap tam olarak tek bir fallback kullanır: engine kolonlar üzerinde grant edemiyorsa, Restricted kolonlar ayrı tablolara taşınır: bank_account_secrets ve payment_token_values.

```
table bank_account_secrets:      // owned by ledgerly_owner
  bank_account_id    text primary key
  iban               text not null
  account_holder_name text not null

table payment_token_values:      // owned by ledgerly_owner
  payment_token_id    text primary key
  token_value         text not null

// reachable by app_payments_service and human_breakglass only;
// outside the ceiling, never part of a JIT request
```

Fallback ile ana tablolar hiçbir Restricted kolon tutmaz; bu yüzden onlar üzerindeki tablo düzeyi bir JIT grant'i Restricted veriye ulaşamaz. Bedel: istek ayrıntısı tablo düzeyine iner ve onaylayan, isteyenini adını verdikleriyle sınırlı değil, tablonun Restricted olmayan her kolonunu, Confidential olanlar dahil, onaylamış olur.

Break-glass ve şemanın üstünde duran işlemler

human_breakglass, Restricted kolonları okuyan tek insan rolüdür. Her tabloyu okur ve iki append-only tablo dışında her tabloya INSERT ve UPDATE yazar; bank_accounts ve payment_tokens'ta yalnızca UPDATE yazar ve yalnızca Restricted olmayan kolonlarda. Acil durumlar içindir, rutin düzeltmeler için değil ve matriste yoktur; çünkü onu yalnızca bir session, kısa bir süre için tutar; hiçbir kalıcı workload tutmaz.

Restricted kolonlar yalnızca bir application yolu üzerinden değıştirilir. Bir insan onları hiçbir zaman ham SQL ile yazmaz, ne bir JIT session'da ne başka bir yerde: `admin-console`, ikinci bir kiři onayladıktan sonra tek yazıcı olan `payments-service`'ten ister ve değışiklik `audit_events`'e kaydedilir. Bu yolu Bölüm 10 anlatır. Bu olgular, grant'lerin yapabileceklerinin sınırını işaretler. Bir grant, bir session'ın hangi tablolara ve kolonlara dokunabileceğini söyler; hangi statement'ı, kaç satırı ya da bu belirli değışikliğin onaylanan değışiklik olup olmadığını söyleyemez. Bu, Yürütme parçasının işidir ve gateway'in her statement'ı nasıl kontrol ettiğini Bölüm 9 ele alır. Grant'lerin nerede bittiğini bilmek, yeni bir yeteneğin nereye ait olduğuna karar vermenize yardım eder.

Kod olarak roller ve drift

Yalnızca veritabanında var olan bir rol řeması, kimsenin review edemeyeceđi bir řemadır. Yukarıdaki grant'ler okunabilecek kadar kısadır ve korudukları řema gibi version control'da durmalı, review üzerinden değışmeli ve pipeline tarafından uygulanmalıdır.

Tanımlar nerede yaşar

İki tür dosya tutardım. Migration'lar tabloları oluşturur ve her tablonun grant'lerini belirtir. Ayrı, review edilen bir erişim tanımı rolleri, roller arasındaki üyelikleri ve tavanı listeler. Pipeline ikisini ayrı adımlar olarak uygular. Eriřim tanımı, pipeline adımının kendi login'i üzerinden `ledgerly_access_admin` içinde uygulanır; broker ve gateway'in kendi login'leri üzerinden kullandığı rolün aynısıdır; hiçbir veri yetkisi taşımaz. Rol oluşturmak tabloları değıştirmekten daha güçlü bir yetkidir; bu yüzden adımının kendi küçük reviewer kümesi olmalıdır. Reviewer'ın sorusu SQL'in geçerli olup olmadığı değil, bu grant'in değışikliği işe yarar kılan en dar grant olup olmadığıdır. Matris açık bir reviewer bunu bir dakikada cevaplayabilir.

Mevcut bir sistemi daraltmak

Ledgerly her řeye sahip tek bir paylaşılan login ile başlar. Oradan matrise tek bir değışiklikle geçmezdim, çünkü fazla dar bir grant production'ı bozar ve ekip neyi kaçırdığını başarısız olana kadar bilemez. Dört adımlı bir yol:

1. **Önce her workload'a, öncekiyle aynı yetkilerle kendi login'ini verin.** Hiçbir şey bozulmaz ve log hemen workload'u adlandırmaya başlar.
2. **Gözlemleyin.** Her login altında kaydedilen statement'ları, workload'un gerçekte nelere dokunduğunu listelemek için kullanın; yavaş aylık job'u içerecek kadar uzun bir dönem boyunca. Bir statement log'u literal değerler tutabilir, bu yüzden onu hassas kabul edin: müşteri verisi ya da bir sorguya yapılandırılmış Restricted değerler içerebilir. Kimin okuyacağını kısıtlayın, saklama süresini belirleyin ve engine sunuyorsa parametrelili statement biçimlerini log'lamayı tercih edin.
3. **Dar rolü tanımlayın ve login'i ona taşıyın;** önce aynı job'ları çalıştıran bir staging ortamında, sonra production'da, eski yetkileri kısa bir süre elde tutarak.
4. **Eski yetkileri kaldırın** ve reddedilen statement'lara göz kulak olun.

Bedel çaba değil, takvim süresidir. Ekipler ikinci adımı atlamaya eğilimlidir ve öngörülebilir sonuç, aylar sonra bozulan nadiren kullanılan bir yoldur.

Drift

Drift, kodun bildirdiği erişim ile veritabanının tuttuğu erişim arasındaki farktır. Bir sorun gece elle yazılmış bir **GRANT** ile çözüldüğünde ortaya çıkar: grant işe yarar, olay biter ve kimse bunu yazıya dökmez. Kontrol, bildirileni gerçek olanla zamanlanmış olarak karşılaştırır ve bir tarafı okuyamadığında yüksek sesle başarısız olmalıdır.

```

function checkRoleDrift():
    declared = loadDeclaredAccess()
    try:
        actual = readActualAccess()
        requests = loadAccessRequests()
        known = loadGatewayLogins()
    on ReadFailed:
        // an unreadable state is not a clean state
        notify("platform", "role drift check could not read a source")
        raise DriftCheckFailed

    for each extra in difference(actual.roleGrants,
declared.roleGrants):
        notify("platform", "grant not in code: " + extra)
    for each login in actual.loginsOwningObjects():
        notify("platform", "login owns objects: " + login)
    for each login in actual.membersOf("human_jit_write"):
        notify("platform", "login is a member of the ceiling: " +
login)
    for each login in actual.allLogins():
        if login is human and login not in known:
            notify("platform", "login unknown to the gateway: " +
login)
        // memberships must match what the login was issued for
        for each role in difference(actual.rolesOf(login),
expectedRoles(login)):
            notify("platform", "unexpected membership: " + role + "
for " + login)
        // a direct grant is legitimate only as part of a live request
        live = unexpiredTriples(requests, login)    // empty for
service logins
        for each g in difference(actual.directGrantsOf(login), live):
            notify("platform", "direct grant without a live request: "
+ g + " to " + login)
        for each login in actual.membersOf("human_breakglass"):
            if not hasOpenBreakGlass(requests, login):
                notify("platform", "break-glass member without a request:
" + login)

```

`expectedRoles(login)`, bir gateway session login'i için `human_readonly`'dir; bir break-glass isteği açıkken ayrıca `human_breakglass`'tır ve broker'ın login'i verdiği tek `app_*` rolüdür. Kontrol bu yüzden elle yazılmış bir GRANT `app_payments_service TO <session login>` ya da GRANT `ledgerly_owner TO <login>` ifadesini, doğrudan bir yetki grant'i gibi yakalar.

Burada `roleGrants` yalnızca kodun bildirdiği rolden role üyelikleri ve yetkiden role grant'leri kapsar. Login'ler kodda değildir: session'larla gelir ve giderler. İnsan login'leri gateway'in kayıtlarına karşı kontrol edilir. Her

login'in, servis login'leri dahil, doğrudan grant'leri, süresi dolmamış `access_request_privilege` üçlülerıyla karşılaştırılır. Bir servis login'inin isteği yoktur; bu yüzden onun üzerindeki her doğrudan grant işaretlenir; gece bir servis login'ine bir yetkinin elle yazılmış `GRANT` ile verilmesi bu şekilde fark edilir. İsteğinden daha uzun yaşayan bir grant ya da açık isteği olmayan bir break-glass üyesi, revoke job'u başarı bildirirse bile işaretlenir. JIT erişim doğrudan grant'lerle, süre dolumu ise bir revoke job'u ile çalışır; job durursa grant'ler kalır ve bu kontrol job'dan bağımsız olarak bunu fark eder (Bölüm 8 eşleşmeyi ele alır). Kontrol yalnızca grant'leri ve üyelikleri okur. Bir session'ın onlarla ne yaptığını görmez; bu yine Yürütme parçasıdır.

Kontrol hiçbir şeyi onarmaz, çünkü production'da bilinmeyen grant'leri otomatik revoke etmek, production'ı gece bozmanın yeni bir yoludur. Bildirilen durumun doğru olup olmadığını da söylemez: doğru bildirilmiş yanlış bir grant geçer. Bu yargı Bölüm 14'ün erişim review'una aittir.

Trade-off'lar ve maliyetler

Bu bölümdeki her kontrolün bir fiyatı vardır ve fiyatı belirli insanlar öder.

Workload başına roller bakım maliyeti getirir. Beş rol, beş kod tabanıyla uyumlu tutulacak beş tanım demektir. Bir servisin yeni bir tabloya ihtiyacı olduğunda, biri rolü aynı değişiklikte günceller; grant koddan sonra gelirse deploy başarısız olur. Hafifletme, grant'i migration'ın parçası yapmaktır. Tek servis ve iki mühendisle bu yük hiç değil değildir; bir owner, bir application rolü ve bir salt okunur rolden oluşan daha küçük bir şema makul bir ilk adımdır.

Dar grant'ler reddedilen statement'lara yol açar. Bazı retler grant'lerdeki hatalardır, bazıları olması gerekenden daha ileri uzanan koddur. Bir ret bir olay sırasında ortaya çıktığında ikisinin bedelini on-call olan öder. Staging'i production ile aynı rollerle çalıştırmak sürprizi azaltır; bedeli görmezden gelmek ekipleri bir grant'i aceleyle genişletmeye götürür ve drift'in başladığı yer bu acelecilik olur.

Kolon düzeyi grant'ler review yükü ekler. Tablo değiştiğinde her kolon listesi bakılmalıdır. Tablo düzeyinde bir grant altında yeni hassas bir kolon varsayılan olarak açığa çıkar; bu yüzden hassas tablolarda adı geçen

kolonlara grant ederim ve ekstra satırları kabul ederim. Engine bu özelliğe sahip değilse, fallback join'lere, migrate edilecek ek tablolara ve daha kaba bir JIT onayına mal olur.

View'lar ve satır kuralları hareketli parça ekler. Biçim ya da izolasyon amaç olduğunda ağırlıklarını hak ederler; dikkatli hissetmek için eklendiklerinde kötü bir takastır. Context'i kaybolabilen bir satır kuralı güvenlik tiyatrosudur: her sorguya sürtünme ekler ve yazıldığı durumda açık kalarak başarısız olur.

Sahiplik disiplini bazı insanları yavaşlatır. Bir tabloyu console'da oluşturmaya alışmış mühendisler şimdi bir migration'dan geçmek zorundadır. Sürtünme amaçlanmıştır ve bir atılabilir geliştirme veritabanında production'dakinden daha az değer taşır. Modeli önce production'a uygulayın ve başka yerlerde yalnızca neyden vazgeçtiğinizi söyleyebildiğinizde gevşetin.

Bu kadar yapıyı ne zaman kullanmamalı. Tek geliştiricisi olan ve müşteri verisi olmayan bir prototipin tam rol şemasına ihtiyacı yoktur. Owner, application ve salt okunur ayrımı, ikinci servise ya da ilk hassas kolona kadar yeterlidir. Şemayı, "bunu kim çalıştırdı?" log'dan cevaplanamadığında ya da bir workload'un credential'ı işinin gerektirdiğinden fazla zarar verebildiğinde büyütün.

Sık yapılan hatalar

- **Application rolünü owner olarak görmek.** Cazip gelir, çünkü migration'lar o zaman startup'ta bir adım eksikle çalışır. Servis o zaman silme ve yeniden grant etme gücünü elinde tutar. Ayrı bir migrator ve login'i olmayan bir owner kullanın.
- **Geçen yılın sorgularına göre grant etmek.** Mevcut en geniş login'den kopyalanmış bir rol, bir zamanlar işe yarayan her yetkiyi korur. Rolü, kendi login'i altında gözlemlenen, workload'un şimdi yaptığından türetin.
- **Kişi başına rol eklemek.** Roller yetki demetleridir ve login'ler kişi başıdır. Bir bireyin adını taşıyan bir rol, şemayı bir personel listesine çevirir. Roller işlemlerle ilgili tutun.
- **Birkaç servis "benzer" diye hepsi için tek rol kullanmak.** Servisler birbirinden ayrışır ve rol her birinin ihtiyaçlarının birleşimini tutar.

Servis başına bir rol birkaç satıra mal olur.

- **Varsayılanları gelecekteki tablolara bırakmak.** Bir default yetki her migration'da bir satır tasarruf ettirir ve bir sonraki hassas tabloyu geniş bir role açar. Varsayılanı boş tutun ve açıkça grant edin.
- **Tavan rolünü yoğun bir hafta için bir login'e grant etmek.** Tavan o zaman dostane bir adı olan kalıcı bir grant'e dönüşür. Tavanı bağlantısız tutun: onu yalnızca istek kontrolü okur ve drift testi her üyeliği işaretler.
- **Hassas bir kolonu bir view'un arkasına saklayıp tabloyu okunabilir bırakmak.** View, temel tabloya erişebilen bir rol için hiçbir şeyi daraltmaz. Tablodan revoke edin ve hiçbir tablo yetkisi olmayan bir rolle test edin.
- **Kimsenin karşılaştırmadığı bildirilmiş bir şemaya güvenmek.** Version control'daki roller sabaha karşı 2'deki GRANT'i durdurmaz. Bildirileni ve gerçeği zamanlanmış olarak karşılaştırın ve karşılaştırmayı okuyamadığında yüksek sesle başarısız kılın.
- **Superuser'ı rutin iş için kullanmak.** Her seferinde işe yarar, yayılmasının nedeni de budur. Rutin bir iş ona ihtiyaç duyduğunda, eksik yetkiyi doğru role ekleyin.

Temel çıkarımlar

- Paylaşılan bir login kanıtı ve kapsamı birlikte yok eder: log kimin hareket ettiğini söyleyemez ve login, en geniş çağırının yaptığı her şeyi yapabilecek şekilde izinli olmak zorundadır.
- Kimliği, login'i ve rolü ayrı tutun: bir principal'ın kimliği vardır, bir login üzerinden bağlanır ve login rollerin üyesidir; kimse bir rol olarak login olmaz.
- `ledgerly_owner` her şeye sahiptir ve login'i yoktur, `ledgerly_migrator` onu yalnızca bir migration sırasında üstlenir ve her workload işinin boyutunda kendi `app_*` rolünü alır.
- Her rolü hiç yetkisiz başlatın, hassas kolonlar sıradan kolonların yanında durduğunda kolon düzeyi grant'leri kullanın, default yetkileri boş tutun ve ayrı secret tablolarına yalnızca engine kolonlar üzerinde grant edemediğinde başvurun.

- `human_jit_write` hiçbir login'e grant edilmeyen bir tavadır: onaylanan her istek bunun bir alt kümesini doğrudan session login'ine grant eder ve hiçbir istek bir Restricted kolon içeremez.
- Roller ve grant'ler version control'a ve review edilen pipeline adımlarına aittir; veritabanını okuyamadığında yüksek sesle başarısız olan zamanlanmış bir drift kontrolü, bildirilen ve gerçek durumu bir arada tutar.
- Her katman birilerine bir şeye mal olur: roller için bakım, dar grant'ler için reddedilen statement'lar, kolon listeleri için review yükü ve engine'de kolon düzeyi grant olmadığında daha kaba bir onay.

Roller, bir principal'ın bağlandıktan sonra ne yapabileceğini söyler, ama bir servisin kim olduğunu nasıl kanıtladığı hakkında henüz hiçbir şey söylemez. Bölüm 4 bir sonraki adımı atar: credential'lar workload'lara nasıl ulaşır, kısa ömürlü ve dinamik credential'lar problemin biçimini neden değiştirir ve yapılandırma dosyasındaki parolanın yerini ne alır.

KISIM II

Uygulama Eriřimi

Bölüm 4 — Credential'lar ve Secret'lar

Ledgerly'de bir cuma öğleden sonrasını düşünün. Backend ekibinden biri başka bir gruba geçmiş ve güvenlik sorumlusu, paylaşılan admin parolasının değiştirilip değiştirilemeyeceğini kibarca soruyor. Platform ekibinden Alex, değiştirilmesi gerektiğine katılıyor. Sonra Alex parolanın nerelerde yaşadığını listelemeye başlıyor: beş servisin tamamının yapılandırması, pipeline'ın değişkenleri, `reporting-job`'un export adımı, bir ekip parola yöneticisi ve birkaç masaüstü client'taki kayıtlı bağlantılar.

Veritabanında değiştirmek bir saniye sürer. Her yerde, aynı anda değiştirmek ise mesai saatlerinde her servisin koordineli biçimde yeniden başlatılması demektir ve bu hafta sonu on-call olan Robin yerinde soruyu soruyor: yoldaki invoice'lara ne olacak? Değişiklik hiç gelmeyen bir "sakin pencere"ye erteleniyor. Parola o kişiden, sonra da bir sonrakinden uzun yaşıyor.

Bu bölüm o cumanın doğurduğu soruyu cevaplar: bir servis credential'ını nasıl alır ve birini kesinti olmadan nasıl değiştirirsiniz? Yürüyen örnekten bir sonraki hedef durum adımını ekler. Servislerin credential'ları kısa ömürlü ya da dinamik olur ve workload'lara paylaşılan bir dosya yerine kimlik üzerinden ulaşır.

Credential'lar beş parçalı modelin Kimlik parçasına aittir ve gereklidir, ama cevap değildir. Kitabın tezi şudur: "Production veritabanı erişimi bir credential dağıtım problemi değildir; bir yetkilendirme, yürütme ve hesap verebilirlik problemidir." Daha iyi credential yönetimi, bir login'i kimlerin elde edebileceğini daraltır. Bir login içeri girdikten sonra ne yapabileceği ya da ne yaptığı hakkında hiçbir şey söylemez; bunlar grant'ler, yürütme ve kayıtlar üzerine olan bölümlerin işidir.

Problem ve neden zor olduđu

Credential, bir principal'ın kimlik doğrulamak için sunduđu şeydir; **secret** ise ifşa edilirse erişim sağlayan her değerdir. Parola ikisidir de. Bölüm 3 her servise kendi rolünü verdi; böylece bir login neyi yapabileceğini söyler. Bu bölüm diğeri yarı hakkındadır: login kim olduğunu nasıl kanıtlar ve bu kanıt ne kadar süre geçerli kalır.

Bir parolanın, kimse tasarlamasa bile bir yaşam döngüsü vardır. Oluşturulur, bir servisin okuyabileceği her yere kopyalanır, aylarca kullanılır ve sonunda değiştirilir ya da unutulur. Her kopya, bir insanın ya da bir process'in onu okuyabileceği bir yerdir ve "bu parolayı kim görebilir?" sorusunun, kopya sayısı kadar cevabı vardır. Başlangıç durumunda cevap, "kabaca backend ekibinin tamamı ve bir servisin yapılandırmasını okuyabilen her şey"dir.

Bariz çare daha iyi bir paroladır: daha uzun, rastgele, daha derli toplu bir yerde saklanan. Bu tahmin etmeye karşı yardımcı olur ve tahmin hiçbir zaman asıl risk olmadı. Risk, değerin durduğu yerlerin sayısıdır ve bunlardan hiçbirinin diğelerine dokunmadan boşaltılamamasıdır. Beş yapılandırma dosyasındaki güçlü bir parola yine beş kopyadır.

İkinci zorluk, bir credential'ı bir zamanlamaya göre ya da şüpheyle yenisiyle değiştirmek anlamına gelen **rotation**'dır. Rotation zordur, çünkü bir credential iki sistem arasındaki paylaşılan durumdur: değeri veritabanı bilir ve her consumer da bilir. Bir tarafı önce değiştirmek diğelerini bozar. Bir parolayı kesinti olmadan değiştiremeyen bir ekip, makul biçimde onu değiştirmeyi bırakır ve kimsenin uygulayamadığı bir rotation politikası takvime bir kayıt ekler, hiçbir riski ortadan kaldırmaz.

Credential'ların yapamadıkları

Reporting job'ının bir saat geçerli, bir sohbet kanalına sızmış bir login'ini düşünün. Bu bölümdeki her ölçüye göre iyi bir credential'dır: benzersizdir, kısa ömürlüdür ve kimlik üzerinden alınır. Bir saati boyunca, yine de rolün grant aldığı customers'ın üç kolonunu (id, created_at ve status) her satırda okuyabilir; çünkü app_reporting_job'ın okumasına izin verilen budur. Kısa ömürler pencereyi daraltır; rolü daraltmaz. Bir sızıntıyı üç şey sınırlar ve hiçbir credential'ın kendisi değildir: taşıdığı **rol** (Bölüm 3), nereden kullanılabileceğine karar veren **ağ erişimi** (Bölüm 7) ve ne

yaptığını kaydeden **log** (Bölüm 14). Yalnızca pooler'ın ağından sunulabilen ve Restricted kolonları göremeyen bir rolle gelen bir login, hiçbir parola politikasının yapabileceğinden daha küçük bir olaydır.

Aşağıdaki en güçlü tasarım için de aynısı geçerlidir. Bir broker, bir lease ve workload identity hepsi bir login'i kimin edinebileceğini cevaplar. Login'in ne yapabileceğine karar vermezler ve izin verilmiş bir login'in zararlı bir statement çalıştırmasını engellemezler.

Bu bir mühendislik sorusudur, bir politika sorusu değil. "Her 90 günde bir rotate et" diyen bir kuralın, beş servisin yeni değeri aynı anda nasıl öğreneceği hakkında söyleyecek bir şeyi yoktur. Cevap, credential'ların nasıl verildiğinde, iletildiğinde ve değiştirildiğinde yatar; bu da ekibin inşa etmesi gereken bir tasarımdır.

Bir credential'ın hayatı

Bir credential'ı doğumundan emekliliğine kadar izlemek ve her durakta onu kimin okuyabileceğini ve neyin ortaya çıkaracağını sormak yardımcı olur. Olağan yerler şunlardır.

Yer	Kimler okuyabilir	Nasıl sızma eğiliminde	Daha iyi yer
Repository'deki yapılandırma dosyası	Repository erişimi olan herkes, geçmiş üzerinden sonsuza dek	Bir fork'a, sohbete, ticket'a kopyalanır	Adıyla referans verilen bir secret manager
Environment variable	Process, çocukları ve process'i ya da crash dump'larını inceleyebilen herkes	Bir debug endpoint'i ya da hata raporu tarafından yazdırılır	Başlangıçta teslim edilen, dar dosya izinli bir dosya ya da bellek
Container image'ı ya da build artifact'ı	Image'ı pull edebilen herkes	Build sırasında içine gömülür, böylece her kopya onu taşır	Image'da hiçbir şey; başlangıçta fetch edilir
CI/CD değişkeni ve job log'u	Pipeline yöneticileri; bazen her job	Bir betik tarafından echo'lanır, başarısız bir komut tarafından yazdırılır	Job'ın kendi kimliğiyle fetch edilen kısa ömürlü değer

Yer	Kimler okuyabilir	Nasıl sızma eğiliminde	Daha iyi yer
Ekip parola yöneticisi	Paylaşılan vault'taki herkes	Alışkanlıkla paylaşılır; kimin okuduğuna dair kayıt yok	Kalan birkaçı için kayıtlı, bireysel erişim
Masaüstü client'ın kayıtlı bağlantısı	O laptop'u kullanan kişi	Kaybolan ya da yedeklenen laptop	Kayıtlı credential yok; bir gateway'den session credential'ları (Bölüm 7)

Tabloda iki şey öne çıkıyor. Birincisi, sızıntılar neredeyse hiç zekice değildir. Kolaylık için yapılmış kopyalardır: bir meslektaş sorunu yeniden üretebilsin diye bir ticket'a yapıştırılan değer, bir deployment'ı debug ederken echo'lanan bir değişken. İkincisi, "daha iyi yer" kolonu çoğunlukla aynı cevaptır: değer tek bir yönetilen yerde yaşar ve consumer'lar ona ihtiyaç duyduklarında fetch eder.

Uyarı: Bir image katmanında, bir repository geçmişinde ya da bir job log'unda görünmüş bir secret, kimse okumamış olsa bile ifşa edilmiş sayılmalıdır. Satırı silmek kopyaları silmez. Çare credential'ı değiştirmektir; bir ekibin, ihtiyaç duyulacağı günden önce çalışan bir rotation yoluna ihtiyac duymasının nedeni budur.

Bir secret'ın adlandırmaya değer bir özelliği daha var: ne kadar süre geçerli kaldığı. Diyelim bir parola üç yıl geçerli: şimdiye kadar yapılmış her kopyadan üç yıllık bir maruziyet taşır. Bir saat süren bir credential'ın eski bir kopyasının hiçbir değeri yoktur. Süreler örnek amaçlıdır. Ömrün güçten daha önemli olmasının nedeni budur.

Secret manager'lar ve bir servis bir secret'ı nasıl okur

Secret manager, secret'ları saklayan, her birini kimin okuyabileceğini denetleyen ve her okumayı kaydeden bir servistir. HashiCorp Vault ve büyük cloud sağlayıcıların secret store'ları örnekler. Örüntü üründen daha önemlidir: değeri tek bir yer tutar, erişime principal başına karar verilir ve bir okuma kayıt bırakır.

Parolayı bir secret manager'a taşımak tek başına okuyucu sayısını azaltmaz. Onların kim olduğunu değiştirir ve her okumaya bir ad verir: belirli bir servis kimliği belirli bir path'i okur ve secret manager bunun ne zaman olduğunu söyler. Bu, inceleme ve revoke için gerçek bir kazançtır, maruziyet için mütevazı bir kazanç.

İlk secret

Bir servisin herhangi bir şeyi okuyabilmesi için secret manager'a kendini kanıtlayacak bir şeye ihtiyacı vardır. O şey saklanmış başka bir parolaysa, problem bir seviye yukarı taşınmıştır ve hiçbir şey çözülmemiştir. Bu bootstrap sorusudur, bazen ilk secret problemi denir.

Cevap **workload identity**'dir: platformun çalışan bir workload'a, örneğin bir servis instance'ına ya da bir job'a verdiği, böylece saklanmış bir parola olmadan kimlik doğrulayabilmesini sağlayan bir kimlik. Workload'u başlatan platform onun adına kefil olur ve secret manager bu beyana güvenir. Servis bu adım için hiçbir zaman uzun ömürlü kendi değerini elinde tutmaz. Workload identity'yi tasarımın geri kalanını mümkün kılan şey olarak görüyorum, çünkü gerilemeyi bitirir.

Başlangıçta okumak ve yeniden okumak

Bir servisin veritabanı credential'ını okumak için iki makul anı vardır: başlangıçta bir kez ya da her bağlantı açışında. Bir kez okumak basittir ve nadiren değişen statik credential'larla çalışır. Credential değiştiğinde başarısız olur, çünkü çalışan process yeniden başlatılana kadar eski değeri tutar. Bağlantı sırasında okumak, bir kimlik doğrulama hatasından sonra secret manager'a tekrar soran bir client üzerinden, yeniden başlatma olmadan rotation'la başa çıkar.

```

function connectWithCredential(service):
    attempts = 0
    while attempts < maxAttempts:
        try:
            slot = readSecret(path: "db/" + service + "/active")
            credential = readSecret(path: "db/" + service + "/" +
slot)
            return openConnection(host: "db-pooler.internal.example",
                                login: credential.login,
                                password: credential.password)
        on AuthenticationFailed:
            // the value may have rotated since the last read; read
again
            attempts = attempts + 1
        on SecretStoreUnavailable:
            // fail the start; never fall back to a value baked into
the image
            raise StartRefused
        raise StartRefused

```

Blok, önemli olan iki hatayı gösterir. Bir kimlik doğrulama hatası yeni bir okumayı tetikler; böylece bir rotation kesintiye dönüşmez. Ulaşılamayan bir secret manager başlangıcı reddeder, çünkü yedek bir kopyaya dönmek, bu tasarımın kaldırdığı dosyayı yeniden yaratır. Bu reddin bir bedeli var ve Trade-offlar bölümü ona geri dönüyor.

Not: Secret manager'ın kendisi bir kapıdır. db/billing-api path'ini kimin okuyabileceği, kimin değiştirebileceği ve okumaların log'lanıp log'lanmadığı, veritabanı için sorulanlarla aynı sorulardır. Her servise her path için okuma erişimi vermek, paylaşılan parolayı fazladan bir adımla yeniden kurmak olurdu.

Kesintisiz rotation

"Bir tarafı önce değiştir, diğeri bozulur" sorununun çaresi, yalnızca tek bir değer çalıştığı bir an hiç olmamasıdır. Bu, sıkça A/B rotation denen **iki credential örüntüsüdür**. Her servisin, ikisi de aynı app_* rolünün üyesi olan iki login'i vardır: A ve B. Herhangi bir anda biri aktiftir, diğeri boşta.

Rotate etmek için boştaki login'i değiştirir, yeni değeri yayımlar, consumer'lara geçmelerini söyler, eski login artık kullanımda olmayana kadar bekler ve sonra onu geçersiz kılarırsınız. Hiçbir noktada bir

consumer'ın elinde, veritabanının kabul etmeyi bıraktığı bir değer bulunmaz. Pseudocode olarak, hata noktaları işaretli:

```
function rotateServiceCredential(service):
    active = readSecret(path: "db/" + service + "/active")    // "a" or
    "b"
    idle = otherSlot(active)
    if openSessions(loginFor(service, idle)) > 0:
        // an earlier rotation did not finish; changing this login
        cuts live sessions
        raise RotationIncomplete
    newPassword = generatePassword()
    // the idle login has no users, so changing it breaks nothing
    setLoginPassword(login: loginFor(service, idle), password:
newPassword)
    try:
        writeSecret(path: "db/" + service + "/" + idle,
                    value: { login: loginFor(service, idle), password:
newPassword })
        writeSecret(path: "db/" + service + "/active", value: idle)
    on SecretStoreUnavailable:
        // the active login is untouched; the idle one holds a value
        nobody can read
        notify("platform", "rotation stopped before publish: " +
service)
        raise RotationAborted
    // consumers re-read on their next connection; wait for the old
    login to drain
    waitUntil(openSessions(loginFor(service, active)) == 0, deadline:
drainDeadline)
    if openSessions(loginFor(service, active)) > 0:
        // the old value still works; the rotation is not finished
        notify("platform", "old login still in use after rotation: " +
service)
        raise RotationIncomplete
    setLoginPassword(login: loginFor(service, active), password:
generatePassword())
    auditLogAppend({ action: "rotate", subject: service, actor:
"rotation-job" })
```

Tasarımı üç ayrıntı taşır. Birincisi, veritabanı ve secret manager ortak bir transaction'lı olmayan ayrı sistemlerdir; bu yüzden bir hatayı sınırlayan şey adımların sırasıdır: canlı olan değer, yerine geçecek olan okunabilir olmadan asla değiştirilmez. Herhangi bir adımdaki hata, önceki credential'ı çalışır bırakır. İkincisi, drain kontrolü. Rotation, yalnızca hiçbir şey eski login'i kullanmadığında biter ve secret'ını hiç yeniden okumayan bir servis o anda bulunur, üç hafta sonraki bir kesintide değil. Üçüncüsü,

son adım eski login'in üzerine kimsenin tutmadığı bir değer yazar, ama yalnızca drain'den sonra çalışır. Herhangi bir session eski login'i hâlâ tuttuğu sürece, eski değer çalışır.

Bedel, artık her servisin iki login'i ve ikisini de bilen bir job'ının olması ve consumer'ın bir hatadan sonra secret'ı yeniden okumak zorunda olmasıdır. Credential'ını başlangıçta bir kez okuyan bir servis rotate etmek için rolling restart'a ihtiyaç duyar ve A/B rotation'ın yine bir takvim olayına dönüştüğü yer o restart'tır. Başlangıç durumunda paylaşılan admin parolasını rotate etmek bu nedenle zordur: ikinci bir login yoktur ve hiçbir consumer bir şeyi yeniden okuyamaz.

Yazıya dökülebilecek en kısa zamanlamayı değil, ekibin sürdürebileceği bir zamanlamayı seçerek rotate ederdim. Şüphelenilen bir sızıntı farklı bir yoldur ve "Bir credential sızdığında"da anlatılır.

Dinamik ve kısa ömürlü credential'lar

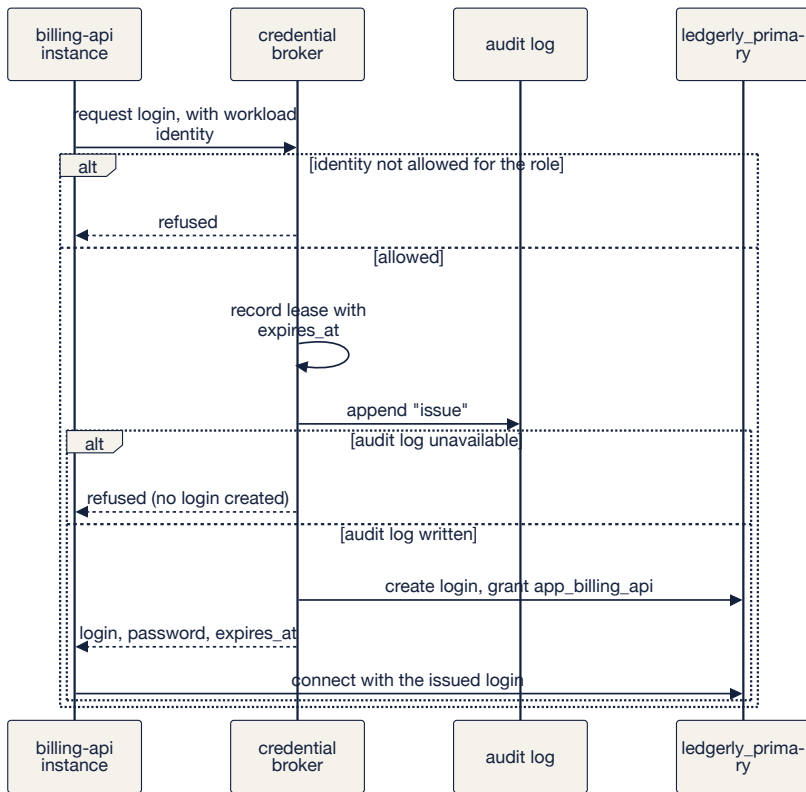
Rotate edilmiş statik bir credential'ın rotation'lar arasında yine uzun bir ömrü vardır. Bir sonraki adım, ömrü rotation'ın bir olay olmaktan çıkacağı kadar kısa yapmaktır. **Kısa ömürlü credential**, kısa, tanımlı bir süre içinde kendiliğinden sona erer. **Dinamik credential**, bir principal için istek üzerine üretilen kısa ömürlü bir credential'dır; böylece iki istek asla aynı değeri almaz. Bütün dinamik credential'lar kısa ömürlüdür; her kısa ömürlü credential dinamik değildir.

Olağan mekanizma bir **credential broker**'dır: istek üzerine veritabanında bir login oluşturan, ona doğru `app_*` rolünde üyelik grant eden ve login'in credential'ını bir **lease** ile birlikte isteyen workload'a teslim eden bir servis. Lease, credential'ı kimin, hangi rol için ve ne zamana kadar tuttuğunu söyleyen bir kayıttır. Lease sona erdiğinde ya da biri onu revoke ettiğinde broker login'i kaldırır.

Ledgerly'de broker, login oluşturan ve silen, rol üyeliklerini yöneten rol olan `ledgerly_access_admin`'in üyesi olan kendi login'iyle bağlanır. Hiçbir tabloya sahip değildir, owner değildir ve onu veri işi için kimse kullanmaz. Bu, sahiplikten daha dar bir güçtür ve yine de geniş bir güçtür: rolü elinde tutan, yönettiği roller içinde, application ve insan rolleri dahil, her şeyi grant edebilir. Broker bu gücün yalnızca küçük bir kısmına ihtiyaç duyar. Ona daha azını vermek rolü bölmek anlamına gelirdi ve bunun

kendi maliyetleri var; burada broker rolü olduğu gibi kullanır. Rol, her biri kendi login'i üzerinden üç principal tarafından kullanılan tek bir roldür: pipeline adımı, bu broker ve gateway. Bölüm 3 bu gücü neyin sınırladığını söyler. Broker için koruma burada yapılan tek beyandır: credential'ı owner'ınki gibi tutulur, ona yalnızca kendi workload identity'si ulaşabilir ve yaptığı her eylem audit log'a gider (Bölüm 14).

Aşağıdaki sıra, `billing-api`'nin bir instance'ının bir login istemesini gösteriyor. İsteğin reddedildiği iki yere bakın: henüz hiçbir şey var olmadan önce ve audit log'a ulaşamadığında.



Pseudocode'da, sırayı doğru tutan şey işlemlerin sırasıdır. `createLogin` gibi fonksiyon adları, kitabın başka yerlerinde olduğu gibi örnek amaçlıdır.

```

function issueServiceLogin(request):
    identity = verifyWorkloadIdentity(request.proof)
    if request.role not in rolesAllowedFor(identity):
        raise IssueRefused
    password = generatePassword()
    login = identity.workload + "-" + randomSuffix()
    expiresAt = now() + ttlFor(request.role)
    // the lease comes first: the sweeper reads it, so no login can
    exist
    // without a record of when it must end
    leaseId = recordLease({ login: login, identity: identity.workload,
                           instance: request.instance, role:
request.role, expires_at: expiresAt })
    try:
        auditLogAppend({ action: "issue", subject: login,
                        actor: "credential-broker", lease: leaseId })
    on AuditLogUnavailable:
        markLeaseVoid(leaseId)
        raise IssueRefused
    // a failure here leaves a lease with no login; dropping a missing
    login is harmless
    createLogin(login, password: password, valid_until: expiresAt)
    grant(request.role, to: login)
    return { login: login, password: password, expires_at: expiresAt }

```

Lease, login var olmadan önce yazılır ve audit girdisi login oluşturulmadan önce yazılır. Lease kaydedildikten sonra veritabanı başarısız olursa, sweeper login'i olmayan bir lease bulur ve onu zaten revoke edilmiş sayar. Audit log kapalıysa hiçbir credential verilmez. Bu bilinçli bir seçimdir ve ulaşılamayan bir audit log'un yeni instance'ların başlamasını durdurduğu anlamına gelir; Bölüm 14 bu bedeli diğer taraftan tartışıyor.

Süre dolumu ve revoke

Bir süre dolumu yalnızca yeni bağlantıların kimlik doğrulamasını durdurur. Zaten açık olan bir session, parolanın süresi dolduktan sonra da çalışmaya devam eder; bu yüzden süre dolumunun iki parçası vardır: login geçersiz kılınır ve üzerindeki açık session'lar kapatılır. Rotation ile bir credential'ı ya da grant'i hemen çalışmaz hâle getiren **revoke** arasındaki fark da budur. Rotation değiştirir; revoke bitirir.

```

function revokeExpiredLeases():
  for each lease in expiredLeases(before: now()):
    try:
      closeSessions(login: lease.login)
      dropLogin(lease.login)
    on LoginNotFound:
      // the login is already gone, which is the goal
    on RevokeFailed:
      // an expired credential that still works is an incident
      notify("platform", "expired login still active: " +
lease.login)
      continue
    markLeaseRevoked(lease.id)
    try:
      auditLogAppend({ action: "revoke", actor: "lease-sweeper",
        subject: lease.login, lease: lease.id })
    on AuditLogUnavailable:
      // the revoke is done; the log catches up from the retry
queue
      queueForRetry(lease.id)

function checkOverdueLeases():
  // scheduled separately, so one failing job cannot hide the other
  if count(expiredLeases(before: now() - graceMinutes)) > 0:
    notify("platform", "leases past expiry and not revoked")
  if now() - lastSuccessfulRun("revokeExpiredLeases") > maxRunAge:
    notify("platform", "lease sweeper has not completed recently")

```

Yenileme de aynı disiplini izler. Çalışan bir instance, lease'i bitmeden önce uzatması için broker'dan ister ve broker yalnızca lease canlıyken ve toplam ömür bir tavanın altındayken kabul eder. Tavan olmadan yenileme, kısa ömürlü bir credential'ı, tesadüfen yeniden verilen uzun ömürlü bir credential'a dönüştürür. Ömrün kendisi bir yargı meselesidir. Sisteminiz için doğru sayıyı bilmiyorum ve üç şeye bağlıdır: bir instance'ın yeniden başlatmalar arasında ne kadar çalıştığı, her instance'ın kaç bağlantı tuttuğu ve broker yük altındayken bir yenilemenin ne kadar sürdüğü. Yalnızca örnek olarak, otuz dakikada bir yenilenen bir saatlik lease düşünün.

A/B rotation'a göre kazanç, sızan bir credential'ın kendiliğinden sona ermesidir. Login, o instance için oluşturulduğundan instance'ı adlandırır; böylece veritabanı log'u "o statement'ı kim çalıştırdı?" sorusunu workload'a kadar cevaplar. Bu, Bölüm 3'ün vaat ettiğini sağlar: tek bir workload instance'ına karşılık gelen bir login.

Workload identity ve IAM tabanlı veritabanı kimlik doğrulaması

Saklanan parolayı tamamen ortadan kaldırmanın bir yolu var. Bazı platformlar, veritabanının platformun kendi kimlik sistemine güvenmesine izin verir: workload platformdan kısa ömürlü bir token ister ve veritabanı bu token'ı parola yerine kabul eder. Cloud sağlayıcıları bunu IAM database authentication gibi adlarla sunar ve bazı veritabanları harici bir identity provider'dan gelen token'ları doğrulayabilir.

Cazibesi, saklanacak, fetch edilecek ya da rotate edilecek bir secret'ın olmamasıdır. Token, ihtiyaç duyulduğunda workload için üretilir, kısa ömürlüdür (tipik olarak dakikalar) ve platformun zaten bildiği kimliğe bağlıdır. İptal platformun işidir: kimliğin token isteme hakkını kaldırmak yeni token'ları durdurur ve zaten verilmiş token'lar, süreleri dolana kadar kabul edilebilir. Veritabanındaki login hâlâ vardır, hâlâ `app_*` rolünün üyesidir ve hâlâ log'da görünür.

Maliyetler belirlidir:

- **Platform bağımlılığı.** Tasarım platformun desteklediği yerde çalışır ve başka hiçbir yerde çalışmaz; bu yüzden iki platformda workload'ları olan bir ekip örüntüyü iki kez kurar.
- **Eşleme.** Birinin hangi platform kimliğinin hangi login ve role karşılık geldiğini söylemesi gerekir. Bu eşleme bir grant'tir ve diğer her grant gibi aynı review'u hak eder (Bölüm 14).
- **Açık bağlantılar.** Yaygın engine'lerde token, bir bağlantı açıldığında kontrol edilir. Her süre dolusunda olduğu gibi, açık bir bağlantı token'ından uzun yaşar ("Süre dolumu ve revoke"a bakın); bu yüzden bağlantı ömrüne bir sınır yine gereklidir (Bölüm 5).

Üç yaklaşımdan hangisinin kullanılacağı, modadan çok platformun neler sunduğuna ve ekibin neyi işletebildiğine bağlıdır.

Yaklaşım	Saklanan secret var mı?	Ömür	Gerektirdikleri	Ana maliyet
Bir secret manager'da statik credential, A/B rotation	Evet, servis başına bir çift	Bir sonraki rotation'a kadar	Secret manager, rotation job'ı	Consumer'lar yeniden okunmalı; rotation job'ı çalışmaya devam etmeli
Bir broker'dan dinamik credential'lar	Yalnızca broker'ın kendisinin	Instance başına dakikalardan saatlere	Broker, ledger'ly_access_admin, lease sweeper	Broker, yüksek değerli ve her zaman erişilebilir olması gereken bir bileşendir
Workload identity token'ları	Veritabanı için hiç	Kısa ömürlü, tipik olarak dakikalar	Platform ve veritabanı desteği	Platform bağımlılığı; token'lar yalnızca bağlanırken kontrol edilir

Bunların hiçbirisi credential problemini ortadan kaldırmaz; her biri onu taşır. Broker'ın kendi login'i statik bir credential'dır, secret manager'da tutulur, broker'ın workload identity'si tarafından fetch edilir ve A/B rotate edilir. Token tasarımı platformun kimlik sisteminin doğru olmasına dayanır. Değişen şey, kaç secret olduğu, her birinin ne kadar süre işe yaradığı ve onları kimin okuyabildiğidir.

TLS ve sertifika doğrulaması

Yukarıdakilerin hepsi bir credential'ın nerede tutulduğu ve ne kadar sürdüğüyle ilgilidir. Veritabanına ulaşmadan önce bir ağı geçer ve şifrelenmemiş bir bağlantı parolayı ve veriyi, yoldaki her şeyin okuyabileceği bir biçimde taşır. Aktarım sırasında şifreleme bunu kapatır ve temel çizgidir. Her ayakta geçerli olmalıdır. Bir istek veritabanına ulaşmadan önce çoğu zaman bir pooler'dan ya da gateway'den geçer ve bunların her biri, kendi şifrelemesi ve kendi sertifika kontrolü olan ayrı bir bağlantıdır. Client'tan pooler'a olan ayağı şifrelemek, pooler'dan veritabanına olan ayak hakkında hiçbir şey söylemez.

Atlaması daha kolay ikinci bir iş var: sunucunun doğru sunucu olduğunu bilmek. Bir bağlantı, veritabanı olmayan bir hedefe şifrelenmiş olabilir. **Sertifika doğrulaması**, client'ın sunucunun sertifikasının güvendiği bir otorite tarafından verildiğini ve sertifikanın ulaşmak istediği host'u adlandırdığını kontrol etmesidir. Onsuz şifreleme, trafiği okuyan birine karşı korur ve bağlantıyı cevaplayan birine karşı hiçbir şey yapmaz.

Önemli olan ayar, client'ın doğrulama modudur. Çoğu veritabanı client'ı şifreleyen ama doğrulamayan bir mod sunar ve süresi dolmuş ve self-signed olanlar dahil her sertifikayla çalıştığı için popülerdir. Bu modu işin yarısı olarak görüyorum: trafiği pasif bir gözlemciden gizler ve sunucunun doğru olduğuna dair yanlış bir güven verir. Çare, sertifika otoritesini her servise dağıtmak ve adı doğrulamaktır.

Bu kendi maliyetlerini getirir. Sertifikaların süresi dolar ve süresi dolmuş bir sunucu sertifikası, üzerinde tam bir tarih olan bir kesintidir; bu yüzden süre dolumu izleme ve prova edilmiş bir yenileme prosedürü gerektirir. Otorite sertifikasını her consumer'a dağıtmak iştir ve otorite değiştiğinde güncellenmesi gerekir. Sertifikanın credential olduğu client sertifikaları, ancak bir şey onları otomatik olarak verirse kısa ömürlerle çalışır.

Bir credential sızdığında

Yukarıdaki her kontrolle bile bir credential sızabilir ve tasarım bir sonraki adımda ne olacağını önceden söylemelidir. İlk adım, paniğin birleştirdiği iki soruyu ayırmaktır: credential'ın çalışmasını nasıl durdururuz ve çalıştığı sürece ne yaptık?

Dinamik bir credential için ilk cevap bir revoke'tur: lease'i bitir, session'ları kapat ve login, broker'ın harekete geçmesi için gereken süre içinde gider. A/B düzenindeki statik bir credential için, şüphelenilen bir sızıntı planlı rotation değildir. Planlı job eski login'in drain olmasını bekler ve sızan bir değer, üzerinde bir session tutan olduğu sürece geçerli kalır. Bir sızıntıda önce sızan login'i kesin: rollerini kaldırın, session'larını kapatın ve parolasını drain'i beklemeden değiştirin; sonra yeni bir değeri boştaki login'e yayımlayın ve aktif işaretçiyi çevirin. Eski login'i hâlâ tutan instance'lar bağlantılarını kaybeder ve secret'ı yeniden okuyana kadar başarısız olur. Bu bir availability maliyetidir ve ben bunu öderdim: çalışmaya devam eden sızmış bir credential daha büyük problemdir.

Secret'ını yeniden okuyamayan bir servis restart gerektirir ve ekip, önem kazandığı günden önce bu servislerin hangileri olduğunu bilmelidir. Başlangıç durumundaki paylaşılan admin parolası için ilk cevap, bu bölümün başladığı kesintidir; bu da işi bir olaydan önce yapmanın en güçlü gerekçesidir.

İkinci soruyu log cevaplar. Tek bir workload instance'ına karşılık gelen bir login, çalıştırdığı statement'ları, dokunduğu tabloları ve aktif olduğu saatleri görmenizi sağlar. Bu, review'u bütün veritabanı yerine tek bir rolün blast radius'una daraltır. Adım adım prosedür, hareket sırası ve nasıl prova edileceği, olaylar sırasındaki erişimi ele alan Bölüm 15'e aittir. Buradaki nokta, revoke'u rutin bir adım olarak kurmuş bir ekibin onu kötü bir günde icat etmeden kullanabilmesidir.

Veri koruma ve ödeme kartı rejimleri, erişimin nasıl grant edildiğini, ne kadar sürdüğünü ve nasıl revoke edildiğini sorma eğilimindedir. Burada anlatılan lease'ler, revoke adımları ve audit girdileri, bu sorulara cevap veren kayıtlardır.

Ledgerly'nin migration planı

Paylaşılan admin parolasından kısa ömürlü credential'lara geçiş, küçük, geri alınabilir adımların bir dizisidir. Onları, her adımın ne kadar riski kaldırdığına ve ne kadar az şeyi bozabileceğine göre sıralardım.

1. **Admin parolasının her kopyasının envanterini çıkarın.** Servisler, pipeline değişkenleri, export adımı, parola yöneticisi, client araçlarındaki kayıtlı bağlantılar. Liste iş planıdır. Bölüm 1'in envanteri başlangıç noktasıdır.
2. **Servis başına, kendi `app_*` rolünde bir login oluşturun.** Bölüm 3'teki rolleri kullanın. Bu aşamada credential'lar statiktir, ama her biri ayrıdır, secret manager'da saklanır ve kimlik üzerinden okunur.
3. **Servisleri admin login'inden tek tek çıkarın.** Blast radius'u en küçük olan `reporting-job` ile başlayın, sonra `invoice-worker`, `admin-console` ve `billing-api`. `payments-service`'i en sona bırakın, çünkü `payment_tokens` ve `bank_accounts`'a ulaşan tek servis odur ve orada bir hata en pahalıya mal olur. Her geçiş, geri alınabilen bir deploy'dur.
4. **Migration'lara ve export adımına kendi login'lerini verin.** Pipeline `ledgerly_migrator` içinde kısa ömürlü bir login kullanır (Bölüm 11);

rolün kendisi kalıcı owner yetkileri taşır ve yalnızca credential kısa ömürlüdür. Export adımı `app_vendor_export`'u kullanır (Bölüm 13). Bu adımdan sonra hiçbir admin login'ini kullanmaz.

5. **Her servisi A/B rotation'a geçirin.** Her servisin, bir kimlik doğrulama hatasından sonra secret'ını yeniden okumasını sağlayın, ilk rotation'ı sakın bir dönemde çalıştırın ve drain kontrolünü izleyin. Drain olmayan bir servis bir bulgudur ve bu adımın amacı o bulgudur.
6. **Kolay yenilenen servisler için broker'ı devreye alın.** Sık yeniden başlayan ve az bağlantı tutan bir servisle başlayın. Ekip lease'lerin nasıl davrandığını öğrendikçe diğerlerini ekleyin. Statik A/B'yi fallback olarak tutun.
7. **Paylaşılan admin login'ini emekliye ayırın.** Onu her yapılandırmadan ve parola yöneticisinden kaldırın. Sahiplik `ledgerly_owner`'a geçtikten sonra (Bölüm 3), yetkilerini sökün ve silin ya da devre dışı bırakın. Admin, bir managed servisin master kullanıcısı gibi, engine'in superuser'ı çıkarsa, Bölüm 3'ün mühürlü alet dediği şey olur. Bu, günlük işte kimsenin kullanmadığı bir credential'dır. Platform ekibinin alabileceği bir yerde tutulur ve her alış kaydedilir. Credential'ını, envanterin tamamı temizlendikten sonra bir kez değiştirin.

Bir ekip beşinci adımdan sonra durabilir ve yine de çok daha iyi durumda olur: her servisin kendi credential'ı vardır, onu bir kimlik üzerinden okur ve kesinti olmadan rotate eder. Broker, onu işletebilen ekipler için bir eklentidir.

Trade-off'lar ve maliyetler

Bu bölümdeki her hamle maliyeti bir olaydan bir rutine kaydırır. Rutinin pahalılaştığı yer burası.

Secret manager'a bağımlılık. Credential'lar bir secret manager'dan gelmeye başladığında, ulaşılamayan bir secret manager yeni instance'ların başlamasını durdurur ve kısa lease'lerle sonunda çalışan instance'ların yenilemesini de durdurur. Bu onu veritabanı yolunun availability'sinin bir parçası yapar. Onu veritabanına gösterdiğiniz özenle işletin: birden fazla instance, test edilmiş bir recovery ve onun var olduğunu bilen bir on-call

rotation'ı. Bir credential'ı lease süresi boyunca cache'lemek kısa bir kesintiye yumuşatır ve revoke edilmiş bir credential'ın hâlâ kullanılabileceği süreyi uzatır.

Bağlantı fırtınaları. Bir servisin her instance'ı aynı dakikada yenilerse ya da bir deploy'dan sonra birlikte yeniden başlarsa, veritabanı yeni bağlantılardan ve kimlik doğrulamalardan oluşan bir patlama görür. Kimlik doğrulama bedava değildir ve kendi yarattığı bir patlama, hiçbir dış neden olmadan diğer her rol için veritabanını yavaşlatabilir; bu bir bağlantı fırtınasıdır. Yenilemeleri jitter ile kaydırın, bağlantı havuzunu yenilemeler arasında kararlı tutun ve bir rolün kaç bağlantı tutabileceğini Bölüm 5'in şekillendirmesine izin verin.

Daha zor debugging. Bir saat var olan bir login, ertesi sabah biri log'u okuduğunda gitmiştir ve `billing-api-4f2a` gibi bir ad lease kaydı olmadan hiçbir şey ifade etmez. Lease'leri ve kimliklerini audit log tutulduğu sürece tutun ve on-call olana login'den instance'a bir arama yolu verin. Bu arama eksikse, kısa ömürlü credential incelemeyi hızlandırmaz, yavaşlatır.

Yüksek değerli bir broker. Broker'ın login'i her application rolünde login oluşturabilir; bu yüzden bir saldırganın çabasına değer. Kodunu küçük ve review edilmiş tutun ve ona, koruduğu şeye gösterdiğiniz dikkati gösterin; daha azını alan bir broker başlı başına bir risktir.

Bunu ne zaman kullanmamalı. Tek servisi, bir avuç mühendisi ve managed bir secret store'u olan bir ekip, secret manager'da tutulan ve yılda iki kez elle A/B rotate edilen ayrı bir statik credential ile iyi hizmet alabilir. Broker, çalıştırılacak bir servis daha ve ekibin öğreneceği bir sistem daha demektir; belli bir büyüklüğün altında, eklenen hareketli parçalar onlediklerinden fazla kesintiye yol açar. Broker kapalıyken kime alarm düşeceğini söyleyemiyorsanız, beklerdim. Platform sunduğunda workload identity token'ları en ucuz adımdır ve yine de eşlemenin review edilmesini gerektirir.

Sık yapılan hatalar

- **Secret manager'ı cevabın tamamı saymak.** Cazip gelir, çünkü geçiş görünür ve hızlıdır. Her servis aynı path'i okuyorsa paylaşılan parola

hâlâ paylaşılyordur. Her servise kendi path'ini ve kendi login'ini verin ve her birini kimin okuyabildiğini review edin.

- **Kimsenin tüketmediği bir rotation.** Bir job değeri rotate eder, ama servisler onu başlangıçta okur ve bir daha okumaz; böylece yeni değeri, eski değeri emekliye ayrılıp bir şey başarısız olana kadar okunmadan durur. Consumer'ların bir rotation'ı aldığını test edin ve drain kontrolünü job'ın parçası yapın.
- **Gömülü bir değere geri dönmek.** Image'daki "bilinen son iyi" bir kopya başlangıcı güvenilir kılar ve tasarımın kaldırdığı dosyayı geri getirir. Başlangıcı reddedin, alarm verin ve bağımlılığı onarın.
- **Doğrulamadan şifrelemek.** TLS'i açmak bir kontrol listesini tatmin eder ve yalnızca şifreleme modu bağlantıyı çalıştırır. Sertifikayı ve host adını doğrulayın ve süre dolumu tarihini izleyin.
- **Tavansız ömürler.** Sınırsız yenileme, uzun ömürlü bir credential'ı yeniden kurar. Bir lease'in toplam ömrüne bir sınır koyun ve instance'ın kimliğiyle yeniden kimlik doğrulamasını sağlayın.
- **Broker'ın rolünü küçük saymak.** `ledgerly_access_admin`, broker'ın işinden daha geniştir; bu yüzden ele geçirilen bir broker, yönettiği rollerin uzandığı yere kadar ulaşabilir. Onu hangi principal'ların çağırabileceğini ve her birinin hangi `app_*` rollerini isteyebileceğini kısıtlayın ve audit girdilerini review edin. Geniş rolü bir sınır olarak değil, kabul ettiğiniz bir maliyet olarak görün.

Temel çıkarımlar

- Bir credential'ın her kopyası bir okuyucudur; bu yüzden en yararlı değişiklik kopya sayısını azaltmak ve kalan her okumaya bir ad vermektir.
- Rotation zordur, çünkü bir credential paylaşılan durumdur; servis başına iki login, diğeri çalışmaya devam ederken birini değiştirmeyi mümkün kılar.
- Consumer'ların almadığı ya da kimsenin çalıştırmadığı bir rotation bir kontrol değil, bir takvim kaydıdır; bu yüzden drain kontrolü ve çalışmayı bırakmış bir rotation job'ı için bir alarm tasarımının parçasıdır.
- Dinamik credential'lar her workload instance'ına, yerleşik bir sonu olan kendi login'ini verir; bedeli bir broker, bir lease kaydı ve secret

manager'a bir bağımlılıktır.

- Workload identity, platformun desteklediği yerde saklanan parolayı kaldırır ve güveni platforma ve kimlik ile login arasındaki eşlemeye taşır.
- Sunucunun sertifikasını ve adını doğrulamak, TLS'in credential'ı korumasını sağlayan şeydir; doğrulamasız şifreleme trafiği gizler ve sunucunun kimliğini denetlenmemiş bırakır.
- Bir credential sızdığında, revoke'un hızı ve log'un netliği olayın ne kadar sınırlı kalacağına karar verir ve bu, sızıntıdan önce yapılan işe bağlıdır.

Credential'lar kimin içeri gireceğine karar verir; kaç bağlantı açtıkları ya da bir statement'ın primary'yi ne kadar tutabileceği hakkında hiçbir şey söylemez. Bölüm 5 bağlantının kendisini bir politikaya çevirir: rol başına limitler, pooling, timeout'lar ve başıboş bir sorguyu bulmayı kolaylaştıran etiketler.

Bölüm 5 — Politika Olarak Bağlantılar

Ledgerly'de sakın bir salı gecesini düşünün. Bir ekip arkadaşı `reporting-job` içindeki gece toplulaştırmasının paralelliğini artırdı, böylece iş sabaha kadar bitecekti. Review'da zararsız görünen bir değişiklikti. Saat 01:40'ta job başlıyor, her worker thread'i için bir bağlantı açıyor ve paylaşılan `admin login`'iyle `primary`'den `payments` ve `invoices` okumaya koyuluyor. Birkaç dakika içinde `billing-api`, saatin çoktan sabah olduğu bir saat diliminde müşterilere hata döndürmeye başlıyor. On-call'daki Robin, müşterilerin yaşadığı bir kesinti için alarm alıyor.

Robin veritabanını açıyor ve hiçbir şeyin bozuk olmadığını görüyor. Primary ayakta, disk etkinliği normal, hiçbir statement yanlış bir şey yapmış değil. Sunucunun sabit sayıda bağlantı slotu var, rapor bunların çoğunu almış ve API kapıda bekliyor. Her çağırın aynı login'i kullandığı için veritabanı sebebin rapor olduğunu söyleyemiyor. Yalnızca slotların dolu olduğunu söyleyebiliyor.

Bu bölüm, kimlik ile veri arasında duran bir soruyu yanıtlıyor: bağlantının kendisini politikanın uygulandığı bir yer hâline nasıl getiririz? Bölüm 1'deki beş parçalı modelde iki parça üzerinde çalışır. Bağlantı, Yol'dur; çünkü servislerin nasıl ve kaç tanesinin geleceğine pooler karar verir. Aynı zamanda Yürütme'nin ilk sınırıdır; çünkü limitler ve timeout'lar, tek bir çağırının ve tek bir statement'ın herkesin payından ne kadar alabileceğini söyler. Boyunca izlenen örnekten gelen adım şudur: bağlantılar bir pooler ile rol başına limitler ve timeout'larla biçimlendirilir.

Problem ve neden zor olduğu

Bağlantı, her şeyden önce bir kaynaktır. Çoğu engine'de her biri sunucuya bellek ve zamanlamadan bir pay kaybettirir, sunucu da yalnızca sabit sayıda bağlantıya izin verir. Limit çağırının değil sunucunun bir

özelliğidir; bu yüzden çok bağlantı isteyen ilk çağırıcı çok bağlantı alır. Varsayılan kurulumda, gece raporunun müşteriye bakan API'den daha az slotu sahip olması gerektiğini söyleyen hiçbir şey yoktur.

Parola bunların hiçbirine dokunmaz. Kimin içeri gireceğine karar verir; kaç slot tuttuğu ya da ne kadar süre tuttuğu hakkında hiçbir şey söylemez. Credential dağıtmaktan fazlasını anlatan bir kitabın bağlantılara bir bölüm ayırmasının nedeni budur.

Üç alışkanlık sorunu kötüleştirir. Birincisi, bağlantı sayısı bir toplam değil bir çarpımdır: bir servisin instance sayısı, her instance'ın pool büyüklüğüyle çarpılır; autoscaler bir instance eklediğinde de yeniden çarpılır. Diyelim ki bir ekip, on instance'a ölçeklenen bir servis için yirmilik bir pool ayarladı. İki yüz bağlantı istemiş olur ve büyük ihtimalle bunu bilerek kararlaştırmamıştır. İkincisi, bir bağlantı statement'ı çalıştığı sürece tutulur; yani yavaş bir sorgu bütün o süre boyunca bir slotu işgal eder, oysa slotlar bir sonraki çağırıcının ihtiyaç duyduğu şeydir. Üçüncüsü, başarısız olan çağırıcı daha çok zorlama eğilimindedir. Timeout'a düşüp hemen yeniden deneyen bir client, yavaş bir veritabanını kapıda toplanmış bir kalabalığa çevirir.

Akla gelen ilk karşılık sunucu limitini yükseltmektir. Bir hafta işe yarar, sonra bir sonraki olayı daha büyük yapar. Daha çok bağlantı, aynı primary üzerinde daha çok eşzamanlı iş demektir; yüz statement'a izin verildi diye veritabanı aynı anda yüz statement çalıştırmakta hızlanmaz.

Kontroller, kimsenin birlikte sahiplenmediği birkaç yerde yaşar: rol tanımı, uygulamanın pool'u, varsa bir pooler ve bir statement'ın ne kadar çalışabileceğine karar veren ayarlar. Availability, Bölüm 2'deki tehdit modelinde söz konusu olan özelliklerden biridir ve onda kazalar baskındır. Paylaşılan bir login her kazayı herkesin kazasına çevirir.

Yani başlangıç durumunda beş servis ve VPN üzerindeki her mühendis, tek bir isim altında tek bir sunucu slotu havuzundan beslenir. Her client kütüphanesinin varsayılanı dışında timeout yoktur ve hangi çağırıcının hangi bağlantıyı tuttuğuna dair bir kayıt yoktur. Aşağıdaki bölümler bunun yerine şunları koyar: rol başına bir bütçe, onu uygulayan bir pooler, hiçbir şeyin bir slotu ne kadar tutabileceğine üst sınır koyan timeout'lar ve her bağlantıya sahibini söyleyen bir etiket.

Rol başına bir bağlantı bütçesi

İlk adım en ucuz olanıdır. Bölüm 3, her servise kendi uygulama rolünü verdi; bu da bir limitin sonunda bir isme bağlanabileceği anlamına gelir. Rol başına limit şunu söyler: bu principal, kaç instance çalıştırırsa çalıştırsın ve ne kadar zorlarsa zorlasın en fazla bu kadar bağlantı tutabilir. Limite ulaştığında o rolden gelen yeni bağlantılar reddedilir ve diğer her rol çalışmaya devam eder.

Politika, bu reddin kendisidir. Başarısızlığı "veritabanı herkes için dolu" durumundan "bu çağırının slotu kalmadı" durumuna taşır; ikincisi Robin'in alması gereken çok daha iyi bir alarmdır. Açgözlü bir çağırının blast radius'u çağırının kendisi olur ve ret çağırını adıyla anar; bu da aynı adımın hesap verebilirlik yarısıdır.

Sayılar bir formül değil, akıl yürütme ister. Ben sunucunun limitinden aşağıya doğru çalışırdım. Önce başka hiçbir şeyin kullanamayacağı bir rezervi çıkarırdım; çünkü sistemi onaranların ve araçların da bağlanması gerekir. Rezerv, platform session'larını ve `ledgerly_access_admin`'i kullanan üç principal'ın login'lerini kapsar (rol tanımlarını uygulayan pipeline adımı, credential broker ve access gateway). Sonra kalanı, her uygulama rolüne en yoğun anındaki ihtiyacına ve biraz da payına göre bölerdim. Aşağıdaki tablo primary için bir örnektir. Sayılar varsayımsaldır, bir öneri değildir.

Principal	Rol	Bağlantı limiti (örnek)	Neden bu büyüklük
<code>billing-api</code>	<code>app_billing_api</code>	80	En yoğun servis; web uygulamasına ve herkese açık API'ye hizmet verir
<code>invoice-worker</code>	<code>app_invoice_worker</code>	30	Kuyruk güdümlü; eşzamanlılığı worker sayısı belirler
<code>payments-service</code>	<code>app_payments_service</code>	30	Düzenli; statement'lar arasında dış sağlayıcılarla konuşur
<code>admin-console</code>	<code>app_admin_console</code>	15	Birkaç insanın yaptığı dar kapsamlı, audit edilen işlemler
Migration'lar	<code>ledgerly_migrator</code>	3	Pipeline'dan, aynı anda tek migration

Principal	Rol	Bağlantı limiti (örnek)	Neden bu büyüklük
Primary üzerindeki insan session'ları	Gateway'in oluşturduğu, JIT ve break-glass türündeki session login'leri	8	Burada yalnızca onaylanmış isteği olan session'lar çalışır (Bölüm 6); rutin okumalar <code>ledgerly_replica_2</code> 'ye gider. <code>human_jit_write</code> ve <code>human_breakglass</code> rolleri limit taşımaz; çünkü hiçbir login ilkinin üyesi değildir, ikincisi ise session başına grant edilir
<code>reporting-job</code>	<code>app_reporting_job</code>	10	Yalnızca geçiş dönemi: başlangıç durumunda ve taşıma sırasında <code>primary</code> 'den okur, hedef durumda bu satır sıfıra iner
Rezerv	erişim yönetimi ve platform	10	Yalnızca onarım ve login verme için
Ayrılmamış	yok	14	Pay; böylece bir yuvarlama hatası kesintiye dönüşmez

Sunucunun 200 bağlantıya izin verdiğini varsayalım. Ayrılmış satırlar 186'ya toplanır ve kalan 14 bilerek atanmadan bırakılır. Rutin insan session'ları bu tabloda yer almaz, çünkü primary üzerinde çalışmazlar. Onların bütçesi, rutin okumaların gittiği `ledgerly_replica_2` üzerinde belirlenir; diyelim orada 20 slot. `reporting-job` satırı yalnızca rapor hâlâ `primary`'den okurken vardır. Hedef durumda `ledgerly_replica_2`'den okur ve o host üzerinde, bağlantı katmanında pooler'da belirlenen kendi bütçesi olur (taşımayı Bölüm 6 anlatır). Rol tanımları limitleri şuna benzer bir notasyonla taşıyabilir; bu notasyon örnek amaçlıdır ve hiçbir engine'e ait değildir:

```
// illustrative notation: the limit belongs to the role, not to a
person
ROLE LIMIT app_billing_api      CONNECTIONS 80;
ROLE LIMIT app_invoice_worker   CONNECTIONS 30;
ROLE LIMIT app_payments_service CONNECTIONS 30;
ROLE LIMIT app_admin_console    CONNECTIONS 15;
ROLE LIMIT ledgerly_migrator     CONNECTIONS 3;
ROLE LIMIT app_reporting_job     CONNECTIONS 10;
```

Not: Engine'ler, bir limitin ya da ayarın neye bağlandığı konusunda farklılaşır. Kimi bağlantıları rol başına, kimi login başına, kimi yalnızca veritabanı ya da sunucu başına sayar. Bazı engine'lerde bir role yazılan limit ya da varsayılan, yalnızca rolün kendisi olarak login olan bağlantılara uygulanır ve o rolün üyesi olan login'lere miras kalmaz. Bölüm 4, her servis instance'ına `app_*` rolünün üyesi olan kendi dinamik login'ini verir; dolayısıyla bu engine'lerde rol düzeyindeki limit ve aşağıdaki rol düzeyindeki timeout'lar sessizce uygulanmaz.

Bunları taşıyabilecek iki yer daha vardır: her login'i oluştururken login başına limiti ve aynı ayarları koyabilen broker ve ayarları taşıyabilen ama bir rolün login'lerini tek bir toplamda birleştiremeyebilen pooler. Aşağıdaki pooler bölümü, bunu yapamadığında rol toplamını elde etmenin üç yolunu sıralar.

İki özellik tabloyu yararlı kılar. Limitler bağlantının başladığı yerde denetlenir; bu yüzden çağırın reddi ilk yavaş statement'tan sonra değil, hemen öğrenir. Ve rezerv her uygulama rolünün dışında durur; böylece en kötü gecede bile platform ekibi bir şeyleri düzeltmek için login olabilir. Bütün sunucuyu dolduran bir bütçe, paylaşılan admin parolasıyla aynı kusuru taşır: biri ona en çok ihtiyaç duyduğu ana kadar çalır.

Toplamların muhafazakâr olması gerekmez. `billing-api` ile `payments-service` nadiren aynı anda zirve yapıyorsa, uygulama limitlerinin toplamının sunucunun izin verdiği kadar fazla olmasına izin verebilir ve iki eşzamanlı zirvenin çıkışabileceğini kabul edebilirsiniz. Bunu ancak rezerv ayrılmışken ve ekip gerçek kullanımı bir süre izledikten sonra yapardım. Varsayılan, toplamı tutan bir bütçedir; çünkü ihlal edilemeyen bir bütçe gece üçte akıl yürütmesi daha kolay olandır.

Pooling: bağlantıların paylaşıldığı yer

Bir pool, açık bağlantılardan oluşan bir küme tutar ve bunları çağıranlara ödünç verir; böylece her istek yeni bir bağlantının bedelini ödemez ve veritabanı sınırlı sayıda bağlantı görür. Bir pool koymak için iki yer vardır.

Neredeyse her servisin veritabanı kütüphanesinde zaten bir pool vardır ve büyüklüğü, ulaşmayı umduğu trafiğe göre değil, bir instance'ın aynı anda yapabileceği işe göre ayarlanmalıdır. Çağırانları kısa süre bekleten küçük bir pool, veritabanına büyük bir pool'dan daha nazık davranır ve kısa bir beklemeden sonra, servisin net bir yanıtı çevirebileceği bir hatayla vazgeçmelidir. Küçük başlayın, çağırانların ne kadar beklediğini izleyin ve büyüklüğü kanıtı dayanarak artırın.

Veritabanının önündeki pooler

Pooler, asıl işi çok sayıda client bağlantısını kabul edip daha az sayıdaki sunucu bağlantısı üzerinden multiplex etmek olan bir veritabanı proxy'sidir. PgBouncer ve ProxySQL yaygın örneklerdir. Servis, pooler'a veritabanıymış gibi bağlanır ve gerçek bir bağlantının ne zaman ödünç verileceğine pooler karar verir.

Kitabın durmadan sorduğu soru burada da geçerlidir: veritabanı kimin adını görüyor? Bölüm 4, her servis instance'ına credential broker'ın oluşturduğu ve servisin `app_*` rolünün üyesi olan kendi dinamik login'ini verdi. Pooler bu modele uymak zorundadır ve ben onu şu biçimde kurardım. Instance, pooler'a kendi login'ini ve credential'ını sunar. Pooler bunları veritabanına karşı doğrular ve sunucu bağlantılarını aynı login altında tutar. Kendi parola deposu yoktur ve her client'ın paylaştığı kendi login'i yoktur.

Rotate edilmiş ya da süresi dolmuş bir credential pooler'a pooler üzerinden değil, veritabanı üzerinden ulaşır. Bir lease bittiğinde ya da revoke edildiğinde broker login'i siler (Bölüm 4); böylece pooler'ın o login altında kimlik doğrulama ya da sunucu bağlantısı açma yönündeki bir sonraki denemesi reddedilir ve pooler, o login için tuttuğu boştaki sunucu bağlantılarını kapatır.

Rol başına limitin yine de nasıl tutulacağını evrensel bir yanıt yoktur ve ben pooler'ınızın bir yanıt verebileceğini varsaymazdım. Birçok pooler limitlerini login ve veritabanı çifti başına tutar, bazıları login başına ya da veritabanı başına bir limit ekler. Birçok farklı login'in bağlantılarını rol düzeyinde tek bir tavanda toplayan genel bir yetenek bilmiyorum; bu yüzden bu bölümün sözüne güvenmek yerine pooler'ınızın dokümantasyonunu okuyun. Sizininki bunu yapamıyorsa rol toplamı başka bir yerden gelmelidir. Üç yer vardır ve her birinin bir bedeli vardır.

- **Pooler, upstream'e rol başına tek bir kalıcı login ile bağlanır.**

Instance'ın dinamik credential'ı o zaman client'ı yalnızca pooler'a karşı doğrular ve pooler'ın login başına limiti, tam olarak rolün limiti olur.

Bedeli, Bölüm 3'ün kimlik modelidir:

- Veritabanı logu rol başına tek bir isim gösterir; dolayısıyla bir instance'a atfetme, pooler'ın loguna ve aşağıda anlatılan uygulama etiketine kayar.
- Pooler dinamik credential'ları kendisi doğrulamak zorundadır; bu da onu broker'ın lease durumunun tüketicisi yapar.
- Ayrıca her rol için, Bölüm 4'ün anlattığı gibi korunacak ve A/B olarak rotate edilecek uzun ömürlü bir credential tutar.
- Broker'da bir instance'ın login'ini revoke etmek artık onun sunucu bağlantılarını kapatmaz; o client'ın session'larını pooler'ın sonlandırması gerekir.

- **Veritabanı, engine üye login'ler için destekliyorsa rol başına bir limit uygular.** Tam isabetlidir, hiçbir şeyin aşamayacağı bir yerde durur ve instance login'leri ile onlara yapılan atıf bozulmadan kalır. Bedeli, yukarıdaki notun dediği gibi desteğin değişmesidir. Çalıştığı yerde ret, bağlantı bir kimlik doğrulamaya mal olduktan sonra veritabanından gelir ve hiçbir şey multiplex edilmez.

- **Broker, rol başına kaç login vereceğini sınırlar.** Her login oluşturulurken login başına bir limit konulduğunda, rolün toplamı login sayısı çarpı bu limiti aşamaz. Bedeli, bunun kesin bir sayı değil bir üst sınır olması ve tavanın autoscaler'ın instance tavanıyla uyuşması gerektiğidir. Tavanı aşan bir scale-out, credential alamayan bir deploy'a dönüşür; bu bir availability problemidir ve eski ile yeni login'in kısa süre üst üste bindiği yenilemelerin tavanın altında yer bulması gerekir.

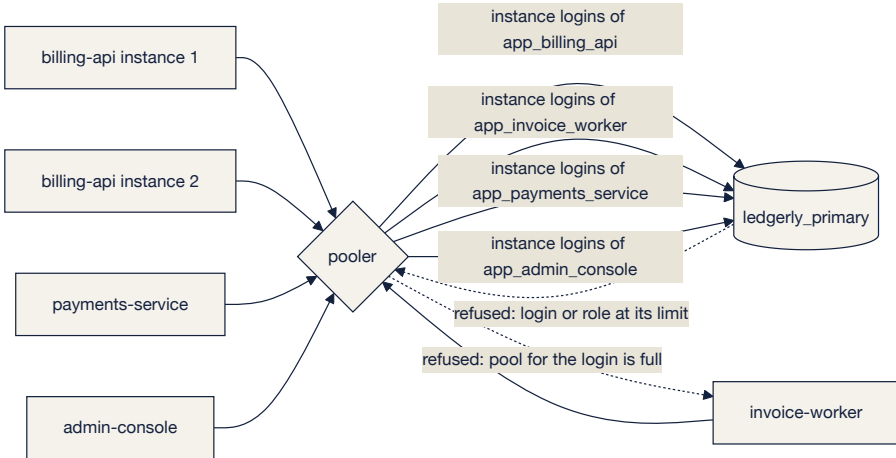
Engine üye login'ler için destekliyorsa önce veritabanı tarafındaki limiti denerdim. Desteklemiyorsa broker tavanı kimlik modelini korur ve bakımı yapılacak bir tavana mal olur. Rol başına tek upstream login'ini ancak bağlantı sayıları instance'lar arasında multiplex etmeyi zorunlu kıldığında seçerdim ve logun artık neyi söylemediğini yazıya dökerdim. Üçü de, her rolün her client'ını tek bir paylaşılan login'e eşleyen tasarımdan iyidir. O tasarım paylaşılan admin hesabını bir katman aşağıda yeniden yaratır: log yine tek bir isim gösterir ve bir limit herkese aynı anda uygulanır.

Önerilen pass-through biçiminde pooler çok az multiplexing sağlar. Transaction modunda bir instance'ın client'ları o instance'ın sunucu bağlantılarını paylaşır ama iki instance birbiriyle paylaşmaz; dolayısıyla bir rolün toplamı en az, canlı instance sayısı çarpı küçük bir asgari değerdir. Pooler'ın sağladığı üç başka şey vardır.

- **Merkezî ret.** Tek bir yer, login ve pool başına, her client kütüphanesi için aynı biçimde hayır der.
- **Her servis için aynı limitler.** Her servisteki pool büyüklükleri ve beklemler tek savunma hattı olmaktan çıkar.
- **Bir ağ darboğazı noktası.** Servisler veritabanına yalnızca pooler üzerinden girer. Hangi host'ların bir TCP bağlantısı açabileceğini söyleyen, Bölüm 6'nın tanımladığı ve Bölüm 7'nin yazdığı ağ politikası pooler host'larını kabul eder, başkasını etmez; böylece bir servisten sıran credential, pooler'ın erişilebilir olmadığı hiçbir yerden işe yaramaz. Bölüm 4'ün dediği gibi erişim alanı bir sızıntıyı sınırlar ve pooler bu sınırın söylenmesinin kolay olduğu yerdir.

Bedeli, aşağıdaki trade-off'larda ele alınan, yolda yeni bir sistemdir. Tek instance'lık tek bir küçük servis için uygulamanın pool'u ve bir rol limiti yeterlidir; instance sayısı ya da tek bir darboğaz noktası ihtiyacı aksini söyleyene kadar pooler eklemeye değmez.

Aşağıdaki diyagram, instance login'lerini veritabanına görünür bırakan pass-through biçimini gösterir. Bir çağırının reddedildiği iki yere bakın: tek bir login'in pool'u dolduğunda pooler'da ve bir login ya da engine destekliyorsa rol limitinde olduğunda veritabanında.



Session modu ve transaction modu

Bir pooler bir sunucu bağlantısını belirli bir süre için ödünç verir ve bu sürenin uzunluğu onun **modudur**. Session modunda client, bağlanmadan bağlantıyı kesene kadar aynı sunucu bağlantısını tutar; pooler bağlanma maliyeti dışında hiçbir şey kazandırmaz. Transaction modunda client bir sunucu bağlantısını yalnızca tek bir transaction süresince tutar ve bir sonraki transaction başka bir bağlantıda çalışabilir. Transaction modu çok daha fazla client'ı az sayıda bağlantı üzerinden multiplex eder ve çekici olmasının nedeni budur.

Gözden kaçması kolay bir bedeli vardır; çünkü bu bedel performansla değil yetkilendirmeye ilgilidir. Bir client'ın session'ında ayarladığı her şey tek bir sunucu bağlantısında yaşar ve transaction modunda client'ın böyle bir bağlantısı yoktur. Transaction dışında yapılan ayarlar bir sonraki statement'ta kaybolabilir ya da daha kötüsü, pooler'ın başkasına ödünç verdiği bir bağlantıda kalabilir. Session durumuna bağlı olan şeyler arasında geçici tablolar, birçok kurulumda prepared statement'lar, advisory lock'lar, session düzeyindeki timeout'lar ve kendi tasarımınızın session'da sakladığı her bağlam değeri vardır.

Bu son madde doğrudan kitaba dokunur. Bölüm 3'te bir satır kuralı `current_case_customer()` gibi bir değere bağlı olabilir ve `app_admin_console` bir insan principal'ın kimliğini audit log'a ve `audit_events`'e taşır. Bu değerlerden biri session başına bir kez ayarlanıp sonra okunuyorsa, transaction modunda eksik olabilir ya da kötü senaryoda bir önceki client'a ait olabilir.

Uyarı: Transaction modunda, session düzeyinde ayarlanmış bir bağlam değeri bir sunucu bağlantısında kalabilir ve bir sonraki client'ın transaction'ı tarafından okunabilir. Onu okuyan bir satır kuralı ya da audit kaydı, o zaman yanlış müşteri ya da yanlış kişi adına işlem yapabilir. Bu tür değerleri transaction'ın içinde, onunla sınırlı olarak ayarlayın ve onları okuyan kodun değer yokken başarısız olmasını sağlayın.

Sağlam duran örüntü, bağlamı transaction içinde ayarlamak ve transaction'la birlikte bitmesine izin vermektir. Sonraki blok biçimi gösterir ve iş yarıda başarısız olduğunda da doğru kalacak şekilde yazılmıştır.

```

function runAsPrincipal(principal, customerId, work):
  transaction:
    // transaction-scoped: it disappears at commit or rollback, so
    // the next client cannot inherit it from this connection
    setLocalContext("acting_principal", principal.id)
    setLocalContext("case_customer", customerId)
    try:
      work()
    on ContextMissing:
      // a row rule or audit write found no value: stop, do not
      guess
      raise ActionRefused
    // the commit happens here, once, after all of the work
    succeeded
    // on any error the transaction rolls back, and the context goes
    with it

```

Veritabanı logu yine instance'ın login'ini gösterir; ancak transaction modunda bir instance'ın client'ları onun sunucu bağlantılarını paylaşır, dolayısıyla log, instance'ın hangi thread'inin ya da isteğinin bir slotu tuttuğunu söyleyemez. Bunu öğrenmek için pooler'ın kendi client görünümüne ihtiyaç vardır ve bunun ilk olaydan önce planlanması gerekir. Timeout'lar ve adlandırma hakkındaki sonraki bölümlerin önemi bundandır.

İnsanlar bu pooler'ı kullanmaz. Veritabanına, her session için bir login oluşturan access gateway üzerinden ulaşırlar (Bölüm 7). Onların limitleri bir sonraki bölümün son kısmının konusudur; çünkü bir insan, bir servisin yapamayacağı bir şeyi yapabilir: limiti kaldırabilir.

Politika olarak timeout'lar

Bir slot, bir statement ya da bir transaction çalıştığı sürece ödünç verilir ve çoğu engine'de varsayılan olarak hiçbir şey ikisini de sonlandırmaz. Timeout'lar, platform ekibinin bir rolün bir kaynağı ne kadar süre tutabileceğini söyleme biçimidir ve bu bölümün Yürütme'yi sınırlayan kısmıdır: çok uzun çalışan bir statement çalıştığı yerde iptal edilir ve slot geri döner. Timeout'lar rol tanımına ve pool yapılandırmasına aittir, her client kütüphanesinin iyi niyetine değil; bölümün bunları politika olarak ele almasının nedeni budur.

Üç timeout veritabanına, ikisi pool'a aittir.

Timeout	Kime ait	Neyi durdurur	Kim hisseder
Statement	Veritabanı	Rolün hiçbir zaman ihtiyaç duymaması gereken kadar uzun süren bir sorgu	O statement'ın çağıranı
Lock wait	Veritabanı	Bir lock'un arkasında kuyruğa girmiş bir statement ve onun arkasındaki herkes	Çağırıcı; ayrıca aynı satırı bekleyen her çağırıcı
Idle in transaction	Veritabanı	Açılıp sonra bırakılmış, lock ve slot tutan bir transaction	Onu açan uygulama
Connection lifetime	Pool	Günlerce açık kalan ve eski credential'ını ya da durumunu koruyan bir bağlantı	Yeniden bağlanmak zorunda kalan pool
Pool acquire wait	Pool	Boş bir bağlantı için sonsuza kadar bekleyen bir çağırıcı	İsteği işleyen servis

Farklı roller farklı değerler ister, çünkü farklı işler yaparlar. `billing-api` içindeki bir istek, sayfa bekleyen bir insandır; orada otuz saniye gerektiren bir statement bir bug'dır. `invoice-worker` bir batch hazırlarken meşru olarak daha uzun bir statement çalıştırabilir. `reporting-job` uzun okumaları tasarım gereği, uzun bir okumanın yazanları engellemediği bir replica üzerinde çalıştırır. Ortak bir varsayılan hepsi için yanlış olurdu; bu yüzden değerler role bağlanır. Aşağıdaki blok örnek amaçlı notasyon kullanır ve değerler varsayımsaldır.

```
// illustrative notation: defaults for the role; see the note above on
// members, because the broker or the pooler may have to apply them
per login
ROLE SETTING app_billing_api      statement_timeout 5s, lock_wait 2s,
idle_in_transaction 10s;
ROLE SETTING app_invoice_worker   statement_timeout 30s, lock_wait 5s,
idle_in_transaction 30s;
ROLE SETTING app_reporting_job     statement_timeout 15m, lock_wait 5s,
idle_in_transaction 60s;
ROLE SETTING human_readonly        statement_timeout 60s, lock_wait 5s,
idle_in_transaction 5m;
ROLE SETTING ledgerly_migrator     statement_timeout none, lock_wait
3s, idle_in_transaction 60s;
```

Son satır, Bölüm 11'in ele aldığı bir migration dersini yansıtır: bir şema değişikliği uzun süre çalışabilir ama bir lock için yalnızca kısa süre beklemelidir.

Sıraya dair iki kural, tek tek değerlerden daha önemlidir. Veritabanının statement timeout'u uygulamanın istek deadline'ından kısa olmalıdır; çünkü uygulama önce vazgeçerse hâlâ çalışan bir statement'ı terk eder ve slot dolu kalır. Pool'un acquire wait'i ise hepsinin en kısıası olmalıdır; böylece çağırılar tıkanıklığı erken öğrenir. Bir deadline önce dolduğunda bağlantı hâlâ çalışan bir statement taşıyor olabilir; bu yüzden servis onu kapatır ve bir sonraki çağırana ödünç vermez.

Idle transaction şaşırtıcı olandır. Bir transaction açıp dış bir ödeme sağlayıcısını çağırır ve transaction hâlâ açıkken bekleyen bir servis, sağlayıcının yavaşlığının veritabanında lock ve slot tutmasına izin verir. Uzun vadeli çözüm, dış çağrıları transaction'ların dışında tutmaktır; bu bir kod değişikliğidir. Timeout, o değişiklik backlog'dayken maliyeti sınırlı tutar.

Haftalarca yaşayan bir bağlantı, doğduğu hâlde ne varsa onu korur; bu yüzden azami ömür, pool'un bağlantılarını düzenli olarak değiştirmesini sağlar. Credential'ın ömründen kısa olmalı ve bağlantılar arasında biraz değişkenlik içermelidir ki hepsi aynı dakikada yeniden bağlanmasın. Bölüm 4, süresi dolan bir credential'ın zaten açık olan bir session'ı neden kapatmadığını açıklar; azami ömür, pool'un kendi session'larını kapatma yoludur.

Bir session'ın kaldıracı olduğu varsayımlar

Rol varsayımı olarak ayarlanan bir timeout bir varsayımdır. Birçok engine'de bir session, statement timeout'u ve read-only modu dahil kendi ayarlarını değiştirebilir ve rol değiştirebilir. Bir servis için bu kabul edilebilirdir: limitler kazalar için vardır ve kod kendi timeout'unu kazayla kaldırmaz. SQL client'ı olan bir insan için kabul edilemez; çünkü limite ihtiyaç duyan session, onu kaldıracı olan session'dır.

Session'ın erişemeyeceği yerde duran iki kontrol vardır. Gateway her insan session'ını limitleri uygulanmış olarak açar ve korunan listedeki ayarları değiştirme isteğini reddeder; gateway'in bunu nasıl yaptığını Bölüm 9 anlatır. Bedeli, bu tür ayarları bağlanırken gönderen client'ların biri onları listeleyene kadar çalışmaz hâle gelmesidir. İnsan session'ları limitlerini gateway'den alır: session login'ini oluştururken session türüne göre login başına bir bağlantı limiti koyar ve bir kişinin kaç session tutabileceğini sayar. Hiçbiri bir rol limiti değildir; çünkü `human_jit_write`, hiçbir

login'in üyesi olmadığı bir tavadır. Hiçbiri client'a bırakılmaz. İkisi de olmayan her limit için, kontroller tablosu bunun bir garanti değil bir varsayılan olduğunu söylemelidir.

Çağırını adlandırmak

Robin bağlantılara baktığında yararlı soru "bu kimin?" sorusudur ve login, ilk yarısını yanıtlar. Rol, bir bağlantının `billing-api`'ye ait olduğunu söyler. Hangi instance'a, hangi release'e ya da zamanlanmış bir job'ın hangi çalıştırmasına ait olduğunu söylemez. Çoğu engine ve pooler, bir client'ın bağlantısına genellikle application name denen kısa bir etiket iliştiirmesine izin verir ve bu etiket veritabanı logunda ve mevcut etkinlik görünümünde yer alır.

Bir konvansiyon, özellikten daha değerlidir. Etikete üç şey koyardım: servis, instance ya da çalıştırma ve release. Örneğin `reporting-job/run-0412/v31`, hangi job'ın, hangi gecenin çalıştırması olduğunu ve hangi build'i söyler. Etiket, "biri kırk bağlantı tutuyor" cümlesini "raporun salı çalıştırması kırk bağlantı tutuyor ve bu yeni build" cümlesine çevirir.

İki uyarı onunla birlikte gelir. Birincisi, etiketi client sağlar ve her şeye ayarlanabilir; bu yüzden tanı ve atıf içindir, yetkilendirme için değil. Kimlik login'dir; etiket zarfın üzerindeki bir nottur. İkincisi, transaction modundaki bir pooler sunucu bağlantılarını client'lar arasında paylaştırır; dolayısıyla sunucu tarafındaki etiket client'ınki olmayabilir. Bazı pooler'lar onu her transaction için iletir, bazıları iletmez ve bu durumda adı taşıyan kayıt, pooler'ın kendi client logudur. Ona güvenmeden önce kurulumunuzun hangisini yaptığını öğrenin.

Etiket ayrıca Bölüm 14'ü besler: bir audit log'u, her satırda yalnızca bir login değil servis ve çalıştırma da olduğunda okumak daha kolaydır. Bedeli, her servisin başlangıcında bir satır ve yeni bir servis ortaya çıktığında korunacak bir konvansiyon daha.

Bağlantılar, bir rolün tüketebileceği tek kaynak değildir. Sıralama ve join için bellek, geçici disk alanı, CPU ve paralel worker sayısı da paylaşılır ve engine'ler bunları rol başına sınırlayıp sınırlayamama konusunda büyük ölçüde farklılaşır. Engine sınırlayamadığında yalıtım yerleşimden

gelmelidir: ağır bir okuyucu bir replica'ya gider; bu, Bölüm 6'nın savıdır. Bunların herhangi birini kimseye vaat etmeden önce engine'inizin rol başına hangi limitleri koyabildiğini kontrol edin.

Ledgerly: raporun API'yi yediği gece

Salı gecesine dönelim. Başlangıç durumu bunu tek cümlede açıklar: `reporting-job`, `primary`'ye karşı `admin` olarak çalıştı; böylece `billing-api` ile aynı isim altında aynı slotlar için yarıştı, ikisinde de limit ve hiçbir şeyde timeout yoktu.

Alex'in planı pooler ile başlamaz. En ucuz olan ve en çok koruma sağlayan şeyle başlar ve ekip öğrendikçe listede aşağı iner.

1. **Rol başına limitler.** Bölüm 3'teki uygulama rolleri yerindeyken her biri, yukarıdakine benzer bir tablodan bir bağlantı limiti alır. Rapor, ilk gece hâlâ `primary`'den okuyor olsa bile API'nin slotlarını bir anda alma yeteneğini kaybeder.
2. **Rol başına timeout'lar.** `app_billing_api` kısa bir statement timeout'u ve kısa bir idle-in-transaction limiti alır; rapor ise API'ye uygulanmayan uzun bir statement timeout'u alır.
3. **Etiketler.** Her servis ve her zamanlanmış çalıştırma kendi etiketini ayarlar. Bir sonraki alarm yalnızca rolü değil, çalıştırmayı adlandırır.
4. **Bir pooler.** Beş servis ona bağlanır ve o, onların tek giriş yolu olur; rol toplamı yukarıdaki üç yerden birinden alınır.
5. **Replica'ya taşıma.** `reporting-job`, kendi bütçesiyle `ledgerly_replica_2`'den okur ve `primary` artık onun işini taşımaz. Taşımanın kendisini Bölüm 6 anlatır.
6. **Ret öncesinde bir alarm.** Platform ekibi yalnızca retleri değil, her pool'un doyunluğunu izler. Saatlerce limitinin, diyelim yüzde doksanında kalan bir pool, hâlâ zaman varken "limiti artırın ya da çağırını düzeltin" der.

Job'ın da kendi başına bir değişikliğe ihtiyacı vardır; çünkü job'ı reddeden bir limit ancak job reddedilmeye iyi tepki verirse yararlıdır. Retten hemen sonra yeniden deneyen bir job, kapıdaki kalabalığın küçük bir biçimidir.

```
function acquireWithBackoff(role):
  for attempt in 1..maxAttempts:
    try: return acquireConnection(role, waitUpTo: acquireWait)
    on PoolEmpty: sleep(backoffWithJitter(attempt)) // slow down,
do not retry harder
raise ServiceBusy
```

Bunlar yerindeyken, raporun paralellliğini artıran aynı değişiklik farklı bir sonuç verirdi. Job limitine ulaşır ve reddedilir, kendini yavaşlatır ve daha uzun sürer; API de bu yüzden slotlarını tüketmez. Bir batch'ten vazgeçen bir job, atladığı şeylerin kaydını da bırakmalıdır; böylece hiçbir şey yarım kurulmuş toplulaştırmaları bütün sanarak okumaz; bu job'ın kendi tasarımıdır ve bu bölümün ötesindedir. Değişikliği onaylayan review yine de sorunsuz geçerci; çünkü tasarım artık bir review'cunun iki servis arasındaki etkileşimi fark etmesine dayanmıyor.

Trade-off'lar ve maliyetler

Rol başına limitler ayar maliyeti getirir. Çok düşük bir limit meşru yükü reddeder ve ret, onu alan servise kesinti gibi görünür. Limitlerin bir sahibi olmalı ve bir servis scale-out olduğunda ya da yeni bir iş yükü ortaya çıktığında bir review yapılmalıdır; çünkü instance sayısı çarpı pool büyüklüğü toplamı, hiçbir tek değişiklik yanlış görünmeden limiti aşabilir. Platform ekibi bu soruların sorulduğu yer olur; bu yararlıdır ve aynı zamanda yeni bir kuyruktur.

Pooler yolda yeni bir sistemdir. Her sorgu artık ondan geçer ve o çöktüğünde her servis çöker. En çok ihtiyaç duyduğunuz olay sırasında erişilebilir olmalıdır; bu yüzden kendi yedekliliğine, izlemesine ve runbook'una ihtiyacı vardır. Transaction modunda ayrıca bazı kodların bağlı olduğu session özelliklerini ortadan kaldırır ve veritabanı logunu okumayı zorlaştırır. Tek instance'lı ve on'luk pool'a sahip tek bir küçük servisiniz varsa, uygulamanın kendi pool'u artı bir rol limiti yeterlidir.

Timeout'lar meşru işi iptal eder. Fazla sıkı ayarlanmış bir statement timeout'u iyi bir statement'ı öldürür ve saf bir yeniden deneme işi de iptali de tekrarlar. Servislerin iptali ele alması gerekir; yazmalarda bu, işlemin zarar vermeden iki kez çalıştırılıp çalıştırılmayacağını düşünmek

demektir. `ledgerly_migrator` ve veri düzeltmeleri (Bölüm 10) için değerler uygulamalarinkinden farklıdır; bu da doğru tutulacak daha fazla yapılandırma anlamına gelir.

Kimsenin izlemediği bir limit hiçbir şeyi korumaz. Hiç tetiklenmeyecek kadar uzun bir `statement timeout`'u ya da hiçbir alarmin kapsamadığı bir rol limiti, yapılandırmaya bir satır ekler ve sınır eklemez. Sınama şudur: kontrol çalışmayı bıraksa kimse fark eder miydi?

Etiketler ucuz ve taklit edilebilirdir. İnsanlara yardımcı olurlar ve disiplin maliyeti getirirler, ama hiçbir şeyi kanıtlamazlar; atıf için onlara dayanan bir log, bir konvansiyona dayanıyordur.

Bu kontrolleri ne zaman kullanmamalı: roller yokken rol başına limit eklemeyin; çünkü paylaşılan `admin` rolündeki bir limit yalnızca herkesi aynı anda sınırlar. Bir servisin normal `statement`'larının ne kadar sürdüğünü bilmeden agresif `timeout`'lar koymayın; çünkü iptal edilen ilk fatura çalıştırması ekibe `timeout`'ları kapatmayı öğretir. Önce normali ölçün, sonra limiti onun üstüne koyun.

Sık yapılan hatalar

- **Bir olayı çözmek için sunucu limitini yükseltmek.** Cazip gelir; çünkü hemen işe yarar ve bugün hiçbir şeye mal olmaz. Başarısızlığı daha büyük bir kalabalığa taşır ve erken sinyali ortadan kaldırır. Her role bir bütçe verin ve sunucu limitini toplam olarak bırakın.
- **Pool'u, sahip olmayı umduğunuz trafiğe göre boyutlandırmak.** Daha çok bağlantı daha çok kapasite gibi gelir, ama veritabanının faydalı eşzamanlılığı sabittir. Pool'u bir instance'ın aynı anda yapabileceği işe göre boyutlandırın.
- **Pooler'da her client'ı tek bir login'e eşlemek.** Kolay yapılandırmadır ve paylaşılan `admin` hesabını geri getirir. Her instance'ın kendi login'ini koruyun ya da rol başına tek bir kalıcı login'i açıkça seçin ve logun artık neyi söylemediğini kabul edin.
- **Session durumuyla transaction modu kullanmak.** Kod session modunda çalışıyordu, bu yüzden kimse onu transaction modunda test etmiyor. Satır kuralları için bağlam, audit için kimlik ve geçici tablolar transaction içinde yaşamalıdır; yoksa o rol için mod session düzeyinde kalmalıdır.

- **Her rol için tek bir timeout kümesi.** Tek bir varsayılan batch için fazla sıkı, API için fazla gevşektir ve olağan sonuç herkes için yükseltilmesidir. Değerleri role bağlayın.
- **Application name'i kimlik saymak.** Etiket client ayarlar. Kimlik doğrulamak için login'i, açıklamak için etiketi kullanın.
- **Sunucuyu uygulama bütçeleriyle doldurmak.** Limitler bütün sunucuya toplandığında platform ekibi olay sırasında bağlanamaz. Her uygulama rolünün dışında bir rezerv tutun.

Temel çıkarımlar

- Bağlantı sınırlı bir kaynak ve bir politika noktasıdır: bir role değil sunucuya ait olan bir limit, ilk açgözlü çağırının herkesin payını almasına izin verir.
- Rol başına bağlantı bütçesi, "veritabanı herkes için dolu" durumunu "bu çağırının slotu kalmadı" durumuna çevirir; bu da kaçak bir job'ın blast radius'unu job'ın kendisine küçültür.
- Her uygulama rolünün dışında bir bağlantı rezervi tutun; böylece sistemi onaran insanlar ve araçlar her zaman bağlanabilir.
- Pooler, servislerin girdiği yerdir; bu yüzden merkezî ret ve ağ darboğazı noktası için doğru yerdir. Birçok pooler bir rolün dinamik login'lerini toplayamaz; bu yüzden rol toplamı, pooler bölümünün sıraladığı gibi, her biri adı konmuş bir bedeli olan veritabanından, broker'dan ya da rol başına tek bir upstream login'den gelir.
- Transaction modu pooling session durumunu ortadan kaldırır; bu yüzden bir satır kuralının ya da audit kaydının bağlı olduğu her bağlam transaction içinde ayarlanmalı ve eksik olduğunda yüksek sesle başarısız olmalıdır.
- Statement, lock wait ve idle-in-transaction timeout'ları role aittir ve veritabanının statement timeout'u uygulamanın deadline'ından önce tetiklenmelidir. Bir session bir varsayılanı kaldırabilir; bu yüzden insanlar için limit, değiştiremeyecekleri yerde uygulanmalıdır.
- Application name'ler bağlantıları açıklanabilir kılar ama hiçbir şeyi kanıtlamaz; bu yüzden kimlik login olarak kalır, etiket bir not olarak kalır.

Bir bağlantı limiti ve bir pooler, primary'ye kaç çağırının ulaştığına karar verir ama üzerine ne kadar iş düştüğünü değiştirmez. Bölüm 6 bir sonraki adımı atar: okumaları replica'lara ve diğer kopyalara taşımak ve her kopyayla birlikte gelen erişim soruları.

Bölüm 6 — Okumalar, Replica'lar ve Veri Kopyaları

Ledgerly'de ay sonundan birkaç gün önce bir salı sabahını düşünün. Veri ekibinden Jordan'dan bir sayı istenmiş: geçen ay kaç fatura kesildi, müşteri segmentine ve bir ödeme denemesinin daha önce başarısız olup olmadığına göre dağılımıyla. Soruyu sormak için doğal yer analytics kopyası olurdu ama o gecelik bir restore ve restore bir günlük. Soru bugünün verisini istiyor. Jordan bir masaüstü client açıyor, paylaşılan admin login'ile bağlanıyor ve bütün ay boyunca `invoices`, `invoice_lines` ve `payments` tablolarını birleştiren bir sorgu çalıştırıyor.

Sorgunun hiçbir yanlışı yok. İşinde iyi biri tarafından yazılmış makul bir soru. Ama bağlantı `primary`'yi gösteriyor, çünkü client'ın hatırladığı host o; sorgu birkaç büyük tabloyu okurken `billing-api`, faturalarını ödemek üzere olan müşterilere hizmet veriyor. Web uygulaması yavaşlıyor, Robin'e alarm düşüyor ve Robin ile Jordan yirmi dakikayı raporla yavaşlamanın aynı problem olduğunu bulmaya harcıyor. Kimse olağandışı bir şey yapmadı. Tasarım okunacak tek bir yer sundu ve her okuma oraya gitti.

Bu bölüm hedef durumun bir adımını ekler: okumalar replica'lara gider ve raporlama `primary`'ye hiç dokunmaz. Bölüm 1'deki beş parçalı modelde bu, Grant ile Yol'un birlikte çalışmasıyla ilgilidir; bu kitapta yetkilendirme denen şey budur. Grant, okuyucunun ne gördüğüne karar verir; yol ise okuyucunun hangi veritabanına ulaşabildiğine. Bölüm ayrıca bu adımdan doğan şeyi de ele alır: her replica ve verinin her diğer kopyası, erişimin tasarlanması gereken yeni bir yerdir. Soru şudur: veri yazıldığından daha fazla yerde okunduğunda, hangi okuma nereye gider, her kopyayı kim okuyabilir ve kopyaların kimsenin sahiplenmediği kapılara dönüşmesini nasıl önlersiniz?

Problem ve neden zor olduđu

Yazmaları alan ve aynı zamanda her okumayı karşılayan bir veritabanı, tek bir kaynak kümesiyle iki iş yapıyordur. Yazmalar primary'yi gerektirir: bir tane vardır, doğruluk kaynağıdır ve yazmalar için ayakta kalmalıdır. Okumalar tekrarlanabilir, dağıtılabilir ve çoğu birkaç saniye eski veriye tahammül eder. Okumaları primary'den uzaklaştırmak yazmaları korur ve yazmaları korumak defteri korumaktır.

Akla gelen ilk yaklaşım bir replica eklemek ve okuma ağırlıklı iş yüklerini ona yöneltmektir. Bu yaklaşık bir hafta işe yarar. Sonrasında üç problem ortaya çıkar ve hiçbirisi replica'nın kendisiyle çözülmez.

Birincisi, replica primary değildir. Çoğu kurulumda replikasyon asenkronundur: primary bir değişikliği commit eder ve replica onu kısa süre sonra uygular. Aradaki farka **replica lag** denir. Geride kalan bir replica'ya düşen okuma, biraz daha eski bir dünya hakkında doğru bir yanıt döndürür. Aylık bir rapor için bu görünmezdir. Bir faturayı yeni ödemiş ve hâlâ "ödenecek" yazan bir sayfaya bakan bir müşteri için bu bir destek talebidir.

İkinci problem, yetkilerin veriyle birlikte yolculuk etmesidir. Fiziksel bir replica tam bir kopyadır; Restricted kolonlar dahil. Replica'yı okuyabilen her şey, grant'ler aksini söylemedikçe primary'nin bir okuyucusunun okuyabileceğini okuyabilir. Sonraki bölüm, bir rolün neden tek bir replica üzerinde daha güvenli hâle getirilemeyeceğini açıklar.

Üçüncü problem, kopyaların çoğalmasıdır. İlk replica uygulama içindir. İkincisi raporlama, sonra insan incelemesi, sonra export adımı içindir. Ardından bir zamanlamayla ya da change data capture ile beslenen bir warehouse gelir. Biri bir snapshot'ı bir staging veritabanına restore eder. Her kopya, kendi gününde gerçek bir yük ya da kolaylık problemine akla yatkın bir yanıtı. Her biri de production verisi taşır, kendi credential'larına ve kendi ağ yoluna sahiptir ve nadiren birinin veriye giden yollar listesinde yer alır. **Yan kapı**, primary'nin erişim kontrollerini atlayan bir veri kopyası ya da export yoludur. Bu bölüm bilerek kurduğunuz kopyalarla ilgilidir; kazara ortaya çıkanlara Bölüm 13 döner.

"Primary üzerinde rapor çalıştırmayın" diyen bir politika Jordan'a yardımcı olmaz. Client tek bir host hatırlar, login hepsinde çalışır ve taze veriyi her zaman primary taşır. Kontrol, okumaların nasıl yönlendirildiğinde ve her

login'in neye bağlanabildiğinde olmalıdır.

Okuma yolunu ayırmak

Okuma yolunu kendi envanteri, kendi rolleri ve kendi kuralları olan ayrı bir erişim yüzeyi olarak ele alarak ve her okumaya şu soruyu sorarak başladım: bu okuma primary'ye ihtiyaç duyuyor mu? Bir okuma, bir şeye karar verdiğinde, örneğin bir ödeme tahsil edilmeden önceki bir bakiyede ya da aynı session'da bir an önce yapılan bir yazmayı görmesi gerektiğinde buna ihtiyaç duyar. Geri kalan her şey bir replica'ya gidebilir. Bir servisin kendi okumalarını nasıl sınıflandırıp yönlendirdiği uygulama tasarımıdır ve bu kitap onu kapsam dışında bırakır.

Erişim kuralı diğer yarıdır ve iki satırdır. Raporlama okumaları ve insan okumaları asla primary'ye geri düşmez: replica kullanılamazsa job bekler ya da başarısız olur ve platform ekibine haber verilir. Ve bunlar için kullanılan login'ler primary'ye hiç bağlanamaz; böylece bir hata onları oraya gönderemez. İki saat gecikmeli çalışan bir gece toplulaştırması bir rahatsızlıktır. Replica kapalı olduğu için sessizce primary'ye geçen bir toplulaştırma ise başlangıç durumunun ilk başta nasıl ortaya çıktığıdır.

Read-only bir rol zararsız bir rol değildir

Okumalar genellikle yük olarak görülür. Aynı zamanda başlıca gizlilik maruziyetidir: sistemden çıkan bir sonuç kümesi hiçbir şeyi değiştirmez, geri alınacak hiçbir şey bırakmaz ve her yere kopyalanabilir. Yazamayan bir rol, kendisine grant edilen her kolonu yine de okuyabilir ve ondan gelen tek bir ağır sorgu primary'yi yerinde tutabilir. "Read-only" ayrıca, session'ın onları geri alıp alamaması bakımından birbirinden farklı üç mekanizma anlamına gelir.

Mekanizma	Neyi durdurur	Session bunu geri alabilir mi?
Yazma ayrıcalığı olmayan bir grant	O rolün yazmaları	Tek başına hayır; ama yazma ayrıcalığı olan bir role geçebilen bir session artık ona bağlı değildir
Read-only transaction varsayılanı	O session'daki yazmaları, varsayılan olarak	Birçok engine'de evet: session bunu geri alabilir ya da read-write bir transaction başlatabilir

Mekanizma	Neyi durdurur	Session bunu geri alabilir mi?
Fiziksel bir replica	O host üzerindeki her yazmayı	Hayır: host yazmaları reddeder ve session'ın bunu değiştirecek bir ayarı yoktur

İnsanlar için üçünün en güçlüsü replica'dır ve rutin insan okumalarının oraya gitmesinin bir nedeni budur. Hiçbirinin sınırlamadığı şey, bir session'ın ne kadar okuyabileceğidir. Bir session üzerindeki row cap ve export'lar için kademelendirme Yürütme'ye aittir ve Bölüm 9 bunları ele alır.

Grant'ler replica'ya yolculuk eder

Güvenlik gözden geçirenlerinin replica hakkında ilk sorduğu soru "onun üzerinde biri ne yapabilir?" sorusudur. Fiziksel replikasyon yapan engine'lerde roller ve grant'ler cluster'ın kataloğunda yaşar ve bu katalog veriyle birlikte kopyalanır. Bir rol primary'de bir kez tanımlanır ve her replica'da görünür. Replica read-only'dir; bu yüzden bir rol tek başına onun üzerinde oluşturulamaz, değiştirilemez ya da grant edilemez. Buradan, bir rolün tek bir replica üzerinde daha güvenli hâle getirilemeyeceği sonucu çıkar: bir replica üzerinde tek bir okuyucunun yüzeyini daraltmak ya bağlantı katmanını ya da ayrı bir kopyayı gerektirir. Bölüm 3'te tasarladığınız yetkiler her replica'da yürürlükte ve bir login'in hangi veritabanına ulaşabileceği bağlantı katmanının kuralıdır.

Bunun yararlı bir yanı var. `human_readonly`, Restricted kolonları saklar ve Confidential olanları Bölüm 12'nin söylediği yerlerde maskeler ve bunu primary'de yaptığı nedenle replica'da da yapar. `app_reporting_job` bir replica üzerinde Restricted kolonlara erişimi olmadan okur. Grant'ler en az yetki gözetilerek kurulduysa, bir replica okuyucuya rolün izin verdiğinden fazlasını vermez.

Riskli bir yanı da var. Başlangıç durumunda replica'lar, her tabloya sahip olan paylaşılan admin login'ini kabul eder. Bir replica üzerinde o login'le okuyan biri her kolonu görür; bu, yazma yeteneği dışında primary'dekiyle aynı maruziyettir. İnsanlar replica'ları tam da bu nedenle güvenli sayma eğilimindedir ve sonuç, en geniş erişimin replica'larda yaşamasıdır.

Her login'i kendi replica'sına yöneltmek

Hedef durum her replica'ya bir amaç verir ve amaç, bağlantıların kabul edildiği yerde uygulanır. Bundan sonra iki terim sabit anlamlarda kullanılır. Bir **ağ politikası**, hangi host'ların bir veritabanı host'una TCP bağlantısı açabileceğini söyler; Bölüm 7 bir tane yazar. Bir **bağlantı politikası** ise hangi login'in hangi veritabanına ve nereden kimlik doğrulayabileceğini söyler.

Bağlantı politikası bir mekanizmaya ihtiyaç duyar; çünkü grant'ler onu taşıyamaz ve ağ politikası login'leri göremez. Bir tane olan engine'lerde mekanizma, her sunucunun kendi yapılandırmasından okuduğu host tabanlı erişim kuralıdır. Bu kural bağlantının kaynağına ve bir login kalıbına ya da role göre anahtarlanır ve replike edilen verinin parçası değildir; dolayısıyla primary ve her replica farklı kurallar tutabilir. Aşağıdaki blok böyle bir politikadır ve üç sunucunun tutacağı kurallar olarak yazılmıştır.

```
connection policy: one rule set per server, not replicated
ledgerly_replica_1:
    allow from pooler hosts      login app_billing_api,
app_admin_console // browsing, support lookups
    deny everything else

    ledgerly_replica_2:
        allow from pooler hosts      login app_reporting_job,
app_vendor_export
        allow from gateway hosts      login pattern: kind readonly //
human_readonly, Chapter 7
        deny everything else

    ledgerly_primary:
        deny from anywhere          login app_reporting_job,
app_vendor_export
        allow from gateway hosts      login pattern: kind jit, kind
breakglass
        deny from gateway hosts      login pattern: kind readonly //
routine reads use replica_2
```

Bedeli, üç kural kümesinin artık rol şemasıyla uyumlu tutulması gerekmesidir; bir sunucuya inip diğerine inmeyen bir değişiklik drift'tir. Kural kümeleri rol tanımlarını uygulayan aynı pipeline'a aittir (Bölüm 3) ve her sunucunun kurallarını politikayla karşılaştıran bir kontrol aynı review'da yer almalıdır. Bir engine'de böyle bir kural yoksa aynı kararlar pooler'a ve gateway'e, bir de onları kabul eden ağ politikasına düşer.

human_readonly ile ledgerly_replica_2'nin eşleştirilmesi, bir sorunu inceleyen bir kişinin raporlamanın kullandığı replica'dan okuması demektir. Bu bilinçli bir trade-off'tur: korunacak, izlenecek ve boyutlandırılacak tek bir replica, karşılığında insan incelemesinin gece job'ıyla kapasite için yarışması. İkisi birbirine karışmaya başlarsa insanlar için üçüncü bir replica makul bir bölünmedir ve bağlantı politikası bir blokla değişir.

Primary üzerindeki ret kuralları, replica'lardaki izinler kadar önemlidir. app_reporting_job ve app_vendor_export'un primary'de işi yoktur. Login'leri oraya bağlanamıyorsa başlangıç durumundaki "bazen primary'den okur" davranışını artık bir yapılandırma hatası geri getiremez. Yanlış host'a yöneltilmiş bir job bağlanamaz; bu görünür ve ucuz bir başarısızlıktır.

Diğer yöndeki istisna, bir JIT ya da break-glass session'ıdır. Her insan session login'i varsayılan olarak human_readonly'nin üyesidir; bu yüzden primary kuralı o role karşı yazılamaz: ya break-glass'ı keserdi ya da hiçbir anlam taşımazdı. Kural, access gateway'in açtığı session türüne karşı yazılır. Rutin bir session ledgerly_replica_2'ye gönderilir. Yazması gereken ya da acil durumda bir replica'nın yeterince hızlı gösteremeyeceği şeyi okuması gereken bir session primary üzerinde açılır ve erişim isteği o session'dan yapılır (modeli Bölüm 7, 8 ve 10 anlatır).

Bu, primary üzerinde bir pencere bırakır: bir istek onaylanana kadar böyle bir session human_readonly'nin yaptığını yapabilir. Üç kural pencereyi küçük tutar ve Bölüm 7 bunları session türlerinin parçası olarak belirtir. Yalnızca politikanın o türü istemesine izin verdiği kişiler onu açabilir. Gerekçe session açılırken loglanır ve session'ın, bir isteğin ardından gelmesi gereken kısa bir açılış penceresi vardır. Hiçbiri gelmezse bir alarm tetiklenir. Yalnızca veriye bakmak isteyen kişi böyle bir session açmaz; replica bunun için vardır ve rutin okumalara primary'de izin verilmemesinin nedeni budur.

Replica'nın da bir host'u var

Fiziksel bir replica veri dosyalarının bir kopyasıdır; dolayısıyla host'ları, volume'leri, snapshot'ları ve replikasyon credential'ları bir veritabanı rolünden geçmeden aynı veriye ulaşır. Replica'lara primary'dekiyle aynı host erişim kurallarını, depolama şifrelemesini ve saklama süresini

uygulardım ve bunu replica oluşturulurken yazıya dökerdim; çünkü sonradan kimse bunu düşünmeyecektir. Snapshot'lara ve backup'lara Bölüm 13 döner.

Raporlama, analytics ve kopya seçimi

Replica'lar, ucuz ve sık okumalar için bir yük problemini çözer. Raporlama ise tersidir: nadir, büyük ve yavaş okumalar. Soru, raporlamanın bir replica'ya mı yoksa başka bir şeye mi ait olduğudur ve yanıt, raporların neye ihtiyaç duyduğuna bağlıdır.

Bir raporun okuyabileceği üç yer

Seçenek	Tazelik	Primary üzerindeki yük	Erişim yüzeyi	Uygun olduğu yer
Bir raporlama replica'sı (<code>ledgerly_replica_2</code>)	Saniyeler ila dakikalar	Yalnızca replikasyon	Primary ile aynı veri ve roller	Operasyonel raporlar, export'lar, inceleme
Zamanlamayla yüklenen ayrı bir kopya (<code>ledgerly_analytics</code>)	Son yükleme kadar eski	Zamanlama başına bir yükleme	Kendi rolleri ve kendi verisi	Veri ekibinin modelleri, dashboard'ları, toplu analizi
Change data capture ile beslenen bir kopya	Saniyeler ila dakikalar	Değişiklik akışının bir okuyucusu	Yeni bir pipeline ve yeni bir depo	Neredeyse gerçek zamanlı analytics, çok sayıda tüketici

Operasyonel raporlamayı bir replica'ya, toplu analizi ayrı bir kopyaya gönderirdim ve veri ekibinin işini replica'ya koymaya yük ile ilgili olmayan bir nedenle direnirdim. Bir replica, production şemasının biçimini ve hassasiyetini taşır. Veri ekibinin Restricted kolonlara ihtiyacı yoktur ve hedef durumda ham Confidential kolonları da görmez; çünkü analytics kopyası yüklenirken maskelenir, yalnızca gösterilmek için var olan görüntüleme ipuçları hariç (nasıl yapıldığını ve istisnanın adını Bölüm 12 verir). Bir replica, replica'nın ne olduğunu değiştirmeden tüketici başına maskelenemez. Ayrı bir kopya ise kitlesine göre kurulabilir.

Analytics kopyasının bu kitapta hiçbir yerde replica olarak adlandırılmamasının nedeni de budur. Primary'den zamanlamayla yüklenir, farklı rolleri vardır ve içeriği kaynaktan farklı olabilir. Ona replica demek aynı kontrolleri taşıdığı varsayımına davetiye çıkarır; başlangıç durumunda ise daha azını taşır.

Change data capture en son başvuracağım seçenektir. Çalışır tutulması gereken bir pipeline, güvenliği sağlanması gereken bir depo, kendine ait bir lag ve üretici ile tüketici arasında bir şema sözleşmesi ekler. Ledgerly büyüklüğündeki bir şirket için önce zamanlanmış yüklemeyi seçerdim ve fikrimi neyin değiştireceğini yazardım: örneğin zamanlamanın izin verdiğinden daha yeni veri isteyen raporlar ya da aynı ihtiyacı olan ikinci bir ekip.

Her kopyanın maliyeti

Tablodaki her seçeneğin aynı gizli kalemi vardır: korunacak yeni bir şey. Bir kopya, verinin yaşadığı bir yer, onu okuyabilen bir login kümesi, bir ağ yolu, bir saklama kuralı ve bir sahiptir. Sahibi olmayan bir kopya bir yan kapıdır; sahibi olup süre sonu olmayan bir kopya ise fark edilmesi daha uzun süren bir yan kapıdır.

Her biri için aynı alanları taşıyan kısa bir kopya kaydı tutmak yardımcı olur. Aşağıdaki blok örnek amaçlı bir notasyondur ve amacı `owner` ve `ends` satırlarının boş bırakılmamasıdır.

```
table data_copy:
  name          text primary key    // "ledgerly_replica_2"
  kind          text not null       // replica, analytics, export,
dump, staging
  source        text not null       // what it was made from
  contains      text not null       // data classes present:
Restricted, Confidential, ...
  readers       text not null       // roles or logins that may
read it
  owner         text not null       // a team, never a person
  ends          timestamp null      // null only for replicas and
the analytics copy
  review_due    timestamp not null
```

Ledgerly için kayıt beş girişle başlar: iki replica, analytics kopyası, muhasebe tedarikçisine giden export dosyaları ve paylaşılan depolamadaki eski dump'lar. Dördü bilerek vardır, dump'lar değildir ve kaydın yararlı

olmasının nedeni budur: dump'lar bir sahip ve yakın bir bitiş tarihi alır. Bölüm 13, veritabanlarının dışında yaşayan kopyalar için aynı kaydı kullanır.

Not: Kimsenin açmadığı bir kayıt kontrol değildir; bu yüzden kopya oluşturan adıma bağlanır. Provisioning adımı, sahibi olan ve tür gerektiriyorsa bitiş tarihi olan bir satır yoksa bir replica, bir snapshot restore'u, bir staging veritabanı ya da bir export job'ı oluşturmayı reddeder. Erişim review'u (Bölüm 14) sonra hangi kopyaların var olduğunu ve her birini kimin okuyabildiğini sorar.

Reddin bir maliyeti vardır: deadline'ı olan bir ekip kopyasını alabilmeden önce biri bir satır doldurur ve ilk haftalar şikâyet üretir. Ayrıca yalnızca provisioning adımından geçerek yapılan kopyaları kapsar; elle yapılan bir kopya hâlâ bir yan kapıdır ve onu Bölüm 13 ele alır.

Yalnızca grant'ler değil, erişim alanı

Kâğıt üzerinde doğru olan bir rol, yanlış login bağlanabiliyorsa yine yanlıştır. Her kopya için grant, kitlenin ihtiyaç duymadığı şeyi saklar (Bölüm 3 ve 12) ve erişim alanı, hangi ağ yollarının ve principal'ların bir bağlantı açabileceğine karar verir. Düz bir ağda herhangi bir servisten sızan bir login her kopyaya ulaşır ve erişim alanı, Bölüm 4'ün dediği gibi bir sızıntının değerini sınırlar. Tablo her yer için erişim alanı kuralını belirtir; host tarafını uygulayan ağ politikası Bölüm 7'ye aittir, yukarıdaki bağlantı politikası ise login tarafını uygular.

Yer	Ulaşılabildiği yer	Ulaşılamaması gereken yer
ledgerly_primary	Pooler host'ları, JIT ve break-glass session'ları için gateway, migration'ları uygulayan pipeline adımı	Raporlama ve export job'ları, rutin insan session'ları, doğrudan web katmanı
ledgerly_replica_1	Pooler host'ları, göz atan servisler için	Rutin okumalar için gateway, analytics yüklemesi, doğrudan web katmanı
ledgerly_replica_2	Raporlama ve export için pooler host'ları, rutin insan okumaları için gateway	Uygulamayı çalıştıran servisler, doğrudan web katmanı

Yer	Ulaşılabildiği yer	Ulaşılamaması gereken yer
ledgerly_analytics	Yükleme job'ı, sonuçları için reporting-job, analytics session'ları için gateway	Uygulamayı çalıştıran servisler

Ledgerly'nin okuma yolunu taşımak

Alex'in Ledgerly'yi başlangıç durumundan bu tasarıma, gece job'ını ilk denemede bozmadan nasıl taşıyacağını düşünün. Sıra, hızdan daha önemlidir ve ben beş adımda ilerlerdim.

1. **Okumaların envanterini çıkarın.** Her servis için çalıştırdığı sorguları listeleyn ve primary'ye ihtiyaç duyan okumalar ile duymayanlar olarak ayırın. billing-api'nin listeleme sorgularının çoğu ikinci türdür. Yazmaları ve yazma sonrası yönlendirmeleri değildir. Sorgu logları, yönlendirme kurallarının ham maddesidir.
2. **Her replica'ya bir amaç verin.** ledgerly_replica_1 ve ledgerly_replica_2 için bağlantı politikasını yazın ve önce yeni login'lere uygulayın; paylaşılan login şimdilik yerinde kalsın.
3. **reporting-job'ı ve export adımını taşıyın.** İkisini de kendi login'leriyle ledgerly_replica_2'ye yöneltin, sonra bu login'leri primary'de reddedin. Eski yolu kapatmadan önce birkaç döngü boyunca yan yana çalıştırın ve çıktıları karşılaştırın.
4. **Göz atma okumalarını taşıyın.** billing-api ve admin-console göz atma okumalarını bir flag arkasında ledgerly_replica_1'e yönlendirin ve payı kademeli olarak artırın.
5. **Paylaşılan login'i replica'larda kapatın.** Orada meşru hiçbir şey onu kullanmadığında, iki replica'ya bağlanma yeteneğini kaldırın. Geri kalanını kalıcı kılan adım budur ve ekiplerin ertelediği adım da budur.

Veri ekibinin yolu da değişir. Jordan'ın ay sonu sorusunun artık iki yanıtı var. Bir günlük sayı yeterliyse human_analytics_reader üzerinden ledgerly_analytics'e gider. Bugünün verisi gerekiyorsa access gateway üzerinden human_readonly ile ledgerly_replica_2'ye gider; loglanan, Restricted kolonları göremeyen ve primary'ye dokunamayan bir session. Jordan'ın kaybettiği şey, bir masaüstü client'ı herhangi bir host'a yöneltip çalışmasını sağlama yeteneğidir. Kazandığı şey ise ağır bir sorgunun replica için bir problem, herkes için hafif bir problem olmasıdır.

Bir replica'yı okuyucularından korumak

İnsanlara ve gece job'ına hizmet eden bir replica ağır sorgular görür ve bazı engine'lerde uzun bir sorgu, yapılandırmaya bağlı olarak replikasyonu geciktirebilir ya da replikasyonun devam edebilmesi için iptal edilebilir.

Kontroller Bölüm 5'tekilerdir ve replica'nın rollerine uygulanır:

`human_readonly` için `app_reporting_job`'dakinden daha kısa bir `statement timeout`'u, `gateway`'in her insan session login'ine koyduğu login başına bir bağlantı limiti ve gece toplulaştırması penceresinin dışında çalışan bir export adımı. Bir lag alarmı ekleyin ve alarmı kayıta replica'nın sahibi olan kişiye gönderin. Kimsenin almadığı bir alarm kontrol değildir.

Trade-off'lar ve maliyetler

Replica, işletilecek bir sistemdir. İzlenmesi, patch'lenmesi, boyutlandırılması ve kurtarılması gerekir ve küçük bir ekip için bir replica ve gecelik bir kopya yeterli olabilir. Erişim argümanı onsuz da geçerlidir: hangi login'lerin primary'ye ulaşp hangilerinin ulaşmayacağına siz karar verirsiniz.

ledgerly_replica_2'yi paylaşmanın bir bedeli var. İnsanlar ve gece job'ı, korunacak tek bir yer karşılığında tek bir replica'nın kapasitesi için yarışır. Üçüncü bir replica'ya bölmek politika açısından ucuz, operasyon açısından pahalıdır.

Bağlantı politikası daha fazla yapılandırmadır. Sunucu başına kuralların rol şemasıyla eşleşmesi ve bir failover'dan, bir rebuild'den ve yeni bir replica'dan sağ çıkması gerekir. Başarısızlık, sunucular arasındaki drift'tir ve karşılaştırma kontrolü onu görünür tutmanın bedelidir.

Bazı ekranlar geride kalacak. Okumaları primary'den uzaklaştırmak lag demektir ve biri bunu fark edecektir. Hangi ekranların geride kalabileceği konusunda destek ve ürünle önceden anlaşın ve bir replica üzerindeki bir incelemede bir şey tutmadığında önce lag'e bakın (Bölüm 15).

Kayıt sürtünme ekler. Bir kopya var olmadan önce bir satır istemek, kopyayı isteyen ekibi yavaşlatır ve bu yavaşlık amaçlanan etkidir. Oluşturma sırasında kimsenin doldurmak zorunda olmadığı bir kayıt, birinin hatırladıklarının listesidir.

Bu bölümün tavsiyesinin geçerli olmadığı durumlar vardır. Primary'sinin payı olan ve ondan okuyan mühendislerin zaten kontrollü bir yoldan bağlandığı küçük bir sistemin yük için bir replica'ya ihtiyacı olmayabilir ve okuma yolunu ayırmanın getirisi daha küçüktür. Her okuma için çok katı tazelik ihtiyaçları olan bir sistem, okumaların çoğunu primary'de çalıştırmak ve çabasını o okuyucuların yapabileceklerini sınırlamaya harcamak zorunda kalabilir. Bir replica ayrıca bir backup değildir; backup'ları Bölüm 13 ele alır.

Sık yapılan hatalar

- **Replica'yı primary'den daha az hassas saymak.** Yazılamaz, bu yüzden geniş erişim toplar: paylaşılan login, herkes için kalıcı bir okuma rolü, daha geniş bir firewall kuralı. Aynı veriyi tutar; bu yüzden aynı grant'leri ve aynı ölçüde dar bir bağlantı politikasını alır.
- **Raporlama ya da insan okumalarının primary'ye geri düşmesine izin vermek.** Geri düşmek, replica kapalıyken raporu çalışır tuttuğu için cazip gelir. Bunun yerine okumayı başarısız kılın ve bu login'leri primary'de reddedin; böylece bir hata onları oraya gönderemez.
- **Bir şeye karar veren okumaları replica'ya göndermek.** Okuma ucuzdur ve replica boştur; bu yüzden kolay bir kazanç gibi görünür. Bayat veriyle verilen ve ardından bir yazmanın geldiği karar, bakiyelerin ve durumların kaymasının yoludur. Okumaları ne kadar ağır olduklarına göre değil, onları neyin izlediğine göre ayırın.
- **Bir insanı replica'ya yöneltip warehouse kurmak.** Veri ekibi o zaman production'ın şemasını, production'ın hassasiyetini ve production'ın kapasite limitlerini taşır. Analytics'e kendi kopyasını ve kendi rollerini verin (Bölüm 12 ve 13).
- **Kayıt tutmadan kopya eklemek.** Her biri, deadline'ı olan bir ekibe yapılmış bir iyilikti. Altı ay sonra kaç tane olduğunu kimse söyleyemez. Sahibi ve bitiş tarihini bir kopya oluşturma parçası yapın.
- **Host'u ve snapshot'ları unutmak.** Roller doğru, ama replica'nın volume snapshot'ı altyapı erişimi olan herkes tarafından okunabiliyor. Host'u, depolamayı ve snapshot'ları her kopyanın kontrollerine dahil edin.

- **Bir rolün replica başına sıkılaştırılabileceğini varsaymak.** Fiziksel replikasyon yapan engine'lerde roller cluster genelindedir. Tek bir okuyucunun yüzeyini bağlantı katmanında ya da ayrı bir kopyayla daraltın.

Temel çıkarımlar

- Yazmalar primary'yi gerektirir ve okumaların çoğu gerektirmez; bu yüzden okuma yolu için ilk soru, bir okumanın bir şeye karar verip vermediği ya da bir yazmayı izleyip izlemediğidir; raporlama ve insan okumaları asla primary'ye geri düşmez.
- "Read-only" üç mekanizmadır ve yalnızca replica bir session tarafından geri alınamaz; bu yüzden read-only bir rol yazmaları sınırlar, bir session'ın ne okuyabileceğini değil.
- Bir replica, primary ile aynı veriyi tutar ve aynı cluster genelindeki rollerle yönetilir; bu yüzden Bölüm 3'teki grant'ler ve bağlantı politikası birlikte bir okuyucunun ne görebileceğine karar verir.
- Bağlantı politikası, kaynağa ve login'e göre anahtarlanan, rol şemasıyla uyumlu tutulan sunucu başına bir kuraldır; her replica'nın tek bir amacı vardır ve paylaşılan admin login'i hiçbirinde kabul edilmez.
- Primary üzerindeki bir JIT ya da break-glass session'ı, onu açabilecek bir kişiyi, loglanan bir gerekçeyi, kısa bir pencereyi ve hiçbir istek gelmediğinde bir alarmı gerektirir.
- Analytics, kitlesi için kurulmuş ayrı bir kopyaya aittir ve ona replica demek, farklı kontroller taşıdığı gerçeğini gizler.
- Her veri kopyasının bir sahibi, bir okuyucu listesi ve replica'lar ile analytics kopyası dışında bir bitiş tarihi olmalıdır; kopya oluşturan adım bunlar olmadan çalışmayı reddeder.

Okuma yolunun artık kendi rolleri, kendi hedefleri ve kısa bir kopya listesi var; ama bir kişinin bunlardan herhangi birine nasıl ulaştığını henüz söylemedik. Bir sonraki bölüm insanlar için içeri giden yolları ele alır ve bir bastion'u, bir veritabanı proxy'sini ve bir access gateway'i karşılaştırır.

KISIM III

İnsan Erişimi

Bölüm 7 — İçeri Giden Yollar: Bastion'lar, Proxy'ler ve Access Gateway'ler

Platform ekibinden Alex'e bir planlama toplantısında sade bir soru soruluyor: "Yarın production veritabanına giden yollardan birini kapatsak, hangisi canımızı yakardı?" Kimse hemen cevap vermiyor. Alex bunu öğrenmeye çıkıyor ve işe ağıdan başlıyor. VPN, her mühendise primary'ye giden bir route veriyor. İki yıl önce bir yüklenici için kurulmuş bir bastion host da var ve hâlâ dinleyen bir portu açık. Bir düzine laptop'taki masaüstü client'larda kayıtlı bağlantılar duruyor; bunlardan biri bastion üzerinden tünel açacak şekilde kurulmuş, bir başkası ise doğrudan VPN üzerinden bağlanıyor.

Sonra Alex, bu hafta on-call olan Robin'e gece veritabanına nasıl ulaşacağını soruyor. Robin iki yolu tereddütsüz sayıyor, üçüncüsünü biraz düşündükten sonra. Her yol çalışıyor, hiçbiri tasarlanmamış ve hepsi aynı paylaşılan admin login'inde bitiyor. Ağ ekibi VPN'de kimlerin olduğunu söyleyebilir. Veritabanı admin login'inin bağlandığını söyleyebilir. Belirli bir bağlantının arkasında hangi kişinin olduğunu kimse söyleyemez ve kimse, herkesinkini kapatmadan tek bir kişinin erişimini kapatamaz.

Bu bölümün sorusu şu: bir kişiyle veritabanı arasında ne durmalı ki ağa ulaşmak ile veriye ulaşmak, her biri üzerinde bir isim taşıyan iki ayrı karar olsun? Bu bölüm access gateway'i ekliyor; bu, Ledgerly'nin hedef durumunun insan tarafındaki ilk adımı. Kitabın beş parçası içinde Yol'u ve o yoldan girenin kimliğini taşır: ağ, kimin kapıyı çalabileceğine karar verir, gateway ise her session'ın üzerine bir isim yazar. Bir gateway session'ının aldığı roller ve bir kişinin okumanın ötesinde nasıl daha fazlasını istediği, Bölüm 8'e aittir.

Problem ve neden zor olduđu

Bir ağ yolu ile bir veritabanı yolu aynı şey gibi görünür, çünkü birincisi ikincisini denemeden önce ihtiyaç duyduğunuz şeydir. VPN'deki bir kişi veritabanı portuna bir bağlantı açabilir. Sonrasında bir şey yapıp yapamayacağına bir kullanıcı adı ve parola karar verir; Ledgerly'nin başlangıç durumunda ise parola paylaşıyor. Ağ "kapıyı çalabilirsin" dedi, veritabanı ise kapıyı kimin çaldığını ayırt edemedi.

Sorunun özü bu. Ağ kontrolleri "hangi makineler hangi makinelerle konuşabilir" sorusunu cevaplar ve bunu iyi yapar. "Bu kişi kim, şu anda ne yapmasına izin var ve ne yaptı" sorusunu cevaplamaz. Bir ağ kuralından bu soruların yerini tutması istendiğinde sessiz bir şekilde bozulur: kural doğru kalır, ama verdiği erişim bayatlar. Takım değiştiren mühendis hâlâ VPN'dedir. Sözleşmesi biten yüklenici, kimsenin review etmediği bir grupta hâlâ durmaktadır.

İnsan yolunu servis yolundan daha zor yapan üç şey daha var.

Birincisi, insanlar en çeşitli client'lardır. Bir servis bilinen bir yerden, bilinen bir driver ile bağlanır. Bir insan komut satırı client'ı, desktop GUI, notebook, tarayıcı tabanlı bir araç ya da geçen salı yazdığı bir script kullanabilir. Her birinin credential'ları hatırlamak için kendine ait bir yolu, auto-commit için kendine ait bir varsayılanı ve sonuçları yerel diske indirmek gibi kendine ait bir alışkanlığı vardır. Bölüm 1, parolayı hatırlayan ve auto-commit yapan client'ları saymıştı; bu alışılmadık bir kurulum değil, kolaylık için yapılmış birçok aracın varsayılan davranışdır.

İkincisi, insanlar en az kontrol edilen yerlerden bağlanır. Bir laptop seyahat eder, uykuya geçer, yedeklenir ve bazen evde paylaşılır. İçinde parola olan kayıtlı bir bağlantı laptop'la birlikte taşınır, bir dosyaya export edilmiş her sonuç kümesi de öyle. Veritabanı bunların hiçbirini göremez.

Üçüncüsü, yol baskı altında kullanılır. Geceleyin veritabanına ihtiyacı olan mühendisin yeni bir araç öğrenecek hâli yoktur; yavaş ya da kırılğan bir yol, imkânı olan herkes tarafından atlanacaktır. Kâğıt üstünde doğru olup sabah üçte eski kısayola yenilen bir tasarım hiçbir şeyi değiştirmemiştir; kısayolun yalnızca caydırılmak yerine kaldırılması gerekmesinin nedeni budur.

"Admin login'ini paylaşmayın" diyen bir politika denetlenemez, çünkü veritabanı log'u admin login'inden başka bir şey göstermez. Her session'ın kendi login'i olduğu bir tasarım denetlenebilir; veritabanına yalnızca tek bir şeyin ulaşmasına izin veren bir ağ kuralı da öyle. Bu bölümün geri kalanı böyle bir tasarımı kurmanın yollarını karşılaştırıyor.

İçeri giden dört yol, karşılaştırmalı

Karşılaştırmadan önce iki terim, çünkü sektörde sözcükler gevşek kullanılıyor ve kitabın bunları sabitlemesi gerekiyor.

Bastion, korunan bir sisteme giden tek ağ yolu olan, sertleştirilmiş bir host'tur. Bağlantıları iletir. Veritabanı protokolünü anlamaz; dolayısıyla bir **SELECT** ile bir **DROP**'u ayırt edemez ve kişi bastion'a kendisi login olmadıysa bağlantının arkasındakinin kim olduğunu bilmez.

Access gateway, bir insanı doğrulayan, kısa ömürlü bir credential veren ve session'ı kaydeden, kimliğin farkında bir araçtır. **Veritabanı proxy'si** ise farklı bir şeydir: veritabanı bağlantılarını havuzlayan, yönlendiren ya da filtreleyen, protokolün farkında bir araçtır. Bir gateway bir veritabanı proxy'sinin üzerine kurulabilir, ama ikisi farklı soruları cevaplar. Proxy protokolü bilir; gateway kişiyi bilir. Bu bölüm her terimi yalnızca bu anlamda kullanır.

Bir ekibin genellikle değerlendirdiği yollar aşağıda; her birinin kişi hakkında ne bildiği ve geride ne bıraktığıyla birlikte. İlk dördü bir kişiyi bir session'ın içine koyar; son ikisi kişiyi session'ın dışında tutar.

Yol	Kişi hakkında ne biliyor	Veritabanı log'u ne gösteriyor	Ana maliyet
Herkese açık endpoint	Veritabanı login'inin ötesinde hiçbir şey	Herhangi bir adresten gelen bir login	Veritabanı portunu bütün internete açar
Veritabanına VPN	Kişinin ağda olduğunu	Paylaşılan ya da kişiye özel login	Ağ üyeliği kimlik yerine geçiyor
Bastion	Hesaplar kişiye özelse, bastion host'a kimin login olduğunu	Kişinin yazdığı login, bastion'ın adresinden	Yamalanacak bir host daha ve hiçbir statement'ı görmez

Yol	Kişi hakkında ne biliyor	Veritabanı log'u ne gösteriyor	Ana maliyet
Access gateway	Şirketin single sign-on'undan gelen kimliği	Tek bir kişiye ve tek bir session'a eşlenen session login'i	Biri ihtiyaç duyduğunda ayakta olması gereken yeni bir sistem
Ürün yolu, SQL yok	Kişinin dahili bir araçtaki rolünü	Servisin kendi login'i; kişi aracın audit izinde	Her görev için bir özellik yazılması gerekir; serbest biçimli sorular için yol yok
Gönder-ve-çalıştır executor'ı	Gönderen kişiyi ve onaylayan reviewer'ı	Executor'ın login'i, review edilmiş bir statement kümesini çalıştırıyor	Her görev review bekler; hiçbir şey etkileşimli değil

Son iki satır, bir veritabanına ulaşmanın yolları değil. Kişi hiçbir zaman bir session tutmaz ve statement, onu önceden kontrol eden bir bileşenden çalışır. Karşılaştırmaya girmelerinin nedeni, birçok görev için daha iyi cevap olmalarıdır: hiç session açmayan bir kişi, açık bir session bırakamaz. Bölüm 1 bu tür örüntülerin yelpazesini tanıtmıştı, Bölüm 9 gateway'in bir statement'ı nasıl kontrol ettiğini, Bölüm 10 ise veri düzeltmesi örüntülerini anlatır; bu bölüm, bu örüntülere uyan görevler dışarı taşındıktan sonra geriye kalan session'ları ele alır.

Herkese açık endpoint

Veritabanını insan kullanımı için herkese açık internete açmazdım ve gerekçesi kısa. Erişilebilir bir port, Ledgerly'nin insanlarıyla ilgisi olmayan otomatik trafiği davet eder ve o veritabanının her credential'ı tahmin etmeye değer bir şeye dönüşür. Geriye kalan tek savunma login'in gücüdür; kitabın öncülü ise login'lerin sızdığıdır.

Yönetilen bir veritabanına yalnızca herkese açık olarak ulaşılabilen ve sağlayıcının kaynak adresler için bir allowlist sunduğu dar durumlar var. Böyle bir durumdaysanız allowlist'i asgari düzey sayın ve sağlayıcı izin verir vermez özel bir yola geçin. Test basit: sizin için hiç çalışmamış biri porta bir bağlantı açabilir mi? Açabiliyorsa, bu kitaptaki diğer her kontrol daha kötü bir noktadan başlar.

VPN

VPN makul bir ağ katmanıdır. Veritabanına kimlerin ulaşabileceğini bir route verilmiş kişilerle daraltır ve çoğu şirkette zaten vardır. Sınırı, neyi kanıtladığıdır. Bir kişinin ya da kişinin laptop'unun VPN'in kontrollerinden geçtiğini kanıtlar. Sonrasında hangi veritabanı login'ini kullandığını söylemez ve statement'ları kaydetmez.

Ledgerly'nin başlangıç durumunda VPN, kontrolün tamamıdır. VPN'dekilerin hepsi primary'ye ulaşabilir ve admin login'i hepsi için aynıdır. VPN'i tutardım, ama ona daha küçük bir iş verirdim: insanları gateway'e ve diğer dahili araçlara taşısın, artık onları veritabanı portuna taşımasını.

Bastion

Bir bastion tek yararlı özellik ekler: ağ kapısının kapatılabileceği tek bir yer. Veritabanının firewall'u yalnızca bastion'ın adresine izin verir; bu yüzden bir bastion hesabını kapatmak, bir kişinin ağ yolunu kapatmak demektir. Bastion'da kişiye özel hesaplar varsa ve oraya kimin login olduğunu kaydediyorsa, kimin ne zaman bağlı olduğuna dair kaba bir kayıt elde edersiniz.

Bastion'ın vermediği şey, ağ katmanının altındaki her şeydir. Bir kişinin üzerinden açık bir tüneli olduğunda, veritabanına bağlantı, kişinin client'ının yaptığı neyse odur. Login yine kişi tarafından yazılır ya da client'ında kayıtlıdır. Statement'lar bastion için görünmezdir. Yani bastion ağ resmini düzeltir ama kimlik resmini düzeltmez; bir ekip bir bastion'a sahip olup yine de "o statement'ı kim çalıştırdı?" sorusunu cevaplayamamasının nedeni budur.

Bir bastion'ın ayrıca sessiz bir bakım sorunu da vardır. Shell'i olan bir host'tur; yani yama, anahtar yönetimi ve var olduğunu hatırlayan bir sahip gerektirir. Girişteki yüklenicinin host'u, bu başarısızlığın olağan biçimidir: kurulduğu gün doğruydı ve sonrasında kimse ona atanmadı.

Access gateway

İnsanları göndereceğim yer access gateway'dir; nedenleri aşağıda.

Kişiyi şirketin single sign-on'una karşı doğrular; dolayısıyla kimlik, şirketin zaten kontrol ettiği kimliğin aynısıdır. Şirketten ayrılma, takım değiştirme ve multi-factor kontrolünden geçememe, kimse veritabanını düzenlemeden gateway'de yürürlüğe girer. Tek bir session'a ait olan ve onunla birlikte ölen bir credential verir; dolayısıyla kaydedilecek parola yoktur. Ve session'ı kaydeder; böylece "o statement'ı kim çalıştırdı?" sorusunun iki yerde cevabı olur: gateway'in session kaydı ve tam olarak bir kişiye eşlenen bir login'i adlandıran veritabanı log'u.

Gateway'ler iki geniş biçimde gelir. Bazıları veri yolunun içinde durur: kişinin client'ı gateway'e bağlanır ve gateway kişi adına veritabanına bağlantıyı açar. Diğerleri yalnızca credential'a aracılık eder ve kenara çekilir; client, kısa ömürlü login ile doğrudan veritabanına bağlanır. İlk biçim statement'ları kaydedebilir ve bağlantıda kuralları uygulayabilir. İkincisi daha basittir ve yolun üzerine daha az şey koyar, ama bu durumda ağın insanların veritabanı portuna ulaşmasına izin vermesi gerekir; bu da bu bölümün başladığı sorunun bir kısmını geri getirir.

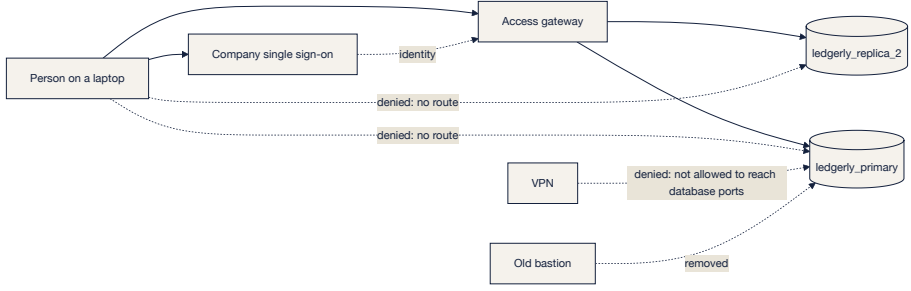
Ledgerly için veri yolunun içindeki biçimi seçerdim, çünkü ağ kuralının mutlak olmasına izin verir: veritabanı host'larına yalnızca gateway'in adresi ulaşabilir. Bedeli, gateway'in her insan statement'ını işlemesi; dolayısıyla kapasitesi ve availability'si önem taşır. Buna aşağıda döneceğim.

Ağ erişimini veritabanı erişiminden ayırmak

Bu bölümdeki en yararlı değişiklik bir ürün değil. Başlangıç durumunun birlikte cevapladığı iki soruyu ayırmak.

Birinci soru **ağ erişimidir**: hangi makineler hangi veritabanı host'una ve portuna bağlantı açabilir. İkincisi **veritabanı erişimidir**: bağlandıktan sonra hangi login ne yapabilir. Başlangıç durumunda birincisini VPN, ikincisini paylaşılan bir parola cevaplıyor. Değişiklikten sonra, birincisini herkes için tek seferde bir ağ kuralı, ikincisini ise her kişi için ayrı ayrı gateway cevaplar.

Diyagram, insanlar için hedef düzenlemeyi gösteriyor. Reddedilen oklara bakın: bugün var olan ve sonrasında var olmayacak yollar bunlar.



O diyagramın arkasında iki uygulama noktası duruyor. Veritabanı host'larındaki ağ kuralı, bağlantılara yalnızca gateway'den, servislerin kendi ağlarından, pipeline'dan ve credential broker'dan izin verir. Gateway ise bir session'ı yalnızca o session'ın türünün izin verdiği veritabanına iletir: rutin bir okuma session'ı `ledgerly_replica_2`'ye gider; bir JIT ya da break-glass isteği yapmak için açılan session ise Bölüm 6'nın ortaya koyduğu gibi `ledgerly_primary`'ye gider.

Ağ kuralını herkesin okuyabileceği kısa bir liste olarak yazardım, çünkü bir erişim review'unun kontrol ettiği şey bu listedir. Bölüm 6 farkı tanımlamıştı: bu liste, hangi host'ların hiç bağlanabileceğine karar veren network policy'dir; connection policy ise hangi login'in hangi veritabanına authenticate olabileceğine karar verir. Listede kimsenin açıklayamadığı bir girdi varsa, o girdi bir yoldur.

```

network policy (illustrative):
  ledgerly_primary:
    allows  access gateway
    allows  service networks of billing-api, invoice-worker,
           payments-service, admin-console
    allows  deployment pipeline
    allows  credential broker, for access administration
    denies  everything else, including the VPN

  ledgerly_replica_1:
    allows  service networks of the read-only services
    allows  replication traffic from the primary
    denies  everything else, including the VPN

  ledgerly_replica_2:
    allows  access gateway
    allows  reporting-job network
    allows  replication traffic from the primary
    denies  everything else, including the VPN

  ledgerly_analytics:
    allows  analytics-job network
    allows  access gateway, masked views only
    denies  everything else, including the VPN

```

Aynı liste, veritabanı host'larından çıkanlar ve gateway'e ulaşabilenlerle devam ediyor.

```

network policy (illustrative, continued):
  egress from the database hosts:
    allows  replication, the audit log, backups
    denies  everything else

  the gateway's own exposure:
    reachable from  the VPN and the office network, nothing else
    reaches        the database hosts, the audit log, single
sign-on

```

Bu modelde VPN'in ne kadar az iş yaptığına dikkat edin. Artık veritabanını korumuyor. İnsanları gateway'e taşıyor, gerisine gateway karar veriyor. VPN daha az önemli; her kişinin erişimi artık adı olan bir session'a bağlı.

Listedeki iki girdi kolayca unutulur. Önce egress. Her yere giden bağlantı açabilen bir veritabanı host'u veriyi dışarı göndermek için kullanılabilir; kısa bir allowlist (replikasyon, audit log, backup'lar) bunu görünür bir olaya çevirir. İkincisi gateway'in kendi maruziyeti: artık erişilebilen tek kapı o; bu yüzden kendisi de kısa bir listede ve VPN'in zaten ulaşmadığı hiçbir yerden erişilebilir olmamalı.

Liste ayrıca ağı çağırıcıları birbirinden ayırabildiğini varsayar. Birkaç servisin aynı adres aralığından geldiği paylaşımlı bir cluster ağında bunu yapamaz: "servislerin ağlarına" izin veren bir kural hepsine izin verir. Bu durumda ağ kuralı, servis ya da gateway olmayan her şeyi dışarıda tutar; bir servisi diğerinden ayıran şey pooler ve login'dir (Bölüm 5 ve 6). Bunu cluster için kural olarak yazardım ki kimse ağ listesini olduğundan daha kesin okumasın. Hangi login'in hangi veritabanına authenticate olabileceği, Bölüm 6'nın connection policy'sidir ve bu bölüm yalnızca ona işaret eder.

Ledgerly'de access gateway

Gateway'in ne yaptığı, yaptığı sırayla ve bir adım başarısız olduğunda ne olduğu burada. Dikkatle test edeceğim kısım bu, çünkü fail open olan bir gateway, daha iyi markalanmış yeni bir paylaşılan parolaya dönüşür.

Session nedir

Bir session, bir kişi authenticate olduğunda başlar. Gateway kimliği kontrol eder, kişinin hangi türde session isteme hakkı olduğuna karar verir (çoğu insan için rutin bir okuma session'ı; JIT ve break-glass türleri yalnızca Bölüm 8'in istekleri üzerinden) ve ardından bir **session login'i** oluşturur; kişinin session'ının kullandığı hesap budur. `ledgerly_primary` üzerinde oluşturulur ve replike olur (Bölüm 6). Session sonra türünün ait olduğu veritabanına bağlanır. Bu login adında kişinin kimliğini taşır; bu yüzden veritabanı log'unda görünür ve varsayılan olarak `human_readonly`'nin üyesidir. Credential'ı bu session için üretilir, gateway'e teslim edilir ve kişiye hiçbir zaman gösterilmez; dolayısıyla kaydedilecek bir şey yoktur.

Bir login adının sabit bir biçimi vardır ki bir log satırı arama yapmadan okunabilsin: session login'i olduğunu söyleyen bir önek, kişinin kısa tanımlayıcısı ve session'ın kendi numarası; `sess-robin-0412` gibi. Ad tanımlar; adı kişiye ve isteğe eşleyen şey gateway'in kayıdır ve session'ın ne yapabileceğine karar vermek için hiçbir şey adı ayırtmamalıdır.

Login oluşturmak ve silmek, rol üyeliklerini vermek ve revoke etmek için gateway'in veritabanında yetkilere ihtiyacı var. Bunları `ledgerly_access_admin` rolü için kendi login'i üzerinden alır. Rol

tanımları için pipeline adımı ve credential broker da bu rolü kullanır; her biri kendi login'i üzerinden (Bölüm 3 ve 4). Bu, gateway'in login'ini hassas bir credential yapar: onu elinde tutan, yönettiği roller içinde her şeyi grant edebilir. Onu owner'ın credential'larıyla aynı yerde ve aynı kurallar altında tutardım, veri yetkisi vermezdim ve her kullanımını audit ederdim (Bölüm 14). Kolaylık için script'lere ya da sohbete kopyalamak, paylaşılan parolayı bir seviye yukarıda yeniden yaratmak olur.

Hangi türde session'ı kim açabilir

`mayRequestKind` bir politikadır ve onu gateway'in kodunun içine gömülü bir koşul olarak değil, bir reviewer'ın okuyabileceği veri olarak tutardım. Tek bir tablo bunu cevaplar: single sign-on grubu, o grubun açabileceği türler, her birinin sürebileceği en uzun süre, indiği veritabanı, varsayılanları (read-only, statement timeout, row cap) ve export edip edemeyeceği.

```
session policy (illustrative):
  group backend-engineers:
    routine          max 8 hours, ledgerly_replica_2, read-only, row
cap, no export
    primary-bound max 2 hours, ledgerly_primary, only to file a
request
  group analysts:
    routine          max 8 hours, ledgerly_analytics, masked views,
export logged
  group contractors:
    routine          max 4 hours, ledgerly_replica_2, masked views,
no export
```

Primary'ye bağlı türler hassas olanlardır, çünkü primary üzerindeki bir session yazma erişiminden bir istek uzaktadır. Dört kural onları nadir tutar: yalnızca adı geçen gruplar bir tane açabilir; loglanmış bir gerekçeyle açılmalıdır; gateway, ondan bir isteğin dosyalanması için yalnızca kısa bir süre, diyelim on beş dakika bekler; ve istek olmadan biten bir session bir alert üretir, çünkü "primary'yi açtı ve hiçbir şey yapmadı" fark edilmeye değer bir örüntüdür. İki saatlik üst sınır yalnızca bir grant'e kadar geçerlidir: Bölüm 8'in `grantJitAccess` ya da `declareBreakGlass` fonksiyonu bir isteği onayladığında gateway o session'ın `ends_at` değerini isteğin `expires_at` değerine sıfırlar ve aşağıdaki `endSession` session'ı o zamanda kapatır. Bölüm 8, diğer taraftan, istekten başlar.

İşlemlerin sırası

Gateway, audit log, veritabanı ve single sign-on ayrı sistemlerdir. Hiçbir transaction bunları kapsamaz; bu yüzden adımların sırası ve yarım kalmış bir session'ın ele alınışı tasarlanmak zorundadır.

```
function openSession(person, wantedKind):
  identity = verifySingleSignOn(person)
  if not mayRequestKind(identity, wantedKind):
    raise SessionRefused
  login = newLoginName(identity)
  endsAt = now() + maxSessionLength(wantedKind)

  // the record comes first: a login with no record is what the
  sweep hunts
  recordSession({ login: login, identity: identity, kind:
wantedKind,
                  ends_at: endsAt, status: "opening" })

  try:
    auditLogAppend({ actor: identity, action: "session-open",
                    subject: login, ends_at: endsAt })
  on AuditLogUnavailable:
    markSessionRefused(login)
    raise SessionRefused

  try:
    // always on the primary: logins are cluster-wide and
  replicate
    createLogin(login, on: "ledgerly_primary", password:
freshSecret())
    grant("human_readonly", to: login)
    // replication lag: wait until the target database knows the
login
    waitUntilLoginVisible(login, on: databaseFor(wantedKind))
  on DatabaseError:
    // a login with no role is harmless, but it must not linger
    dropLoginIfExists(login)
    markSessionRefused(login)
    raise SessionRefused
  markSessionOpen(login)
  return connectionFor(login, databaseFor(wantedKind))
```

Bloğu hata yönetimi için okuyun. Kayıt, login var olmadan önce yazılır; böylece ortada bir çökme olursa geride "opening" diyen ve login'i olmayan bir kayıt ya da kaydın açıkladığı bir login kalır. Audit girdisi login'den önce gelir: log deposuna ulaşılamıyorsa hiçbir session açılmaz. Bu seçimin bir maliyeti var; trade-off bölümü bunu adlandırıyor. Rutin bir okuma session'ı için gateway, `ledgerly_replica_2` login'i alana kadar bekler, çünkü aksi hâlde replikasyon gecikmesi ilk bağlantının ara sıra başarısız

olmasına yol açardı. Login'i oluşturmak ya da rolü grant etmek başarısız olursa gateway temizlemeye çalışır ve session'ı reddeder. Temizlik bile başarısız olursa, yetkisiz bir login var olur ve aşağıdaki sweep onu kaldırır.

Bir session'ı bitirmek

Bir session üç yoldan biriyle biter: kişi bağlantıyı keser, idle limiti geçer ya da `ends_at` geçer. Son ikisi önemli, çünkü insanlar sekmeleri açık bırakır. Açık bir bağlantı, süresi dolmuş bir credential'dan uzun yaşar (Bölüm 4); bu yüzden gateway bağlantıları kendisi kapatır ve sonra login'i kaldırır. Aynı fonksiyon şirketten ayrılan kişiye de hizmet eder: single sign-on'dan gelen bir deprovisioning olayı `endSession`'ı çağırır ve maksimum session süresi, o olayın hiç gelmediği durum için sınırdır.

```
function endSession(login, reason):
  // the single owner of a session's end; Chapter 8's expiry calls
  it too
  try:
    closeConnections(login)
  on CloseFailed:
    // a drop that fails on a live connection is retried by the
    sweep
    notify("platform", "connections not closed: " + login)
    // requestsOf returns only requests not yet marked revoked
    for each id in requestsOf(login):
      try:
        revokeRequestAccess(id, from: login)
      on LoginNotFound:
        // nothing is left to revoke
      on RevokeFailed:
        notify("platform", "session grants not revoked: " + login)
        return
    markRevoked(id)
    auditOrQueue({ action: "revoke", actor: "gateway",
                    subject: login, request: id })
  try:
    revokeIfExists("human_readonly", from: login)
    dropLoginIfExists(login)
  on DatabaseError:
    notify("platform", "session login not removed: " + login)
    return
  markSessionEnded(login, reason)
  auditOrQueue({ action: "session-end", subject: login, reason:
    reason })
```

Buradaki `auditOrQueue`, kaydı audit log'a ekler ve log'a ulaşamıyorsa tekrar denemek üzere kuyruğa alır ve platform ekibine bildirir. O noktada revoke ve drop zaten gerçekleşmiştir; dolayısıyla bir log kesintisi girdiyi

geciktirir, erişimin kaldırılmasını asla geciktirmez. Zaten olmayan bir login başarı sayılır: Bölüm 8'in süre dolumu yolunun daha önce bitirdiği bir session burada temiz biçimde kapanır, kaydı bitmiş olarak işaretlenir ve hiçbir alert tetiklenmez.

Sweep, şimdiden bitmiş olması gereken her session için `endSession`'ı çağırır.

```
function sweepSessions():
    // endSession reports a stuck login and returns, so one failure
    does not stop the loop
    for each s in sessionsPastEnd(before: now()):
        endSession(s.login, "expired")
    for each s in sessionsStuckOpening(before: now() - openingGrace):
        endSession(s.login, "abandoned")
    for each login in loginsWithoutRecord():
        endSession(login, "orphaned")
```

Önemli olan sıradır. Bir isteğin grant'leri saklanan kaydından revoke edilir ve yalnızca revoke işe yaradıktan sonra revoke edilmiş diye işaretlenir; login ancak ondan sonra kaldırılır, çünkü bazı engine'ler hâlâ yetki tutan bir login'in silinmesini reddeder. Başarısız bir revoke ya da drop platform ekibine bildirir ve kaydı bir sonraki sweep için bırakır; başarısız bir audit yazımı kuyruğa alınır ve döngüyü durdurmaz. Sweep, hâlâ "opening" olan bir session'a bir tolerans süresi tanır; böylece yalnızca birkaç saniyelik bir login'i kaldırmaz. Var olan login'leri gateway'in kayıtlarıyla, grant'leri de saklanan üçlülerle karşılaştıran Bölüm 3'ün drift check'i kendi takviminde çalışır; böylece durmuş bir job diğerini gizlemez.

Hangi veritabanı, hangi rol

Her session türünün ait olduğu yere iner. Rutin bir session, `ledgerly_replica_2`'ye bağlanan bir `human_readonly` login'idir. Bir JIT ya da break-glass isteği yapmak için açılan session `ledgerly_primary` üzerinde açılır ve Bölüm 8, isteğin o session'dan nasıl yapıldığını ve grant'lerin onunla birlikte nasıl bittiğini gösterir. Gateway, Bölüm 6'nın bir connection policy olarak yazdığı hedef kuralını uygular: türü bilen gateway'dir, başka bir tür için tasarlanmış bir login'i reddeden de veritabanı host'udur.

Bir kişinin aynı anda iki session'ı açık olabilir; biri rutin, biri bir istekle. Bunu mümkün tutardım ve session'ları görünür biçimde ayrı tutardım ki kimse ayrıcalıklı olanı gelişigüzel bir bakış için kullanmasın. İnsani maliyet, kişinin client'ında ikinci bir bağlantıdır. Yararı, ayrıcalıklı session'ın bulunmasının kolay, kısa ve bitirilmesinin kolay olmasıdır.

Gateway ne kaydeder

Ledgerly'nin gateway'i veri yolunun içinde durduğu için, bir session'ın gönderdiği her statement'ı başlangıç, bitiş ve etkilenen satırlarla birlikte kaydedebilir. Bu kaydı, tarif ettiği kişilerin düzenleyemeyeceği bir yerde sakladım; Bölüm 14 nerede olduğunu anlatır. Bir kayıt şöyle görünür:

```
statement record (illustrative):
  login:      sess-robin-0412
  person:     robin
  kind:       jit
  request:    req-0412
  fingerprint: UPDATE invoices SET status = ? WHERE id IN (?)  //
  literals redacted
  param_hash: 9c1e...
  rows:       12
  started_at: 03:21:07
  ended_at:   03:21:07
```

Ödeme verisi ve finansal raporlama rejimleri, production verisine kimin hangi yoldan dokunduğunu sorma eğilimindedir; session kaydı bu sorunun cevabıdır ve yorumun sahibi okurun hukuk ve uyum ekipleridir. Bilmeye değer iki sınır var. Bir statement log'u ne istendiğini söyler, ne döndüğünü söylemez ve tam sonuç kümelerini saklamazdım, çünkü log'u verinin bir kopyasına çevirirlerdi. Fingerprint aynı nedenle parametrelerin değerlerini değil, statement'ın biçimini ve parametrelerinin bir hash'ini taşır. Ve kimsenin okumadığı bir log, hiçbir şeyi korumadan kaydeder; bu yüzden kimin ve ne zaman bakacağına karar verin: JIT isteği tutmuş session'lardan haftalık bir örnek bir başlangıçtır, erişim review'u ise diğer fırsattır.

Kapasite ve bağlantı bütçesi

Her insan session'ı aynı zamanda bir veritabanı bağlantısıdır ve Bölüm 5'in bütçesi insanlara, örnek olarak diyelim yirmi bağlantılık sabit bir dilim vermişti. Rutin mühendis session'larının hepsi `ledgerly_replica_2`'ye

indiđi için, insan yuvasının host'u o replica'dır ve sayının nihayet uygulandıđı yer onun bađlantı limitidir. Gateway bu dilimi kendisi uygulamalı: kiři bařına bir limit, session türü bařına bir limit ve bütçe tükendiđinde net bir ret. Gateway'deki bir ret, bir insana giden bir mesajdır. Bütçe bittiđinde veritabanındaki bir ret ise bir servisi dıřarı itebilir. Limiti gateway'e koymak, insan bütçesinin servisin sorunu olmasını önler.

Laptop ve yerel kopya

Gateway veritabanına giden yolu kontrol eder. Veri bir laptop'a ulařtıktan sonra ona ne olduđunu kontrol etmez ve riskin sessiz bir kısmı tam orada yařar.

Laptop'ta iki řey kalır. Biri bađlantı kurulumudur: kayıtlı host, kayıtlı login ve bařlangıç durumunda kayıtlı parola. Diđeri, kiřinin bir sonuçla yaptıđı řeydir: export edilmiř bir dosya, yapıřtırılmıř bir tablo, "kurcalamak" için birkaç bin satırın yerel bir kopyası. Her biri Bölüm 6 anlamında bir veri kopyasıdır; production verisinin sahibi ve süre sonu olmayan bir yerde yařadıđı bir yer.

Bunları, en ucuzdan en pahalıya üç katmanda ele alırdım.

Client'ın hatırlayabildiđi řeyi kaldırın. Session credential'ları tek bir session için üretildiđinden ve hiç gösterilmediđinden, bir masaüstü client'ın kaydedecek parolası yoktur. Kayıtlı bađlantı artık bir gateway adresi ve kiřinin single sign-on'unu tutar; primary'ye iřaret eden eski kayıtlı girdiler migration sırasında, neyin nerede silineceđini gösteren kısa bir listeyle silinebilir. Bu katman neredeyse hiřbir řeye mal olmaz ve en yaygın sızıntıyı ortadan kaldırır.

Göreve uyan varsayılanlar seçin. Gateway, session'ları read-only varsayılanlarla ve Bölüm 5'teki bir statement timeout ile açabilir. Ayrıca bir statement'ın döndürdüđü satırlara da üst sınır koyabilir: akıtırken satırları sayar, sınırdaki cursor'ı kapatır, kiřiye söyler ve olayı statement kaydına yazar; bu sınıra tekrar tekrar çarpan bir session, haftalık örneđin ilk baktıđı řeydir.

Kişinin kaldırabildiği varsayılanlar varsayılan kalmaz. Bir client, timeout'u yükselten ya da read-only'yi kapatan bir SET gönderebilir; bu yüzden gateway bir guarded setting listesi tutar ve bunları değiştiren her statement'ı reddeder. Maliyeti, bazı client'ların bağlanırken böyle statement'lar göndermesi ve bozulmasıdır; bu, migration'da adlandırılacak bir maliyettir ve gateway'in hangi statement'ları reddedeceğine nasıl karar verdiğini Bölüm 9 anlatır. Auto-commit yapan client'lar, session'ın commit edecek yazma yetkisi olmadığında daha az tehlikelidir. Gateway'in kendisinin işlettiği tarayıcı tabanlı bir konsol bir adım daha ileri gider, çünkü bir export düğmesi var olup loglanmadıkça hiçbir şey indirilmez. Bu konsol yerel bir client'tan daha yavaş ve daha az keyiflidir; onu yalnızca bakan ve kendi araçlarına hiç ihtiyaç duymayan insanlar için bir seçenek olarak sunardım, bir zorunluluk olarak değil.

Neyin dışarı çıktığına açıkça karar verin. Bir sonucun bir session'dan export edilip edilemeyeceği kendi başına bir kuraldır ve Bölüm 12'nin maskeleme ve minimizasyon çalışmasına ve Bölüm 13'ün export yollarına aittir. Gateway kuralı uygulayabilir, ama gereksiz kılamaz. Bir kişi bir ekranı okuyabilir, fotoğrafını çekebilir ya da bir değeri yeniden yazabilir ve buna karşı maliyet açısından verimli bir kontrol bilmiyorum. Gateway, kazara dışarı çıkan veri miktarını azaltır ve kasıtlı export'ları görünür kılar; dışarı çıkmayı imkânsız kılma.

Not: Mühendisleriniz analiz için büyük extract'lara ihtiyaç duyuyorsa, cevap genellikle gateway'de daha büyük bir export limiti değildir. Bunun için tasarlanmış, maskelenmiş bir analytics kopyasıdır; Bölüm 6 ve 12 bunu anlatır. Gateway, toplu veri taşıma için yanlış araçtır.

Tek kapı, tek hata noktası

Tek bir giriş noktasını açıklamak kolay, loglamak kolay ve kapatmak kolaydır. Aynı zamanda tek bir hata noktasıdır ve bunu önermeden önce açıkça söylemeye değer. Gateway kapalıysa kimse ön kapıdan veritabanına ulaşamaz; başka bir şey zaten bozulmuşken veritabanına en çok ihtiyaç duyan on-call mühendis de dahil.

Bu maliyeti ortadan kaldıran bir cevap yok. Onu başa çıkabileceğiniz bir yere taşıyan cevaplar var.

Gateway'i küçük tutun. Authenticate eder, login oluşturur, iletir ve kaydeder. Her ek özellik, kapalı olması için bir sebep daha demektir. Sonuç dönüştürmeyi ve bir onay arayüzünü aynı servise eklemeye direnirdim; bunları gateway'in çağırdığı ama bir okuma session'ı başlatmak için bağımlı olmadığı ayrı bileşenler olarak çalıştırdım. Bir statement çalışmadan önce kontrol ediliyorsa, o kontrol de kendi başına bir bileşendir (Bölüm 9).

Birden fazla instance çalıştırın ve state'i kalıcı bir yere koyun. Aynı adresin arkasında, session kayıtlarını paylaşan iki instance, bir restart'ın ya da kötü bir deploy'un her session'ı bitirmemesi demektir. Session deposu ve audit log da bağımlılıklardır ve availability'leri gateway'in availability'sinin parçasıdır. Yukarıdaki açılış fonksiyonunun log'a ulaşamadığında bir session'ı reddetmesinin nedeni budur: insan erişimini log'a bağlayan kasıtlı bir seçimdir ve ekibin bilerek yaptığı ve yazıya döktüğü bir seçim olmalıdır.

Degraded mod'ların ne olduğuna karar verin. Planlanacak iki başarısızlık var. Biri, veritabanları sağlamken gateway'in kapalı olması. Diğeri, gateway çalışırken audit log'un kapalı olması. Her biri için, sorumlu birinin insan erişiminin durup durmayacağına ya da daha dar bir yoldan devam edip etmeyeceğine ve bu arada nelerin kaydedileceğine önceden karar vermiş olması gerekir. Kitabın gateway kesintisi için kararı, ikinci bir kapının olmadığıdır: break-glass dahil, insan erişimi durur, çünkü bir break-glass session'ı daha ağır bir rolü olan bir gateway session'ıdır. Bunu nadir tutmanın yolu yukarıdaki yedeklilik; gateway'in dışında bir acil durum login'i değil. Bölüm 8 gerekçeyi ve bir ekip aksine karar verirse ne yapacağını verir, Bölüm 15 ise kesintinin prova edilmesini anlatır. Bir audit log kesintisinde, replica üzerindeki okumalar girdileri buffer'layabilir, yazmalar buffer'layamaz (aşağıdaki trade-off'lara bakın).

Gateway'e production gibi davranın. Bir sahibi, bir on-call yolu, bir insanın okuyabileceği bir health check'i ve rollback'li bir deploy süreci vardır. Sahibi platform ekibidir. Gateway tek bir mühendisin yan projesiyse, tek hata noktası o mühendistir.

Gateway ayrıca farklı bir riski de yoğunlaştırır: gateway'in ele geçirilmesi, ulaşabildiği her şeye erişimin ele geçirilmesidir. Bu yüzden `ledgerly_access_admin` login'i daha önce anlatıldığı gibi korunur ve bu, gateway'den kaçınmak için bir neden değildir: başlangıç durumunda, ekipteki herkese kopyalanmış paylaşılan bir parola aynı konumdadır.

Ledgerly'nin çok yoldan tek yola geçişi

Ledgerly'nin durumundaki bir ekip her şeyi bir günde değiştirmezdi. Adımları, her noktada çalışan bir yolun elde kalacağı bir sırayla atardım.

1. **Yolları envanterleyin.** Bugün bir kişinin herhangi bir Ledgerly veritabanına ulaşabildiği her yolu listeleyin: VPN route'ları, bastion, kayıtlı client girdileri, admin login'iyle bağlanan her script. Girişteki Alex'in turu bu adımdı ve platform ekibinin tuttuğu yazılı bir listeye bitmelidir.
2. **Gateway'i eski yolların yanında ayağa kaldırın.** Başlangıçta session login'lerini login'lerin tanımlandığı primary üzerinde oluşturur, onlara `human_readonly` grant eder ve onları `ledgerly_replica_2`'ye bağlar; başka hiçbir şey yapmaz. Eski yollar hâlâ çalışır. Birkaç kişi günlük okumaları için gateway'i kullanır ve nelerin zahmetli olduğunu bildirir.
3. **Önce okumayı taşıyın.** Okuma, insanların yaptıklarının çoğudur ve denemesi en az risk taşır. Zahmetli kısımlar düzeltildikten sonra replica'yı VPN'e kapatın ve ona işaret eden kayıtlı girdileri silin.
4. **Primary'yi en son taşıyın.** Primary'ye doğrudan erişim, insanların bir olay sırasında dayandığı şeydir; bu yüzden onu ancak Bölüm 8'in JIT ve break-glass yolları çalışıp prova edildiğinde kapatırdım. Daha erken kapatmak, yerine geçecek olan hazır olmadan acil durum yolunu elinden almak olurdu.
5. **Eski bastion'ı ve admin login'inin erişimini kaldırın.** Yüklenicinin bastion'ı kapatılır. Paylaşılan admin login'i insan ağlarından kabul edilmez olur. Servisler, Bölüm 4'ün çalışması onların yerini alana kadar kendi login'lerini korur.
6. **Reddedilenleri test edin.** Eski yolların her biri için biri onu dener ve başarısız olduğunu doğrular. Kimsenin test etmediği reddedilmiş bir yol, bir varsayımdır.

Migration'ın maliyeti çoğunlukla insanidir. İnsanlar yeni bir bağlantı akışı öğrenmek zorundadır ve bazı script'leri kalıcı bir login varsaydıkları için bozulur. İkisinin de çalıştığı bir pencere planlayın ve eski yolun kapanacağı tarihi yazın. Tarih olmazsa eski yol açık kalır, çünkü onu bırakmak her zaman daha ucuz görünür.

Trade-off'lar ve maliyetler

Yeterli olabilecek daha küçük bir tasarım. Gateway, kritik yola eklenmiş yeni bir sistemdir ve on-call yolunu ve deploy'larını kadrolayamıyorsanız, daha küçük bir tasarım daha iyi bir başlangıçtır. Dört mühendis özel bir ağ üzerinden tek bir veritabanına ulaşıyorsa ve ekibinizin zaten işlettiği bir şeyin verdiği kişiye özel, kısa ömürlü login'ler kullanıyorsa, değerin çoğuna zaten sahip olabilirsiniz. Fikrinizi neyin değiştireceğini yazın: katılan bir yüklenici, ikinci bir veritabanı, session kaydı talebi ya da log'un kimin yaptığını söyleyemediği ilk olay. Gateway bu koşullara bir cevaptır, bir dönüm noktası değil.

Fail closed'un bir bedeli var. Audit log'a ulaşamadığında bir session'ı reddetmek, kaydı eksiksiz tutar ve log deposunu her insan session'ının bir bağımlılığı yapar. Bazı ekipler okuma session'larını yerel bir buffer'la açıp girdileri sonra göndermeyi tercih eder. Bu, replica üzerindeki read-only session'lar için savunulabilir bir seçimdir, yazan session'lar için daha zayıf bir seçim. Bunu seçerseniz hangi session'lara uygulandığını söyleyin ve buffer'ı izleyin.

Statement kaydı bir veri kopyasıdır. Fingerprint kaydı literal'leri redakte eder, ama yine de tabloları ve kolonları adlandırır ve redakte edilmemiş bırakılan her alan, örneğin serbest metin bir not ya da client'ın verdiği bir yorum, bir değer taşıyabilir. Log'a, bir kaydın içerebileceği en hassas şeyin sınıflandırmasıyla davranın, kimin okuyabileceğini kısıtlayın ve bir sahibi olan bir saklama süresi belirleyin.

Gecikme ve sadakat. Veri yolundaki bir gateway bir hop ekler ve her client özelliğini ya da her protokol uzantısını desteklemeyebilir. Bazı araçlar bir aracı üzerinden tuhaf davranır: çok büyük transferler, uzun ömürlü cursor'lar ve alışılmadık authentication akışları olağan yerlerdir. Eski yolun kapanacağına söz vermeden önce insanların gerçekten kullandığı araçları test edin.

Gateway ne zaman kullanılmaz. Müşteri verisi olmayan tek servisli bir prototipin buna ihtiyacı yoktur. İnsan erişiminin tek yolu, kimlik, kısa ömürlü credential'lar ve loglama sağlayan yönetilen bir konsol olan bir sistemin önünde ikinci bir katmana ihtiyaç olmayabilir. Gerçek ihtiyaç ağ kapısını kapatabileceğiniz tek bir yerse ve korunan sistemin kendi güçlü, kişiye özel authentication'ı varsa, bastion makul bir cevaptır. Her iki durumda da aynı sorular geçerlidir: kişi kim, erişim ne kadar sürüyor ve kayıt nerede.

Sık yapılan hatalar

- **VPN'i kimlik katmanı saymak.** Cazip, çünkü VPN kimlerin içinde olduğunu zaten biliyor ve herkeste var. Ağ üyeliğini kanıtlar; hangi veritabanı login'inin kullanıldığını ya da ne yaptığını değil. VPN'i gateway'e ulaşma yolu olarak tutun ve her session'a bir kişiyi bağlayan bir katman ekleyin.
- **Veritabanının önüne bir bastion koyup işi bitmiş saymak.** Ağı düzene sokar, login'i paylaşılan ve statement'ları görünmez bırakır. Bastion'ı yaptığı iş için seçin, tek bir kapatılabilir ağ kapısı; kimlik ve kaydı başka yerde arayın.
- **Migration sırasında eski yolu açık bırakmak.** İki yolu bir süre tutmak ihtiyatlı görünür ve öyledir de, ama kapanış tarihi olmazsa eski olan yenisinden uzun yaşar. Tarihi yazın, duyurun ve sonrasında eski yolun başarısız olduğunu test edin.
- **Sonunu kimsenin uygulamadığı bir session açmak.** Yalnızca dokümantasyonda var olan bir session limiti, daha uzun adlı kalıcı bir grant'tir. Session'ları `ends_at` 'te bitirin, login'i silmenin yanında bağlantıları da kapatın ve var olmaması gereken session'ları ve login'leri bulan bir sweep çalıştırın.
- **Degraded mod kararını atlamak.** Kapalıyken ne olacağına karar vermeden gateway kuran bir ekip, baskı altında karar verecektir; genellikle eski paylaşılan yolu "bir dakikalığına" yeniden açarak. Acil durum yoluna önceden karar verin, yazın ve prova edin.
- **Her şeyi kaydedip hiçbir şeyi okumamak.** Her statement'ı yakalamak titizlik gibi hissettirir. Onu örnekleyen bir sahip ve ona bakan bir erişim review'u olmadan log, amacı olmayan bir veri kopyasıdır. Kimin okuyacağına ve neyi arayacağına karar verin.

- **Ağ listesini olduğundan daha kesin okumak.** Paylaşımlı bir cluster ağında kural bir servisi diğerinden ayıramaz ve ayırabildiğini varsayan bir reviewer, görüldüğünden daha geniş bir yolu onaylar. Bunun yerine çağırınları hangi kontrolün ayırdığını yazın.

Temel çıkarımlar

- Ağ erişimi ve veritabanı erişimi ayrı sorulardır ve ikisini de aynı kontrolle cevaplayan bir tasarım, kimin ne yaptığını söyleyemez.
- Bir bastion ağ yolunu tekil yapar ama veritabanı protokolünü anlamaz; bu yüzden tek başına login'i paylaşılan, statement'ları görünmez bırakır.
- Bir access gateway, bir kişiyi single sign-on ile doğrular, o kimliği taşıyan bir session login'i oluşturur ve session bittiğinde onu kaldırır; veritabanı log'unun tek bir kişiyi adlandırmasını sağlayan budur.
- Veritabanı host'larına yalnızca gateway'in ulaşmasına izin veren ağ kuralı, gateway'i tek kapı yapan kontroldür ve bir erişim review'unun okuyabileceği kısa bir liste olmalıdır.
- Tek bir giriş noktası tek bir hata noktasıdır; bu yüzden gateway küçük olmalı, birden fazla instance'ta çalışmalı, platform ekibine ait olmalı ve olaydan önce verilmiş bir degraded mod kararıyla eşleşmelidir.
- Gateway, bir laptop'a ulaşmış veriyle kişinin ne yaptığını kontrol edemez; kayıtlı credential'ları kaldırabilir, read-only varsayılanlar koyabilir ve export'ları görünür kılabilir, gerisi maskeleye ve minimizasyona aittir.
- Önce okumayı taşıyın, primary'yi en son kapatın, her eski yol için bir kapanış tarihi koyun ve reddedilen yolların gerçekten reddedildiğini test edin.

Gateway her kişiye üzerinde bir isim olan ve varsayılanı okumak olan bir session verir; ama okumak nadiren işin tamamıdır. Sonraki bölüm, birinin sınırlı bir süre için `human_readonly`'den fazlasına ihtiyaç duyduğunda ne olduğuna bakıyor: just-in-time erişim, break-glass erişim ve onayın, onu bekleyen insanlara neye mal olduğu.

Bölüm 8 — Just-in-Time ve Break-Glass Erişim

Ledgerly'de bir salı gecesini düşünün, sabahın üçünü biraz geçmiş. Bir alert, `invoice-worker` yeniden başladığından beri bir grup faturanın `sending` durumunda takılı kaldığını söylüyor ve grup temizlenmeden sabah turunun hatırlatma e-postaları gitmeyecek. Hiçbir şey ifşa olmadı, hiçbir şey kaybolmadı; sorunun maliyeti bir gecikme. On-call backend mühendisi Robin, `access gateway` üzerinden rutin bir session açıyor. Session read-only ve bir replica üzerinde çalışıyor; Robin'in ilk on dakika için istediği de bu: bak, say, anla.

Bakmak sorunu doğruluyor ve düzeltme için `invoices` üzerinde tek bir kolona bir `UPDATE` gerekiyor. Robin tek statement'lık düzeltmeyi yazıyor ve bunun için bir istek dosyalıyor. Onaylayan kişi uyuyan ikinci on-call mühendisi ve alarm henüz karşılanmadı. Robin'in elinde bir laptop, bir runbook ve verilecek bir karar var: cevabı beklemek ya da soruyu dolanmanın bir yolunu bulmak. Eski Ledgerly'de bu karar hiç var olmazdı, çünkü paylaşılan admin parolası bir yapıştırma uzaklığındaydı. Yeni Ledgerly'de dolanmanın yolu tasarlanmak zorunda, geri dönüşün yolu da.

Bu bölüm, çalışan örnekten hedef durumun sonraki adımlarını ekliyor: insanların kalıcı yazma erişimi yok, yazmalar Bölüm 3'teki tavanın içinde just-in-time grant'ler olarak geliyor ve nadiren kullanılan, her zaman review edilen bir acil durum rolü var. Kitabın beş parçası içinde ağırlıklı olarak Grant'tir: kim neyi onaylar, hangi statement için, ne kadar süreyle. Robin'in sorusunu cevaplar: erişim ihtiyaç duyulduğunda nasıl ortaya çıkar, görevle birlikte nasıl biter ve normal yol çok yavaş olduğunda ne olur?

Problem ve neden zor olduđu

Kalıcı erişim, kimsenin bir daha istemesi gerekmeyeceğine dair bir sözdür. Bir ay boyunca kullanışlıdır ve sonrasında kimse sözün neyi kapsadığını hatırlamaz. İnsanlar takım değiştirir ve grant'lerini korur; bir olay için genişletilmiş bir rol geniş kalır. Bir erişim review'u geldiğinde (Bölüm 14), reviewer'ın karşısında bir görevi olmayan grant'ler vardır ve "bu hâlâ gerekli mi?" sorusunun cevabı "muhtemelen"dir. Kalıcı bir grant'in maliyeti, kullanılmadığı her gündür, çünkü kapsadığı şeyin blast radius'unu hâlâ taşır.

Bariz yerine geçen şey **just-in-time (JIT) erişim**dir: belirli bir görev için ve sınırlı bir süre verilen erişim. Söylemesi kolay ve beş yerde ince biçimde yanlış yapması kolaydır.

Kapsam. Kolay uygulama, bir login'e bir saatliğine bir rol grant eder. Bu rol `human_jit_write` ise login tavanının tamamını alır: bir kolonun işi için altı tabloda update hakkı; ve grant, üzerine zamanlayıcı konmuş kalıcı bir grant olur. Kapsam grant'lerin kendisinde, her seferinde tek bir istek için yaşmalıdır.

Süre dolumu. Yaygın engine'lerde grant'ler kendiliğinden sona ermez. Bir şeyin onları revoke etmesi gerekir, genellikle zamanlanmış bir job; bir job durabilir, credential'ını kaybedebilir, bir login'de başarısız olup devam edebilir ya da yanlış listeye karşı çalışabilir.

Onay. Her isteği okuyan ikinci bir kişinin insani bir maliyeti var: biri uyandırılır, biri bekler. Onaylar çok yavaşsa insanlar daha hızlı yolu bulur; bu da kitabın emekliye ayırmaya çalıştığı paylaşılan paroladır. Onaylayan her zaman saniyeler içinde cevap veriyorsa, onay artık maruziyeti azaltmaz.

Acil durum. Hızlı hareket etmenin onaylı bir yolu olmayan bir erişim şeması, onaysız bir yol büyütür. Soru, hızlı yolun tasarlanmış, loglanmış ve review edilmiş mi, yoksa doğaçlama mı olduğudur.

İşlem. Bir grant bir tabloyu ve bir kolonu adlandırır, hangi satırların ya da hangi statement'ın olduğu hakkında hiçbir şey söylemez. "`invoices`" üzerinde `UPDATE, status`" okuyan bir onaylayan bir yeteneği onaylamıştır

ve onun içinde çalışan her şey onayın dışındadır. Onayın "ikinci bir kişi ne olacağını gördü" anlamına gelmesi için isteğin işlemi adlandırması ve grant'ın ondan çıkması gerekir.

"Erişim geçici olmalı" diyen bir politika, revoke job'u kapalıyken sisteme ne yapacağını söylemez. Bölümün geri kalanı grant'leri, kayıtları ve zamanlayıcıları bunu yapacak şekilde birbirine bağlıyor.

Just-in-time erişim: bir işlemi adlandıran istekler

Bir JIT isteği beş ögeden kurulur; her biri yukarıdaki başarısızlıklardan biri yüzünden oradadır. Her biri, hizmet ettiği kitap modeli parçasıyla etiketlenmiştir.

- Tavan (Grant), `human_jit_write`, bir isteğin en fazla ne isteyebileceğini tanımlar. Hiçbir login'e grant edilmez (Bölüm 3).
- Bir yazma isteği (Grant), kendisinin ait olduğu review edilmiş düzeltmeyi `fix_id` içinde adlandırır. (yetki, tablo, kolonlar) üçlülere o düzeltmeden, tavanın içinde türetilir ve `access_request_privilege` içinde saklanır.
- Onay (Grant) statement'ındır ve grant sınırlarını uygular: onaylayan statement'ı okur, gateway tam olarak türetilmiş üçlülere isteyen session login'ine grant eder ve onaylanan statement kümesi `approved_hash` olarak kaydedilir. Süre dolumu tam olarak bu üçlülere, aynı kayıttan okuyarak revoke eder.
- Bir JIT session'ındaki yazmalar (Yürütme) yalnızca `approved_hash` ile eşleşmeyen her şeyi reddeden runner'dan iletilir.
- Bağımsız bir reconciliation (Kayıt), veritabanının tuttuğu grant'leri onları haklı çıkaran isteklerle karşılaştırır; böylece durmuş bir job, job'un kendisinden başka bir şey tarafından fark edilir.

Onları, bir isteğin karşılaştığı sırayla ele alacağım.

Bir istek neyi içerir

Bir istek, onu kullanacak session'dan yapılır ve o session primary üzerinde çalışır. Bir session'ın en fazla bir canlı isteği olur; penceresinden uzun süren bir görev, yeni bir istekle yeni bir session'da devam eder. Rutin inceleme `human_readonly` ile replica üzerinde kalır (Bölüm 6) ve

primary'ye bağlı bir session'ı kimin açabileceğini Bölüm 7 söyler. Gateway hangi login'in hangi kişiye ait olduğunu bilir; bu yüzden kimse bir principal ya da bir login yazmaz.

Robin'in isteğinin, henüz kimse cevaplamadan önceki biçimi şöyle:

```
access_request:
  id: "req-0412"
  kind: "jit"
  requester: "robin"
  requester_login: "sess-robin-0412"
  approver: null
  fix_id: "fix-0093"
  approved_hash: null // set at approval, never by the
runner
  reason: "alert-7731: invoices stuck in sending"
  expires_at: now() + 1 hour // illustrative window
  status: "pending"

access_request_privilege: // derived from fix-0093, not
typed
  ("UPDATE", "invoices", "status")
```

Dikkat edilecek üç şey var. Gerekçe, kontrol edilebilir bir şeye işaret ediyor: bir alert ya da bir ticket. Grant dar ve Robin'in bir `SELECT` üçlüsüne ihtiyacı yok, çünkü session zaten `human_readonly` tutuyor. Ve Robin üçlüyü yazmadı: `fix-0093`'ten türetildi; dolayısıyla grant ile onaylayanın okuduğu statement birbiriyle çelişmez. Yalnızca `SELECT` üçlülere tutan bir isteğin, örneğin Confidential bir kolonu maskesiz okumak için, düzeltilmesi ve hash'i yoktur.

Bir grant, satırları değil yetkileri ve kolonları kısıtlar. Yukarıdaki `UPDATE`, session'ın görebildiği her faturanın `status` değerini değiştirebilir. Bir JIT grant'i, bir hatanın neye ulaşabileceğini kolon ve zamanla sınırlar; satırla sınırlayan şey, onaylayanın okuduğu ve runner'ın yürüttüğü statement'tır, Bölüm 10'daki önce-say disipliniyle.

İsteği statement'a bağlamak

Düzeltilme, filtresi ve beklenen satır sayısıyla birlikte review edilmiş bir statement kümesidir; Bölüm 10'un anlattığı biçimdedir ve istek onu `fix_id` ile adlandırır ve `approved_hash`'ini taşır. Hash, düzeltmenin merge kaydından ya da onaylanmış istek kaydından gelir, runner'ın girdisinden asla gelmez ve bir JIT yazma session'ında gateway yazmaları

yalnızca runner'dan iletir. Yalnızca `SELECT` üçlülere tutan bir isteğin düzeltilmesi, hash'i ya da runner'ı yoktur ve onaylayan kolon listesini okur. Bölüm 9 kuralı tam olarak belirtir ve gateway'in her statement'ı nasıl kontrol ettiğini anlatır.

Bir isteği kontrol etmek

Gateway, kimseden onay istenmeden önce isteğin biçimini kontrol eder ve grant zamanında, onaylayanla birlikte bir kez daha kontrol eder.

```
function checkRequestShape(request):
    // runs before anyone is asked to approve
    // live = approved and not yet revoked; this request is not yet
    counted
    if request.status != "pending" or
    hasLiveRequest(request.requester_login):
        raise GrantRefused
    if request.expires_at is null or request.expires_at <= now():
        raise GrantRefused
    // a window longer than the gateway allows is refused, not clamped
    if request.expires_at > now() + maxJitWindow:
        raise GrantRefused
    for each p in request.privileges:
        // an empty column list is never read as "all columns"; table-
        level
        // triples exist only on engines using the secrets-tables
        fallback
        if p.privilege != "DELETE" and isEmpty(p.columns):
            if not tableLevelAllowed(p.table):
                raise GrantRefused
        // refuses a table outside the ceiling and any Restricted column
        if not withinCeiling("human_jit_write", request.privileges):
            raise GrantRefused
    checkOperationBinding(request)
```

```

function checkOperationBinding(request):
    if hasWriteTriple(request.privileges):
        fix = fixRecord(request.fix_id)    // null when fix_id is null
        if fix is null:
            raise GrantRefused
        // the triples come from the fix; a typed list that differs is
        refused
        if request.privileges != triplesFromFix(fix):
            raise GrantRefused
        else if request.fix_id is not null or request.approved_hash is not
        null:
            // a SELECT-only request has no fix and no hash
            raise GrantRefused

function checkJitRequest(request):
    // runs again at grant time, once an approver is named
    checkRequestShape(request)
    if request.approver is null or request.approver ==
    request.requester:
        raise SelfApprovalNotAllowed
    if request.fix_id is not null:
        if request.approver == fixRecord(request.fix_id).author:
            raise SelfApprovalNotAllowed

```

Kendi kendini onaylama satırları, four-eyes'ın kodda uygulandığı yerdir: isteyen ya da düzeltmeyi yazan, cevap veren olamaz. Bekleyen bir isteğin onaylayanı yoktur; biçim kontrolünün bu satırları dışarıda bırakmasının nedeni budur; onaylayan onay anında doldurulur ve tam kontrol grant zamanında çalışır.

Aşırı uzun bir pencereyi sessizce kısaltmak yerine reddederdim: kısaltılmış bir pencere, isteyenin yirmi dakikası varken bir saati olduğuna inanmasına yol açar ve sürpriz onarımın ortasında gelir.

`withinCeiling`, Bölüm 3'ün tavanının uygulanabilir hâle geldiği yerdir. Tablosu tavanın dışında olan, yetkisi o tabloda sunulmayan ya da kolon listesi `iban`, `account_holder_name` veya `token_value` içeren bir üçlüyü reddeder. `bank_accounts` ve `payment_tokens` üzerinde `INSERT` ve `DELETE`'i reddeder, çünkü yeni ya da kaldırılmış bir satır `Restricted` değerleri belirler ya da yok eder. Kolon listesi olmayan, `DELETE` dışındaki her üçlüyü reddeder; `secrets-tables fallback`'ini kullanan bir engine üzerindeki tablo seviyesi üçlü hariç. Ve hiçbir insan rolünün yazmadığı `ledger_entries`, `audit_events` ya da erişim isteği kayıtlarına dokunan her şeyi reddeder.

Bir şey bilerek tavanın içinde kalır. `bank_accounts` ya da `payment_tokens` üzerinde `UPDATE (customer_id)`, `withinCeiling`'den geçer, çünkü `customer_id` yalnızca bir satırı sahibine bağlar ve Restricted değildir. Satırı başka bir müşteriye taşımak yine de hesabın ya da token'ın kimin olduğunu değiştirir; bu yüzden yürütme politikası onu ürün yoluna yönlendirir (Bölüm 9). Onaylayanın kontrol listesi bunu elle yakalar.

Kim onaylar, ne kadar süreyle, hangi kapsamda

Bir JIT isteğinin üç değişkeni var: onaylayan, pencere ve kapsam.

Onaylayan, istekteki tabloları anlayan ve isteyen ya da düzeltmenin yazarı olmayan bir kişidir. Ledgerly için `invoices`'ın doğal onaylayanı başka bir backend mühendisidir, genellikle ikinci on-call mühendisi. `payments` ya da `payment_tokens` üzerindeki her şey için doğal onaylayan, `payments-service`'in sahibi olan takımdan biridir, çünkü o takım, oradaki değişmiş bir `status`'ün aşağı akışta neyi tetikleyeceğini bilir. Platform ekibi erişim araçlarını işletir ve sahibi olmadığı veri için onu varsayılan onaylayan yapmazdım: darboğaz olur ve değişikliği yargılayamayan bir onaylayana dönüşür.

```
function chooseApproverPool(request):
  owners = emptySet()
  for each p in request.privileges:
    owners.add(owningTeam(p.table))
  pool = eligibleApprovers(owners)
  // neither the requester nor the author of the fix may answer
  pool.remove(request.requester)
  if request.fix_id is not null:
    pool.remove(fixRecord(request.fix_id).author)
  if pool is empty:
    raise NoApproverAvailable
  return pool
```

Boş bir havuz görünür biçimde başarısız olur ve asla daha geniş bir havuza düşmez. Bunun dışında havuzun sırası yükseltme sırasındır: alarm önce ilk kişiye gider, sonra sıradakine, aynı bekleyen istek içinde; ve istek ancak cevapsız olarak `expires_at`'e ulaştığında reddedilir.

Pencere, en kötü görevin değil, tipik görevin uzunluğunda olmalıdır. Diyelim bir veri düzeltmesi genellikle yirmi dakika sürüyor, form bir saat sunuyor ve gateway'in maksimumu dört saat. Bir grant'i uzatmak, eskisinin düzenlenmesi değil, yeni bir onaylı yeni istektir; çünkü bir

düzenleme tek bir onayın süresiz uzamasına izin verirdi. Bir onarım penceresinden uzun sürdüğünde session biter ve açık her transaction onunla birlikte rollback olur; bu yüzden tek bir transaction olarak yazılmış bir onarım (Bölüm 10), yarısı uygulanmış bırakmak yerine işini temiz biçimde kaybeder.

Kapsam, üçlüler ve onların çıktığı statement'tır. Onaylayan ikisini de okur ve onay ekranı üçlülerini olacakları grant'ler olarak gösterir: "invoices (status) üzerinde UPDATE, 04:12'ye kadar". Bir SELECT içinde Confidential bir kolon adlandıran bir istek, o kolonu maskesiz gösterir ve ekran bunu yalnızca kolon listesinde değil, sözcüklerle de söyler.

Onay, gateway'in kendi ekranında, tıklama anında yeni bir multi-factor kontrolüyle single sign-on üzerinden gerçekleşir; bir sohbet mesajının altında verilen bir tepki olarak değil: onay kontrolün kendisidir, bu yüzden onu kaydeden şeyin kimin verdiğini bilmesi gerekir.

Onaylayan neyi kontrol eder

Onaylayanın kararı "bu iyi bir fikir mi?" sorusundan daha dardır: statement'ı onayla ve sınırlarını grant'in uygulamasına bırak. Bu kontrol listesi, Bölüm 10'daki düzeltme için reviewer'ın kontrol listesini erişim tarafından yansıtır.

1. Gerekçe kontrol edilebilir bir alert ya da ticket mı ve hâlâ açık mı?
2. Ekrandaki statement gerekçenin söylediğini yapıyor mu? İsteyenin özetini değil, statement'ı okuyun.
3. Filtre satırları sınırlandırıyor mu ve beklenen sayı isteyeninin gördüğüyle eşleşiyor mu?
4. Türetilmiş üçlüler statement'ı kapsıyor ve fazlasını kapsamıyor mu? Secrets-tables fallback'i altında "fazlası değil", "tablonun Restricted olmayan tüm kolonları" demektir ve ekran bunu söyler.
5. Pencere bu görev kadar uzun mu ve maksimumun altında mı?
6. Statement bank_accounts ya da payment_tokens üzerinde customer_id'ye ya da herhangi bir Restricted kolona dokunuyor mu? O zaman bu bir JIT veri düzeltmesi değildir. Ürün yoluna gönderin ve isteği reddedin.
7. Ben isteyen de yazar da değil miyim ve bu onayı gelecek hafta reviewer'a açıklamaktan rahat olur muyum?

Bunlardan birini cevaplayamayan bir onaylayan, istek bekleyen durumdayken isteyene sorar. Sessizlik onay değildir; bir kolon listesi de değildir.

Grant etmek

Onay, dört kaydın uyuşması gereken andır: gateway'in istek kaydı, audit log girdisi, veritabanındaki grant'ler ve session. Hiçbir transaction bunları kapsamaz; bu yüzden sıra önemlidir ve her başarısızlığın geride ne bıraktığı da.

```
function grantJitAccess(request):
    checkJitRequest(request)

    // the hash comes from the records, as "Binding the request"
    explains
    hash = approvedHashFor(request)
    // the approved record comes first: the revoke job reads it, so a
    grant
    // can never exist without its triples and its end
    markApproved(request.id, approved_hash: hash)

    // audit next: if the log is unreachable, no grant is made
    try:
        auditLogAppend({ actor: request.approver, action: "grant",
                        subject: request.requester_login,
                        request: request.id, fix: request.fix_id,
                        approved_hash: hash, triples:
request.privileges,
                        expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused

    // the session's end follows the approved window, as "Ending the
    grant" says
    setSessionEnd(request.requester_login, request.expires_at)

    // separate systems, no shared transaction: a failure part-way
    leaves an
    // approved record with no grant or only some triples
    for each p in request.privileges:
        grant(p.privilege, on: p.table, columns: p.columns,
            to: request.requester_login)

    // the window may end between the check and the grants: end it now
    if request.expires_at <= now():
        revokeRequestAccess(request.id, from: request.requester_login)
```

Başarısızlık modlarına göre okuyun. Audit log kapalıysa istek reddedilir ve hiçbir şey grant edilmez: kaydedilemeyen bir grant review edilemez ve görünmez bir onarım yaratmaktansa bir onarımı durdurmayı tercih ederim. Gateway kayıttan ve log girdisinden sonra ama grant'lerden önce ölürse, istek onaylanmıştır ve hiçbir etkisi yoktur; bu hiçbir şey ifşa etmez. Üç üçlünden ikisinden sonra ölürse, login ikisini tutar, kayıt üçünü tutar ve sonraki revoke üçünü de geri alır, çünkü hiç grant edilmemiş bir üçlüyü revoke etmek zararsızdır.

Son satırlar bir race'i ele alır. Pencere onayda yeniden başlamaz; bu yüzden biçim kontrolü `expires_at`'ten sonra gelen bir onayı reddeder; geriye kalan, pencere kapanmadan bir an önce geçen bir onaydır ve bu satırlar onun grant'lerini hemen geri alır.

Audit girdisi üçlülerle birlikte düzeltmeyi ve hash'i de taşır; böylece onaylayan, statement ve pencere tek bir kayıta durur. Ödeme verisi ve finansal raporlama rejimleri, production verisindeki bir değişikliği kimin, hangi dayanakla ve ne kadar süreyle onayladığını sorma eğilimindedir ve bu kayıt üçünü de cevaplar; yorumun sahibi hukuk ve uyum ekipleridir.

Grant'i bitirmek

Bir grant üç katmanda biter ve hiçbirine tek katman olması için güvenilmez.

Birinci katman session'dır. Gateway onu `expires_at`'te bitirir ve login'i kaldırır; hiçbir zamanlanmış job devrede değildir: erişimi sınırlayan saat, onu oluşturan bileşene aittir. Primary'ye bağlı bir session, Bölüm 7'nin iki saatlik üst sınırıyla açılır; bu yüzden bir grant'te, `grantJitAccess` ya da `declareBreakGlass` tarafından, gateway session'ın `ends_at` değerini istegin `expires_at` değerine sıfırlar ve Bölüm 7'nin `endSession`'ı session'ı o zamanda kapatır; dolayısıyla iki saatten uzun bir pencere erken kesilmez.

```
function endSessionAtExpiry(session):  
    // one live request per session, so its end is the session's end  
    waitUntil(session.request.expires_at)  
    // never raises; a failure leaves the record for the next sweep  
    endSession(session.login, "expired")
```

Bitirmenin kendisi `endSession`'dır; bağlantıları kapatan, `session`'ın isteklerinin `grant` ettiğini `revoke` eden ve `login`'i kaldıran tek yerdir ve `sweepSessions` aynı fonksiyonu çağırır (Bölüm 7).

İkinci katman `revoke` job'udur; aynı kaydı okur ve isteğin `grant` ettiğini kaldırır. `revokeRequestAccess` işini `access_request_privilege`'dan türetir: bir JIT isteği için saklanan her üçlüyü `revoke` eder; bir `break-glass` isteği için `human_breakglass` rolünü `revoke` eder. Onunla çalışabilecek ayrı bir listeyi okumaz ve eksik bir `login`'i zaten `revoke` edilmiş sayar.

```
function revokeExpiredGrants():
  // expiredGrants: kind jit or breakglass, status approved,
  expires_at
  // passed, not yet revoked
  for each item in expiredGrants(before: now()):
    try:
      // revokes what the request granted: its stored triples
    for "jit",
      // the role human_breakglass for "breakglass"
      revokeRequestAccess(item.request_id, from: item.login)
    on LoginNotFound:
      // the login went with the session; nothing is left to
    revoke
    on RevokeFailed:
      // an expired grant still live is an incident; retried
    next run
      notify("platform", "expired grant still active: " +
        item.id)
      continue
    markRevoked(item.request_id)
    try:
      auditLogAppend({ action: "revoke", actor: "revoke-job",
        subject: item.login, request:
        item.request_id })
    on AuditLogUnavailable:
      // the revoke is done; the log catches up from the retry
    queue
      queueForRetry(item.request_id)
      notify("platform", "revoke not yet logged: " + item.id)
```

Audit log kapalıyken de `revoke` devam eder. Asimetri kasıtlıdır: log girdisi olmayan bir `grant` bir maruziyettir; girdisi olmayan bir `revoke` ise sonra doldurulabilecek bir boşluktur.

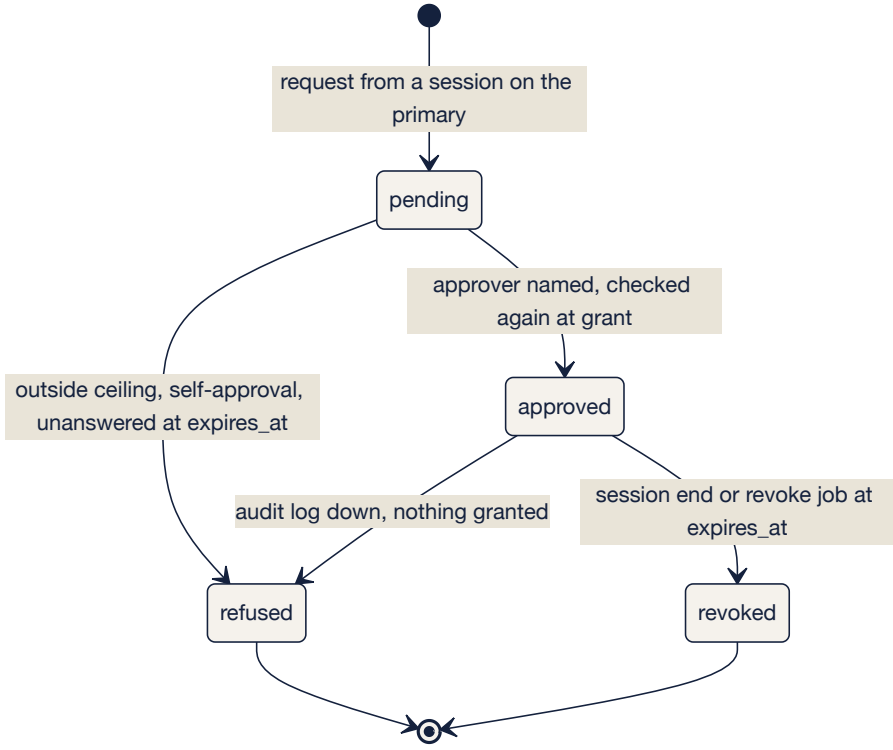
Üçüncü katman `reconciliation`'dır ve farklı türden bir şeydir. Erişimi `revoke` etmez. Dışarıdan, ilk iki katmanın işini yapıp yapmadığını sorar:


```
function checkOverdueGrants():
  // scheduled separately from revokeExpiredGrants, so one failure
  // cannot hide the other
  if count(expiredGrants(before: now() - graceMinutes)) > 0:
    notify("platform", "grants past expiry and not revoked")
  if now() - lastSuccessfulRun("revokeExpiredGrants") > maxRunAge:
    notify("platform", "revoke job has not completed recently")
```

Onunla birlikte, Bölüm 3'ün drift check'i veritabanının gerçekte tuttuğu grant'leri okur ve istekler içindeki süresi dolmamış üçlülerle karşılaştırır. Arkasında canlı bir istek olmayan doğrudan bir grant, tavanın üyesi olan bir login, açık bir break-glass isteği olmayan `human_breakglass` üyesi: her biri, revoke job'u başarı bildirse bile işaretlenir. Overdue check, job'un işlem yapmış olması gereken kayıtları izler; drift check ise job'un değiştirmiş olması gereken veritabanını izler. Hiçbir şeyi değiştirmeden "bitti" bildiren bir job yalnızca ikincisiyle yakalanır.

Reconciliation kendi başına hiçbir şeyi onarmaz ve bunu öyle tutardım: bilinmeyen grant'lerin otomatik revoke edilmesi, geceleyin production'ı bozmanın bir yoludur. Platform ekibine bildirir ve bir insan karar verir.

Bir isteğin içinden geçtiği durum küçüktür ve onu çizmek her katmanın nerede devreye girdiğini gösterir. `approved`'dan çıkışlara bakın: bir revoke ya da audit log kapalıyken ve hiçbir şey grant edilmemişken bir `ret`; `revoked`'dan hiçbir şey geri dönmez.



Kendi kendini onaylama ve on-call için otomatik onay yok

Kendi kendini onaylama her zaman reddedilir. `checkJitRequest` `SelfApprovalNotAllowed` yükseltir ve kıdem ve aciliyet dahil hiçbir şey onu geçersiz kılmaz. Acil durumun farklı bir yolu vardır; bu, bu bölümün sonraki yarısıdır.

Otomatik onay, öbür cazip kestirmedir, çünkü sürtünme geceleri en yüksektir. Okumalar için onu kurmazdım: her session login'i zaten `human_readonly` tutuyor ve bir JIT `SELECT`'in eklediği tek şey maskesiz gösterilen bir Confidential kolondur; bu da tam olarak bir onaylayanın bakması gereken şeydir. Bir yazma, servislerin sıradaki işlem yapacağı şeyi değiştirir ve bir makine `status`'ün doğru kolon olup olmadığını söyleyemez. Yazmaları bir insan onaylayanda tutun ve çabayı o onaylayanı ulaşılır kılmaya harcayın.

Daha sonra otomatik bir onay eklerseniz, gerçekten yeni bir şey taşıyan bir okuma için, kural açık bir kolon listesi şart koşmalıdır ve boş ya da eksik bir liste isteği görünür biçimde reddetmelidir. `Secrets-tables fallback`'i

altında tablo seviyesinde bir okumanın hiç kolon listesi yoktur; bu yüzden "adı geçen Confidential kolon yok"u kabul edilebilir sayan bir kural, tablodaki her Confidential kolonu onaylar.

Otomasyondan gelen istekler ve kod olarak politika

Yazmak isteyen bir job, bir runbook script'i ya da bir AI agent, diğerleri gibi bir principal'dır: session'ını kendi workload identity'si altında açar, bir düzeltmeyi adlandırır ve bir insan onaylayanı bekler; kayıt hem otomasyonu hem de onu tetikleyen kişiyi adlandırır (bir session içinde neler yapabileceğini Bölüm 9 anlatır). Onay mantığının kendisi koddur. `withinCeiling`, `chooseApproverPool`, `maxJitWindow` ve biçim kontrolleri repository'de yaşar, review'dan geçer ve drift check'in baseline'ında yer alır; bir konsolda elle düzenlenmiş bir tavan, asıl sorunun bir seviye yukarıdaki hâlidir.

Kötüye kullanım durumları

Üç durum yukarıdaki kontrollerle sınırlanır ve hiçbirleriyle ortadan kaldırılmaz. İsteyen ile onaylayan arasındaki gizli anlaşma, kayıttaki kolonlara, statement'a ve pencereye daraltılır ve çift, audit girdisinde görünür; birbirini tesadüfün çok üstünde bir sıklıkla onaylayan bir çift, erişim review'unda (Bölüm 14) ortaya çıkar. Sosyal mühendisliğe uğramış bir onaylayan ("req-0412'yi şimdi onayla, batch bordroyu tıkıyor") kontrol listesiyle karşılaşılır: statement'ı gateway'in ekranında okuyun, gerekçedeki alert'i kontrol edin ve başka bir kanaldan gelen hiçbir şeye cevap vermeyin. Ele geçirilmiş bir single sign-on token'ı bir JIT session'ında yazamaz, çünkü gateway yazmaları yalnızca runner'dan iletir ve runner hiç onaylanmamış bir hash'i reddeder. Kalan risk, grant edilen üçlüler içindeki okumalarda ve onaylayanı olmayan break-glass'ta durur; onu sınırlayan, beyan anındaki step-up kontrolü ve alert'tir.

Break-glass erişim

Break-glass, tanımlı bir logging seti ve zorunlu review'u olan acil durum erişimidir. Adı, önü camla kaplı yangın düğmesinden gelir: ihtiyacı olan herkesin ulaşabildiği ve kırılması herkes tarafından görülebilen.

Kitabın modelinde bu, `human_breakglass` rolüdür ve Restricted kolonları okuyan tek insan rolüdür. Her tabloyu okur ve iki append-only tablo dışındaki her tabloya `INSERT` ve `UPDATE` yazar; `bank_accounts` ve `payment_tokens` üzerinde yalnızca `UPDATE` yazar ve yalnızca Restricted olmayan kolonlarda. `ledger_entries` ya da `audit_events`'e asla yazmaz, çünkü hiçbir insan rolü yazmaz. Rutin düzeltmeler için bir yol değildir ve bir veri düzeltmesi yolu değildir: bir veri düzeltmesi planlanır, review edilir ve sınırlanır (Bölüm 10); break-glass tasarım gereği bunların hiçbirisi değildir.

Bir break-glass isteği kendi kendine beyan edilir. Onaylayanı yoktur. Session'daki kişi bir acil durumun var olduğunu söyler, bir gerekçe verir ve gateway isteği onaylı olarak işaretleyip rolü aynı session login'ine `expires_at`'e kadar grant eder. Aynı revoke job'u ve aynı session zamanlayıcısı onu geri alır.

Onaylayan, JIT için kontrolün tamamıdır; bu yüzden onaylayanı olmayan bir acil durum rolü tuhaf görünebilir. Break-glass, onaylayan havuzunun cevap vermediği durumlar için var ve bir onay adımı bu başarısızlığı yeniden üretti. Kontrol, erişimden önceden erişimin etrafına taşınır: beyan anında bir alert, aşağıdaki logging seti, kısa sabit bir pencere ve istisnasız izleyen bir review.

```

function declareBreakGlass(session, reason):
    // a hijacked token alone must not be enough to declare
    requireFreshMultiFactor(session.principal)
    // one live request per session: a break-glass declaration
    supersedes it
    supersedeLiveRequest(session.login)
    // the window is the gateway's fixed maximum; there is no argument
    for it
    window = breakGlassMaxWindow
    request = newRequest(kind: "breakglass", requester:
session.principal,
                                requester_login: session.login, approver:
null,
                                reason: reason, expires_at: now() + window)
    markApproved(request.id)
    try:
        auditLogAppend({ actor: session.principal, action:
"breakglass",
                                subject: session.login, request: request.id,
                                expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused
    setSessionEnd(session.login, request.expires_at)
    grant(human_breakglass, to: session.login)
    // paged after the grant: a paging outage must not block an
    emergency;
    // the review queue below is the net for a page that never arrived
    queueReview(request.id, reviewer: "not-the-requester")
    notify("security", "break-glass declared: " + request.id)
    notify("platform", "break-glass declared: " + request.id)

```

O bloktaki üç karar bir açıklamayı hak ediyor.

Pencere gateway tarafından sabitlenmiştir; Ledgerly'nin örneğinde iki saattir. İstek üzerine daha uzun bir pencere sunulmaz, çünkü "biraz daha uzun lazım" bir istisnayı alışkanlığa çevirir. Acil durum pencereden uzun sürerse kişi yeniden beyan eder ve ikinci beyan yeniden alert üretir; kasıtlı sürtünme budur.

Audit log kontrolü JIT'tekiyle aynıdır. Log kapalıysa beyan reddedilir. Bu maliyeti kabul ediyorum, çünkü kaydı olmayan bir break-glass session'ı, bir review'un yeniden kuramayacağı tek şeydir ve rolü katlanılabılır kılan review'dur. Normal yol ile log'un ikisi de kapalıyken on-call'un ne yapacağı Bölüm 15'e aittir.

Alert, güvenlik iletişim kişilerine ve platform ekibine, birinin Ledgerly'nin sahip olduğu en geniş erişimi açtığını bildiren bir bildirimdir, izin için bir istek değil. Sağlıklı bir sistemde okunacak kadar nadirdir; haftalık tetikleniyorsa okunmaz olacaktır ve break-glass, tasarlanmış bir yolu hak eden bir şey için kullanılıyordur.

Break-glass logging seti

"Ağır loglama", tanımlanana kadar bir standart değildir; bu yüzden bir break-glass session'ı için logging seti şu dört şeydir:

- Her statement; literal'leri redakte edilmiş bir fingerprint ve parametrelerin bir hash'i olarak, başlangıç, bitiş ve etkilenen satırlarla. Redaksiyon burada en çok önem taşır: bu session Restricted kolonları okur ve log'daki bir literal onları kolon seviyesinde korumanın olmadığı bir yere kopyalardı.
- Girdiler, olurken. Her statement çalışırken audit log'a eklenir, sonunda toplanmaz; böylece yarıda kesilen bir session de kaydını bırakır.
- Her yazmada tek satırlık bir not. Gateway notu olmayan bir yazmayı reddeder; bu birkaç saniyeye mal olur ve reviewer'a niyeti statement'ın yanında verir.
- Diğer her session'la aynı kayıt: login, kişi, istek ve gerekçe; böylece Bölüm 14'ün üç kaydı birleştirilebilir.

Bir risk kalıyor ve duruma göre farklılaşıyor. Bugün, gateway olmadan ya da yalnızca credential'a aracılık eden bir gateway'le, bir break-glass session'ı, yazabildiği bir tabloda sınırsız bir **UPDATE**'in hâlâ mümkün olduğu tek yerdir. Pencere, satır sayıları, girdiler ve review hasarı sınırlar ve açığa çıkarır; engellemezler. Hedef durumda Bölüm 9'un gateway'i yazma session'larını auto-commit kapalı ve bir row cap ile çalıştırır, matrisi bu hatta sınırsız bir yazmayı deny eder ve notsuz bir yazma reddedilir. Mümkün kalan, cap'in içine sığan ve yine de yanlış satırlara çarpan bir yazmadır. Cap'in boyutlandırılması burada en zordur, çünkü bir acil durum tanım gereği planlanmış görev değildir; politika bir cap koyana kadar sınırsız yazma kabul edilmiş bir risktir ve review'un var olma nedenidir.

Sonradan review

Her break-glass kullanımı, onu kullanan kişi dışında biri tarafından review edilir. Review bir duruşma değildir; bir tasarım review'unun sorduğu soruları sorar: bu, normal yolun karşılayamadığı bir acil durum muydu; session ne okudu ya da yazdı; Restricted kolonları okudu mu ve bu gerekli miydi; ve bir JIT isteği aynı ihtiyacı karşılayabilir miydi, karşılayabilirse o isteği yeterince hızlı yapacak olan neydi?

Son soru karşılığını veren sorudur. Her review ya break-glass'ın doğru araç olduğunu doğrular ya da normal yolda onu tek araç yapan bir boşluk bulur. Boşluğu kapatmak (eksik bir onaylayan, kısa bir pencere, tavanın sunmadığı bir tablo) sonraki kullanımı azaltır. Review'un kendi saatine de ihtiyacı var:

```
function checkBreakGlassReviews():
  for each r in breakGlassRequests(endedBefore: now()):
    review = reviewFor(r.id)
    if review is null and now() - r.expires_at > reviewDeadline:
      notify("security", "break-glass review overdue: " + r.id)
    if review is not null and review.reviewer == r.requester:
      notify("security", "break-glass reviewed by its own user:
" + r.id)
```

Kimsenin yapmadığı bir review, sessizce var olmayı bırakmış bir kontroldür; bu yüzden son tarihi bir alert yükseltir.

Gateway olmadığında

İnsanlar veritabanına yalnızca access gateway üzerinden ulaşır (Bölüm 7) ve hem JIT hem de break-glass grant'leri gateway tarafından, onun oluşturduğu bir session login'ine yapılır. Bir gateway kesintisi bu yüzden aynı zamanda bir break-glass kesintisidir. Bu, tasarımda bir trade-off'tur ve Bölüm 7 aynı kararı gateway'in tarafından belirtir.

Alternatif ikinci bir kapıdır: gateway'in dışında bir acil durum login'i. Gerektiği gece cazip, diğer her gece tehlikelidir; çünkü gateway olmadan çalışan bir credential, onun loglaması, kimliği ve süre dolumu olmadan çalışır; daha iyi bir adı olan paylaşılan admin parolasıdır.

Her şeyi zayıflatarak gateway'in yokluğunu atlatılabilir kılmaktansa gateway'i kaybetmeyi zorlaştırmayı tercih ederim: birden fazla instance, az bağımlılık (audit log, istek kayıtları), prova edilmiş bir başarısızlık ve on-

call'un ilk gördüğü bir health sinyali. Tüm instance'lar aynı anda kapalıysa hiçbir insan veri yolu kalmaz ve burada bir tane icat etmezdim. Bölüm 3'ün mühürlü aracı engine seviyesindeki işler için vardır, Bölüm 7'nin insanlara kapattığı bir ağ route'una ihtiyaç duyar ve append-only ile Restricted korumalarını atlar; bu yüzden bir veri erişim yolu değildir. Bölüm 15 o durumda ne yapılacağına geri döner.

Seçim bedava değil. Kırılğan bir gateway'i olan bir ekip, olgunlaşırken makul biçimde ikinci bir yol isteyebilir. O zaman ne olduğunu, credential'ını kimin alabileceğini, kullanımının nasıl alert ürettiğini ve onu neyin emekli edeceğini yazın: sahibi ve bitiş tarihi olan bir yol bir karardır, olmayan ise bir artıktır.

Engine'in kolon seviyesinde grant'i olmadığında

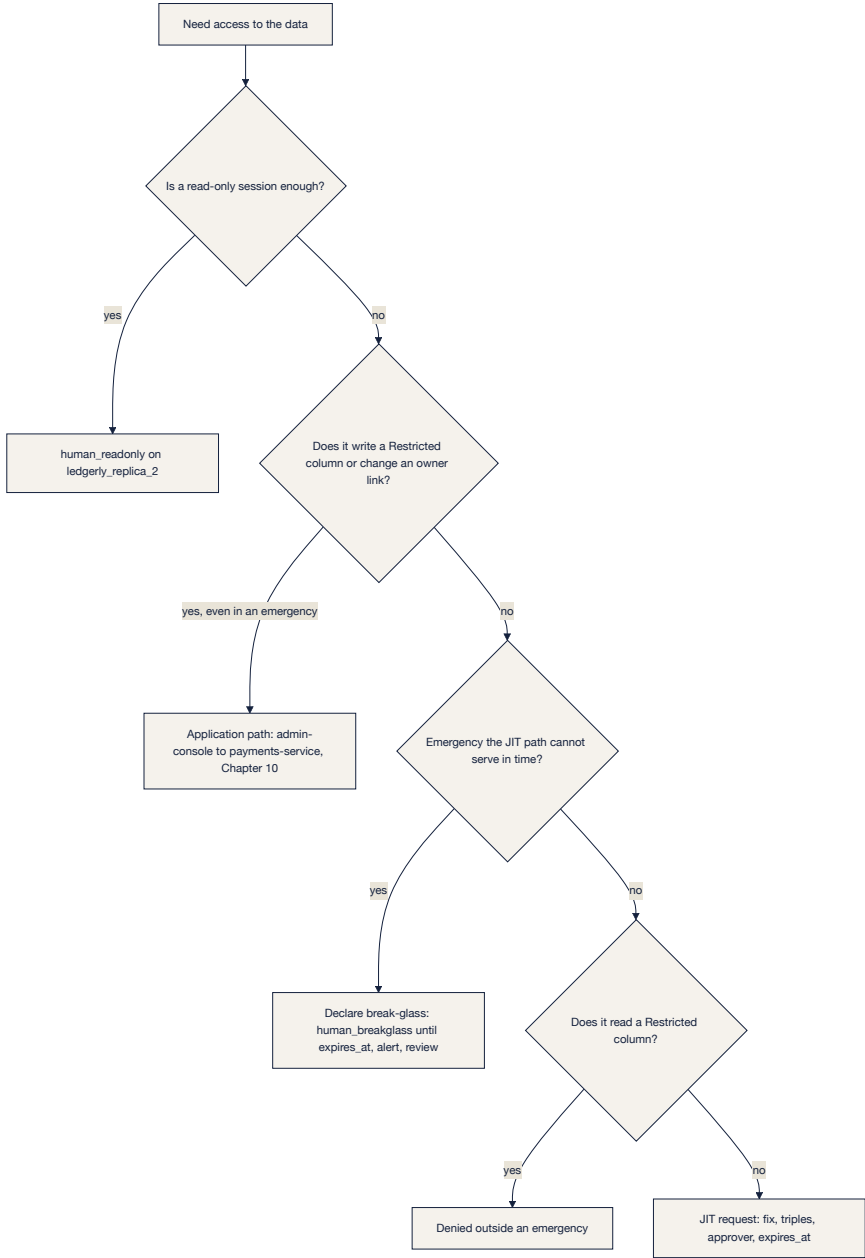
Yukarıdaki her şey, engine'in adı geçen kolonlar üzerinde bir yetki verebildiğini varsayar. Bölüm 3, bunu yapamayan engine'ler için kitabın kullandığı tek fallback'i tanıtmıştı: Restricted kolonlar tavanın dışındaki ayrı tablolarda, bank_account_secrets ve payment_token_values içinde yaşar ve yalnızca app_payments_service ve human_breakglass tarafından erişilebilir. Ana tablolar o zaman hiçbir Restricted kolon tutmaz; bu yüzden bank_accounts ya da payment_tokens üzerinde tablo seviyesinde bir JIT grant'i hiçbirine ulaşamaz. Böyle bir isteğin kolon listesi yoktur: columns null'dır ve üçlü "payment_tokens üzerinde UPDATE" olarak okunur.

Bedel onaylayana düşer; o, tablonun Confidential olanlar dahil Restricted olmayan her kolonunu onaylamıştır. İsteyen expires_on'u değiştirmek istediye, onaylayan provider ve last_four'u da onaylıyor demektir. Onay ekranı "bu tablonun Restricted olmayan tüm kolonları" demelidir ve onaylayanın okuduğu statement ağırlığın daha fazlasını taşır. Bir ekibin sık açmayı beklediği bir kolon grubu kendi küçük tablosunu hak eder.

Fallback, Restricted veri kuralını değiştirmez. Bir kişi onu hâlâ asla ham SQL ile yazmaz; değişiklik, ikinci bir kişi onayladıktan sonra admin-console ve payments-service üzerinden geçer (Bölüm 10).

Yolu seçmek

Sabahın üçündeki bir kişinin hangi kapıyı kullanacağını bilmesi gerekir ve cevabın yeri runbook'ta kısa bir akıştır. Aşağıdaki diyagram onun biçimi. Hiçbir insan veri yolunun hizmet etmediği iki uca bakın: acil durumda bile ürün yoluna giden bir Restricted kolona yazma ve acil durum dışında reddedilen bir Restricted kolonu okuma. Tasarım bunu söyler, kararı baskı altındaki kişiye bırakmaz.



Sabah üçte Ledgerly

Robin'e dönelim. Takılı satırlar rutin session'da sayıldı ve sayı runbook'la eşleşiyor. Yazma, `fix-0093`'e bağlı, bir saatlik, açık alert'e atıf yapan istektir. `chooseApproverPool` ikinci on-call mühendisini ve takımın

lead'ini bu sırayla seçer ve alarm ilkinde gider. Eski Ledgerly burada paylaşılan parolayı açardı. Yenisinde bir bekleme var ve bekleme, tasarımın işlemesidir.

Kimse cevap vermezse runbook her şeye karar veren soruyu sorar: bu, onaylayan havuzunun zamanında karşılayamayacağı bir acil durum mu? Takılı hatırlatmalar sabaha kadar bekleyebilir; bu yüzden Robin alert'e bir not bırakıp duruyor; eski parolanın hiç izin vermediği bir sonuç.

Bunun yerine payments-service'in başarısız olduğunu, ödeme tahsilatının her müşteri için durduğunu ve teşhisin hiçbir JIT isteğinin içermeyeceği bir Restricted kolona ihtiyaç duyduğunu varsayın. Bu, break-glass için bir durumdur. Robin, alert'i gerekçe göstererek beyan eder; gateway human_breakglass'i sabit penceresi için grant eder; güvenlik iletişim kişilerine ve platform ekibine haber verilir. Ertesi sabah Robin olmayan biri log'u okur ve review sorularını sorar; bunlardan biri, teşhisin neden bir Restricted kolona ihtiyaç duyduğudur. Cevap, gelecek çeyrekte daha güvenli bir teşhis view'una yol açabilir.

Trade-off'lar ve maliyetler

Sürtünme bedeldir ve bir şey satın aldığı yerde harcanmalıdır. Bir JIT isteği isteyene en iyi ihtimalle yazmak için bir dakikaya, onaylayana okumak için bir dakikaya, en kötü ihtimalle uyandırılmış bir mühendise mal olur. Üç ayda iki kez olan bir görev için bu ucuzdur. Günde iki kez olan bir görev için, operasyonel kılığa girmiş bir ürün özelliğidir ve Bölüm 10 onu admin-console'a nasıl taşıyacağınızı anlatır.

Bir düzeltmeye bağlamak hazırlığa mal olur. İsteğin dosyalanabilmesi için düzeltmenin yazılmış olması gerekir; bu, tek başına bir grant'ten daha yavaştır. Çalışabilecek olanı değil, çalışacak olanı onaylayanın bedelidir.

Onay yorgunluğu, onaylayanın değil, tasarımın başarısızlığıdır. Her seferinde aynı kişiden saniyeler içinde gelen bir onay tiyatrodur: ya istekler önemsizdir ve human_readonly onları kapsamalıdır ya da onaylayan okumayı bırakmıştır. Çare daha küçük istekler, grant'lerin ne olacağını gösteren bir ekran, tabloları bilen onaylayanlar ve kimsenin varsayılan olmadığı kadar büyük bir havuzdur. İki sinyal erişim review'una

aittir: isteklerin ne sıklıkla reddedildiği ya da geri gönderildiği ve aynı onaylayanın ne sıklıkla cevap verdiği. Sıfır ret oranı, isteklerin iyi olduğu kadar kimsenin bakmadığı anlamına da gelebilir.

Maliyeti önce on-call mühendis taşır. Onaylayan uyandırılır, isteyen bekler ve takım nadiren kullandığı bir runbook öğrenir. Onaylayan havuzunu bir rotation yapın, yayımlayın ve içindeki insanların **invoices** ya da **payments** hakkındaki bir isteği okuyacak bağlama sahip olduğundan emin olun.

Gateway, her yazmanın bir bağımlılığıdır. Revoke job'u, audit log ve istek kayıtlarının hepsi bir grant'in bağımlılıklarıdır; biri kapalıyken grant'ler durur; bu maruziyeti sınırlar ve olaya müdahaleyi yavaşlatır. Bölüm 7 ihtimalleri nasıl azaltacağınızı anlatır.

Reconciliation gürültü üretir. Drift check ve overdue check, bir dakika geç çalışan bir revoke gibi meşru durumları işaretleyecektir. Tolerans süresini, bir işaretin bir anlam taşıyacağı şekilde ayarlayın; susturulmuş bir kontrol hiç olmamasından kötüdür, çünkü ekip kapsandığına inanır.

Break-glass'ın sosyal bir maliyeti var. Bir tane beyan eden kişi, güvenlik iletişim kişilerine normal yolun başarısız olduğunu söylüyordur. Review bir yargılama gibi hissettirirse insanlar gerçek acil durumlarda tereddüt eder; bir tasarım review'u ise etmezler. Rolü kullanılabilir tutan, suçlayıcı olmayan review'dur.

Bunu ne zaman kullanmamalı. Düşük riskli tek bir veritabanı olan ve Restricted verisi olmayan dört mühendislik bir takımın ilk günden onaylayan havuzlarına ve bir reconciliation job'una ihtiyacı yoktur. Bir gerekçesi, bir zaman sınırı, onaylayan bir kişisi ve bir kaydı olan manuel bir süreç, süre dolduğunda grant'ler kaldırılıyorsa ve biri kontrol ediyorsa bir süre için yeterli olabilir. Örüntüyü başarısızlıklarının sırasıyla kurun: önce süre dolumu, sonra kayıt, sonra onaylayan, sonra alert.

Sık yapılan hatalar

- **Tavan rolünü bir login'e "bu tek görev için" grant etmek.** O zaman her JIT session'ı tavanın tamamını tutar ve isteğin kapsamı süstür. Üçlüleri session login'ine grant edin ve **human_jit_write**'i hiçbir

login'in ait olmadığı bir tavan olarak tutun; drift check bir üyeyi işaretler.

- **Statement'ı değil, üçlülere onaylamak.** Bir kolon listesi kısa ve okuması kolaydır ve onu onaylamak bir yeteneği onaylamaktır. Yazma isteğini review edilmiş bir düzeltmeye bağlayın, üçlülere ondan türetin ve `approved_hash`'i düzeltmenin kaydından alın.
- **Ayrı bir listeden revoke etmek.** Sabit kodlanmış bir tablo listesi eninde sonunda isteğin grant ettiği bir tabloyu kaçıracaktır. Revoke'u, grant'i yürüten kayıt olan `access_request_privilege`'dan türetin.
- **Yalnızca revoke job'una güvenmek.** Bir yıldır her gece çalışan bir job'u izlemeyi bırakmak kolaydır. Session'a kendi bitiş zamanını verin ve grant'leri isteklerle karşılaştıran bağımsız bir kontrol çalıştırın.
- **Audit girdisini grant'ten sonra yazmak.** Log kapalıysa kimsenin göremediği bir grant var olur. Önce kaydı, sonra log girdisini, sonra grant'leri yazın ve başarısız bir log girdisinin grant'i durdurmasına izin verin.
- **On-call isteklerinin otomatik onayı.** Okumalar istek gerektirmez ve bir yazma ikinci bir kişi gerektirir. Beklemeyi iyi bir onaylayan havuzuyla kısaltın.
- **Break-glass'ı rahat olacak kadar uzun yapmak.** Bir iş günü içinde hiç dolmayan bir pencere, çalışan bir role dönüşür. Maksimumu gateway'de sabitleyin ve ikinci bir beyanın yeniden alert üretmesine izin verin.
- **Logging setini tanımsız bırakmak.** Bir reviewer, kaydı hiç belirtilmemiş bir session'ı değerlendiremez. Seti ilk kullanımdan önce adlandırın.
- **Hiçbir şey ters gitmediği için review'u atlamak.** Normal yolun onarıldığı yer review'lardır. Review'a bir son tarih verin ve geçtiğinde alert yükseltin.
- **Gateway'in kötü günü için ikinci bir kapı kurmak.** Gateway'in dışındaki bir acil durum login'i, onun kimliğini, loglamasını ve süre dolumunu atlar. Önce availability'ye yatırım yapın; ikinci bir yol kaçınılmazsa ona bir sahip, bir alert ve bir bitiş tarihi verin.

Temel çıkarımlar

- Kalıcı erişim en çok kullanılmadığı günlerde maliyetlidir; bu yüzden yararlı soru, şimdi kimin hangi görev için erişime ihtiyaç duyduğu ve sistemin görevin bittiğini nasıl anlayacağıdır.
- `human_jit_write` bir tavadır ve hiçbir login'e grant edilmez; bir yazma isteği review edilmiş düzeltmesini (`fix_id`) adlandırır, üçlüler düzeltmeden türetilir ve onaylayan statement'ı onaylar; runner'ın eşleşmesi gereken hash'i (`approved_hash`) o statement'ındır.
- Revoke, grant'i yürüten aynı `access_request_privilege` kaydından türetilir, session'ın kendi bitiş zamanıyla uygulanır, bir revoke job'uyla desteklenir ve ikisine de güvenmeyen bağımsız bir reconciliation tarafından kontrol edilir.
- Yazmalar four-eyes gerektirir: isteyen ya da düzeltmenin yazarı olmayan, gateway'in ekranında yeni bir multi-factor kontrolünün ardından cevap veren bir onaylayan. On-call için otomatik onay yoktur.
- Break-glass, sabit bir maksimum pencere, tanımlı bir logging setli, beyan anında bir alert'li ve kullanıcıdan başka biri tarafından zorunlu review'lu, kendi kendine beyan edilen bir `human_breakglass` session'ıdır. Ne rutin bir yoldur ne de veri düzeltmesi yolu; kabul ettiği tek risk bugün sınırsız bir `UPDATE`, hedef durumda ise Bölüm 9'un gateway'i bir cap uyguladığında cap içindeki yanlış satırlardır.
- Onaylı bir hızlı yolu olmayan bir tasarım onaysız bir tane büyütür; bu yüzden acil durum yolu tasarlanır, loglanır ve prova edilir ve gateway'e bağımlılığı adlandırdığınız bir maliyettir.
- Engine kolonlar üzerinde grant edemediğinde, `secrets-tables fallback`'i Restricted kolonları her istekten uzak tutar ve onaylayan, Restricted olmayan kolonların tüm tablolarını onaylar.

Bir grant, bağlı olsa bile, bir session'ın neye dokunabileceğini ve ne kadar süreyle dokunabileceğini söyler; arkasındaki statement'ın ne kadar dikkatle kontrol edildiğini değil. Bölüm 9 buna dönüyor: gateway bir statement çalışmadan önce onunla ne yapar. Bölüm 10 ardından işin kendisine dönüyor: doğru satırların değişmesi, yanlış satırların değişmemesi ve ikinci bir kişinin nedenini görebilmesi için bir veri düzeltmesi nasıl çalıştırılır.

Bölüm 9 — Aracılı Yürütme: Sorgu Analizi ve Tehlikeli İşlemler

Ledgerly'nin destek ekibinin chat aracında bir asistan var. Sam ondan, bir billing retry'ından kalan mükerrer faturaların bir partisini iptal etmesini istiyor. Asistan kendi kimliğiyle çalışarak bir düzeltme taslağı hazırlıyor: `invoices` üzerinde, retry'ın batch etiketine göre `WHERE` koşulu olan bir `UPDATE`. Veri ekibinden Jordan statement'ı inceleyip onaylıyor. Statement geçerli bir SQL, grant onu kapsıyor, onaylayan okudu ve etiket Jordan baktığı anda doğrudu.

Buna rağmen statement, düzeltmenin adını verdiği on iki satırdan fazlasıyla eşleşiyor. İki gün önce ikinci bir job aynı batch etiketini yeniden kullanmış ve statement'ın metninde bunu söyleyen hiçbir şey yok. Kimse özensiz davranmadı: Sam görevi doğru anlattı, asistan kendisinden istenen şeyi yazdı, Jordan da kendisine gösterileni inceledi. Eksik olan, "onaylandı" ile "commit edildi" arasındaki bir adımdı; yani yalnızca statement'ın ne dediğine değil, veriye ne yapacağına bakan bir yer.

Bölüm 3'ten Bölüm 8'e kadar, bir grant'i kimin, hangi yol üzerinden ve ne kadar süreyle tutabileceğine karar verdik. Bu bölüm beş parçalı modelin Yürütme parçasını ekliyor: gateway'in bir session teslim edip onu elinde tutana güvenmek yerine her statement'ı kendisinin ele aldığı adım. Bölümün cevapladığı soru şu: onay tek başına statement'ın ne yapacağını söyleyemiyorsa, onaylanmış bir statement ile veri arasında ne durmalı?

Problem ve neden zor olduğu

Bir session handout, bir kişiye bir login verir ve client'ının veritabanıyla doğrudan konuşmasına izin verir. Bundan sonrası client'ın işidir: ne yazıldığı, kaç satıra dokunulduğu, hemen commit edilip edilmediği. Grant hangi tablolara ve kolonlara erişilebileceğini sınırlar, ama o erişim alanının

içinde tek bir statement'ın ne yaptığını hiçbir şey sınırlamaz. Başlangıç durumunda bir masaüstü client yazılan her şeyi gönderir ve anında commit eder.

Aracılı yürütme bu düzeni tersine çevirir. Statement, kişinin client'ı olmayan bir bileşene gider ve çalışıp çalışmayacağına, hangi sınırlarla çalışacağına ve ne zaman commit edileceğine o bileşen karar verir. Bölüm 7'deki gateway zaten veri yolunun üzerindedir; bu iş için doğal yer orasıdır. Statement'ı yine kişi yazar, ya da bir otomasyon; değişen, hiçbir statement'ın bir sonraki bölümdeki döngüden geçmeden veriye ulaşmamasıdır.

Bu bir anahtar değil, bir yelpazedir. En ucuz biçimi, gateway'in uyguladığı birkaç ayarla (statement timeout ve row cap gibi) verilen bir session handout'tur. Bir sonraki biçim, her statement'ı iletmeden önce kontrol eder. En çok aracılı biçim her statement'ı ayrı ayrı onaylar ve bir worker'dan çalıştırır; böylece kişi yazma için hiçbir zaman bir veritabanı session'ı tutmaz. Bazı araçlar bir zaman penceresini değil, her sorguyu onaylar. Bu yazı yazılırken, örneğin QueryProxy statement'ı ikinci bir kişiye onaylatır ve bir worker'a çalıştırır; böylece geliştirici veritabanı credential'ını hiç tutmaz. Bu, interaktif bir session'ın hızını, tam olarak o statement'a bağlı bir onayla takas eder; diğer sorgu onay araçları da benzer şekilde çalışır.

Aracılık üç nedenle zordur. Birincisi, SQL anlatım gücü yüksek bir dildir: aynı etki birçok biçimde yazılabilir ve bir statement'ın "tehlikeli" olup olmadığı çoğu zaman kontrol edenin okumadığı veriye bağlıdır. İkincisi, her kontrolün gecikme, yanlış ret ve onu aşmayı öğrenen insanlar biçiminde bir maliyeti vardır. Üçüncüsü, fazla güvenilen bir kontrolün, kimsenin gözden geçirmedeği şeye dönüşmesidir. Bu bölümün benimsediği pozisyon şudur: statement analizi hataları azaltır, yetkilendirme sınırı değildir. Sınır grant'lerdir. Gateway'in işi, sık görülen yanlış statement'ı kazayla çalıştırmayı zorlaştırmak ve her yazmanın, yazanın dışında bir şeyin uyguladığı bir sınır taşımasını sağlamaktır.

Gateway'in her statement için döngüsü

Bir session'ın gönderdiği her statement, insan olsun olmasın, aynı döngüden geçer. Sıra, tek tek adımların hiçbirinden daha önemlidir; bu yüzden parçalar açıklanmadan önce pseudocode olarak veriliyor.

```
function handleStatement(session, statement):
    if statement.changesSetting(guardedSettings):
        return refuse(session, "guarded setting")
    verdict = evaluateStatement(session, statement)
    // the record comes first: a statement that cannot be recorded
    does not run
    try:
        recordIntent(session, statement, verdict)
    on RecordUnavailable:
        return refuse(session, "record unavailable")
    if verdict.action == "deny":
        return refuse(session, verdict.reason)
    if verdict.action == "gate":
        return routeToGate(session, statement, verdict) // approval,
    or the product path
    if statement.isRead:
        return streamRows(session, statement, cap: session.rowCap)
    // cursor closes at the cap
    // auto-commit is off: the runner owns the transaction of a fix,
    the gateway owns the rest
    tx = transactionFor(session)
    result = execute(tx, statement)
    tx.rowsSoFar = tx.rowsSoFar + result.rows // counted across
    every statement of the fix
    if tx.rowsSoFar > verdict.bound:
        failTransaction(tx) // its owner can
    only roll back
    return refuse(session, "more rows than approved")
    showRowCount(session, tx.rowsSoFar) // before COMMIT,
    while undo is free
    if tx.ownedByGateway:
        if not countAccepted(session, tx.rowsSoFar, verdict.bound):
            rollback(tx)
            return refuse(session, "count not accepted")
        commit(tx) // the commit point
    for a person or a migration
        recordOutcome(session, statement, result) // failure is
    queued and alerts; a commit stands
    return result
```

Guarded setting'ler. Gateway her session'ı limitleri uygulanmış olarak açar: statement timeout, read-only varsayılan, session türüne göre rol ve bağlantı limiti (Bölüm 5). Bir client bunların herhangi birini kaldıran bir SET gönderebilir; bu yüzden gateway bir guarded setting listesi tutar ve

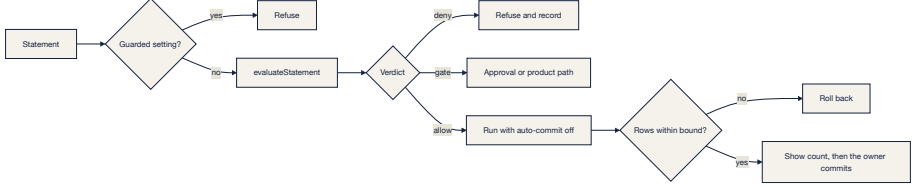
bunlardan birini deęiřtiren statement'ı reddeder. "Override edilemez" sözcüğünü hak etmek için bunu zorlayan bir mekanizma gerekir. Rol varsayılanı olarak ayarlanmış bir timeout, gateway SET'i reddetmedikçe ya da platform onu sabitlemedikçe session tarafından kaldırılabilir; yani ikisinden de yoksun bir limit bir varsayılandır, garanti deęildir. Maliyeti, bazı client'ların bağlanırken bu tür ayarlar göndermesi ve biri bunları listeye ekleyene kadar çalışmaz hale gelmesidir.

Row cap. Okumalarda gateway satırları stream ederken sayar, cap'e gelince cursor'ı kapatır, kişiye söyler ve olayı kaydeder. Cap, bir kişinin bir günde yalnızca řu kadar satır görebileceęi anlamına gelmez; kararlı bir kişi bir tabloyu dilimler halinde sayfa sayfa gezebilir. Cap'in söyledięi, tek bir yanlış sorgunun bütün bir tabloyu bir laptop'a çekmemesidir. Büyük ve meşru extract'ların yeri, Bölüm 13'teki export yolu ile Bölüm 12'deki maskeli analytics kopyasıdır; orada onaylayanı tier belirler.

Yazmalarda auto-commit kapalı. Bir yazma session'ı her yazmayı açık bir transaction içinde çalıştırır. Bir kişi ya da bir migration için transaction'ı gateway açar; bir veri düzeltmesi için Bölüm 10'daki runner açar ve gateway düzeltmenin her statement'ını o tek transaction'a iletir. Her iki durumda da gateway etkilenen satırları bütün statement'lar boyunca sayar ve sınırla karşılaştırır; COMMIT kararı ise transaction'ın sahibinde kalır: bir düzeltmenin her statement'ını kendi başına commit eden bir gateway runner'ın tek transaction'ını bozar ve dry-run'ını gerçek bir deęişikliğe çevirirdi. Sayı sınırı aştığında gateway statement'ı reddeder ve transaction'ı başarısız olarak işaretler; böylece sahibi yalnızca rollback yapabilir. Bu, Bölüm 1'deki geri alınabilirliğin ucuz biçimidir: bir satıra dokunması beklenen ama dört yüz bildiren bir update, geri alma hâlâ bedavayken durdurulur. İnteraktif bir session için "gösterilir", kişinin sayıyı görüp onaylaması demektir. Runner ve otomasyon için "gösterilir", gateway'in sayıyı onayın kaydettięi sınırla karşılaştırması ve herhangi bir fazlalıkta transaction'ı başarısız kılması demektir. Maliyet, ek adım ve kişi karar verirken tutulmaya devam eden transaction lock'larıdır; kimsenin cevaplamadıęı bir onay istemi bir satır kilidini tutar, bu yüzden istemin rollback yapan bir timeout'u vardır.

Önce kayıt. Gateway niyeti statement çalışmadan önce, sonucu sonra yazar. Gateway'in log'unu ve veritabanını kapsayan bir transaction yoktur, bu yüzden garantiyi taşıyan şey sıradır. Niyet yazılamazsa statement

çalışmaz. Sonuç bir commit'ten sonra yazılamazsa commit geçerli kalır, kayıt kuyruğa alınır ve platform ekibine alarm gider; yalnızca hiçbir şey ters gitmediğinde çalışan bir log bu adı hak etmez. Kaydın nerede durduğunu Bölüm 14 anlatır.



Statement'ı sınıflandırmak

Gateway bir statement'la ne yapacağına `evaluateStatement` içinde karar verir. Statement'ı ne yaptığına ve nerede yaptığına göre sınıflandırır, bir sonraki bölümden sonraki bölümdeki matrise bakar ve matrisin bir sınırla izin verdiği bir yazma için sınırı bulup planner'ın tahminini onunla karşılaştırır. Her ret ve her gate matristen gelir; fonksiyonun, parser'ın dışında kendi erken return'leri yoktur.

```

function evaluateStatement(session, statement):
    parsed = parse(statement)
    if parsed.failed or parsed.isMultiStatement:
        return deny("cannot be analyzed")           // fail closed
    // nine classes, one per matrix row; the most specific class wins
    class = classify(parsed)
    cell = matrix[class][session.kind]              // deny, gate,
allow or allow with a bound
    if cell.action != "allow":
        return cell                                // a deny, or a
gate with its route
    if class == "read":
        return allow(cap: session.rowCap)
    if cell.bound == "fix":
        binding = checkBinding(session, statement)  // the next
statement of the approved fix
        if binding.action == "deny":
            return binding
        bound = binding.bound
    else:
        bound = session.writeBound()                // the break-glass row cap, or
the backfill rule's limit
        if bound is none:
            return deny("no approved count")
        estimate = plannerEstimate(parsed)           // asks the
planner; runs nothing
        if estimate > bound * tolerance:
            return deny("estimate exceeds the approved count")
        return allow(bound: bound)

```

Savunmaya değer üç tercih var. Birincisi, parser'ın okuyamadığı statement reddedilir. Gateway analiz edemediğini çalıştırsaydı, analiz yalnızca hiçbir zaman sorun olmamış statement'lara uygulanmış olurdu. Tek bir string içinde statement batch'leri gönderen bir client bu kuralda bozulur ve bu, migration'da adı konulması gereken bir maliyettir; cevap, onları tek tek göndermektir.

İkincisi, sınıflandırma kişinin ya da aracın adına değil, etkiye bakar. Bir migration'daki **DELETE** ile bir JIT session'daki **DELETE** aynı sınıftır ve her lane'in onunla ne yapabileceğine matris karar verir. Her yol için ayrı bir kural kümesi yerine tek bir tabloya sahip tek bir sınıflandırıcı olması, politikayı gözden geçirilebilecek kadar küçük tutar.

Üçüncüsü, planner'ın tahmini beklenen sayıyla karşılaştırılır, ama yalnızca erken bir ret olarak. Planner istatistiklerden tahmin eder ve istatistikler eskir; bir tahmin çok yüksek olabildiği kadar kolayca çok düşük de olabilir. Tahmin, gateway'in açıkça yanlış olan bir statement'ı lock almadan önce

reddetmesini sağlar. Asıl kontrol, COMMIT'ten önce transaction içinde karşılaştırılan gerçek sayıdır. Tolerans, platform ekibinin belirleyip kaydettiği bir parametredir; ben sıkı başlamayı ve gevşetmeyi ancak yanlış retler tahmin edilerek değil ölçülerek görüldüğünde yapmayı tercih ederim.

Statik analizin göremedikleri

Bir statement'ın metni, satırlar üzerindeki etkisi hakkında çok az şey söyler. Bir parser WHERE'in eksik olduğunu, bir predicate'in her zaman true olduğunu, statement'ın dokunmaması gereken bir tabloya dokunduğunu ya da bir fonksiyon çağırdığını görebilir. Bir predicate'in bugün kaç satırla eşleştiğini göremez; baştaki sorun da buydu: geçerli bir filtre, artık kimsenin beklediğinden fazla satırla eşleşen bir etikette.

Analizin çözemeyeceği birkaç durum daha var. Bir filtre geçerli olup yine de geniş olabilir; sözdizimi yüzünden değil, veri yüzünden bir tablonun çoğuyla eşleşir. Bir join satırları çoğaltabilir. Bir view ya da bir fonksiyon, zararsız görünen bir adın arkasına bir yazma ya da daha geniş bir okuma saklayabilir ve gateway, rolün çağırmasına izin verilen bir fonksiyonun gövdesinin içeri göremez. Tetiklenen bir yan etki statement'tan sonra gerçekleşir ve statement'ın metninde görünmez. Üstelik bir statement biçim olarak tam doğru, anlam olarak yanlış olabilir: filtreye göre doğru satırlar, niyete göre yanlış satırlar; bunu hiçbir araç yargılayamaz.

QueryProxy gibi açık kaynak araçlar statement'ları gönderim anında kontrol eder; WHERE'i eksik ya da her zaman true olan bir UPDATE veya DELETE'i reddeder ve sınırsız SELECT'leri cap'ler. Bu yazı yazılırken kendi dokümantasyonu bu parser'ı yetkilendirme sınırı olarak değil bir savunma katmanı olarak tanımlıyor; bu bölümün vurguladığı nokta da budur ve diğer gateway'ler ile onay araçları benzer yaklaşımlar izler.

Bundan çıkan bir iş bölümüdür. Analiz sık görülen hataları ucuza ve erken yakalar. Grant'ler (Bölüm 3) neye erişilebileceğini belirler; dolayısıyla gözden kaçan bir statement yine de rolle sınırlıdır. Transaction içindeki sayı kontrolü, verinin yazarla aynı fikirde olmadığı durumu yakalar. Statement'ı ve beklenen sayıyı okuyan ikinci bir kişiyle onay ise biçimin

doğru, niyetin yanlış olduğu durumu bazen yakalar. Bunların hiçbiri tam değildir ve tasarım, örtüşmelerine güvenir; hiçbirini sınır olarak ilan etmez.

Aynı sınır, Bölüm 13'teki export tier'ını da şekillendirir. Gateway bir export'un tier'ını sonuç kolonlarının adlarından değil, sorgunun okuduğu kaynak kolonlardan hesaplar ve çözemediği bir kaynak, export edilebilecek en yüksek tier sayılır; bunun bile kararlaştırılamadığı yerde export reddedilir. Bu, parser'ınkiyle aynı fail-closed kuralıdır: analizin okuyamadığını geçirmez.

Deny, gate ve allow matrisi

Her statement sınıfı, her lane'de üç cevaptan birini alır. **Deny**, gateway'in reddettiği ve reddi kaydettiği anlamına gelir. **Gate**, statement'ın gönderildiği gibi çalışmadığı anlamına gelir: henüz sahip olmadığı bir onaya ihtiyacı vardır ya da farklı bir yola aittir. **Allow**, yukarıdaki döngünün limitleri altında çalışır demektir; bir hücre "bound" diyorsa yazmaya yalnızca hücrenin adını verdiği kaynaktan gelen bir sınırın içinde izin verilir ve gerçek sayı COMMIT'ten önce onunla karşılaştırılır. Sütunlar kitabın anlattığı lane'ler, satırlar ise sınıflandırıcının ürettiği statement sınıflarıdır.

Statement sınıfı	İnsan okuma session'ı	JIT write	Break-glass	Veri düzeltmesi runner'ı	Migration
Okuma, row cap dahilinde	Allow	Allow	Allow	Allow	Allow
Sınırlı yazma, onaylı sayı dahilinde	Deny	Allow, düzeltmeye bağlı (yalnız runner)	Allow, bir row cap'e bağlı (not zorunlu)	Allow, onaylı hash'e ve sayıya bağlı	Allow, backfill kuralına bağlı
Sınırsız yazma	Deny	Deny	Deny	Deny	Gate: belirlenmiş onaylayan
Owner-link değişikliği	Deny	Gate: ürün yolu	Gate: ürün yolu	Gate: ürün yolu	Gate: belirlenmiş onaylayan

Statement sınıfı	İnsan okuma session'ı	JIT write	Break-glass	Veri düzeltmesi runner'ı	Migration
Restricted kolona yazma	Deny	Gate: ürün yolu	Gate: ürün yolu	Gate: ürün yolu	Gate: belirlenmiş onaylayan
Export	Gate: tier'a göre	Deny	Deny	Deny	Gate: belirlenmiş onaylayan
Şema değişikliği (DDL)	Deny	Deny	Deny	Deny	Allow: review'dan geçen lane
Yetki değişikliği	Deny	Deny	Deny	Deny	Gate: platform review'u
ledger_entries ya da audit_events tablosuna yazma	Deny	Deny	Deny	Deny	Gate: platform review'u

Satır satır okumak tasarımı gösterir. İnsan okuma session'ının yazma yolu yoktur, bu yüzden her yazma sınıfı orada deny'dır ve tek gate export tier'ıdır. Bir JIT write session'ı yazmaları yalnızca runner'dan iletir; dolayısıyla sınırlı yazma hücreleri açık bir kapı değil, bağlayıcı bir allow'dur. Break-glass, bir kişinin doğrudan yazdığı tek lane'dir; bu yüzden sınırlı yazma orada yalnızca bir row cap altında ve bir notla serbesttir ve bu cap'in büyüklüğü kitaptaki en zor boyutlandırma kararıdır: bir acil durum planlı görev değildir, dolayısıyla cap görevden seçilemez. Cap, break-glass rolünün onarmak için makul biçimde neye ihtiyaç duyacağına göre belirlenmeli ve her kullanımdan sonra gözden geçirilmelidir. Sınırsız yazma orada deny'dır. Geriye, cap'in içinde kalan ve yanlış satırlara denk gelen bir yazma kalır; Bölüm 8'in belirttiği gibi, review'un ortaya çıkarmak için var olduğu kabul edilmiş bir risktir. Bölüm 8 ayrıca, bir gateway cap'i uygulamaya başlamadan önce bugün neyin geçerli olduğunu da söyler.

Owner-link değişikliği kendi satırını hak eder. bank_accounts ya da payment_tokens üzerinde customer_id'nin UPDATE'i JIT tavanından geçer, çünkü kolon bir satırı sahibine bağlar ve Restricted değildir. Yine de satırı başka bir müşteriye taşımak, o ödeme hesabının ya da ödeme token'ının kime ait olduğunu değiştirir. Bu yüzden sınıflandırıcı bunu

kendi sınıfı olarak adlandırır ve migration dışındaki her lane, onu ürün yoluna yönlendiren bir gate ile cevap verir: `admin-console`, `payments-service`'ten ister, ikinci bir kişi onaylar ve değişiklik `audit_events` içine kaydedilir. Normal lane'deki bir migration için bunu belirlenmiş bir onaylayana gate'ler; Bölüm 11'in işaretlenen her statement için yaptığı gibi. Hızlandırılmış lane ise onu reddeder; bir şema değişikliğinin sahipliği yeniden atamakla pek işi olmaz ve toplu bir yeniden atama gerekirse bu, çalıştırılacak bir statement değil, inşa edilecek bir ürün özelliğidir. Restricted kolonlara yazmalar (`iban`, `account_holder_name` ve `token_value`) aynı yolu izler; bu yüzden hiçbir lane bir insanın bunları ham SQL ile yazmasına izin vermez. Tek migration istisnası, Restricted kolonların Bölüm 11'deki ayrı secret tablolarına taşınmasıdır; bu da reddedilmez, belirlenmiş bir onaylayana gider.

Migration sütunu, Bölüm 11'in anlattıklarıyla örtüşür. Onun sınıflandırıcısı statement'ları belirlenmiş bir onaylayan ya da platform review'u için işaretler ve yalnızca hızlandırılmış lane'de reddeder. Bu tabloda işaretler, bir kişinin evet demesi gereken gate'ler olarak görünür; allow'lar ise review'dan geçmiş şema değişikliği ve kendi kuralının içinde kalan backfill'dir. Append-only tablolar da aynıdır: onlara dokunan bir migration platform ekibine gider, diğer lane'ler onu deny eder. Yasaklı statement'lar için ikinci bir kural kümesi, bununla uyumlu tutulması gereken ikinci bir politika olurdu; bu yüzden migration pipeline'ı sınıflandırıcıyı ve matrisi yeniden kullanır ve yalnızca kendi lane kurallarını ekler.

Matris bilerek küçüktür. Dokuz sınıf ve beş lane bir sayfaya sığar ve bir sayfaya sığan politika okunur. Ona eklenen her yeni sınıf ya da lane, birinin savunması gereken bir karardır ve olağan başarısızlık, her olay için bir istisna biriktirip sonunda neyi yasakladığını kimsenin söyleyemediği bir matristir. Ben onu version control'da tutmayı, kod gibi gözden geçirmeyi, sahibinin platform ekibi, yazma satırları için zorunlu reviewer'ın ise veri ekibi olmasını tercih ederim.

Onayı statement'a bağlamak

"`invoices` üzerinde bir yazma"yı kapsayan bir onay, `invoices` üzerindeki her yazmanın geçmesine izin verir. Statement'a bağlı onay, onayın bu statement'ı adıyla belirtmesi ve gateway'in diğer hepsini reddetmesi

demektir. Bölüm 8 isteği, Bölüm 10 runner'ı anlatır; bu bölüm, ikisini birbirine bağlayan kuralı bir kez söyler.

Bir yazma isteği bir `fix_id` ve bir `approved_hash` taşır. Hash, review edilen statement'ların hash'idir; bir düzeltme birden fazla statement içeriyorsa bütün küme üzerinden alınır ve kaynağı, kontrolünden daha önemlidir. Düzeltmenin merge kaydından ya da onaylanmış istek kaydından gelir ve hiçbir zaman runner'ın girdisinden gelmez. Runner kendi hash'ini sağlasaydı kendi işini kendisi kontrol ediyor olurdu ve statement'taki bir değişiklik, eşleşen bir hash'le birlikte gelirdi. İkinci bir kişinin onayladığı kayıttan çıkan bir hash, yazan dışında birinin kefil olduğu bir hash'tir.

```
function checkBinding(session, statement):
    request = loadApprovedRequest(session.requestId)    // from the
store, not from the caller
    if request is none or request.status != "approved":
        return deny("no approved request")
    if request.fixId is none:
        return deny("no fix named")                    // a SELECT-
only request has no fix and no hash
    fix = loadMergeRecord(request.fixId)                // the
statements, read from the record
    if hashOf(fix.statements) != request.approvedHash:
        return deny("record differs from the approved hash")
    next = fix.statements[session.transaction.statementCount]
    if normalize(statement) != normalize(next):
        return deny("statement differs from the approved one")
    return allow(bound: fix.expectedRows)              // the count
comes from the hashed record
```

Gateway statement'ları tek tek kontrol eder, ama hash kümeyi kapsar: statement'ları merge kaydından yükler, hash'lerinin onaylanmış hash olduğunu doğrular ve yalnızca o sıradaki bir sonraki statement'ı kabul eder. Sınır, aynı kayıttaki beklenen satır sayısıdır; isteğin taşıdığı bir sayı değildir, dolayısıyla hash sayıyı da korur. Commit runner'ın transaction'ında kalır; bu yüzden birkaç statement'lı bir düzeltme ya bütünüyle uygulanır ya da hiç uygulanmaz.

İki kural bağlamayı tamamlar. Bir JIT write session'ında gateway yazmaları yalnızca runner'ın bağlantısından iletir. Kişinin kendi client'ı, verilen üçlüler içinde yine okuyabilir, ama oradan gönderilen bir yazma reddedilir; aksi halde hash, bir client'ın atlayabileceği bir kontrol olurdu. Bir guard ancak session başka türlü yazamıyorsa bir mekanizmadır. İkinci

kural ters durumdur: yalnızca `SELECT` üçlülere tutan bir JIT isteğinin düzeltilmesi ve hash'i yoktur, dolayısıyla bir yazmanın bağlanacağı hiçbir şey yoktur ve yazma deny edilir. Okuma isteği, farklı bir statement göndererek yazma isteğine dönüştürülemez.

Gateway yalnızca bir credential'a aracılık ettiğinde ve client doğrudan bağlandığında bunların hiçbirisi uygulanabilir değildir ve kitap bunu açıkça söyler: hash o zaman bir niyeti kaydeder ve hiçbir şeyi engellemez. Bağlama, veri yolunun bir özelliğidir ve yalnızca gateway o yolun üzerindeyse vardır.

Runbook durumu, şablonu olan durumdur. Bir runbook'un merge kaydı statement şablonunun hash'ini saklar ve istek yalnızca anahtar listesini taşır. Runner anahtarları saklanan şablona yerleştirir. Bu durumda gateway hash'i yerleştirilmiş metne değil, şablona karşı kontrol eder: gönderilen statement'ın, yalnızca anahtar listesi doldurulmuş onaylı şablon olduğunu doğrular ve anahtar sayısını, isteğin üçlülereyle birlikte sınır olarak kullanır. Gateway, işin yapılmasını isteyen çağırandan hiçbir zaman bir statement ya da hash almaz.

İnsan olmayan çağırılar

Job'lar, runbook betikleri, ChatOps bot'ları ve AI agent'lar statement gönderir ve hiçbirisi bir kişi değildir. Bu bölümdeki kural, onların aynı döngüden, kendi principal'ları olarak geçmesidir. Otomasyon için daha hafif bir yol yoktur, çünkü bir hatanın makine hızında tekrarlandığı yer otomasyondur.

Otomatik bir çağırının, bir kişinin session'ı yerine platformdan gelen bir workload identity'si ve read-only bir varsayımı vardır. Session'ını kendi principal'ı altında açar; dolayısıyla okumaları, herkesinki gibi aynı row cap, timeout ve guarded setting'lere tabidir. Yazmak isterse bir kişiyle aynı rotayı izler: bir düzeltmeyi adıyla belirtir, otomasyonu tetikleyenden başka bir kişi olan bir onaylayan onu onaylar ve yazmaları beklenen bir sayıyla runner üzerinden geçer. Bölüm 8'deki iki kişi kuralı burada da geçerlidir: otomasyondan düzeltmeyi hazırlamasını isteyen kişi onu onaylayan kişi olamaz ve kişi olmayan bir çağırının otomatik onayı yoktur. Bir statement öneren agent onu önermektedir; onay, bir insanın onu okumasıdır.

Kayıt iki tarafı da adıyla belirtir: statement'ı gönderen otomasyonu ve onu tetikleyen kişiyi. Bu, hesap verebilirliğin yarısıdır. Yalnızca "asistan yaptı" diyen bir kayıt izi bir makinede bitirirdi, yalnızca "Sam istedi" diyen bir kayıt ise isteği hangi aracın SQL'e çevirdiğini gizlerdi. İki ad da saklanır ve erişim review'u ikisini de sorar.

Bir AI agent, diğer çağırarlarda olmayan bir kaygı daha ekler: statement'ları, bir kısmını kendisinin yazmadığı metinden üretilir. Bir ticket, bir log satırı ya da bir müşteri mesajı, statement'ı üreten bağlama girebilir. Savunma, metni daha dikkatli okumak değildir. Agent'ın yetkilerinin okuduğu şeyle değişmemesidir. Grant'leri aynıdır, yazmaları hâlâ tam statement'ın bir kişi tarafından onaylanmasını gerektirir ve agent'ın sahip olmaması gereken bir yoldan gelen statement, örneğin runner dışında bir yazma, aynı matris tarafından reddedilir. Agent neyi önermeye ikna edilmiş olursa olsun, onu onaylayamaz ve sınır olmadan çalıştıramaz.

Read-only bir agent için işe yarayan limitler olağan olanlardır: sıkı bir row cap, kısa bir timeout, Restricted kolonlar yerine bir view ya da maskeli bir kopya ve principal başına bir bütçe; böylece ters giden bir döngü paylaşılan bütçeyi değil, kendi payını tüketir. Yazan bir agent için işe yarayan limit, yazmalarının az ve küçük olmasıdır, çünkü her biri bir kişi gerektirir. Bu, agent'ı yazmalar için işe yarayamayacak kadar yavaşlatıyorsa doğru sonuç, onayın gevşetilmesi değil, görevin agent'ın çağıracağı bir ürün özelliğine dönüşmesi gerektiğidir.

Trade-off'lar ve maliyetler

Aracılık zamana mal olur. Her statement bir bileşenden daha geçer ve gate'lenmiş bir yazma bir kişiyi bekler. Replica üzerindeki interaktif okumalarda eklenen gecikme küçüktür ve çoğunlukla görünmezdir. Yazmalarda amaç budur: gecikme, ikinci bakışın gerçekleştiği yerdir. Acil durumlar için onu kaldırmazdım; break-glass lane'inin gate'ini hızlı ve gürültülü yapar, sonra gözden geçirirdim.

Uyumluluğa mal olur. Statement'ları parse eden ve parse edemediğini reddeden bir gateway bazı client'ları, bazı driver'ları ve bazı alışkanlıkları bozar. SET statement'larını, multi-statement string'leri ve alışılmadık sözdizimini reddetmek, çalışmaz hale gelen şeylerin bir listesini üretir.

Liste, rollout'tan önce, mevcut session'ların statement kayıtlarından toplanırsa, destek ticket'larıyla keşfedilmesinden daha kısa olur. Gateway'in verdict'lerini kaydettiği ama yalnızca en açık durumları reddettiği bir dönem için bütçe ayırın ve yanlış retler azaldıkça sınıf sınıf enforcement'a geçin.

Ayakta olması gereken bir bileşene mal olur. Statement kontrolü her yazmanın yolundadır ve Bölüm 7 bu bileşenin neden küçük kalması gerektiğini anlatır. Kontrolün çöktüğü yerde hasarı sınırlayan varsayılan, yazmaların durması ve replica üzerindeki okumaların sürmesidir. Fail-open bir kontrol ekibe kontrolün isteğe bağlı olduğunu öğretir ve ilk kesinti politikayı bitirir.

Bakımı yapılacak bir politikaya mal olur. Matris, guarded setting listesi, tolerans ve row cap'lerin hepsi birinin sahip olduğu parametrelerdir. Sahibi olmayan bir politika, son olayın neye ihtiyaç duyduysa ona doğru kayar. Küçük, gözden geçirilmiş ve versiyonlanmış tutulacak tek artifact matristir.

Mühendislikle ödenemeyen bir maliyet de var: analiz yanlış bir güvenlik hissi verir. WHERE'siz UPDATE'i reddeden bir parser'ı olan bir ekip, geçen bir statement'ı zarar veremeyecek bir statement saymaya başlayabilir. Matris gateway'in neyi reddettiğini söyler, bu bölümün başı ise geçen bir statement'ın yine de ne yapabileceğini gösterir. Sayı kontrolü ve ikinci kişi bu durum için vardır ve parser iyi diye inceltilmemelidir.

Sık yapılan hatalar

Analyzer'ı sınır saymak. Analyzer, sık görülen yanlış statement'ı yakalayan bir katmandır. Neye erişilebileceğine grant'ler karar verir ve bu yükü analyzer'a taşıtan bir tasarım, bir statement'ın geçerli ve yanlış olduğu ilk anda başarısız olur.

Hash'i runner'ın ya da çağırının sağlamasına izin vermek. Statement'la birlikte gelen bir hash yalnızca statement'ın kendisiyle eşleştiğini kanıtlar. Hash, merge ya da onaylanmış istek kaydından gelir ve kontrol, çağırının düzenleyemeyeceği gateway'de çalışır.

Yazma session'larında auto-commit'i açık bırakmak. Auto-commit açıkken satır sayısı, olay olduktan sonraki bir rapordur. Onu kapatmak ve sayıyı COMMIT'ten önce göstermek, yanlış bir sayıyı ucuz kılan şeydir.

Parser'ın bir statement'ı okuyamadığı durumda fail-open davranmak. Analiz edilemeyen çalıştırmak, analizin yalnızca zararsız durumları kapsamasına yol açar. Reddedin ve değişmesi gereken client'ların kısa bir listesini tutun.

Otomasyon için daha hafif bir yol. Job'lara ve agent'lara, güvenildikleri ya da acele ettikleri için döngüyü atlayan bir kestirme vermek, kontrolleri hacmin en yüksek olduğu yerden kaldırır. Otomasyon aynı döngüden, kendi principal'ı altında geçer.

Her olay için bir istisna biriktiren matris. Her istisna, sahibi olmayan bir karardır. Tabloyu var olan sınıflar ve lane'lerle sınırlı tutun ve bir durum uymadığında tasarımı değiştirin ya da özelliği inşa edin.

Row cap'i maruziyet limiti olarak okumak. Cap tek bir sorguyu sınırlar, bir kişinin bir günde okuyabileceğini değil. Endişe maruziyetse, bu Bölüm 12 ve Bölüm 13'teki maskeleyme ve tier'lama işine aittir.

Temel çıkarımlar

- Yürütme, bir statement'ın veriye nasıl ulaştığıdır: onu kimin gönderdiği, önce bir şeyin onu inceleyip incelemediği, etkisini neyin sınırladığı ve nasıl commit edildiği. Grant'ler bu soruları cevaplamaz.
- Aracılı yürütme, statement ile veri arasına bir bileşen koyar. En ucuz biçimi birkaç uygulanan ayardır; en güçlüsü her statement'ı onaylar ve bir worker'dan çalıştırır.
- Her statement, bir kişiden ya da bir otomasyondan gelsin, tek bir döngüden geçer: guarded setting'ler, değerlendirme, önce kayıt, okumalar için row cap ve yazmalar için satır sayısı COMMIT'ten önce gösterilen kapalı auto-commit.
- Analiz hataları azaltır ve yetkilendirme sınırı değildir. Sınır grant'lerdir ve transaction içinde kontrol edilen gerçek satır sayısı, parser ile planner'ın yakalayamadığını yakalar.
- Statement sınıfı ile lane'in tek ve küçük matrisi, her cevabı deny, gate ya da allow olarak verir. Owner-link değişikliği ve Restricted kolonlara

yazmalar gate'lenir ve ürün yoluna yönlendirilir.

- Onay statement'a bağlıdır: `approved_hash`, merge ya da onaylanmış istek kaydından gelir, hiçbir zaman runner'dan gelmez ve bir JIT write session'ında gateway yazmaları yalnızca runner'dan iletir.
- Job'lar, runbook'lar, ChatOps ve AI agent'lar kendi kimlikleri, read-only bir varsayılanı ve aynı döngüsü olan principal'lardır. Kayıtları hem otomasyonu hem de onu tetikleyen kişiyi adıyla belirtir.

Döngü ve matris, gateway'in bir statement'la ne yaptığını söyler.

Onaylanmaya değer bir düzeltmenin nasıl yazılacağını söylemez: sayının nasıl kanıtlanacağı, snapshot'ın nasıl saklanacağı, rollback'in nasıl planlanacağı ve ikinci bir kişinin nasıl review edeceği. Bölüm 10 bu işe yönelir.

Bölüm 10 — Ad-hoc Sorgular ve Veri Düzeltmeleri

Ledgerly'de bir Perşembe öğleden sonrasını düşünün. `invoice-worker`'daki bir bug düzeltilip deploy edilmiş, ama canlıdayken bir parti faturayı, e-postalar gitmiş olmasına rağmen `sending` durumunda bırakmış. Destek bunları soruyor, müşterilerin hatırlatmaları durmuş durumda ve on-call backend mühendisi Robin onarımın biçimini tek satırda görebiliyor: etkilenen faturaların durumunu `sent`'e geri al, başka hiçbir şeye dokunma.

Robin access gateway üzerinden bir session açıyor. Varsayılan session read-only, bu yüzden o satır çalışamaz. Robin'in yazma erişimi istemesi gerekiyor ve bu istek tabloları ve kolonları adıyla belirtmek zorunda. Statement primary'ye dokunmadan önce başka birinin ona bakması gerekiyor. Bu arada saat işliyor ve onarım o kadar küçük ki etrafındaki süreç işin kendisinden büyük görünüyor.

Bu bölüm hedef durumun bir adımını ekliyor: yazmalar, gözden geçirilmiş bir veri düzeltmesi prosedürü ve `human_jit_write` tavanı içindeki just-in-time grant'ler aracılığıyla yapılır. Bölüm 7 insanlara tek bir kapı verdi, Bölüm 8 o kapıyı sınırlı bir süre için açan grant'leri kurdu. Burada kapı açıldıktan sonra bir kişinin ne yapması gerektiğini soruyoruz: nasıl okumalı, nasıl yazmalı ve ikinci tür işi bir Perşembe öğleden sonrası sakın sakın yapılabilecek kadar nasıl güvenli kılmalı.

Bölüm 1'deki beş parçalı modelde bu bölüm **Yürütme** ile ilgilidir. Robin onarımın ihtiyaç duyduğu her credential'a zaten sahip, bu yüzden risk kimin credential tuttuğunda değil. Risk, grant'in neyi yetkilendirdiğinde, tek bir statement çalıştığında ne yapacağına, onu önce kimin okuduğunda ve geriye hangi kanıtın kaldığında. Bölümün sorusu şu: bir kişinin production verisini değiştirmesinin en küçük, en kolay gözden geçirilebilir yolu nedir ve değişiklik ne zaman tümüyle bir kişinin işi olmaktan çıkmalıdır?

Problem ve neden zor olduđu

Gerçek veri tutan her sistem eninde sonunda kendi kodunun onaramayacağı bir duruma varır. Bir worker iki adım arasında çöktü. Bir migration bir kolonu yarı dolu bıraktı. Bir müşteri yanlışlıkla bir başkasıyla birleştirildi. Uygulamada "bunu düzelt" için bir endpoint yok, çünkü kimse buna ihtiyaç duyulacağını beklemedi ve endpoint'i yazmak onarımdan uzun sürerdi. Bu yüzden bir kişi bir statement yazıp production'a karşı çalıştırır. **Veri düzeltilmesi** budur: uygulamanın onaramayacağı bir durumu onarmak için, normal uygulama yolunun dışında bir kişi tarafından production verisinde yapılan değişiklik.

İlk zorluk, tehlikeli aracın ve yararlı aracın aynı araç olmasıdır. "Kaç fatura takıldı?" sorusunu cevaplayabilen bir SQL session'ı, bir anahtar sözcük daha ekleyerek hepsini değiştirebilir. Bölüm 1'deki başlangıç durumunda GUI client'ı paylaşılan admin login'ini tutar, her statement'ı auto-commit eder ve vurgulanan metin neyse onu çalıştırır. Bir soru ile geri alınamaz bir değişiklik arasındaki mesafe tek bir sözcüktür.

İkinci zorluk, işi yapan kişinin genellikle baskı altında olması ve genellikle haklı olmasıdır. Her düzeltmeyi bir tehdit gibi ele alan bir süreç insanlara onu dolaşmayı öğretir ve dolaşmak asıl riskten kötüdür: kalıcı bir grant, paylaşılan bir parola, production'ın bir laptop'taki kopyası. Her birini rutin sayan bir süreç ise kimseye iki kez bakmayı öğretmez ve yüzde bir ihtimalle yanlış olan statement her satıra ulaşır. Güvenli yol kolay yol olmak zorundadır.

Üçüncü zorluk, risk biriminin kişi değil statement olması ve bir statement'ın kendi büyüklüğü hakkında neredeyse hiç bilgi taşımamasıdır. `UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` ile `WHERE` koşulu kesilmiş aynı satır birkaç karakterle ayrılır. Metinde "bu dokuz bin satıra dokunacak" diyen hiçbir şey yok; bunu yalnızca veri söyler ve süreç soruyu önce sormadıkça veri olaydan sonra okunur.

Dördüncüsü, cevabın tek başına bir politika olamamasıdır. "Update etmeden önce satırları say" bir kez okunur ve ilk yoğun güne kadar uygulanır; bu yüzden yolun içine inşa edilmelidir: sayı yanlışsa commit etmeyi reddeden bir runner ve başka türlü yazamayan bir session.

Ad-hoc sorguların daha sessiz bir problemi var. **Ad-hoc sorgu**, bir kişinin yazıp bir kez çalıştırdığı, herhangi bir uygulamanın dışındaki sorgudur. Read-only bir session'da hiçbir şeyi değiştiremez, ama çok şey okuyabilir, veritabanı için çalıştırması pahalı olabilir ve sonuçları bir ekran görüntüsünde, bir spreadsheet'te ya da bir chat mesajında sistemden çıkar.

Varsayılan: read-only bir session

Bir insan session'ı, access gateway üzerinden `ledgerly_replica_2`'ye bağlı, `human_readonly` üyesi bir login olarak başlar. Rolün kapsadığı tabloları okuyabilir; Restricted kolonlar saklanır ve Confidential kolonlar Bölüm 12'nin söylediği yerlerde maskelenir. Yazamaz ve kişinin yazdığı hiçbir şey bunu değiştirmez.

Araştırmaların çoğu bu session'ın içine sığar ve kişinin hiçbir şey istemesi ya da kimseyi beklemesi gerekmez. Bu varsayılanı yalnızca maliyet gerekçesiyle savunurdum: yaygın görevin onay gerektirmediği bir tasarımı, nadir görev için onay hâlâ bir anlam taşır.

Üç alışkanlık paylaşılan replica'yı ve sonuçlarını güvende tutar. **Sorguyu sınırlayın**: rol düzeyindeki statement timeout (Bölüm 5) ve gateway'in row cap'i (Bölüm 7) başıboş durumu durdurur ve önce sayı ve örnek seçmek kazara export'ları küçük tutar. **Soruyu cevaplayan en azını seçin**: `id`, `status`, `due_on`, `SELECT *`'dan iyidir; kısmen daha hızlı olduğu için, kısmen de soru gerektirmedikçe Confidential kolonları sonucun dışında tuttuğu için. **Sonuçları veri kopyası sayın**: bir ticket'a ya da spreadsheet'e yapıştırılan sonuç, farklı erişim kurallarına tabi bir production verisi kopyasıdır (Bölüm 13 bu tür kopyaları yan kapı olarak ele alır); bu yüzden kişisel içerik taşıyan satırlar değil, tanımlayıcıları ve sayıları yapıştırın ve onları üreten sorgunun adını verin.

Bir uyarı bir sonraki bölüm için önemlidir. Replica, primary'nin bir miktar gerisindedir; genellikle azdır, bazen değildir. Orada alınan bir sayı, primary'nin şu anda tuttuğunun tahminidir. Araştırma için bu iyidir. Bir yazmanın kaç satıra dokunacağına karar vermek için değildir; aşağıdaki veri düzeltmesi prosedürü kesin sayıyı primary üzerinde, yazmayı yapan transaction'ın içinde alır.

Yazmak farklı bir eylemdir

Bir okuma tekrarlanabilir, bırakılabilir ve görmezden gelinebilir. Bir yazma, o anda çalışan ve yeni duruma, kimse kontrol etmeden önce göre davranacak servisler dahil, diğer her principal'ın gördüğünü değiştirir. Yanlış güncellenmiş bir `status` bir hatırlatma e-postası başlatabilir; yanlış güncellenmiş bir `due_on`, bir müşteriye ne borçlu olduğunun söylenişini değiştirebilir. Kitabın yazmaya kendi yolunu, kendi onayını, kendi zaman sınırını ve kendi kaydını vermesinin nedeni budur.

Bu yol, Bölüm 8'deki just-in-time erişimdir ve burada önemli olan bir özelliği vardır: bir yazma isteği, gözden geçirilmiş bir düzeltmeye bağlıdır. İstek düzeltmenin `fix_id`'sini ve bir `approved_hash`'i taşır ve (yetki, tablo, kolonlar) üçlülere isteyen tarafından yazılmaz, düzeltmeden türetilir. Bir isteğin isteyebileceği en fazla şey, `human_jit_write`'in tanımladığı tavadır. Üç sonuç düzeltmenin nasıl yazılacağını şekillendirir.

Birincisi, **istek düzeltmeden kurulur**. `invoices.status`'u güncelleyen bir düzeltme tam olarak bir üçlü gerektirir: `status` kolonu için `invoices` üzerinde `UPDATE`. Bir sonraki bölümdeki guard yazmadan önce okur ve bu okuma `human_readonly`'nin zaten sahip olduğu `invoices` üzerindeki `düz SELECT`'i kullanır; dolayısıyla hiçbir `SELECT` üçlüsü gerekmez. İstisna, `customers` üzerindeki `billing_address` gibi bir Confidential kolonun snapshot'ını alan düzeltmedir: `human_readonly` altında kolon maskelidir ve maskeli değerlerin snapshot'ı rollback için işe yaramaz; bu yüzden istek onun için bir `SELECT` üçlüsü belirtir ve onaylayan bunu görür.

İkincisi, **Restricted bir kolon istekte hiç görünmez**. `iban`, `account_holder_name` ve `token_value` adıyla belirtilemez ve `bank_accounts` ile `payment_tokens` üzerinde `INSERT` ve `DELETE` mevcut değildir, çünkü bir satır eklemek ya da silmek Restricted değerleri belirler ya da yok eder. Tavanın içinde duran ve yine de olağan bir veri düzeltmesi olmayan bir değişiklik daha var: `bank_accounts` ya da `payment_tokens` üzerinde `customer_id`'nin `UPDATE`'i. Kolon Restricted değildir, ama bir satırı taşımak o ödeme hesabının ya da ödeme token'ının kime ait olduğunu değiştirir. Yürütme politikası bu owner-link değişikliğini gate'ler ve aşağıda anlatılan ürün yoluna yönlendirir (Bölüm 9). Bir düzeltme bunlardan herhangi birini gerektiriyorsa, o bir veri düzeltmesi değildir.

Üçüncüsü, **onaylayan yalnızca üçlüleri değil, statement'ı görür**. Bölüm 9 statement'a bağlı onayın bu kuralını ve gateway'in bunu nasıl uyguladığını belirtir. Bir veri düzeltmesi için onay ekranı, üçlülerin yanında statement'ı, filtreyi ve beklenen sayıyı gösterir ve tüm kolonlar için bir istek, farklı türden bir güven istemektir. Dar bir etki alanıyla yazılan düzeltmeler daha hızlı onaylanır; istenen teşvik budur.

Takılan faturaların düzeltmesi için istek kaydı, Bölüm 3'teki engine'den bağımsız notasyonla:

```
// request record for the stuck-invoice fix; values are illustrative
access_request:
  id          "req-0187"
  kind        "jit"
  requester   "robin"
  approver    "alex"
  fix_id      "FIX-0031"
  approved_hash "a41c...07" // set at approval, from the fix's
merge record
  reason      "reset invoices left in 'sending' by worker bug"
  expires_at  now() + 1 hour

access_request_privilege: // derived from FIX-0031, not typed
  ("req-0187", "UPDATE", "invoices", "status")
```

Kolon düzeyinde grant'i olmayan engine'lerde, Restricted kolonlar zaten tavanın dışındaki ayrı tablolarda durur (Bölüm 3). Ana tablodaki bir JIT isteği o zaman tablo düzeyinde bir grant'tir ve onaylayan, Confidential olanlar dahil, onun Restricted olmayan her kolonunu onaylamıştır. Grant artık statement'ın etki alanını sınırlamaz; bu yüzden review'un daha büyük bir kısmını statement'ın kendisi taşır.

Bir veri düzeltmesinin anatomisi

Veri düzeltmesi, sabit bir sıraya sahip kısa bir prosedürdür. Sıra, tek tek adımların hiçbirinden daha önemlidir, çünkü her adım bir sonrakinin doğru yapılmasını ucuzlatmak için vardır.

1. **Niyeti belirtin.** Tek bir cümle: hangi durum yanlış, hangi durum doğru ve hangi satırlar kapsamda. Tek cümleyle yazılamıyorsa, düzeltme muhtemelen iki düzeltmedir.
2. **Etkilenen satırları bir SELECT ile sayın.** "Kırk fatura" ile "dokuz bin" farklı konuşmalardır ve sayı, bir hatanın kendini ilk gösterdiği yerdir.

3. **Beklenen satır sayısını yazın.** Düzeltme, değiştirmeyi beklediği sayıyı beyan eder. Guard budur.
4. **Bir before snapshot alın.** Düzeltmenin değiştireceği kolonların tam olarak primary key'leri ve eski değerleri, session'dan uzun yaşayan bir yerde saklanır.
5. **Değişikliği bir transaction içinde çalıştırın.** Değişen satır sayısını beklenen sayıyla karşılaştırın. Farklıysa rollback yapın.
6. **Aynı transaction içinde bir postcondition kontrol edin.** Durumun artık niyetin söylediği olduğunu kanıtlayan bir sorgu; örneğin "listedeki hiçbir faturanın durumu hâlâ `sending` değil".
7. **Commit edin, sonra kaydedin.** Commit, diğer principal'ların değişikliği gördüğü noktadır. Transaction içinde okunan after durumu, before snapshot'ın yanında kayda girer.

Pseudocode guard'ı kişinin hafızasına değil, runner'a koyar. Veritabanını, snapshot store'u ve audit log'u ayrı sistemler olarak ele alır; dolayısıyla hiçbir transaction veritabanından fazlasını kapsamaz. Runner'a bir istek id'si verilir ve başka hiçbir şey verilmez; ilk fonksiyon, o isteğin hiç çalışıp çalışamayacağına karar verir.

```
function loadApprovedFix(request_id, session):
    request = loadRequest(request_id)           // the gateway's record,
not the runner's input
    if request.status != "approved" or request.expires_at <= now():
        raise RequestNotLive
    if request.requester_login != session.login: // granted to this
session's login only
        raise RequestNotForThisSession
    fix = loadFixScript(request.fix_id)
    // approved_hash was set at approval from the fix's merge record,
so
    // the script is compared with the record, not with itself
    if hashOf(fix.statements) != request.approved_hash:
        raise ScriptNotReviewed
    if request.privileges != fix.privileges:
        raise GrantsDoNotMatchFix
    return fix
```

İkinci fonksiyon onu çalıştırır ve yaptığı her şey ilkiyle sınırlıdır:

```

function runDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    // the authoritative count, on the primary, before anything is
    written
    before = readRows(fix.target_filter, fix.columns, using: session)
    if count(before) != fix.expected_rows:
        raise RowCountMismatch
    saveSnapshot(fix.id, before)
    transaction:
        // re-read under the lock: a changed value must fail, not pass
        locked = readRowsLocked(fix.target_filter, fix.columns, using:
session)
        if locked != before:
            rollback()
            raise RowsChangedSinceSnapshot
        // one source: the change runs on the same filter that was
        counted
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        if changed != fix.expected_rows or not
fix.postcondition(session):
            rollback()
            raise FixGuardFailed
        after = readRows(keysOf(before), fix.columns, using: session)
        // commit point: other sessions see the change only after this
        line
        commit()

    try:
        saveAfterState(fix.id, after)
        auditLogAppend({ actor: session.login, action: "data-fix",
                        request: request_id, fix: fix.id, rows:
changed })
    on AuditLogUnavailable or SnapshotStoreUnavailable:
        queueForRetry(fix.id)
        notify("platform", "fix committed, record pending: " + fix.id)

```

Bu bloklardaki birkaç ayrıntı bilinçlidir.

Runner hiçbir zaman kendi hash'ini sağlamaz. İsteğin adını verdiği düzeltmeyi yükler ve statement'larının hash'ini, istek kaydındaki `approved_hash` ile karşılaştırır; bunu Bölüm 9 merge kaydına kadar izler. Statement'ı yeniden yazmak, review ile yürütme arasında metni değiştirir ve review önceki metne iliştilmişti.

Snapshot, transaction'dan önce kaydedilir. Snapshot store'un bir arızası o zaman düzeltmeyi hiçbir şey değişmeden durdurur. Maliyet, satırların snapshot ile yazma arasında değişebilmesidir: bir worker o aralıkta bir faturayı `sending`'den `failed`'a taşıyabilir ve key yine de filtreye eşleşirdi.

Transaction satırları bir lock altında yeniden okur. Düzeltmenin değiştirdiği kolonlarla birlikte bütün sonucu snapshot ile karşılaştırır. Satır kümesindeki ya da bir değerdeki herhangi bir fark çalıştırmayı RowsChangedSinceSnapshot ile iptal eder ve yazar taze bir sayımla yeniden başlar. Satırları bir filtreyle lock edemeyen engine'lerde yeniden okuma yine transaction içinde çalışır; aralığı daraltır ve kapatmaz; geriye beklenen sayı ve postcondition kalır.

Kayıt commit'ten sonra yazılır. Snapshot store ve audit log başka sistemlerdir, bu yüzden after durumu commit gerçekleşikten sonra kaydedilir. Bu, Bölüm 8'deki JIT grant'ın tersi bir sıradır; orada bir grant log girdisi olmadan var olamaz.

Sıra burada savunulabilir, çünkü Ledgerly'nin gateway'i veri yolunun üzerindedir (aşağıda): runner'ın statement'larını iletir ve session'ı kaydeder; böylece audit append başarısız olsa bile commit edilmiş bir değişiklik kayıtlıdır. Girdi yeniden denenir ve platform ekibine haber verilir. Yalnızca credential'lara aracılık eden bir gateway'de statement'ların session log'u olmazdı ve audit girdisinin önce gelmesi gerekirdi.

Bir dry-run, sayımın gösteremediğini gösterir. Sayım bir SELECT'tir; bu yüzden statement'ın harekete geçirdiği şeyi kaçırr: başka yere yazan bir trigger, bir cascade, bir rule. Gerçek bir dry-run, değişikliği bir transaction içinde çalıştırır ve rollback yapar:

```
function dryRunDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    if count(readRows(fix.target_filter, fix.columns, using: session))
    != fix.expected_rows:
        raise RowCountMismatch           // a wrong filter never reaches
the locks
    transaction:
        countsBefore = countsIn(fix.watch_tables, using: session)
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        report = { rows: changed,
                    sample: sampleAfterValues(fix, using: session),
                    related: diff(countsBefore,
countsIn(fix.watch_tables, using: session)) }
        rollback()           // nothing is kept; the row locks are held
until here
    return report
```

Rapor, satır sayısını, after değerlerinden bir örneği ve düzeltme header'ındaki `watch_tables` içindeki satır sayılarının değişimini içerir. Onaylanmış bir istek ve yazma grant'leri gerektirir; bu yüzden pull request review edilirken var olamaz; onaylı pencere içinde çalışır, çalışırken aynı satır lock'larını tutar ve bir e-posta gibi veritabanı dışındaki etkileri göremez. Bir sürprizin maliyetinin yüksek olduğu yerde yazar onu pencerenin ilk dakikalarında çalıştırır, raporu isteğe ekler ve onaylayan, bir duraklama pahasına, commit adımından önce okur.

Bir guard ancak session başka türlü yazamıyorsa bir mekanizmadır. Ledgerly'nin gateway'i veri yolunun üzerindedir (Bölüm 7) ve bir JIT write session'ında yazmaları yalnızca runner'ın bağlantısından iletir; bu yüzden hash kontrolü, sayım ve snapshot atlanamaz. Transaction runner'ın kalır: gateway her statement'ı ona iletir ve satırları bütün düzeltme boyunca sayar, ama runner adına hiçbir zaman commit etmez; bir postcondition'ın başarısız olması her statement'ı bu yüzden rollback eder. Gateway'in her statement'ı merge kaydıyla nasıl kontrol ettiğini ve incelemesinin nerede bittiğini Bölüm 9 anlatır. Bedel, her JIT write'ın runner'ın ve gateway'in ayakta olmasına bağlı olmasıdır. Yalnızca credential'a aracılık eden bir gateway'de runner bir gelenekten ibarettir; `WHERE` koşulsuz bir statement'a karşı koruma değildir.

Snapshot nereye gider

Before snapshot bir veri kopyasıdır ve her kopya gibi bir sahibe ve bir süre dolumuna ihtiyaç duyar. `invoices` üzerindeki `status` değerlerinin snapshot'ı Internal veridir. `customers` üzerindeki `billing_address` gibi bir Confidential kolonun snapshot'ı Confidential'dır ve store'un erişim kuralları en az kaynak tablo kadar sıkı olmalıdır.

Snapshot'ları düzeltmenin kaydıyla birlikte, platform ekibinin belirlediği bir saklama süresiyle tutardım: "bu düzeltme doğru muydu?" sorusu cevaplanabilecek kadar uzun, snapshot'ın müşteri verisinin ikinci, yönetilmeyen bir kopyasına dönüşmeyeceği kadar kısa. Fatura durumları için bu günlerdir, bir billing address için belki bir tam fatura döngüsü. Bir sayı seçin, yazın ve süresinin dolmasına izin verin.

Rollback planı

Bir before snapshot, birisi onun nasıl kullanılacağını yazana kadar bir rollback planı değildir. Plan, düzeltmenin header'ında bir ya da iki cümledir; örneğin "status'u aynı key'ler için snapshot'tan geri yükle" ve snapshot'ı mekanik olarak bir düzeltme betiğine çevirmek mümkün olmalıdır.

Rollback'in kendisi bir veri düzeltmesidir. Aynı istekten ve review'dan geçer, çünkü diğer principal'lar bu arada yeni duruma göre davranmış olabilir ve eski değerleri geri yüklemek meşru güncellemelerin üzerine yazabilir. Bu yüzden bir rollback runner'ı, mevcut değerlerin hâlâ orijinal düzeltmenin kaydettiği after durumuyla eşleştirdiğini kontrol eder ve o zamandan beri değişmiş satırlara dokunmayı reddeder. "Bu satırlar başka bir duruma geçti, ayrı ele alın" diyemeyen bir ters düzeltme, bir sonraki onarımı doğurur.

Bazı değişikliklerin rollback'i yoktur ve header bunu söylemelidir. Bir e-posta, bir webhook ya da bir ödeme sağlayıcısına yapılan çağrı, bir kolonu geri yükleyerek geri alınamaz. Bunlar için konuşma düzeltme çalışmadan önce yapılır: bu durum değişikliği hangi etkileri tetikliyor ve tepki veren servisler önce durdurulmalı mı?

İki tablonun olağan anlamda rollback'i yoktur, çünkü hiçbir insan rolü onları düzenleyemez. ledger_entries ve audit_events append-only'dir. Ledger'a yapılan bir düzeltme yeni bir girdidir; yalnızca payments-service'in düzeltme yolu üzerinden verilir, hiçbir zaman bir insan login'ile verilmez; dolayısıyla orada bir değişiklik gerektirdiği görünen bir düzeltme, bir grant için değil o yol için bir istektir. Her iki tabloda her UPDATE ve DELETE'i reddeden bir trigger, bir grant yanlılıkla genişletilse bile kuralın geçerli kalmasını sağlar. Bir maliyeti var: ledger_ly_owner trigger'ı yine de drop edebilir; bu yüzden gerçek review noktası bu tablolara dokunan migration'larda kalır (Bölüm 11).

Bir DELETE, rollback'i daha zor olan aynı prosedürdür

Yukarıdaki anatomi UPDATE için yazılmıştır. invoice-worker'daki bir retry'nin invoice_lines içine mükerrer satırlar eklediğini ve onarımın onları sildiğini varsayın. invoice_lines üzerindeki DELETE, human_jit_write tavanının içindedir ve üç bakımdan farklıdır.

Grant satırları değil tabloyu sınırlar, çünkü bir `DELETE` üçlüsünün kolonu yoktur; bu yüzden sınır filtreden, beklenen sayıdan ve review'dan gelir ve filtre bir key listesi olmalıdır. Snapshot, satırın tamamıdır; çünkü sonradan değerleri başka hiçbir şey tutmayacaktır ve Confidential kolonları olan bir tablo snapshot'ı tablo kadar hassas yapar. Rollback bir yeniden eklemedir; bu da kendi isteği olan başka bir veri düzeltmesidir. Bir `INSERT` üçlüsü kolonlarını adıyla belirtir, dolayısıyla yeniden ekleme isteği `id` ve `created_at`'in bunların arasında olup olmadığına karar verir. Değillerse, geri yüklenen satır her ikisi için yeni değerler alır ve eski tanımlayıcılara referans veren her şey geri yüklenmez. Ellerse, onaylayan bir key ile bir zaman damgasının yazıldığını görür; bunu hiçbir olağan düzeltme yapmaz ve engine'in sağlanan bir key'le ne yaptığı, key'leri nasıl ürettiğine bağlıdır. Header, rollback'in hangi seçimi yaptığını belirtir.

```
fix:
  id          "FIX-0044"
  intent      "Remove the duplicate invoice_lines rows created
by a retry.
               No other row is deleted."
  target_filter "id IN (<ids of the duplicates>)"
  change       "DELETE"
  expected_rows 12                      // illustrative
  snapshot      "whole rows"
  privileges    ("DELETE", "invoice_lines") // no columns for
DELETE
  rollback      "re-insert from the snapshot as a new fix; its
INSERT triple
               leaves out id and created_at, so the rows get new
values and
               references to the old ids are lost"
  effects       "invoice totals are recomputed by the worker on
next run"
  watch_tables  ("invoices")
```

Tanımlayıcıların kaybedilmesinin kabul edilemez olduğu yerde, reviewer rollback'ten daha ucuz bir koruma ister: başka bir satır hâlâ o kalemi işaret ederken silmeyi reddeden bir foreign key ya da satırı kaldırmayan farklı bir onarım.

Review: four-eyes, bir repository'de

Veri düzeltmesi bir kez çalışan koddur; bu yüzden mühendislerin zaten güvendiği kontrolü alır: bir pull request. Düzeltme, bir header'ı, bir reviewer'ı ve bir geçmişi olan, bir repository'deki bir betiktir ve geçmiş, altı ay sonra bir durumu kimin ve neden değiştirdiğini cevaplar.

```
// data-fixes/2026-10-06-FIX-0031-reset-sending-invoices.fix
fix:
  id          "FIX-0031"
  intent      "Invoices stuck in 'sending' after the worker bug
              are set to 'sent'. No other column changes."
  target_filter "status = 'sending' AND updated_at < '<cutoff>'"
  change      "SET status = 'sent'" // applied to
target_filter only
  expected_rows 40
  postcondition "no invoice matches target_filter"
  privileges    ("UPDATE", "invoices", "status")
  rollback      "restore status from snapshot for the same ids,
              only where status is still 'sent'"
  effects       "reminders resume for these invoices"
  watch_tables  ("invoices") // read by the dry run
  author       "robin"
  reviewer      "alex"
```

`privileges` alanı repository'yi Bölüm 8'e bağlar. İsteğin üçlüləri ondan kopyalanır; böylece onaylayan isteğe ve düzeltmeye yan yana bakar ve `loadApprovedFix`, üçlüləri düzeltmeden farklı olan bir isteği reddeder. Yetkilerinin adını vermediği bir kolona dokunan statement çalışma zamanında başarısız olur; bu, amaçlanan sonuçtur: grant, review'un arkasındaki ikinci savunma hattıdır.

Review edenin okudukları

Review edenin işi "bu iyi bir fikir mi?" sorusundan daha dardır.

- **Statement niyetle eşleşiyor mu?** Tablolarda, kolonlarda ve filtrede uyuşmalıdırlar.
- **Filtre spesifik mi?** Key'e göre seçen filtre en güvenlisidir; duruma ve zaman kesim noktasına göre seçen filtre sayı uyuşuyorsa kabul edilebilir; hiç kısıtlaması olmayan filtre görüldüğü anda reddedilir.
- **Beklenen sayı makul mü ve nereden geldi?** `expected_rows: 40` gören bir reviewer, pull request'te sayım sorgusunun çıktısını

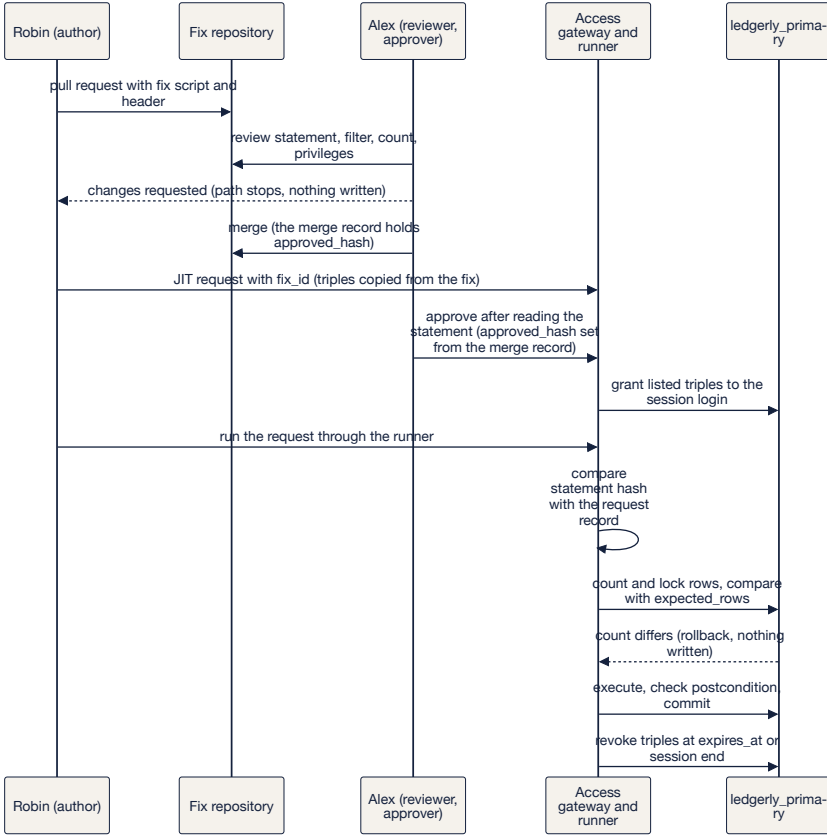
isteyebilir. Kaynağı gösterilemeyen bir sayı, kimsenin kontrol etmediği bir sayıdır.

- **Yetkiler yalnızca statement'ın ihtiyaç duyduğunu mu belirtiyor?** Satır satır karşılaştırın.
- **Düzeltilme bir Restricted kolona, bir append-only tabloya ya da tavanın dışındaki bir tabloya dokunuyor mu?** Dokunuyorsa bu bir veri düzeltmesi değildir ve review, doğru yolu gösteren bir işaretle biter.
- **Bu değişiklik neyi tetikliyor?** Hatırlatmalar, e-postalar, kolonu okuyan job'lar. Bu soru sessiz durumları yakalar.
- **Önizleme niyetle uyuyor mu?** Pull request, sayım sorgusunun çıktısını ve satırların ile after değerlerinin bir SELECT önizlemesini içerir; dry-run raporu isteğe daha sonra eklenir ve bir trigger ya da cascade yalnızca orada görünür.
- **Rollback cümlesi kullanılabilir mi?** Bir DELETE için, yeniden eklemenin neyi geri yükleyemeyeceğini söyler.

Review, kitabın anlamında four-eyes'tır: riskli bir eylem çalışmadan önce ikinci bir kişi review eder. Merge metni sabitler: review'dan sonra betikte yapılan değişiklik farklı bir hash üretir ve runner onu reddeder. JIT isteğinin onaylayanı ile düzeltmenin reviewer'ı iki ayrı eylemdir. Büyük bir ekipte iki kişiyi tercih ederdim; böylece onaylayan erişimi yeniden okumak yerine düzeltmeye karşı kontrol eder; küçük bir ekipte çoğunlukla aynı kişi olurlar, hiçbiri yazar değilse bu kabul edilebilir. Yazar hiçbir zaman kendi düzeltmesini ya da kendi grant'ini onaylamaz.

Sıra

Diyagram, bir istekten commit edilmiş bir değişikliğe giden yolu gösterir. Yolun hiçbir şey yazılmadan durabileceği yerlere bakın: review, hash karşılaştırması ve runner'ın guard'ları.



Değişiklik Restricted bir kolona olduğunda

Tavan, herhangi bir JIT isteğinde Restricted kolonları yasaklar (gerekçesini Bölüm 3 verir); bu yüzden bütün bir onarım sınıfının SQL yolu yoktur. `bank_accounts` üzerinde bir müşterinin `account_holder_name`'inin yazım hatasıyla girildiğini düşünün. Bir kişi bunu bir JIT session'ında düzeltemez, çünkü kolon istekte adıyla belirtilemez. Break-glass session da yapamaz: süreç ne derse desin, `human_breakglass` Restricted kolonlarda hiçbir yazma yetkisine sahip değildir.

Yol uygulamadır. `admin-console`, `bank_accounts`'un tek yazarı olan `payments-service`'ten değişikliği yapmasını ister; `admin-console`, `payments-service` bunu gerçekleştirmeden önce ikinci bir kişinin onaylamasını şart koşar ve değişiklik `audit_events` içine kaydedilir. Bu bir veri düzeltmesi değil, bir ürün yoludur: bir formu, bir doğrulaması, bir neden alanı, bir onaylayanı ve bir kaydı vardır ve bir insanın değil, servisin kendi rolü olarak çalışır.

Bu, bir SQL statement'ından daha pahalıdır ve öyle olması gerektiğini düşünüyorum. Bu değerler, maruz kalmaları müşterilere geri alınması zor biçimlerde zarar veren değerlerdir ve uygulama, doğrulamanın ve yan etkilerin tamamının yaşadığı tek yerdir. Maliyet, değişikliğe ihtiyacı olan kişi için dakikalardır; kazanç, elle yazılmış bir onarımın artık hatalı biçimli bir banka hesap numarası sokamamasıdır. `payment_tokens` içindeki yanlış bir token da benzer şekilde yamalanmaz, `payments-service` tarafından sağlayıcının kendi akışı üzerinden değiştirilir.

Break-glass bir veri düzeltmesi yolu değildir

`human_breakglass`, JIT tavanının ulaşmadığı yerde yazar ve Restricted kolonları okuyan tek insan rolüdür (kesin yetkileri Bölüm 3'tedir). Kâğıt üzerinde, JIT yolunun reddettiği bir onarımı yapmanın elverişli bir yolu gibi görünür. Değildir ve nedeni rolün sahip olmadıklarıdır.

Break-glass kendi kendine beyan edilir. Onaylayanı, `fix_id`'si ve `runner`'ı yoktur ve güvenceleri Bölüm 8'in tanımladığı logging set'i ile zorunlu review'dur. Süren zararı durduran, sonradan review edilen sınırlayıcı bir eylem, rolün var olma amacıdır (olay tarafını Bölüm 15 kapsar). Verinin kalıcı onarımı farklı bir eylemdir ve her zaman bir veri düzeltmesidir: bir break-glass session'ında çalıştırıldığında, statement çalışmadan önce onu okuyan tek kontrolü atlar ve bunu, satır guard'ı olmayan bir `UPDATE`'in mümkün olduğu bir rolde yapar. Bölüm 8 bunu kabul edilmiş bir risk olarak belirtir ve onarımları role yönlendirmek onu olağan hale getirirdi.

İkinci neden teşviklerdir. Break-glass bir onarım yapabilirse hızlı lane haline gelir ve alarmı anlamını yitirir. Break-glass "gerektiren" bir veri düzeltmesi başka bir şeyin işaretidir: gece onay döngüsü çok yavaştır, tavanda düzeltmenin meşru olarak ihtiyaç duyduğu bir kolon yoktur ya da düzeltme uygulama yoluna aittir. Bir onarım bekleyemiyorsa, zararı mevcut en küçük eylemle durdurun ve baskı kalktıktan sonra veri düzeltmesini normal yoldan çalıştırın.

Tekrarlanan bir düzeltmeyi ürün özelliğine çevirmek

Veri düzeltmesi bir semptomdur: uygulamanın geri alamayacağı bir duruma ulaşılabilir. İlk olay bir düzeltmedir; üçüncüsü kimsenin açmadığı bir özellik isteğidir. Düzeltme betiklerinin bir dizinini onarımın türüne

göre etiketleyerek tutardım ve aynı etiket yeniden görüldüğünde birisi bunun `admin-console`'a ait olup olmadığını sorar. "Üç kez"de kesin bir şey yok: yılda iki kez on satırlık bir onarım hiçbir zaman bir ekranı haklı çıkarmayabilir; oysa her hafta çalışan, her seferinde review gerektiren ve destek tarafından bir dakikada yapılabilecek bir onarım yazılmayı bekleyen bir özelliktir.

Baştaki takılan faturaları ele alın ve durumun yinelandığını varsayın. Bölüm 3'ün grant tablosu `app_admin_console`'un `invoices` üzerinde neler yapabileceğini zaten gösteriyor: yalnızca `status` üzerinde `SELECT` ve `UPDATE`. Ürün sürümü, fatura sayfasında, dar bir grubun kullanabildiği, tek bir faturanın durumunu sıfırlayan, bir neden isteyen ve eylemi insan principal'ın kimliğiyle `audit_events`'e yazan bir eylemdir. Kendi kontrollerini taşır: yalnızca takılı durumdaki faturalar, aynı anda yalnızca bir tane ve bir kişinin bir saatte sıfırlayabileceği sayıya bir limit.

Dönüşüm riski, her onarımda bir istek, bir review ve bir runner bedeli ödeyen statement'tan, bir kez review edilen, test edilen ve versiyonlanan koda taşır. Destek eylemi bir veritabanı session'ı olmadan kullanabilir; bu yüzden özelliğe dönüşen her onarım, artık insan erişimi gerektirmeyen yinelenen bir yazmadır.

```
function resetStuckInvoice(actor, invoice_id, reason):
  if not actor.canResetInvoices:
    raise NotAuthorized
  if reason is empty:
    raise ReasonRequired
  if countResetsBy(actor, within: "1 hour") >= resetLimit:
    raise RateLimited

  transaction:
    invoice = readInvoiceLocked(invoice_id)
    if invoice.status != "sending" or invoice.updated_at >
      cutoffForStuck():
        rollback()
        raise NotStuck
    setInvoiceStatus(invoice_id, "sent")
    // commit point: the status change and the business record
  agree
    appendAuditEvent({ actor_principal: actor.id, action: "reset-
      stuck-invoice",
                      subject_id: invoice_id, reason: reason })
    commit()
```

Audit event ve status değişikliği ikisi de `ledgerly_primary` içindedir, bu yüzden tek bir transaction ikisini kapsar; `audit_events` uygulamanın iş kaydıdır, Bölüm 14'teki audit log değildir. Yukarıdaki runner'da audit log ayrı bir sistemdir ve güvenliği işlemlerin sırası taşır.

Her onarım bir özellik olmamalı. Herkesin basabileceği bir düşmanın risk profili, iki kişinin review ettiği bir betikten farklıdır; bu yüzden para, token ya da banka bilgisini değiştiren onarımlar sık olsalar bile four-eyes yolunda kalır. Bir onarım, ancak en kötü durumu eylemin tasarımıyla sınırlıysa self-servis bir eylem olur; aksi halde ürünün içinde bir onayı olan bir istek olur. Test, onarımın tekrarlanıp tekrarlanmayacağı ve şu anda onu yapabilen tek kişinin bir insan olup olmadığıdır. İkisinin de cevabı evetse sprint bir maliyet değil, bir yatırımdır.

Öngörebildiğiniz onarımlar için runbook'lar

Bazı düzeltmeler ayrıntılar değişse de türce tekrarlanır: bir faturayı doğru müşteriye yeniden atamak, başarısız bir export'u yeniden oynatmak, mükerrer bir satırı onarmak. Bunlar için doğru artifact bir **runbook**'tur: tekrarlanabilir bir görev için yazılı, izlenen bir prosedür. Parametreleri boş bırakılmış bir düzeltme betiğiyle aynı iskelete sahiptir. Onu tetikleyen semptomu, onu doğrulayan tanı sorgularını, düzeltmenin şablonunu ve ihtiyaç duyduğu yetkileri, beklenen sayının kaynağını, kontrol edilecek etkileri, rollback'i ve kimin yazıp review edebileceğini belirtir. Ayrıca ne zaman durup yükseltileceğini de söyler: "tanı sayısı bu paragraftaki sayıyı aşıyorsa, bu bilinen problem değildir".

Bir runbook ayrıca bir ön onay taşıyabilir; böylece öngörülebilir bir onarım gece bir onaylayana beklemez. Beş koşul buna karar verir ve bunları kişi değil runner kontrol eder:

- filtre, bir durum ve bir kesim noktasına göre değil, key'e göre, bir id listesiyle seçer;
- beklenen satır sayısı en fazla N'dir; runbook'un kendi onarımı için belirlediği bir sayı (takılan faturalar için diyelim 20; örnek amaçlı);
- üçlüler tam olarak runbook'un sabit kümesidir;
- onarım, runbook'un `effects` alanının kaydettiği gibi, dışarıya giden hiçbir etki tetiklemez;

- statement, yalnızca key listesi yerleştirilmiş runbook şablonudur ve şablonun hash'i runbook review edilirken onaylanmıştır.

Runbook'un merge kaydı bu şablon hash'ini saklar. Key listesi, isteğin taşıdığı tek parçadır; runner onu saklanan şablona yerleştirir ve sonucu isteğin üçlüleryle kontrol eder; kendi çağırından hiçbir zaman bir hash ya da statement almaz (yerleştirilmiş statement'ı gateway'in nasıl kontrol ettiğini Bölüm 9 kapsar). Koşullardan herhangi biri başarısız olursa istek, bir onaylayanla tam review'a döner.

Maliyet, ön onaylı bir çalıştırmanın o gece kimse key listesini okumadan yürütülmesidir; bu yüzden runbook'un satır limiti yanlış bir listeyi sınırlar ve runner dışında biri kaydı bir sonraki iş gününde okur. Bir runbook ayrıca eskir: şekli o zamandan beri değişmiş bir tablo için olan bir runbook, eskimiş bir statement'a review görünümü verir. Her runbook'u bir tarihe ve bir sahıbe bağlayın ve başarısız bir çalıştırmayı, atlanacak bir rahatsızlık değil, onu gözden geçirme çağrısı sayın.

WHERE'siz bir UPDATE, suçlamadan analiz edilmiş

Ledgerly'yi başlangıç durumunda düşünün. Bir mühendis, paylaşılan admin parolasını hatırlayan ve auto-commit yapan bir masaüstü client'la bağlanıyor. Görev takılan faturalar. Statement editörde iki satıra yazılmış: ilk satırda UPDATE, ikincisinde WHERE. Client vurgulanan seçimi çalıştırıyor ve seçim yalnızca ilk satırı kapsıyor. Statement çalışıyor ve invoices içindeki her satırın, diyelim dokuz bininin, durumu sent oluyor.

Bu hikâyenin bir sürümü, birine daha dikkatli olması söylenerek biter. Bu hiçbir şey öğretmez, çünkü aynı koltuktaki bir sonraki mühendis, bir soru ile geri alınamaz bir değişiklik arasında aynı mesafeyle karşılaşır. Yararlı soru, tasarımdaki neyin tek bir statement'ın her satıra ulaşmasına izin verdiğidir.

- **Login her yere yazabiliyordu.** Paylaşılan admin rolü her tablonun sahibidir; bu yüzden bir hata, görevin ihtiyaç duyduğu tabloyla ya da kolonla sınırlı değildi.
- **Client kendi kendine commit etti.** Statement kesinleşmeden önce kimse "dokuz bin satır etkilendi"yi görmedi.

- **Sayı hiç sorulmadı.** Hiçbir şey kişinin kaç satır beklemesi gerektiğini bilmesini şart koşmadı; bu yüzden olaydan sonra yazdırılan sayının bir anlamı yoktu.
- **Statement'ı başka kimse okumamıştı.** Tek okuyucu, niyeti metne geri okuyan yazardı.
- **Before snapshot yoktu.** Eski durumlar gitmişti ve onarımın onarımının bir başlangıç noktası yoktu.
- **Log "paylaşılan rol" diyordu.** "Bunu kim çalıştırdı?" sorusunun cevabı yalnızca chat geçmişi üzerinden vardı.

Kurtarma bir backup'tan başlar, o zamandan beri olanları mutabık hale getirir ve aşağıdaki herkesi bilgilendirir (Bölüm 13 ve Bölüm 15 bu mekanikleri ele alır). Tasarım, maliyetini statement yazılmadan önce belirlemişti.

Şimdi aynı öğleden sonrayı hedef durumdan geçirin. Robin düzeltmeyi header'ı ve beklenen 40 sayısıyla yazıyor ve Alex review ediyor. JIT isteği `invoices` üzerinde `UPDATE`'i, `status` kolonunu belirtiyor ve grant bir saat sürüyor. Filtre review edilen betikten düşürülmüş olsaydı, runner beklenen 40'a karşı dokuz bin sayar ve hiçbir statement çalışmadan reddederdi. Kişinin kendi client'ına yazılan bir statement yazma olarak iletilmezdi (Bölüm 9'daki gateway kuralı) ve düz bir `human_readonly` session'ında `UPDATE` hiç çalışmazdı.

Guard bir düzeltmeyi doğru yapmaz. Bir hata sınıfını gürültülü yapar: statement, yazarın söylediğinden farklı sayıda satıra dokundu. Tam olarak 40 satıra dokunan ama yanlış 40'a dokunan bir statement geçer; review, postcondition ve snapshot'ın ayrı katmanlar olarak var olmasının ve Bölüm 9'un analizin neyi göremediğini söylemesinin nedeni budur. Guard ayrıca yalnızca sayısı kadar iyidir. Çalıştırmanın kendi çıktısından kopyalanan beklenen sayı, runner'a kendisiyle aynı fikirde olmayı öğretir; bu yüzden sayı, reviewer'ın bağımsız olarak çalıştırabileceği bir sorgudan, ideal olarak statement yazılmadan önce gelmelidir. Engine'in kolon düzeyinde grant'leri olduğu yerde (Bölüm 3), dar bir grant guard'ın kaçırdığı durumların neye ulaşabileceğini sınırlar.

Trade-off'lar ve maliyetler

Prosedür, bir statement yazmaktan yavaştır. Bir istek, bir review, bir onay ve bir runner iyi bir günde dakikalar ekler ve onaylayan uykudayken çok daha fazlasını. Bir Perşembe öğleden sonrası bu kabul edilebilir. Geceleyin, bir alarm altında, insanları break-glass'a doğru yönelten maliyettir ve tasarım dikkatini hak eder: rotation'daki bir reviewer, nitelikli onarımlar için yukarıdaki runbook ön onayı ve ekibin bir gece onayı için bekleyeceği en uzun sürenin, ihtiyaç duyulan ilk geceden önce yazılmış belirtilmiş bir limiti.

Review bir reflekse dönüşebilir. Saniyeler içinde verilen bir onay, maruziyeti azaltmadan sürtünme ekler; Bölüm 8 tam argümanı yapar. Bu bölüme özgü sinyal pull request'tir: reviewer'lar sayım sorgusunun çıktısını hiç istemiyor ve bir statement'ta hiç değişiklik talep etmiyorsa, hiçbir şey okumuyor ya da yalnızca kusursuz düzeltmeleri review ediyorlardır. Savunmalar, bir artifact gerektiren kontrol listesi maddeleridir: pull request'teki sayım çıktısı ve önizleme, istekteki dry-run raporu ve statement'ın yetkilerle açık bir karşılaştırması.

Transaction'lar lock tutar. Runner transaction'ı review'dan sonra başlatır ve saniyeler içinde bitirir. Onu "güvenli olsun diye" önce açan ve kararı sonradan insana bırakan bir prosedür diğer yazarları bekletir, pooler'ın bağlantılarını doldurur ve Bölüm 5'in timeout'larının ilgisiz işte tetiklenmesine yol açar. İnsan kararını transaction'ın dışında tutun.

Büyük düzeltmeler batch gerektirir. Milyonlarca satıra dokunan bir düzeltme, uzun süre tutulan lock'lar ve büyük bir transaction log olmadan tek bir transaction olarak çalışamaz. Olağan tepki, onu her biri kendi sayımı ve guard'ı olan, birkaç bin satırlık (örnek bir boyut, öneri değil) batch'lere bölmektir; runner'ın aynı kontrolleri tekrarlayan bir batch modu vardır. Ortada bir başarısızlık veriyi karışık bir durumda bırakır; bu yüzden düzeltme yeniden çalıştırılabilecek biçimde yazılmalıdır: filtre yalnızca hâlâ eski durumda olan satırları seçer. Büyük bir onarım yine bir veri düzeltmesidir; yalnızca yeni bir yapının ilk doldurulması bir migration'a aittir (Bölüm 11).

Snapshot'ların ve düzeltme repository'sinin sahibi olmalıdır. Süre dolumu olmayan bir snapshot bir riski başka bir riskle takas etmiştir ve Confidential kolonlar için ikincisi daha büyük olabilir. Düzeltme betikleri

dizini production'ın nasıl savrulduğunu kaydır ve etikete göre bir index olmadan kimse bir problemin önceki örneğini aramaz.

Hepsini ne zaman kullanmamalı. Hassas hiçbir şey tutmayan bir veritabanına sahip küçük bir ekipte, her durum değişikliği için bir pull request, riskin haklı çıkardığından fazla törendir; önce saymak ve bir transaction kullanmak yine de kendini öder. Production verisi olmayan bir geliştirme ya da staging veritabanında hiçbir geçerli değildir ve `ledgerly_analytics` gibi her gece yeniden kurulan bir kopya için yanlış bir yazmanın maliyeti bir yeniden kurulumdur. Prosedür, yanlışmanın maliyetiyle ölçeklenir ve primary, bu maliyetin en yüksek olduğu yerdir.

Sık yapılan hatalar

- **WHERE koşulunu guard saymak.** Cazip, çünkü koşul yazarın niyetinin yaşadığı yerdir. Ama bir koşul metindir ve metin kesilebilir, yanlış yazılabilir ya da kısmen çalıştırılabilir. Çare, değişen sayıyı beyan edilmiş bir beklentiyle karşılaştıran ve uyumsuzlukta rollback yapan bir guard'dır.
- **Replica'da sayıp primary'de update etmek.** Replica geride kalır, bu yüzden iki sayı haklı bir nedenle farklı olabilir. Çare, kesin sayıyı primary üzerinde, transaction içinde almak ve replica sayısını yalnızca keşif için kullanmaktır.
- **Review edilen statement'ı yeniden yazmak.** Review eden bir dosyayı onayladı, bir kişi onu bir editöre kopyalayıp "madem buradayım" diyerek bir şeyi değiştirdi. Çare, betiğin hash'ini istek kaydındaki `approved_hash` ile karşılaştıran bir runner ve session'ın yazmalarını yalnızca o runner'dan ileten bir gateway'dir. Yalnızca credential'lara aracılık eden bir gateway'de guard bir gelenektir ve review onu taşımak zorundadır.
- **"Güvenli olsun diye" geniş bir grant istemek.** Onayı, bir statement'a karşı yapılan kontrol yerine bir güven yargısına çevirir ve bir hatanın neye ulaşabileceğini genişletir. Çare, statement'ın ihtiyaç duyduğu üçlülere istemek ve durum değişirse ikinci bir istek yapmaktır.
- **Onaylayan yavaş diye break-glass kullanmak.** Yol daha kısadır ve düzeltme acildir. Çare, yavaş gece onayını çözmek ve break-glass'ı tanımlandığı acil durumlar için saklamaktır.

- **Restricted bir kolona "özel" bir session ile yazmak.** Bir `iban`'ı doğrudan güncellemek müşteriye bir iyilik gibi gelir ve doğrulamayı, onayı ve uygulamanın kaydını tek adımda ortadan kaldırır. Çare, daha yavaş olsa bile uygulama yoludur.
- **Kimsenin yönetmediği bir snapshot kaydetmek.** Snapshot kötü bir düzeltmeye karşı korur ve paylaşılan bir klasörde Confidential verinin yönetilmeyen bir kopyasına dönüşür. Çare bir sahip, kaynak tabloyla eşleşen bir erişim kuralı ve bir süre dolumu tarihidir.
- **Düzeltilme küçük diye rollback cümlesini atlamak.** Aceleyle geri alınanlar küçük düzeltmelerdir. Çare, çalıştırmadan önce yazılmış, snapshot'ın neyi geri yüklediğini ve nerede yüklememesi gerektiğini söyleyen tek satırlık bir plandır.
- **Tekrarlanan bir düzeltmeyi sonsuza dek betik olarak bırakmak.** Sonuç, her seferinde primary'de bir insan gerektiren yinelenen bir yazmadır. Çare, düzeltme dizinindeki etiketlere periyodik bir bakış ve karar vermektir; karar "henüz değil" olsa bile.
- **Niyetin doğruluğu için review etmek, statement'ın doğruluğu için değil.** Onarımın yapmaya değip değmediğini tartışan bir reviewer statement'ı hiç okumayabilir. Çare, dar soruları sırayla soran bir kontrol listesidir.

Temel çıkarımlar

- Read-only bir session araştırma için doğru varsayılandır ve yalnızca yaygın görev onay gerektirmiyorsa işe yarar.
- Yazmak, okumaktan farklı bir eylemdir; bu yüzden kendi isteğinden, review'undan, zaman sınırından ve kaydından geçer. İstek, merge kaydından alınmış bir `fix_id` ve bir `approved_hash` taşır, üçlüleri düzeltmeden türetilir ve onaylayan statement'ı görür.
- Bir veri düzeltmesinin güvenli sırası sayım, snapshot, transaction, satır sayısı guard'ı, postcondition, commit'tir ve bunu kişinin hafızası değil runner uygulamalıdır; gateway session'ın yazmalarını yalnızca runner'dan iletir.
- Guard bir hata sınıfını gürültülü yapar ve bir düzeltmeyi doğru yapmaz; review, dar grant'ler ve before snapshot'ın ayrı katmanlar olarak var olmasının nedeni budur.

- Restricted bir kolondaki değişiklik bir veri düzeltmesi değildir: ikinci bir onaylayan ve `audit_events` içinde bir kayıtlı `admin-console` üzerinden `payments-service`'e gider ve break-glass bunun etrafından dolaşan bir yol değildir.
- Tekrarlanan bir elle düzeltme, eksik bir ürün özelliğidir ve `admin-console` eylemine dönüşen her biri, artık primary'de bir kişi gerektirmeyen yinelenen bir yazmadır.
- Bir statement her satıra ulaştığında yararlı soru, tasarımıdaki neyin buna izin verdiğidir ve cevap bir kişi değil, eksik guard'ların bir listesidir.

Veri düzeltmeleri tabloların içindekileri değiştirir; sonraki değişiklik türü tabloların kendisini değiştirir. Bölüm 11, şema değişikliklerine, onları kimin hangi kimlikle çalıştırdığına ve bir günün onarımının bir sonrakinin kesintisi olmaması için onları geriye uyumlu yapma alışkanlığına yönelir.

Bölüm 11 — Şema Değişiklikleri ve Migration Erişimi

Ledgerly'de bir perşembe öğleden sonrasını düşünün. Faturalama ekibi, vergiyi toplamdan ayrı gösterebilmek için `invoices` tablosuna yeni bir kolon istiyor ve değişiklik küçük: bir kolon, bir default, eski satırlar için bir backfill. Bu hafta on-call olan Robin pull request'i inceleyip onaylıyor. Pipeline migration'ı `ledgerly_migrator` olarak çalıştırıyor, deploy bitiyor ve yaklaşık doksan saniye boyunca `billing-api`'ye gelen, bir faturaya dokunan her istek takılıp ardından hata veriyor.

Migration'da yanlış olan bir şey yoktu, kimse de olağan dışı bir şey yapmadı. Uzun süren bir `invoice-worker` batch transaction'ı, harici bir mail servisini beklerken açık kalmış ve tablo üzerinde bir lock tutmuştu; migration onun arkasına kuyruğa girdi, migration'dan sonra gelen her statement de migration'ın arkasına girdi. Platform ekibinden Alex nedeni çabuk buluyor. Ardından daha zor soru geliyor: migration, her tablonun owner'ının gücüyle çalıştı, başında kimse yokken çalıştı ve bu güçle ledger arasında yalnızca bir diff'in review'u vardı. Şema değişiklikleri primary'ye nasıl ulaşmalı, hangi kimlikle ulaşmalı ve bunların erişim modelinde bir delik ya da bir kesintiye dönüşmesini nasıl önleriz?

Bu bölüm hedef durumun bir adımını ekliyor: migration'lar pipeline'dan `ledgerly_migrator` olarak çalışır ve şemayı değiştirme gücü başka hiçbir şeyde yoktur.

Bölüm 1'deki beş parçalı modelde migration, en geniş sınıra sahip **Yürütme**'dir. Pipeline, tasarım gereği owner'ın yetkilerini taşır; dolayısıyla risk, credential'ı kimin tuttuğunda değil. Risk, migration'ın ne yapmaya yetkili olduğunda, statement'larının nasıl yürütüldüğünde ve geriye hangi kaydın kaldığında. Migration'lar için bu, her lane'de izin verilen kısa bir statement sınıfları listesi (yetkilendirme), bir statement'ın ne zaman ve

nasıl çalışacağına karar veren, kısa bir lock wait ve bir retry içeren bir yol (yürütme) ve owner'ın yeniden yazamayacağı bir kayıt (hesap verebilirlik) demek.

Problem ve neden zor olduğu

Şema değişikliği, çoğu ekibin sıradan bir günde bir veritabanına yaptığı en ayrıcalıklı işlemdir. Sahiplik, owner'a bir tabloyu silme, bir kolonun tipini değiştirme, grant'leri yeniden yazma ve birçok engine'de herkes için geçerli kısıtlamaları aşma imkânı verir. Bölüm 3 bu gücü, login'i olmayan bir rol olan `ledgerly_owner`'a verdi ve `ledgerly_migrator`'ın onu yalnızca bir migration çalışırken üstlenmesine izin verdi. Bu düzenleme şemanın kime ait olduğunu yanıtlar. Bir migration'ı kimin başlatabileceğini, çalışan bir migration'ın ne yapabileceğini ya da o çalışırken herkesin başına ne geleceğini yanıtlamaz.

Tek bir faaliyetin içinde üç ayrı risk var.

Birincisi **yetkidir**. Pipeline'ın migration adımını tetikleyebilen herkes, fiilen production'da owner düzeyinde statement'lar çalıştırabilir. Kalıcı olan şey yetkidir; Bölüm 3'ün tarif ettiği anlamda: kısa ömürlü bir credential üzerinden kullanılan kalıcı bir owner yetkisi. Bu yetkinin sınırları usule dayanır: credential yalnızca pipeline'da yaşar, job'la birlikte sona erer ve pipeline onu kullanmanın onaylı tek yoludur. Bu sınırlar ancak pipeline'ın kendi erişim kontrolleri kadar sağlamdır; bu da, kimse çizmemiş olsa bile pipeline'ı veritabanının güvenlik sınırının bir parçası yapar.

İkincisi **availability'dir**. Birçok şema değişikliği lock gerektirir ve hasar lock'ta olur. Bir lock için beklemek zorunda kalan statement yalnızca beklemekle kalmaz: birçok engine'de arkasından gelen statement'lar da kuyruğa girer; dolayısıyla ilk saniyede lock'unu alamayan bir migration, önündeki en eski transaction açık kaldığı sürece tabloyu herkese kapatabilir. Değişikliğin büyüklüğü neredeyse hiç önemli değildir. Yanlış transaction yoldaysa tek kolonluk bir değişiklik, bir yeniden yazım kadar yıkıcı olabilir.

Üçüncüsü **drift'tir**. Migration'larla tanımlanan ama elle de düzenlenen bir şema, kimsenin tam olarak anlatamadığı bir şemadır. Kötü bir gecede bir console'dan eklenen hotfix index'i yararlıdır; bir yıl sonra ise ortamlar

arasında kimsenin hatırlamadığı bir farka ve hiçbir dosyayla eşleşmeyen grant'lere dönüşür.

Her biri tasarımıyla azaltılabilir. Pipeline ile sınırlanmış bir migration rolü, hızlı başarısız olan bir lock wait, hiçbir zaman uzun lock gerektirmeyen bir değişiklik biçimi ve canlı şemayı beklenen şemayla karşılaştıran bir kontrol; hepsi inşa edebileceğiniz şeylerdir. "Migration'larda dikkatli olun" diyen bir politika bu listede yok.

DDL'i kim çalıştırır ve hangi rolle

DDL (data definition language), satır okuyan ve yazan statement'ların aksine, şema nesnelerini oluşturan, değiştiren ve silen statement ailesidir. Bu bölümün kuralı kısa: DDL, pipeline'dan `ledgerly_migrator` olarak çalışır ve hiçbir uygulama rolü onu çalıştıramaz.

Uygulama rollerinde DDL yoktur

Bölüm 3'teki `app_*` rollerinin hiçbirisi bir tablonun owner'ı değildir ve tablo oluşturamaz, değiştiremez ya da silemez. Bunu söylemek kolay, korumak biraz maliyetli; çünkü birçok framework başlangıçta tablo oluşturabilir ya da değiştirebilir ve bazı ekipler bunu, deploy'dan bir adımı kaldırdığı için açar. Bu rahatlığın bedeli, servisin artık sahipliğe ihtiyaç duymasındır; ele geçirilmiş ya da şaşırmış bir servis de tabloyu silip yeniden grant verebilir. Production'da otomatik şema yönetimini kapatırdım; onu, silinmiş bir tabloyu kimsenin açıklamak zorunda kalmadığı geliştirme veritabanlarında tutardım.

Gözle görülür yan etkisi şu: şema değişikliği içeren bir deploy artık iki adımdır; önce migration, sonra servis rollout'u. İkisi arasındaki sıralama, aşağıdaki expand and contract örüntüsünün yanıtladığı bir tasarım sorusudur.

Migrator ve owner

`ledgerly_migrator`'ın pipeline'ın kullandığı bir login'i var; `ledgerly_owner`'ın üyesidir ve kendine ait hiçbir veri yetkisi taşımaz. Üyelik migrator'a owner'ın yetkilerini verir ama migrator'ı owner yapmaz. Birçok engine'de yeni bir nesne, oluşturulduğu anda hangi rol aktifse ona ait olur; dolayısıyla migrator'ın kendi login'iyle çalışan bir migration,

tablolarını migrator'ın sahipliğinde bırakırdı. Bu yüzden kural her engine'de aynıdır: bir migration her zaman açıkça `ledgerly_owner`'a geçer, çünkü sahiplik miras alınan yetkileri değil, aktif rolü izler. Oluşturduğu her nesne böylece `ledgerly_owner`'a ait olur ve migration bittiğinde migrator'ın kendi rolüne döner. Hiçbir engine'e ait olmayan bir notasyonla:

```
function runMigration(migration):
    login = fetchMigratorLogin(identity: pipeline.workloadIdentity)
    connection = connect(login: login, database: "ledgerly_primary")
    // lock_wait 3s, statement timeout none (Chapter 5)
    applyRoleSettings(connection, role: "ledgerly_migrator")
    if isApplied(connection, migration.id):
        close(connection)
        return // a retry never re-runs
    applied work
    // external record first: the owner can rewrite the database's own
    record
    auditLogAppend({ actor: login.name, action: "migration-start",
                    migration: migration.id, commit: migration.commit
    })

    try:
        assumeRole(connection, "ledgerly_owner")
        execute(connection, migration.statements)
        recordApplied(connection, migration.id)
    on LockWaitExceeded or DatabaseError as error:
        // owner rights end with the connection on every path
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-
failed",
                    migration: migration.id, error: error.kind })

        raise error
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-applied",
                    migration: migration.id })
```

Dört ayrıntı önemli. Credential bu job'a, pipeline'ın workload identity'si tarafından çalışma anında verilir (Bölüm 4'teki örüntü) ve job'la birlikte sona erer; böylece kopyalanmış bir pipeline değişkeni aylarca çalışan bir owner credential'ı tutamaz ve repository'de de böyle bir şey bulunmaz. Owner rolüne geçiş, tek bir migration'ın statement'larıyla sınırlıdır. Bir migration'ın uygulandığının kaydı aynı çalışmada yazılır; böylece veritabanı kendi sürümünü bilir ve aşağıdaki drift kontrolü buna dayanır. Harici kayıt da ilk owner statement'ından önce yazılır, çünkü

`recordApplied`, owner'ın yeniden yazabileceği bir veritabanında durur. Bunun bedeli, ulaşamayan bir audit store'un migration'ı başlamadan durdurmasıdır. Bu, Bölüm 10'daki veri düzeltmesinin tersi bir sıradır; orada gateway'in session log'u statement'ları zaten kaydediyor. Burada owner'ın session'ında olanları pipeline'dan başka hiçbir şey görmüyor, bu yüzden kayıt önce gelir. Her start kaydının `migration-failed` ya da `migration-applied` olarak eşleşen bir sonu vardır ve `isApplied`, başarısız olan adım son audit append'i olsa bile `retry`'ı güvenli kılar.

Bazı engine'lerde sürümü kaydetmek ve DDL'i çalıştırmak tek bir transaction'ı paylaşabilir; bazılarında ise bir DDL statement'ı çevresindeki transaction'ı bitirir. Hangisine sahip olduğunuzu bilin. Bir hata şemayı değiştirilmiş ama sürümü kaydedilmemiş bırakabiliyorsa, migration ikinci kez çalıştırılmasının zararsız olacağı biçimde yazılmalıdır.

İnsanlar migrator'ı tutmaz

İnsanlar `ledgerly_migrator` olarak asla bağlanmaz. Bir kişinin şema değişikliğine ihtiyacı varsa migration'ı yazar, pipeline de çalıştırır. Bu sürtünme gibi geliyor ve öyle de, ama işe yarar bir özellik kazandırıyor: owner'ın gücünün her kullanımı, bir commit'i, bir review'u ve bir log'u olan bir pipeline çalışmasına karşılık gelir. Migrator credential'ı aynı zamanda bir ekip parola yöneticisinde duruyorsa bu özellik kaybolur; migrator'a sahip olmanın nedeni de onunla birlikte kaybolur.

Pipeline neye ulaşabilir

Pipeline'a migrator credential'ını vermek, büyük bir gücü birçok kişinin etkileyebildiği bir sisteme taşır. Bunu açıkça söylemekte yarar var, çünkü bu tasarımın sessizce ters gidebileceği yer burası.

Pipeline ne yapabilir

`runMigration`'ı çalıştıran bir job, şemadaki herhangi bir nesneyi oluşturabilir, değiştirebilir ya da silebilir; owner'ın sahip olduğu nesnelerdeki herhangi bir grant'i değiştirebilir ve iki append-only tabloyu, `ledger_entries` ve `audit_events`'i değiştirebilir. Hiçbir uygulama rolü ve hiçbir insan rolü, `human_breakglass` dahil, bu tablolardaki satırları

düzenleyemez; yapısal tek istisna, owner'ın ve dolayısıyla çalışan bir migration'ın onları değiştirebilmesidir. Bu istisnanın kontrolü bir yetki değildir. Review'dur.

Dolayısıyla pipeline hakkındaki soru şu: onu bir şey çalıştırmaya kim zorlayabilir? Buna deployment branch'ine merge eden kişiler, pipeline tanımını düzenleyebilen kişiler, credential'ı tutan bir job'a adım ekleyebilen kişiler ve job çalışırken ortamını okuyabilen herkes dahil.

Erişimi sınırlamak

Bu erişimi beş yolla sınırlardım; her birinin ne kadarını ortadan kaldırdığına göre kabaca sıralı olarak.

1. **Credential'ı yalnızca bir job tutar.** Production ortamının migration job'ı, migrator login'inin verilebildiği tek yerdir ve her job'ın okuyabildiği bir değişkenle değil, job'ın kimliğiyle verilir. Review edilmemiş branch'lerin build'leri, test job'ları ve pull request pipeline'ları onu asla almaz.
2. **Pipeline tanımı, migration'lar gibi review edilir.** Migration job'ında, izinlerinde ya da onu tetikleyen branch kurallarında yapılan bir değişiklik, owner'ın erişiminde yapılan bir değişikliktir ve bir migration'ı kontrol edecek reviewer'ın aynısına ihtiyaç duyar.
3. **Tetikleyici korumalı bir branch'tir.** Ona merge etmek, yazar olmayan bir kişinin review'unu gerektirir; bu, Ledgerly'nin şema değişiklikleri için four-eyes kontrolüdür. Grant'lere, sahipliğe ya da append-only tablolara dokunan migration'larda reviewer platform ekibinden olmalıdır.
4. **Credential kısa ömürlüdür.** Başarısız olup içinde bir secret bulunan bir log bırakan job, süresi çoktan dolmuş bir secret bırakır.
5. **Her çalışma kaydedilir.** Pipeline, neyi uyguladığını, hangi login ile, hangi commit'ten, pipeline'ın yeniden yazamayacağı bir store'a yazar ve start kaydını migration çalışmadan önce yazar. Bu store'un nerede durduğunu Bölüm 14 anlatıyor.

Review'un kendisi kısmen mekanik olabilir. Küçük bir kontrol, migration'ı bir kişiden önce okuyabilir ve daha yakından bakılması gereken statement'ları işaretleyebilir:

```

function classifyMigration(migration, lane):
    flags = []
    refusals = []
    for each statement in migration.statements:
        if statement.touches("ledger_entries") or
statement.touches("audit_events"):
            add(flags, "append-only table: platform review required")
            if statement.kind in ["GRANT", "REVOKE", "CHANGE OWNER",
                                "CHANGE DEFAULT PRIVILEGES", "CREATE
VIEW"]:
                add(flags, "changes access: platform review required")
            if statement.kind in ["CREATE TABLE", "ADD COLUMN"] and
(statement.table is new or statement.table in
sensitiveTables) and
not hasExplicitGrants(migration, statement.table):
                add(flags, "new structure without a grant decision")
            if statement.kind in ["INSERT", "UPDATE", "DELETE"]:
                add(flags, "data change in a migration: backfill rule
applies")
            if statement.kind in lockHeavyKinds:
                add(flags, "long lock: scheduled window and a named
watcher")
            if statement.kind in ["DROP TABLE", "DROP COLUMN"] and not
migration.isContractStep:
                add(flags, "destructive change outside a contract step")
            if lane == "expedited" and statement.kind not in
expeditedAllowlist:
                add(refusals, "not allowed in the expedited lane: use the
normal lane")
    return flags, refusals

```

Normal lane'de kontrol hiçbir şeye karar vermez. Reviewer'ın dikkatinin erişimi değiştiren, lock tutan ya da veriyi yok eden satırlara düşmesini sağlar ve bir grant'ın unutulmasını görünür kılar. Her işaretle merge'ü engelleyen ve hiçbirini onaylamanın yolu olmayan bir kontrol bir ay içinde atlatılır; bu yüzden bir işaret, adı belli bir onaylayan gerektirir ve değişikliği yasaklamaz. Kontrolün gerçekten reddettiği tek yer hızlandırılmış lane'dir: boş olmayan bir `refusals` listesi job'ı çalışmadan önce başarısız kılar ve hiçbir onaylayan bunu geçersiz kılamaz. Nedenini sonraki bölüm açıklıyor.

Normal lane hiçbir şeyi kesin olarak reddetmez ve bu bir tercihtir: yasaklı statement'ların katı bir listesi, Bölüm 9'daki deny, gate ve allow matrisinin yanında ikinci bir politika olurdu. `ledger_entries`'i silmek iki kez işaretlenir, hem append-only bir değişiklik hem de yıkıcı bir değişiklik olarak, ve platform ekibi buna evet demek zorundadır.

Neden bilet sahibi bir kişi olmasın

Bazı ekipler, her migration'ı onaydan sonra bir DBA'nın elle uygulamasını tercih eder. Ayda bir iki şema değişikliği olan küçük bir veritabanı için bu makul bir tercih olabilir ve pipeline'ın erişim sorusu ortadan kalkar. Bedeli, kişi yokken, değişikliğin birkaç ortama aynı şekilde uygulanması gerektiğinde ya da elle yapılan adım review edilenden saptığında ortaya çıkar. Pipeline'ın zayıflığı erişiminin genişliğidir; elle yapılan yolun zayıflığı ise uygulananın onaylananla aynı olmasını hiçbir şeyin zorlamamasıdır. Yukarıdaki sınırlarla pipeline'dan yanayım ve pipeline'ın kendi erişim listesini kısa tutmak imkânsız çıkarsa bu tercihi yeniden gözden geçirirdim.

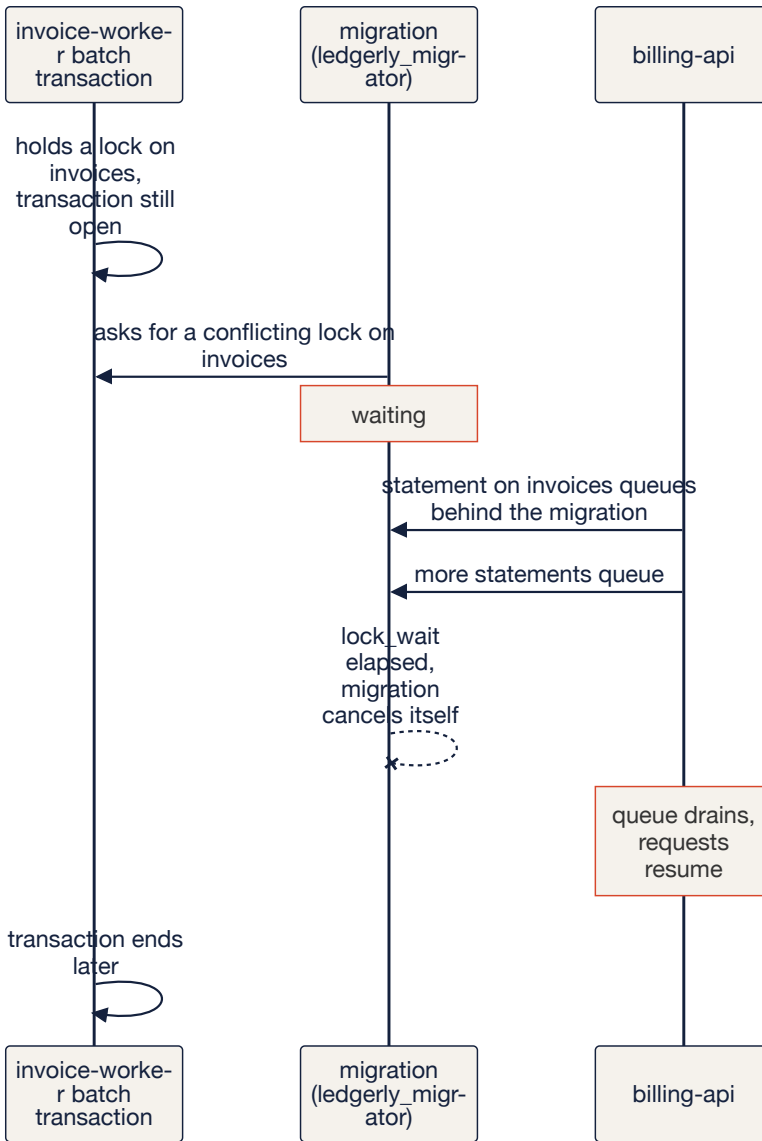
Lock'lar ve availability riski

Şimdi Ledgerly'deki o doksana saniyeye dönelim. Arkasındaki mekanizma engine'ler arasında ortaktır ve savunma bir avuç alışkanlıktan kurulur.

Bekleyen bir migration bir tabloyu nasıl kapatır

`invoices`'ı değiştiren bir migration, olağan okuma ve yazmalarla çakışan bir lock'a ihtiyaç duyar. Bu lock'u ancak çakışan bir lock tutan hiçbir transaction açık değilken alabilir. Uzun bir transaction açıksa migration bekler. O beklerken tablodaki yeni statement'lar migration'ın bekleyen isteğinin arkasına girer, çünkü öne geçmelerine izin vermek migration'ı aç bırakırdı. Tablo, eski transaction bitene ya da migration vazgeçene kadar fiilen kapanmış olur.

Aşağıdaki sıra olayların düzenini gösteriyor. Kuyruk büyümeden önce lock wait'in migration'a bir çıkış verdiği noktaya bakın.



Önemli olan kendi kendini iptal etmedir. Lock wait olmazsa migration, o batch transaction açık kaldığı sürece bekler ve **billing-api** o sürenin tamamında kuyrukta durur. Kısa bir lock wait ile migration birkaç saniye sonra başarısız olur, kuyruk boşalır ve pipeline daha sonra yeniden deneyebilir.

Kısa bir lock wait ve sabırlı bir retry

Bölüm 5, `ledgerly_migrator` için `lock_wait 3s` ve `statement timeout` olmamasını belirledi. Bu eşleşme bilinçlidir. Bir şema değişikliği işini yapmak için uzun süreye ihtiyaç duyabilir, ama başlamak için çok kısa süre beklemelidir. Retry de bu durumda pipeline'a aittir:

```
function applyWithRetry(migration, maxAttempts, pauseSeconds):
  attempt = 1
  while attempt <= maxAttempts:
    try:
      runMigration(migration)
      return
    on LockWaitExceeded:
      // a clean no-op only if nothing had committed: one
transactional
      // unit, or a locking step that comes first
      if not migration.retrySafe:
        notify("platform", "partial migration, not retried: "
+ migration.id)
        raise MigrationBlocked
        notify("platform", "migration waiting on a lock: " +
migration.id)
        sleep(pauseSeconds * attempt)
        attempt = attempt + 1
      // the table is busy for longer than the retry window; a person
decides
      notify("platform", "migration blocked after retries: " +
migration.id)
      raise MigrationBlocked
```

Bununla ilgili birkaç not. Başarısız bir lock wait, ancak henüz hiçbir şey commit edilmemişse temiz bir no-op'tur; bu yüzden bir migration `retrySafe` işareti taşır: tek bir transaction birimidir ya da lock alan adımı ilk sıradadır. Her DDL statement'ının kendi başına commit edildiği bir engine'deki çok adımlı bir migration körü körüne yeniden denenmez, çünkü ikinci deneme birincinin işiyle karşılaşır. Durur ve bir kişi karar verir. Bekleme süresi deneme sayısı ile artar; böylece tekrarlanan denemeler kendileri bir yük örüntüsüne dönüşmez. Ve son durum olan `MigrationBlocked` insani bir sorudur: hangi transaction'ın yolda olduğunu bulun ve beklemeye, onu iptal etmeye ya da yeniden zamanlamaya karar verin. Bu karar birinin session'ını öldürmeyi içerir; bu da başlı başına bir erişim sorusudur ve bir olay sırasında yaşandığında Bölüm 15'e aittir.

Uzun statement'lar ve online değişiklikler

Bazı işlemler, çalıştıkları sürece bir lock tutar; örneğin bir kolonun tipini değiştirmek için büyük bir tabloyu yeniden yazmak. Bunlarda sorun lock wait değil, süredir. Hangi statement'ların hızlı olduğu engine'e göre değişir ve yanıt engine'in sürümüyle de değişir; bu yüzden her birini kontrol edin. Online şema değişikliği araçları ve bloklamayan index build'leri uzun lock'tan kaçınır, ama nesne oluşturup sildikleri için her migration gibi aynı owner erişimine ihtiyaç duyarlar ve migrator olarak, pipeline'ın içinde çalışırlar. `classifyMigration` bu statement'ları uzun lock olarak işaretler ve uzun lock'lu bir migration, migration'ın açıklamasında yazılı, zamanlanmış bir pencere ve adı belli bir izleyen kişi alır. Bu, otomasyonun başarısızlığı değildir; bazı değişiklikler birer olaydır.

Expand and contract

En güvenli şema değişikliği, eski ve yeni uygulama kodunun ikisinin de yaşayabildiği değişikliklerdir. Bunun örüntüsü **expand and contract**'tır: şemayı eklemeli adımlarla değiştirin, uygulamayı karşıya taşıyın ve ancak o zaman artık kullanılmayana kaldırın. Bu kitap için önemli, çünkü her adım küçük bir lock'a sahip küçük bir migration'dır ve çünkü deploy'ları geri alınabilir tutar.

Ledgerly'de yalnızca biçimi göstermek için kullanılan varsayımsal bir değişiklik düşünün: `invoices` üzerindeki `total_minor`'ı `amount_minor` olarak yeniden adlandırmak. Kanonik şema `total_minor`'ı korur ve örnek, kolonun yeniden adlandırıldığı bir varyantı varsayar. Adımlar şöyle görünür:

1. **Expand.** Bir migration, kısa bir lock ile `amount_minor`'ı nullable bir kolon olarak ekler. Eski kod onu yok sayar.
2. **Write both.** `billing-api` ve `invoice-worker`'ın yeni sürümleri eski kolonun yanı sıra yeni kolona da yazar; böylece eski ve yeni sürümler yan yana çalışabilir.
3. **Backfill.** Eski satırlara yeni kolon için değerler verilir; aşağıda anlatıldığı gibi.
4. **Read the new column.** Sürüm yanlış davranırsa eski kolon hâlâ doğrudur; bu yüzden rollback bir onarım değil, bir deploy'dur.

5. **Contract.** Hiçbir şeyin `total_minor`'ı okumadığı ya da yazmadığı bir sürenin ardından son bir migration onu siler. Bu, yıkıcı olan tek adımdır ve ayrıca kapıdan geçirilir: yıkıcı olarak review edilir ve yeni kolonu getiren sürümde asla yapılmaz.

Expand and contract zaman ve dikkat maliyeti getirir. Tek migration olabilecek bir değişiklik, birkaç deploy'a yayılan dört ya da beş migration'a dönüşür ve ara adımlarda uygulama her iki biçim için de kod taşır. Tek servisi ve bakım penceresi olan küçük bir ekip için tek bir migration daha iyi seçim olabilir. Aynı tabloları okuyan beş servisi ve işin kabul ettiği bir penceresi olmayan Ledgerly için lock'u kısa, rollback'i ucuz tutan şey bu örüntüdür.

Migration içindeki backfill'ler ve dışındaki veri düzeltmeleri

Adım 3, verinin toplu bir değişikliğidir ve aynı kural, toplamın net ve vergi olarak bölünmesi için de geçerli olurdu; tek bir farkla: eski tabloda vergi rakamı yok, dolayısıyla o backfill hesaplanamaz ve yeni kolonlar bir veri düzeltmesi onları doldurana kadar boş kalmak zorunda olurdu. Bölüm 10 veri düzeltmesini, uygulamanın onaramadığı bir durumu onarmak için, bir kişinin normal uygulama yolunun dışında production verisinde yaptığı değişiklik olarak tanımlıyor; bu yüzden kitabın burada bir çizgiye ihtiyacı var: bir şema değişikliğine ait olan backfill, migration'ın içinde `ledgerly_migrator` olarak çalışır; verideki diğer her toplu değişiklik bir veri düzeltmesidir ve Bölüm 10'u izler.

Bir backfill'i şema değişikliğine ait yapan nedir?

- Tek amacı, yeni yapıya ilk değerlerini vermektir; bunu migration'da belirtilen bir kuralla yapar.
- Migration'da açıklanır, onunla birlikte review edilir ve pipeline tarafından uygulanır.
- Yazdığı her değer, tabloda zaten bulunan verilerden yeniden hesaplanabilir.

Bunlardan biri yanlışsa, o bir veri düzeltmesidir. Yeniden adlandırma çizginin nereden geçtiğini gösteriyor: `amount_minor`'a yazılan her değer aynı satırın `total_minor`'ıdır; migration'ın belirttiği ve bir reviewer'ın kontrol edebileceği bir kural. Ekibin backfill yaparken bazı eski faturaların `total_minor` değerinin yanlış olduğunu fark ettiğini varsayın. O

toplamları düzeltmek şema değişikliğinin parçası değildir. Etkilenen satır sayısıyla, beklenen satır sayısı üzerinde bir korumayla ve ikinci bir reviewer ile Bölüm 10'u izler; çünkü yanlış bir değeri koruyan bir kopya yine de doğru bir backfill'dir. Migrator'ın gücü büyüktür ve her zahmetli değişikliğin güzergâhı olmamalıdır.

Büyük bir tabloda backfill batch'ler hâlinde çalışır; her biri sınırlı bir satır kümesi üzerinde kısa bir transaction'dır. Replica lag büyüdüğünde duraklar (Bölüm 6), lag temizlenmezse görünür biçimde vazgeçer ve iş kaybetmeden durdurulup yeniden başlatılabilir.

Aynı kural iki append-only tablo için de geçerlidir. Bir migration ledger_entries'e kolon ekleyebilir, ama o kolonu düzeltilmiş değerlerle doldurmak için mevcut kayıtları yeniden yazmaz; bir düzeltme yeni bir kayıttır ve payments-service'in düzeltme yolundan verilir. O tablolara dokunan bir migration yukarıdaki sınıflandırma adımı tarafından işaretlenir ve review için platform ekibine gider.

Hotfix DDL ve drift

Her ekip er geç, bir index'in yavaş bir sorguyu düzelteceği ve pipeline'ın, diyelim, yirmi dakika sürdüğü geceyle karşılaşır. Soru, erişim modelinin bu kısayol hakkında ne söylediğidir.

Console kısayolu yok, o yüzden hızlı bir lane kurun

Hedef durumda kimse owner'ın login'ini tutmaz. human_breakglass INSERT ve UPDATE yazar ve DDL taşımaz; superuser de günlük işte kullanılmayan, mühürlü bir araçtır (Bölüm 3). Dolayısıyla primary'deki bir kişinin elle index eklemek için meşru bir yolu yoktur. Bu bilinçlidir ve soruyu pipeline'a taşır: ihtiyaç acilken review edilmiş bir değişikliği ne kadar hızlı çalıştırabilir?

Pipeline'a bir bypass yerine **hızlandırılmış lane** verirdim. Lane aynı job'ı, aynı credential'ı ve aynı kaydı kullanır; fark zamanlamadır, kimin onaylaması gerektiğinde değil. Four-eyes sürer: yazar olmayan bir reviewer yine onaylar ve yazar tek başına merge etmez. Gevşeyen şey, zamanlanmış bekleme süresi ve kısa bir statement sınıfı allowlist'i için platform ekibi şartıdır; platform ekibinin bir sonraki iş günü yapacağı takip review'u ile. classifyMigration allowlist'i uygular: bir index

oluşturmak, bir index silmek ve nullable bir kolon eklemek uygundur. Grant'leri, sahipliği ya da bir view'u değiştiren, append-only bir tabloya dokunan ya da veriyi yok eden bir statement uygun değildir, çünkü platform review'u tam olarak bu değişiklikler için vardır ve içinde biri bulunan bir migration lane'de reddedilir ve normal lane'den geçer. Bölüm 15 olaylar sırasındaki erişimi anlatıyor; burada önemli olan, aciliyetin sürecin hızını değiştirdiği, DDL'i çalıştıran kimliği ya da tehlikeli bir statement'ın göreceği review'u değiştirmedir.

Bedeli, hızlandırılmış şeridin bazı dikkat adımlarını atlamak için kalıcı bir seçenek olması ve böyle bir seçeneğin kullanılmasıdır. Ne sıklıkla kullanıldığını izleyin ve artan bir sayıyı, ekibin daha az dikkatli hâle geldiğinin değil, normal şeridin fazla yavaş olduğunun işareti sayın.

Drift

Drift, migration'ların tarif ettiği şema ile production'da bulunan şema arasındaki herhangi bir farktır. Elle uygulanan bir hotfix ile, uygulandıktan sonra düzenlenmiş bir migration ile, kimsenin hatırlamadığı bir aracın owner düzeyinde yaptığı bir değişiklik ile ve erişim tanımının dışında değiştirilen grant'ler ile girer. Şemadaki drift bir sonraki migration'ı şaşırtır. Grant'lerdeki drift, hiçbir review'un görmediği bir erişim değişikliğidir.

Zamanlanmış bir kontrol ikisini de bulabilir:

```

function detectDrift(environment):
    expected = buildExpectedSchema(appliedMigrations(environment))
    actual = readLiveSchema(environment)

    findings = []
    for each difference in compare(expected, actual):
        add(findings, difference)
    for each difference in compare(expectedGrants(),
readLiveGrants(environment)):
        add(findings, difference)

    if length(findings) > 0:
        notify("platform", "schema or grant drift in " + environment)
        recordCheckRun(environment, findings)

function checkDriftCheckAge(environment):
    // scheduled separately, so a stopped drift job cannot hide itself
    if now() - lastSuccessfulRun("detectDrift", environment) >
maxRunAge:
        notify("platform", "drift check has not completed recently")

```

Bunu yararlı kılan iki şey var. Beklenen durum, migration'lardan ve Bölüm 3'teki erişim tanımından türetilir; böylece karşılaştırılacak tam olarak tek bir doğruluk kaynağı vardır. Ve kontrol yalnızca tabloları değil grant'leri de kapsar, çünkü hiçbir dosyada olmayan bir grant, bu kitap için daha önemli drift türüdür. Bir fark bulunduğunda çözüm ikisinden biridir: kalması gerekiyorsa değişikliği kaydeden bir migration yazmak, kalmaması gerekiyorsa onu geri alan bir migration yazmak. Drift'i, neyin uygulandığının kaydını düzenleyerek çözmek, değişikliği review etmek yerine gizler.

Not: Drift'i, düzeltici bir statement'ı doğrudan superuser ya da owner olarak uygulayarak "düzeltmeyin". Farkı kaldıran statement, kendisi production'a yapılmış review edilmemiş bir değişikliktir. Onu bir migration'a koyun ve lane'den geçmesine izin verin.

Yeni tablolar, yeni kolonlar ve arkasından gelen grant'ler

Bölüm 3, yeni tablolar için boş bir default seçti: bir tablo, bir migration bir grant belirtene kadar owner dışında kimse tarafından okunamaz. Bu bölüm o kararın diğer yarısıdır, çünkü karar ya tutar ya da unutulur ve bunun olduğu yer migration'lardır.

Grant'ler migration'da yaşar

Bir tablo oluşturan her migration, grant'lerini aynı dosyada belirtir. Reviewer böylece bir değişikliğin erişim üzerindeki bütün etkisini tek bir diff'te görür; boş default'un gerekçesi de budur. Yeni bir tablo olan `invoice_adjustments` için (açıklayıcı bir tablo, kanonik şemanın parçası değil), dosya şöyle görünür. Serbest metin `reason` kolonu, `invoice_lines.description` gibi, en kötü durum için Confidential olarak sınıflandırılmıştır; bu yüzden tablonun owner'ı dışındaki okuyuculara onu dışarıda bırakan kolon düzeyinde grant'ler verilir:

```
CREATE TABLE invoice_adjustments (  
  id          text primary key,  
  invoice_id  text not null,  
  reason      text not null,  
  delta_minor integer not null,  
  created_at  timestamp not null  
);  
  
// the migration states every grant it means to give  
GRANT SELECT, INSERT ON invoice_adjustments TO app_billing_api;  
// reason is Confidential free text: left out of the broad read grants  
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON  
invoice_adjustments TO app_reporting_job;  
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON  
invoice_adjustments TO human_readonly;
```

Bir grant'i unutmamanın iki olası sonucu vardır ve aradaki fark, boş default'un nedenidir. Boş bir default ile tabloya ihtiyaç duyan servis ilk sorgusunda başarısız olur. Bu gürültüdür, bir testte ya da ilk deploy'da bulunur ve tek satırlık bir migration ile düzeltilir. Geniş bir default yetkiyle ise unutulmuş bir grant diğer yönde sessizdir: sonraki tablo default'un adını verdiği herkes tarafından okunabilir ve hiçbir şey başarısız olmadığı için kimse fark etmez. Gürültülü bir başarısızlık, sessiz bir ifşadan ucuzdur.

Başarısızlık ancak bir şey sorguyu çalıştırırorsa gürültüdür. Bu yüzden pipeline'a, her uygulama rolünün kullanması gereken tabloları kullanabildiğini kontrol eden bir migration sonrası smoke adımı ekledim. Uygulama rollerinin login'i yoktur, dolayısıyla adımın bir mekanizmaya ihtiyacı var. Engine destekliyse her uygulama rolünün üyesi olan ve sırayla her birine geçen bir test login'i kullanırdım; alternatifi rol başına bir test login'idir. İki seçim de pipeline'ın erişimini genişletir, çünkü herhangi bir servis rolüne dönüşebilen bir login, bir saldırgan için tek bir

servis credential'ından daha değerlidir. Bu yüzden job'ın workload identity'si tarafından verilir, job'la birlikte sona erer ve yalnızca bu adım için kullanılır. Karşılığında adım, unutulmuş grant'i servis rollout'undan önce yakalar.

Zaten grant'leri olan tablolardaki yeni kolonlar

Yeni bir kolon, tablosunun nasıl grant edildiğine göre farklı davranır.

Tablo grant biçimi	Yeni bir kolon...	Sonuç
Tablo düzeyinde grant	grant'e sahip her rol tarafından okunabilir	Yeni bir hassas kolon hemen ifşa olur
Kolon düzeyinde grant	adı verilene kadar kimse tarafından okunamaz	Yeni bir kolonun kullanılabilmesi için grant edilmesi gerekir

`customers`, `bank_accounts` ve `payment_tokens` gibi hassas kolonları olan tablolar için bu, engine'in desteklediği yerlerde kolon düzeyinde grant'leri savunur; Bölüm 3'ün yaptığı seçim de budur. Yeni kolon böylece kapalı başlar ve onu ekleyen migration, hangi rollerin okuyabileceğine ve Confidential mı Restricted mı olduğuna ekibin karar verdiği yerdir (sınıflandırmayı Bölüm 12 ele alıyor). Bu tablolardan birine hiçbir grant statement'ı olmadan kolon ekleyen bir migration bir soruyu tetikler ve sınıflandırma kontrolü bu soruyu sorabilir.

Engine'de kolon düzeyinde grant yoksa Bölüm 3'teki yedek çözüm geçerlidir: Restricted kolonlar `bank_account_secrets` ve `payment_token_values` gibi ayrı tablolarda yaşar. Yeni bir Restricted kolon bu tablolardan birine aittir, asla ana tabloya değil; çünkü ana tablonun grant'leri tablo düzeyindedir ve JIT tavanı onun çevresine çizilmiştir. Bunu yanlışlıkla ihlal etmek kolaydır, çünkü herkesin zaten bildiği tabloya kolon eklemek doğal harekettir; bu da yedek çözümü bilen bir reviewer'ın dışında kimsenin fark edeceği türden bir hatadır.

Hassas tablolar üzerindeki view'lar

Birçok engine'de bir view, owner'ının yetkileriyle çalışır; bu da Restricted kolonları olan bir tablo üzerindeki bir view'un, okuyucularının kendi rollerinin göremeyeceği şeyi açığa çıkarabileceği anlamına gelir. `classifyMigration`'ın `CREATE VIEW`'u bir erişim değişikliği olarak

işaretlemesinin, birini oluşturan migration'ın platform ekibinden geçmesinin ve hassas bir tablo üzerindeki bir view'un bir kolaylık olarak değil bir erişim değişikliği olarak değerlendirilmesinin nedeni budur.

Trade-off'lar ve maliyetler

Pipeline yüksek değerli bir hedeftir. Owner düzeyinde DDL çalıştırabilen ve birçok kişinin düzenleyebildiği bir sistemin elinde bulunan bir credential, gücün yoğunlaşmasıdır. Review edilmiş bir pipeline, owner düzeyinde erişimi olan birinin review edilmemiş bir şey yapmasını engellemez; o şeyi sonradan görünür kılar ve bu, görüldüğünden daha zayıf bir kontroldür. Bu bölümdeki sınırlar riski azaltır ama ortadan kaldırmaz. Pipeline'ınızın erişim listesi uzunsa ve kimse sahiplenmiyorsa, geri kalanın hiçbirine güvenmeden önce bunu düzeltin.

Review'lar gecikme ekler ve ancak reviewer kadar iyidir.

Migration'larda four-eyes kontrolü küçük değişiklikleri yavaşlatır ve özellikle reviewer haftada, diyelim, on migration görüyorsa ve her biri bir kolonsa bir formaliteye dönüşebilir. Sınıflandırma kontrolü tehlikeli satırları göstererek yardım eder. Dikkatin yerini tutmaz. Her zaman saniyeler içinde onaylanan bir review, izlenmesi gereken işarettir ve Bölüm 8 bunun tam savını yapıyor.

Expand and contract değişiklik sayısını çoğaltır. Tek bir mantıksal değişiklik birkaç deploy'a dönüşür. Bakım penceresi olan, tek servisli ve küçük tablololu bir ekip için buna değmeyebilir. Örüntü, tabloyu birçok şey okuduğunda ve bir değişikliğin süresince kimse onları durduramadığında karşılığını verir.

Kısa bir lock wait migration'ları başarısız kılar. İstenen davranış budur ve bir maliyettir. Yoğun bir tabloda bir migration birkaç denemeye ihtiyaç duyabilir ve sonuncudan sonra ne olacağına birinin karar vermesi gerekir. Bekleme süresini iş yükünüz için fazla kısa ayarlarsanız pipeline, platform ekibini gürültü için çağırır. Onu kendi uzun transaction'larınızın nasıl görüldüğüne göre ayarlayın ve bunu runbook'ta yazın.

Drift kontrolleri, birinin sahiplenmesi gereken bulgular üretir.

Kimsenin okumadığı bir kanala uyarı veren bir kontrol hiçbir şey bulmaz. Kontrolü açmadan önce her bulgu sınıfının kime ait olduğuna karar verin.

Bu tasarımı ne zaman kullanmamalı. Atılacak bir veritabanı, bir prototip ya da müşteri verisi olmayan tek mühendislik bir proje için, owner rolüne sahip bir migrator ve review edilmiş bir pipeline durumun gerektirdiğinden fazladır. Migration'ları bir geliştiricinin makinesinden çalıştırın, production veritabanını kritik yolun dışında tutun ve veriye sizden başka biri bağımlı olduğunda pipeline'ı ekleyin. Bu bölümdeki tasarım, şemanın servisler tarafından paylaşıldığı ve yanlış bir değişikliğin müşterilere bir şeye mal olduğu nokta içindir.

Sık yapılan hatalar

- **Servisin başlangıçta kendi tablolarını oluşturmaya ya da değiştirmesine izin vermek.** Deploy'dan bir adımı kaldırır ve servise, silme ve yeniden grant verme gücünü de içeren sahiplik verir. Production'da otomatik şema yönetimini kapalı tutun ve migration'ları migrator ile ayrı bir adım olarak çalıştırın.
- **Migrator credential'ını "acil durumlar için" bir ekip parola yöneticisinde tutmak.** Cazip gelir, çünkü pipeline bazen en kötü anda başarısız olur. Her sahibi bir owner'a dönüştürür ve sahipliğin her kullanımının bir pipeline çalışmasına denk geldiği özelliği ortadan kaldırır. Bunun yerine hızlandırılmış şeridi kurun.
- **Bir migration'ı lock wait olmadan çalıştırmak.** Migration, önündeki en eski transaction kadar bekler ve arkasındaki her statement de bekler. Migrator'a kısa bir lock wait verin ve retry'ı pipeline'a koyun.
- **Bir veri düzeltmesini backfill'in içine saklamak.** Tablo zaten açıkken kötü değerleri düzeltmek verimli hissettirir. Migration'ın review'u bir kolon tanımı için tasarlanmıştı, müşterilerin tutarlarında yapılan bir değişiklik için değil. Değişiklik, yeni yapının belirtilmiş bir kuralla ilk doldurulması değilse, bir veri düzeltmesidir ve Bölüm 10'u izler.
- **Eski kolonu, yenisini getiren sürümde silmek.** Bir migration'dan tasarruf sağlar ve rollback'i ortadan kaldırır. Contract adımını ayrı, ertelenmiş ve yıkıcı olarak review edilmiş tutun.
- **Yeni tabloların grant satırı olmadan çalışması için default yetkilere güvenmek.** Geniş bir default, unutulmuş bir grant'i sessiz bir ifşaya çevirir. Default'u boş tutun ve her migration grant'lerini kendisi belirtsin.

- **Kolon düzeyinde grant'i olmayan bir engine'de ana tabloya Restricted kolon eklemek.** Tablonun grant'leri tablo düzeyindedir ve JIT tavanı onların etrafında kurulmuştur. Restricted kolonları ayrı secret tablolarına koyun.
- **Bir drift bulgusunu gürültü saymak.** Production'da bulunan ve kodun hiçbir yerinde olmayan bir grant, kimsenin review etmediği bir erişim kararıdır. Onu ya kaydeden ya da geri alan bir migration ile çözün.
- **Bir migration'ı uygulandıktan sonra düzenlemek.** Eski sürümü çalıştıran ortamlar artık yenisini çalıştıranlardan farklıdır. Bunun yerine yeni bir migration ekleyin.

Temel çıkarımlar

- DDL, pipeline'dan `ledgerly_migrator` olarak, tek bir migration süresince owner olarak çalışır ve hiçbir uygulama rolü herhangi bir DDL yetkisi taşımaz.
- Migrator kalıcı bir owner yetkisi taşır ve onu kısa ömürlü bir credential üzerinden kullanır: onu bir job tutar, workload identity ile verilir, her çalışma başlamadan önce kaydedilir ve insanlar onu asla kullanmaz.
- Pipeline owner'ın gücüne ulaşabildiği için, kendi erişim listesi ve tanımı bir migration gibi review edilir; grant'lere ya da append-only tablolara dokunan migration'lar platform ekibine gider ve hızlandırılmış lane, `classifyMigration`'ın uyguladığı yalnızca bir statement sınıfları allowlist'ini kabul eder.
- Pipeline'da retry ile birlikte kısa bir lock wait, bloke olmuş bir migration'ın bir tabloyu kapatmasını önler; online araçlar ve eklemeli adımlar ise aksi hâlde uzun süre lock tutacak değişiklikleri ele alır.
- Expand and contract eski ve yeni kodun birlikte çalışmasını sağlar ve rollback'i bir deploy yapar, bedeli daha fazla migration'dır; yeni yapıyı belirtilmiş bir kuralla dolduran bir backfill migration'ın içinde çalışır ve verideki diğer her toplu değişiklik bir veri düzeltmesidir ve Bölüm 10'u izler.
- Şemadaki ya da grant'lerdeki drift, canlı durumun migration'ların ve erişim tanımının tarif ettiğiyle karşılaştırılmasıyla bulunur ve sessiz bir düzenlemeyle değil, bir migration ile çözülür.

- Her migration, oluşturduğu şeyin grant'lerini belirtir ve default'ları boş tutar; böylece unutulmuş bir grant bir tabloyu sessizce ifşa etmek yerine gürültüyle başarısız olur.

Şema değişiklikleri, veri üzerindeki en büyük gücü bir kişinin değil, bir pipeline'ın tuttuğu tek yerdir. Sonraki kısım, veritabanına kimin ulaşabildiğinden onun içinde ne olduğuna dönüyor: Bölüm 12, hangi kolonların korumayı hak ettiğini, onları nasıl sınıflandıracağımızı ve onlara ihtiyaç duyan işi bozmadan nasıl maskeleyeceğimizi ya da saklayacağımızı soruyor.

KISIM IV

Veriyi Korumak

Bölüm 12 — Hassas Veri: Sınıflandırma, Maskeleye ve Minimizasyon

Ledgerly'de bir pazartesiye düşünün. Bir müşteri destek ekibine yazıp bir ödemenin hesabına ulaşmadığını söylüyor ve kayıtlı banka hesabının doğru olup olmadığını soruyor. Destekten Sam, `admin-console`'da müşterinin sayfasını açıyor ve hesap numarasının tamamını ekranda buluyor. Sohbetinde geri okumak yanıt vermenin en hızlı yolu, ticket'a ekran görüntüsü koymak da vakayı bir meslektaşına devretmenin en hızlı yolu. Hikâyedeki kimse makul olmayan bir şey yapmıyor ve öğleden sonra bittiğinde numara bir sohbet log'unda ve bir ticket ekinde duruyor.

İki masa ötede veri ekibinden Jordan, `ledgerly_analytics` üzerinde bir rapor hazırlıyor ve oradaki `customers` tablosunun hiçbir raporun ihtiyaç duymadığı iletişim adlarını, e-postaları ve telefon numaralarını taşıdığını fark ediyor. On-call olan Robin, bir hatırlatmanın neden başarısız olduğunu bulmak için bir uygulama log'u okuyor ve alıcının e-posta adresini hata mesajının içinde buluyor. Her ifşa küçük ve üçünün biçimi ortak: veri, kimsenin orada olmasına karar vermediği bir yere ulaştı.

Bu bölüm hedef durumun bir adımını ekliyor: hassas kolonlar sınıflandırılır, default olarak maskelenir ya da hariç tutulur ve analytics kopyası ham Restricted veri tutmaz. Yanıtladığı soru, her kişinin ve her kopyanın gerçekte neyi görmesi gerektiği ve sistemi bunu gösterecek, fazlasını göstermeyecek biçimde nasıl kuracağımız.

Bu kitabın beş parçalı modelinde (**Kimlik, Grant, Yol, Yürütme, Kayıt**) bölüm iki parça üzerinde çalışıyor. Grant, bir rolün hangi kolonları okuyabileceğine karar verir; Yürütme ise bir değerin yoldan hangi biçimde döneceğine. Hiçbiri, bir credential'ın bir kişiye nasıl ulaştığına bağlı değil; tam hesap numarasını okuyan, mükemmel korunmuş bir login yine de onu ifşa etmiş olur.

Problem ve neden zor olduđu

Önceki bölümlerdeki kontroller kimin hangi tabloya ve hangi kolonlara ulaşacağına karar verir. Bir kolonun neden önemli olduğunu bilmezler. `customers` üzerindeki bir grant bir durum bayrağıyla bir iletişim e-postasını aynı görür ve şema ikisine de `text` der. Bir kolon tanımındaki hiçbir şey `invoice_lines.description`'ın bir kişinin ad yazmış olabileceği serbest metin olduğunu ya da `iban`'ın, ifşasının bir şeyi rotate ederek geri alınamayacağı türden bir değer olduğunu söylemez.

İkinci zorluk, insanların verinin bir kısmına ihtiyaç duyması ve kimsenin göremeyeceğini söyleyen bir kuralın geçici çözümler doğurmasıdır. Destek, ödeme hesapları hakkındaki soruları yanıtlamak zorundadır. On-call, bir hatırlatmanın neden başarısız olduğunu bulmak zorundadır. Veri ekibi müşterileri segmente göre saymak zorundadır. Onaylı yol hiçbir şey göstermediğinde onaysız olanlar ortaya çıkar: ekran görüntüleri, ad-hoc export'lar, "proje bitene kadar" kalıcı bir grant. Yararlı soru bir kişinin bir değeri görüp göremeyeceği değil, görevin değerin hangi biçimine ihtiyaç duyduğudur: tamamı, son rakamları, bir evet ya da hayır ya da bir sayı.

Üçüncü zorluk, verinin tabloda kalmamasıdır. Bir analytics store'una kopyalanır, log satırlarına yazılır, hata mesajlarında yankılanır, ticket'lara yapıştırılır ve export'larla gönderilir. Primary üzerindeki her kontrol veritabanının kenarında durur ve değer yoluna devam eder. Yanıtın sisteme yerleştirilmesi gerekmesinin ve bir politika olarak kalamamasının nedeni budur. "Kişisel veriyi özenle işleyin" diyen bir politika bir migration pipeline'ı, zamanlanmış bir yükleme ya da bir log formatter tarafından yürütülemez. Kararlar çalışan şeylere yazılmalıdır: şemaya, grant'lere, yüklemeye ve loglama koduna.

GDPR, KVKK ya da PCI DSS gibi veri koruma ve ödeme kartı rejimleri bir soru ailesi sorma eğilimindedir: hangi kişisel verileri tutuyorsunuz, neden, nereye gidiyor, kimler görebiliyor ve ne kadar süreyle. Şemada tutulan bir sınıflandırma, bunları yanıtlayabileceğiniz biçimdir. Herhangi bir rejimin yorumu hukuk ve uyum ekiplerinize aittir; bu bölüm yalnızca yanıtların bulunabilir olmasını sağlayan mühendisliği anlatır. Katalog, sistemin ne yaptığının kanıtıdır, herhangi bir rejimi karşıladığının iddiası değildir; bir erişim review'unun hangi kanıtları toplayabileceğine Bölüm 14 dönüyor.

Şemada yaşayan sınıflandırma

Veri sınıflandırması, veriyi adlandırılmış sınıflara ayırır ve her sınıf nasıl işleneceğine karar verir. Ledgerly dört sınıf kullanır.

Sınıf	Ne anlama gelir	Ledgerly örnekleri	Default işleme
Restricted	İfşası, müşterilere ya da şirkete geri alınması zor zararlar verir	iban, account_holder_name, token_value	human_readonly'den hariç tutulur; hiçbir just-in-time (JIT) isteğinde yer almaz; yalnızca bir uygulama yolundan değiştirilir; analytics kopyasında yoktur
Confidential	Müşteriler ve iletişim kişileri hakkında kişisel ya da ticari veri	contact_email, billing_address, provider_reference, invoice_lines.description	human_readonly için maskelenir ya da hariç tutulur; maskesiz yalnızca bir onaylayanın olduğu JIT SELECT'te; analytics kopyasında maskelenir ya da düşürülür; gösterilmek için var olan görüntüleme ipuçları hariç (provider, expires_on; aşağıdaki karar tablosuna bakın)
Internal	Kişisel içerik taşımayan operasyonel veri	amount_minor, status, due_on, account_code	İhtiyaç duyan roller okuyabilir; analytics'e kopyalanır
Public	Yayımlanması güvenli	Sekiz tablonun hiçbirinde yok	Yalnızca bütünlük

Public satırı bilerek boştur. "Bu yayımlanabilir mi?" sorusunun sessizlikten başka bir yanıtı olsun diye var.

Dört kural sınıfları kullanılabilir kılar. Herhangi bir araç seçmeden önce yazıya dökeceğim kısım bunlar.

- **Tabloları değil, kolonları sınıflandırın.** customers bir durum bayrağı ve bir telefon numarası tutar ve ikisi farklı işleme ihtiyaç duyar. Yalnızca envanter amacıyla, bir tablo kolonları arasındaki en yüksek sınıfı alır.

- **Serbest metni en kötü durum için sınıflandırın.** İnsanların yazdığı bir kolon er geç insanların yazdığı her şeyi tutar. `invoice_lines.description`, kişisel veri içerebildiği için Confidential'dır ve hiçbir filtre hangi satırların içerdiğini söyleyemez.
- **Türetilmiş bir değer, biri aksini savunana kadar kaynağının sınıfını devralır.** Bir `iban`'ın keyed hash'i, bir kişi öyle olmadığını savunana kadar Restricted'dır. `payment_tokens.last_four`, gösterilmek için var olduğundan Restricted değil Confidential'dır ve bu karar bir kez verildi ve yazıya geçirildi.
- **Bilinmeyen Confidential'dır.** Kimsenin sınıflandırmadığı bir kolon Confidential sayılır ve geçici olarak işaretlenir; böylece bitmemiş iş görünür olur ve default kapalı olandır.

Migration'larla aynı repository'deki bir katalog

Bir wiki'de yaşayan sınıflandırma drift eder, çünkü şemayı değiştiren hiçbir şey wiki'yi okumaz. Onu migration'ları tutan repository'de bir dosya olarak tutardım ve kolonu ekleyen pull request'te değiştirirdim. Bazı engine'ler kolonlara etiket ya da yorum da koymaya izin verir; pipeline okuyabildiği sürece her iki biçim de işe yarar.

```
// illustrative catalog entry, one per column
table column_classification:
  table_name          text not null
  column_name         text not null
  class               text not null    // Restricted,
Confidential, Internal, Public
  reason              text not null    // one sentence, for the
reviewer
  owner                text not null    // a team, never a person
  human_readonly       text not null    // readable, masked,
withheld
  analytics_copy       text not null    // copy, transform, drop
  reviewed_on          date null        // null means provisional
```

Katalog iki kontrolü mümkün kılar. Birincisi her migration'a karşı, Bölüm 11'de anlatılan işaretlerin yanında çalışır. Bir reviewer'dan bakmasını isteyen o işaretlerin aksine bu, merge'ü engeller; çünkü bir kolonu sınıflandırmak bir satıra mal olur ve eksik bir satır, bütün kontrolün yakalamak için var olduğu şeydir.

```

function checkColumnClassification(migration, catalog):
    problems = []
    for each column in addedColumns(migration):
        entry = catalog.find(column.table, column.name)
        if entry is null:
            add(problems, "unclassified: " + column.table + "." +
column.name)
            continue
        if entry.class == "Restricted" and not
inRestrictedHome(column):
            // a main table whose grants are table-level cannot
withhold it
            add(problems, "Restricted outside its home: " +
column.name)
            if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
                add(problems, "Restricted column would reach analytics")
            if problems is not empty:
                raise MergeBlocked(problems)

```

Buradaki `inRestrictedHome`, grant'lerin kolonu hariç tutabildiği bir tablo demektir: kolon düzeyinde grant'leri olan bir ana tablo ya da bunlara sahip olmayan engine'lerde Bölüm 3'teki ayrı secret tabloları. Kolonu ekleyen kişi emin değilse çıkış yolu kontrolü atlamak değil, kolonu `reviewed_on` boş olarak Confidential sınıflandırmak ve sonraki erişim review'unun karara bağlamasına izin vermektir.

İkinci kontrol zamanlanmış olarak çalışır ve canlı şemayı katalogla karşılaştırır; Bölüm 11'in canlı şemayı migration'larla karşılaştırmasıyla aynı biçimde. Kötü bir gecede elle oluşturulmuş bir kolon, sahibi olan bir bulgu olarak görünür ve bulgu sessiz bir düzenlemeyle değil, bir migration ve bir katalog satırıyla çözülür.

Minimizasyon: tutmadığınız veri

Minimizasyon, kimsenin ihtiyaç duymadığı şeyi toplamamak, kopyalamamak ya da saklamamaktır. Bu bölümdeki en ucuz kontroldür, çünkü hiç saklanmamış bir değer bakımını yapılacak bir maskesi, bulunacak bir kopyası ve redact edilecek bir satırı yoktur. Buradaki diğer her kontrol verinin var olduğunu varsayar; bu ise olması gerekip gerekmediğini sorar.

Her hassas kolon için bu sırayla dört soru sorardım.

1. Herhangi bir özellik ya da yükümlülük bu değere ihtiyaç duyuyor mu?
Yanıt "işe yarayabilir" ise yanıt hayırdır.
2. Bunun yerine bir vekil tutabilir miyiz? Ledgerly, ödeme araçlarının kendilerini değil, onların yerine geçen, provider tarafından verilmiş token'lar olan `payment_tokens`'ı tutar. Bu tek karar, şemadan bütün bir değer sınıfını kaldırır.
3. Daha azını tutabilir miyiz? `last_four` gibi bir görüntüleme ipucu, destek ekibinin ve müşterilerin aracı tutmadan tanınmasını sağlar.
4. Daha kısa süre tutabilir miyiz? Kapanmış bir müşterinin iletişim bilgilerinin bir amacı, son fatura kapanana ve itiraz pencereleri geçene kadar vardır; ondan sonra yalnızca kısa, tanımlı bir süre saklanır ve silinir.

Verinin nerede yaşadığına dair iki karar, herhangi bir saklama kuralından daha önemlidir. Birincisi **append-only tabloların kişisel veri taşımamasıdır**. `ledger_entries`, ikisi de Internal olan `amount_minor` ve `account_code`'u tutar ve bir düzeltmenin yeni bir kayıt olabilmesinin nedeni budur: bir kişiyle ilgili bir talebi yerine getirmek için tablodaki hiçbir şeyin silinmesi gerekmez. `audit_events` aynı nedenle konusuna opak bir `subject_id` ile atıfta bulunur. İkincisi serbest metindir. Sabit bir değer kümesine sahip yapılandırılmış bir alan sınıflandırılabilir ve maskelenebilir; bir not kutusu ikisi de değildir; bu yüzden bir özellik bir cümle yerine bir kod, bir durum ya da bir ürün referansı kullanabiliyorsa onu seçerdim.

Saklama bir iştir, bir politikadaki cümle değil. Bir şeyin zamanlanmış olarak çalışması, süresi geçmiş satırları bulması ve silmesi ya da anonimleştirilmesi gerekir ve o şeyin kendi dar yetkilerine, kendi audit kayıtlarına ve bir sahibine ihtiyacı vardır. Verileriniz için doğru süreyi bilmiyorum, bu kitap da bilmiyor. Bu, sözleşmelerinizin, düzenleyicilerinizin ve kendi itiraz sürecinizin gerektirdiğine bağlıdır ve bir mühendis işi yazmadan önce hukuk ekibinizin bunu yanıtlaması gerekir. Bir mühendisin söyleyebileceği şey, backup'ların, export'ların ve analytics kopyasının da eski veriyi tuttuğu ve yalnızca primary'ye ulaşan bir silmenin bitmediğidir. Bölüm 13 buna dönüyor.

Maskeleme: dinamik ve statik

Maskeleme, gerçek değer görünmesin ama biçim kullanılabilir kalsın diye bir değeri değiştirir ya da gizler. **Redaction** kaba versiyonudur: değer bir çıktıdan ya da log'dan kaldırılır. İki tür maskeleme, nerede etki ettikleri bakımından ayrılır.

	Dinamik maskeleme	Statik maskeleme
Ne değişir	Bir sorgunun döndürdüğü	Kopyanın kendisi
Ne zaman	Okuma anında, okuyucu başına	Yükleme anında, bir kez
Ham değer hâlâ saklanıyor mu	Evet, aynı store'da	Kopyada hayır
Tipik kullanım	Bir replica üzerinde human_readonly	Analytics kopyası, staging verisi
Nasıl başarısız olur	Engine davranışı, filtreler, doğrudan grant'ler	Bayat ya da fazla kırılmış bir kopya

İnsanlar için dinamik maskeleme

human_readonly, her session'ın başladığı default'tur; bu yüzden maskelemesi çoğu kişinin karşılaştığı maskelemedir. Aşağıdaki notasyon açıklayıcıdır, çünkü engine'ler maskelemeyi çok farklı biçimlerde ifade eder: bazıları kolon üzerinde bir maske sunar, bazıları bir unmask yetkisi, bazıları tablo başına sabit bir view.

```
// illustrative notation. The grant lists the columns the role may
read;
// each mask rule changes what a read through the role returns
GRANT SELECT (id, created_at, status, contact_name, contact_email,
               contact_phone, billing_address) ON customers TO
human_readonly;

MASK RULE contact_name_initials  ON customers (contact_name)
  FOR human_readonly USING initials();
MASK RULE contact_email_partial  ON customers (contact_email)
  FOR human_readonly USING keep_first(1);
MASK RULE contact_phone_tail    ON customers (contact_phone)
  FOR human_readonly USING keep_last(2);
MASK RULE billing_address_hidden ON customers (billing_address)
  FOR human_readonly USING redact();
MASK RULE provider_reference_tail ON payments (provider_reference)
  FOR human_readonly USING keep_last(4);

// free text has no shape to mask, so it is withheld: not in the
column list
// (illustrative column list)
GRANT SELECT (id, invoice_id, created_at) ON invoice_lines TO
human_readonly;
```

Fonksiyon seçimi, kolonun ne için var olduğundan çıkar. Bir teknik kendi başına daha iyi ya da daha kötü değildir; her birinin bir yeri ve bir başarısız olma biçimi vardır.

Teknik	Okuyucunun aldığı	Uygun olduğu yer	Yardımlı kesildiği yer
Kısmi	İlk ya da son birkaç karakter	Bir ekran, yapıştırılmış bir sonuç	Kısa değerler büyük ölçüde kendilerini gösterir
Redaction	Sabit bir işaret ya da hiçbir şey	Adresler, serbest metin	Hiçbir şeyi doğrulayamaz
Keyed pseudonym	Kararlı bir vekil	Yüklemeler arası join'ler	Anahtarı rotate etmek join'leri bozar
Kabalaştırma	Bir ülke ya da bir ay gibi daha geniş bir değer	Analytics	Birleştirilen kolonlar bir kişiyi ayırt edebilir (Bölüm 13)

Düz bir hash listede yok. Bir telefon numarası gibi, her adayın denenebileceği kadar küçük bir kümeden alınan bir değer hash'i, adaylar denenerek tersine çevrilebilir; bu yüzden bir pseudonym, veritabanının dışında tutulan bir anahtara ihtiyaç duyar.

Her Confidential kolon maskelenmez. Gösterilmek için var olan bir kolon okunabilir kalır: `payment_tokens.provider`, `last_four` ve `expires_on`, bir kişinin aracı tanınmasını sağlayan görüntüleme ipuçlarıdır ve onları maskelemek insanları yalnızca yerine geçtikleri Restricted değere doğru iter. İletişim alanları, fatura adresi ve `provider_reference` bir kişiyi ya da bir işlemi tanımlar ve maskelenir. `invoice_lines.description` hariç tutulur.

Bu, Bölüm 8'in dayandığı kuralı verir. `human_readonly` zaten okur; bu yüzden bir JIT `SELECT` üçlüsü yalnızca rolün maskelediği ya da hariç tuttuğu bir Confidential kolon için gereklidir ve eklediği şey o kolonun maskesiz olmasıdır. Restricted kolonlar hiçbir JIT üçlüsünde görünmez; dolayısıyla burada onlar için hiçbir şey değişmez. İstek ve onaylayanın ekranı bu yüzden tablo listesinden ayrı, açık bir satır taşınmalıdır: "Maskesi kaldırılan kolonlar: `contact_email`"; böylece onaylayan, isteğin yalnızca hangi tablolara dokunduğunu değil, hangi maskeleri kaldırdığını da görür.

Uyarı: Engine'inizin, `human_readonly` üzerinden bir kolonu okuyan ve aynı zamanda doğrudan bir kolon grant'ine sahip olan bir login'i nasıl çözdüğünü kontrol edin. Maske kazanırsa, bir JIT `SELECT` yeni bir şey göstermez ve onayın bir anlamı kalmaz. Ham değerler her okuyucu için kazanırsa maske süstür. İkisine de güvenmeden önce iki durumu bir kopya üzerinde test edin.

Tekniğin yanında iki sınır durmalıdır. Birincisi, birçok engine'de bir maske bir sorgunun döndürdüğüne uygulanır, koşullarının karşılaştırdığına değil; bu yüzden maskeli bir kolon üzerinde filtre uygulayabilen bir okuyucu, sayımlar ve eşleşmeler üzerinden değerleri hakkında bir şeyler öğrenebilir. Dinamik maskelemeyi, bir ekran, bir ekran görüntüsü ya da yapıştırılmış bir sonuç gibi sıradan ifşalara karşı koruma olarak ele alın. İkincisi, bir çıktıyı maskelemek saklanan değer için hiçbir şey yapmaz; o değer `primary`'nin her backup'ında, replica'sında ve dump'ında kalır.

İlk sınır, çıkarım için bir karar kuralı verir. Bir okuyucunun değeri filtreleyerek ne kadar ucuza geri kazanabileceğini sorun: `contact_email` üzerinde bir avuç eşitlik denemesi ucuzdur ve maskeli bir telefon önekinde göre müşteri sayısı da ucuzdur. Ucuz olduğu ve okuyucunun değeri öğrenmemesi gerektiği yerde kolonu maskelemeyin, hariç tutun ya da o

rol için maskeli kolon üzerindeki predicate'leri reddedin. Pahalı olduğu yerde, örneğin uzun bir serbest değeri binlerce loglanmış statement ile her seferinde bir karakter yoklamak gibi, maskeleme artı statement kaydı makul bir kontroldür.

Sonuç akışında maskeleme

Bazı erişim yolları engine'in maskelemesine güvenemez. Kişi ile veritabanı arasında duran bir gateway, satırlar ekrana ya da bir sonuç dosyasına ulaşmadan önce kuralları sonuç akışının kendisine uygulayabilir. Bu, engine'in maskeleme özelliği olmadığı yerlerde yardımcı olur ve kurallar arkasındaki her engine için aynı olabilir.

Bir akış kuralı, bir değerin hangi kolondan geldiğini bilmeye bağlıdır ve `SELECT api_key AS k` gibi bir alias, kolon adıyla eşleşen kuralları bozar. QueryProxy gibi sonuçları maskeleyen araçlar, değerle de eşleşen kuralları ekler; bu ikinci bir katmandır, grant'in yerine geçen bir şey asla değil. Bu yazının yazıldığı sırada bu araç, ad tabanlı kuralların bu sınırlamasını kendisi belgeliyor ve diğer araçlar da benzer yaklaşımlar izliyor. Ledgerly için sonuç, bir akış maskesinin Bölüm 3'teki kolon grant'lerinin arkasında durması ve bir rolün asla görmemesi gereken bir kolonun bir desene bırakılmayıp grant tarafından hariç tutulmasıdır.

Not: Kolon düzeyinde grant'i olmayan engine'lerde Ledgerly, Bölüm 3'teki yedek çözümü kullanır: Restricted kolonlar `bank_account_secrets` ve `payment_token_values` içinde durur. `human_readonly`'yi maskelemek o zaman genellikle tablo başına, bir migration ile bir kez yapılan ve `ledgerly_owner`'ın sahip olduğu, rolün temel tabloda değil view üzerinde SELECT tuttuğu sabit bir maskeli view demektir. Temel tablodaki tablo düzeyinde bir JIT grant'i, Confidential olanlar dahil, Restricted olmayan her kolonu ham gösterir ve onaylayan tam olarak bunu kabul etmiştir. Bu, istek başına yapılan bir view değil sabit bir view'dur ve Confidential kolonlar için bir maskeleme mekanizmasıdır. Restricted olanlar için ikinci bir yedek çözüm değildir: onlar için tek yedek çözüm secret tablolarıdır.

Analytics kopyası için statik maskeleme

Analytics kopyası, maskelemenin en ucuz ve en yararlı olduğu yerdir, çünkü ham değer kopyanın dışında tamamen bırakılabilir. Jordan'ın raporu müşteri segmentlerine, sayılara ve tutarlara ihtiyaç duyar. Adlara, e-postalara ya da telefon numaralarına ihtiyaç duymaz; bu yüzden çoğu Confidential kolon için doğru işlem drop'tur ve transform, analistlerin kanıtlanabilir biçimde kullandığı birkaç kolon içindir.

Yükleme bir servis değil, zamanlanmış bir job'dır ve erişim review'undaki diğer her principal gibi kendi login'i ve `app_analytics_load` rolü altında çalışır. Rol, `SELECT`'i yalnızca katalogun Restricted olarak işaretlenmediği kolonlarda tutar; yani birinci koruma grant, ikincisi kataloğtur. Bağımsız değildirler, çünkü grant kataloğun Restricted işaretlerinden türetilir: yanlışlıkla Internal olarak sınıflandırılmış bir kolon ikisini de geçer. Grant'in eklediği şey, bir transform girdisi yanlış olsa bile Restricted işaretli bir kolonun okunamamasıdır ve bağımsız kontrol, kopyayı planla karşılaştıran `verifyMasked`'taki şema assertion'ıdır. Bu grant'in içinde yükleme, kataloğa bakarak kolon kolon karar verir ve fail closed davranır.

```
function loadAnalyticsTable(table, catalog):
    plan = []
    for each column in sourceColumns(table):
        entry = catalog.find(table, column)
        if entry is null:
            // an unclassified column never travels; the load says so
            notify("platform", "not loaded, unclassified: " + table +
"." + column)
            continue
        if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
            raise CatalogInvalid
            if entry.analytics_copy != "drop":
                add(plan, { column: column, treatment:
entry.analytics_copy })
            // build beside the live table, check, then swap: a failure leaves
the
            // previous load in place, so the copy never holds a half-masked
table
            next = table + "_next"
            writeMasked(next, from: table, plan: plan)
            if not verifyMasked(next, plan):
                dropTable(next)
                raise MaskingCheckFailed
            swapTable(next, into: table)
```

`verifyMasked` iki kontrol yapar. Şemayı assert eder: hedefin kolon kümesi plana eşit olmalı ve içinde hiçbir `Restricted` ya da düşürülmüş kolon bulunmamalıdır. Sonra satırlardan örnek alır ve dönüştürülmüş bir kolon hâlâ kaynak değerine eşitse başarısız olur. Şema `assertion`'ı, planın listelemediği bir yoldan gelmiş bir kolonu yakalar; örnekleme yanlış yapılandırılmış bir transform'u yakalar. İkisi de kopyanın hassas hiçbir şey tutmadığının kanıtı değildir, çünkü örnek yalnızca bazı satırları okur.

`Fail closed`'un bedeli görünürdür: bir geliştirici kolon ekleyip kataloğu unuttuğunda migration kontrolünün onu durdurmuş olması gerekirdi ve durdurmadıysa analytics tablosu o kolon olmadan yüklenir ve veri ekibi bir boşluk fark eder. Bunu kabul ederdim. Eksik kolonlu bir rapor bir ticket'tır; maskesiz bir kolonu olan bir kopya ise kimsenin fark etmediği bir ifşadır.

`Join`'ler için bir `keyed pseudonym` kendi başına bir karara layıktır. Analistlerin müşterinin kim olduğunu bilmeden bir müşteriye yüklemeler boyunca izlemesini sağlar ve yalnızca anahtar veri ekibinin erişiminin dışında ve değişmeden kaldığı sürece çalışır. Anahtarın `rotation`'ı her `pseudonym`'i değiştirir; böylece geçmiş artık `join` edilmez. Birçok ekip için opak `customer_id` bu işi zaten görür ve ikinci bir vekil, kazanç olmadan korunacak bir anahtar ekler.

Tokenization ve uygulama düzeyinde şifreleme

Tokenization, bir değeri yalnızca kontrollü bir servisin geri eşleyebildiği bir vekille değiştirir. Maskeleyme değildir, çünkü eşleme vardır ve genellikle şifreleme de değildir, çünkü vekil normalde değerden hiç hesaplanmaz. Vekili bir anahtar altında değerden hesaplayan şemalar başka bir adla şifrelemedir ve aynı anahtar özenini gerektirir. Ledgerly bundan zaten bir yerde yararlanır: `token_value` bir provider'ın token'ıdır, eşlemeyi provider tutar ve tek çağırın `payments-service`'tir. Ledgerly veritabanı bu yüzden aracı hiç tutmaz.

Banka hesap numaraları için ikinci bir tokenization servisi kurmak daha büyük bir adımdır. Ödeme yoluna, ödemeler ayaktaiken ayakta olması gereken bir sistem ekler, bir token'ı geri eşleyebilecek çağırınların listesine ihtiyaç duyar ve her eşleme, birinin review edebilmesi gereken bir olaydır. Ledgerly'nin ölçeğinde kurmazdım. Kolonları kısıtlamak, onları okuyan tek

servisi kısıtlamak ve değerleri şifrelemek, işletilecek daha az şeyle korumanın çoğunu verirdi. Düzenlemeye tabi bir ortak ya da bir ödeme provider'ı bunu gerektiriyorsa, gereksinim karar verir.

Uygulama düzeyinde şifreleme, servisin bir değeri veritabanı görmeden önce şifrelemesi demektir; böylece roller, replica'lar, backup'lar ve dump'ların hepsi ciphertext tutar. Ayırt edici gücü budur: değeri, bir grant'ın ulaşamadığı veri kopyalarında korur (Bölüm 13). Depolama katmanının diskte ya da bir snapshot'ta yaptığı şifreleme, kaybolmuş bir volume'e karşı korur ve bir sorguya karşı hiçbir şey yapmaz; ekipler onu sıkça erişim kontrolüyle karıştırır. Uygulama düzeyinde şifreleme gerçek bir sınıra daha yakındır ve daha ağır bir taahhüttür.

- **Anahtar kimin okuyabileceğine karar verir.** Veri ancak anahtarı kadar korunur. Decrypt iznini `payments-service`'in workload identity'sine verin, başka hiçbir şeye vermeyin; böylece karar, kimin hangi yoldan decrypt edebileceği sorusu olur. Anahtar hiyerarşileri, rotation ve anahtar backup'ı kendi başına bir disiplindir ve anahtar servisinin dokümantasyonu bunları bu kitaptan daha iyi anlatır.
- **Kolon sorgu özelliklerini kaybeder.** Ciphertext üzerinde olağan biçimde arama yapamaz, sıralayamaz, aralık filtresi uygulayamaz ya da benzersizlik zorlayamazsınız; bu yüzden yalnızca tanımlayıcıyla geri okunan kolonları seçin.
- **İstek üzerine decrypt edebilen bir servisten bunu isteyebilirsiniz.** Şifreleme sınırı servise taşır; böylece çağırانları, hızı ve audit kayıtları kontrol olur ve audit kaydı, aşağıdaki gibi, decrypt'ten önce yazılır.

```
function readPayoutAccount(accountId, onBehalfOf):
    row = selectBankAccount(accountId)           // the stored value is
    ciphertext
    // the audit entry comes first: if the log is unreachable, nothing
    is decrypted
    try:
        auditLogAppend({ actor: "payments-service", action: "decrypt",
                        subject: row.id, on_behalf_of: onBehalfOf })
    on AuditLogUnavailable:
        raise DecryptRefused
    // the key service checks the caller's workload identity, not the
    request
    return keyServiceDecrypt(row.iban, keyId: row.key_id)
```

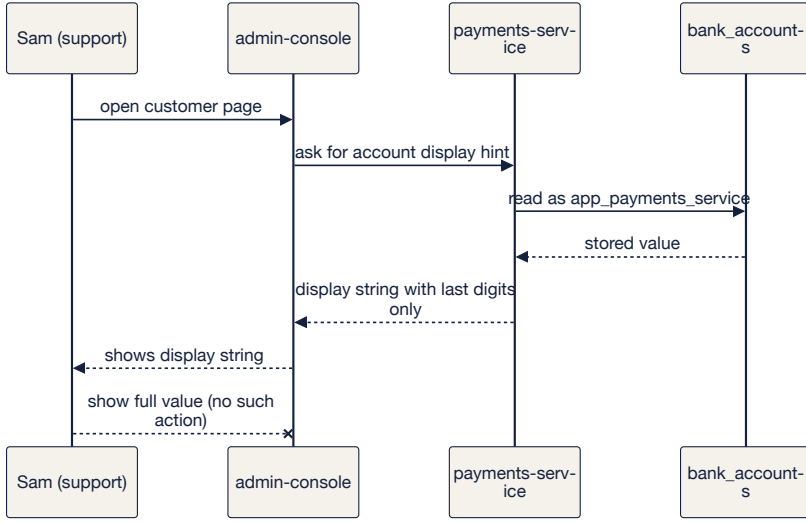

Şifrelerseniz matris değişmez. `human_breakglass` yine Restricted kolonları okur, ama okuduğu ciphertext'tir ve decrypt etmek, rolün taşımadığı bir anahtar izni gerektirir. Plaintext'e ihtiyaç duyan bir olayın anahtar erişimi isteyip isteyemeyeceğine, hangi onaylayanla, önceden karar verirdim ve bunu Bölüm 8'in break-glass runbook'una yazardım. Bu seçimi olaya bırakmak, bir kuralın doğaçlama üretilmesinin yoludur.

Görmeden çalışmak: destek vakası

Sam'e dönelim. Amaç, Sam'in müşteriye tam olarak yanıtlayabilmesi ve tam değeri hiç görmemesi. Çoğu destek sorusunun değerden daha azına ihtiyaç duyduğu ortaya çıkar.

Destek ne yapmalı	Neyi görmesi gerekir	Nasıl sağlanır
Müşteriye hangi hesabın kayıtlı olduğunu söylemek	Son rakamlar	<code>payments-service</code> bir görüntüleme string'i döndürür; ya da ipucu, <code>last_four</code> 'un token'lar için olduğu gibi, hesap yazılırken saklanır
Yazan kişinin iletişim kişisi olduğunu doğrulamak	Bir evet ya da hayır	Grant'e zaten sahip olan servisteki dar bir kontrol; müşterinin söylediğini saklanan değerle karşılaştırır ve "eşleşti" ya da "eşleşmedi" yanıtı verir
Müşterinin faturalarına bakmak	Durumlar, tutarlar, vade tarihleri; yalnızca bu müşteri için	<code>app_admin_console</code> , <code>invoices</code> 'in Internal kolonlarını okur ve Bölüm 3'teki, vakanın müşterisine dair satır kuralı satırları daraltır
Yanlış bir hesap numarasını düzeltmek	Hiçbir şey ve elle değil	Uygulama yolu: <code>admin-console</code> , <code>payments-service</code> 'ten ister, ikinci bir kişi onaylar ve değişiklik <code>audit_events</code> 'e kaydedilir

Diyagram ilk satırı gösteriyor. Son oka bakın: var olmayan bir eylem, var olup reddedilen bir eylemden daha güçlü bir kontroldür.



Başlangıçta bir "göster" düğmesi kurmadım. Gerekçe isteyen ve tıklamayı loglayan bir göster makul görünür ve bir ay içinde ona tıklamak bir alışkanlık, log'u ise kimsenin okumadığı bir şey olur. Gerçek bir ihtiyaç ortaya çıkarsa, Bölüm 8'in her maskesiz Confidential okuma için yaptığı gibi onu onaylayanı olan bir istek yapın ve ne sıklıkla kullanıldığını sayın. Her gün kullanılan bir özellik, görüntüleme biçiminde bir şeyin eksik olduğunu söylüyordur.

On-call olan kişi aynı kuralı farklı bir baskı altında izler. Başarısız bir hatırlatmayı debug eden Robin'in, bir alıcı adresinin reddedildiğini bilmesi gerekir ve adrese nadiren ihtiyaç duyar. Robin'in maskesiz

`contact_email`'e ihtiyacı olduğunda bu, kolonu adıyla belirten bir JIT SELECT'tir ve kolon listesini okuyan bir onaylayanı vardır. On-call okumaları asla otomatik onaylanmaz: Confidential kolonları atlayan bir kural hiçbir şey vermeyen bir isteği onaylar; onları içeren bir kural ise anlam taşıyan tek kontrolü kaldırır. Bedeli kötü bir saatte beklemektir ve bunu ödemenin yolu, yaygın durumlarda değere duyulan ihtiyaç ortadan kaldırmaktır; bunu sonraki bölüm ele alıyor.

Log'larda ve hata mesajlarında hassas veri

Log'lar, kimsenin tasarlamadığı veri kopyasıdır. Birçok bileşen tarafından yazılır, birçok kişi tarafından okunur, uzun süre saklanır ve erişim modelinin hiçbir zaman parçası olmamış araçlarla aranır. Sızıntının

çoğunu dört yer taşır.

- **Uygulama log'ları ve hata mesajları.** "Adrese gönderilemedi" gibi bir değerden kurulmuş mesaj, değeri log'a, hata takipçisine ve bazen bir müşterinin ekranına koyar. Sürücüler ve ORM'ler, debug loglama açıkken bağlı parametreleri de loglayabilir.
- **Veritabanı log'ları.** Statement loglama literal'leri kaydedebilir ve bir slow-query log'u bir sorgunun metnini içindeki değerlerle birlikte tutar.
- **Statement kayıtları.** Erişim gateway'i, bir kişinin gönderdiği her statement'ı kaydeder (Bölüm 7) ve bir koşula değer yazan kişi o değeri kayda koymuş olur.
- **Ticket'lar ve sohbet.** Bir sorunu açıklamak için yapıştırılmış bir sonuç, yanlış kitleye sahip ve bitiş tarihi olmayan bir kopyadır.

Uyarı: Bir statement log'u literal'leri tutar; bu yüzden Bölüm 14'teki audit log ve gateway'in statement kayıtları kendileri hassas veri store'larıdır. Statement'ları literal'leri redact edilmiş fingerprint'ler olarak kaydedin, store'u en az Confidential olarak sınıflandırın, kimin okuyabileceğini kısıtlayın ve ona bir saklama süresi verin; yoksa veriyi izleyen kontrol onun bir kopyasına daha dönüşür. Mekanizma Bölüm 14'e aittir.

Uygulama loglaması için tercih ettiğim kural, bir scrubber değil bir allowlist'tir. Bir scrubber hassas değerlere benzeyen desenleri arar ve kaldırır; yazarının düşündüğü biçimleri yakalar ve gerisini kaçırır. Bir allowlist görünebilecek alanları adlandırır ve geri kalan her şeyi düşürür; böylece yeni bir alan, biri onu loglamayı seçene kadar sessizdir.

```
function buildLogEvent(event, allowedFields):
  safe = {}
  dropped = 0
  for each field in event.fields:
    if field.name in allowedFields:
      safe[field.name] = field.value
    else:
      dropped = dropped + 1
  // the count is the signal: a field dropped every day is either a
  // missing allowlist entry or a value somebody tried to log
  safe["dropped_field_count"] = dropped
  return safe
```

Hata mesajları aynı disiplini izler. Bir hata, başarısız olan değeri değil, bir kod ve bir fatura id'si, bir müşteri id'si ve bir teslimat durumu gibi tanımlayıcılar taşır. Robin'in hatırlatma vakasının ham adrese ihtiyaç duymaması da böyle olur: başarısızlık kaydı "recipient rejected" gibi bir durum ve fatura id'sini tutuyorsa log, soruyu adres olmadan yanıtlar ve gateway'in adresi kimin okuduğuna dair kaydı, okuyan olduysa, tek bir yerde kalır.

Log pipeline'ındaki redaction bir güvenlik ağıdır ve bir şeyleri kaçıracaktır. Atlamayacağım kontrol testlerdeki bir canary'dir: uygulamayı her hassas alanda tanınabilir biçimde bir yer tutucuyla çalıştırın ve herhangi bir log çıktısında görünürse build'i başarısız kılın. Allowlist'i önemli olduğu yerde test eder ve birkaç satıra mal olur. Log değerler yerine tanımlayıcılar tuttuğunda debug yavaşlar ve ortaya çıkan "bunu neden göremiyorum" mesajlarının gideceği yer, bir geliştiricinin production'da parametre loglamasını yeniden açması yerine, `dropped_field_count`'tur.

Ledgerly'nin kolon kararları

Aşağıdaki tablo bölümü bir araya getiriyor. Sekiz tablonun hassas kolonları için, bir erişim review'unun okuyabileceği biçimde karar kayıdır. O kolonda "Hariç tutulur" ve "Erişim yok" ikisi de rolün kolonda hiçbir grant'i olmadığı anlamına gelir; dolayısıyla onu adıyla anan bir sorgu reddedilir. `human_readonly` kolonu bir session'ın default'unu belirtir; JIT kolonu bir istegin kolonu adıyla anıp anamayacağını belirtir.

Kolon	Sınıf	human_readonly	JIT SELECT	Analytics kopyası
bank_accounts.iban, account_holder_name	Restricted	Erişim yok	Asla	Düşürülür
payment_tokens.token_value	Restricted	Hariç tutulur	Asla	Düşürülür
payment_tokens.provider, last_four, expires_on	Confidential	Okunabilir	Gerekmez	provider ve expires_on tutulur, last_four düşürülür

Kolon	Sınıf	human_readonly	JIT SELECT	Analytics kopyası
customers. contact_name, contact_email, contact_phone	Confidential	Maskeli	Kolonu adıyla anar, maskesiz	Düşürülür
customers. billing_address	Confidential	Redact edilmiş	Kolonu adıyla anar, maskesiz	Düşürülür
payments. provider_reference	Confidential	Son dört karakter	Kolonu adıyla anar, maskesiz	Düşürülür
invoice_lines. description	Confidential	Hariç tutulur	Kolonu adıyla anar, maskesiz	Düşürülür
amount_minor, currency, status, due_on, account_code	Internal	Matrisin grant ettiği yerde okunabilir	Gerekmez	Kopyalanır

İki girdi bir yoruma layıktır. Restricted kolonlar analytics kopyasından düşürülür ve `human_breakglass` dışındaki hiçbir insan rolüne sunulmaz; onları değiştiren tek yol, ikinci bir kişinin onayıyla uygulama yoludur (yürütme politikasının gate'li işlemleri oraya nasıl yönlendirdiğini Bölüm 9 anlatıyor). Aynı yol owner bağlantısını da alır: `bank_accounts` ve `payment_tokens` üzerindeki `customer_id` Restricted olmadığı için JIT tavanının içinde kalır, ama bir satırı başka bir müşteriye taşımak hesabın kimin olduğunu değiştirir; bu yüzden bir owner bağlantısı değişikliği gate'lidir ve ürün yoluna yönlendirilir, sıradan bir veri düzeltmesi olarak asla yapılmaz. Ve engine'de kolon düzeyinde grant yoksa aynı tablo yine geçerlidir, tek bir değişiklik: Restricted kolonlar secret tablolarında yaşar, JIT kolonu "tablo düzeyi" der ve onay ekranı bunu sözcüklerle söyler.

Trade-off'lar ve maliyetler

Sınıflandırma bakımdır. Yazıldığı gün doğru olan bir katalog hemen eskimeye başlar. Birinin her sınıf kararını sahiplenmesi, geçici olanları review etmesi ve sınıflandırılmamış bir kolonun doğurduğu soruyu

yanıtlaması gerekir. Kimse kataloğu review etmezse katalog, eksiksiz ve yanlış olan türden bir kayda dönüşür; bu da dürüst bir boşluktan daha kötüdür.

Maskelene debug süresine mal olur. Robin ham e-postayı göremez; bu yüzden tam karakterlerine bağlı bir bug bir JIT isteğine ve bir onaylayana ihtiyaç duyar. Bu bölümün kapattığı erişim için bu bedeli kabul ediyorum. Bedava değil ve bunu her gece ödeyen bir ekip bir kestirme yol aramaya başlar; bu yüzden maskesiz okumanın ne sıklıkla istendiğini ölçün ve nedeni düzeltin.

Dinamik maskelene engine'e bağlıdır. Filtreler, doğrudan grant'ler ve view'lar etrafındaki davranış farklıdır ve engine'inizde test etmediğiniz bir maske bir umuttur. View tabanlı varyant, migration'ların temel tablolarla uyum içinde tutması gereken nesneler ekler.

Fail closed boşluk üretir. Sınıflandırılmamış kolonları reddeden analytics yüklemesi, biri sınıflandıranı kadar veri ekibini bir kolonsuz bırakır. Bu beklemenin kime ait olduğunu belirleyin ve sınıflandırma review'unu boşluk haftalar değil saatler sürece kadar hızlı tutun.

Uygulama düzeyinde şifreleme özellikleri kaldırır ve bir bağımlılık ekler. Anahtar servisi kolonun her okumasının yolundadır ve kesintisi bir ödeme kesintisidir. Audit kaydı önce geldiği için ulaşılamayan bir audit log da bu okumaları durdurur; bu, bilinçli bir bağlantı ve bir availability bağımlılığı daha demektir. Sorgu özelliklerinin kaybı aramayı diğer kolonlara iter ve anahtar backup'ı test edilmesi gereken bir şey daha olur.

Buradaki bazı kontroller güvenlik tiyatrosuna dönüşebilir. Her zaman tıklanan bir unmask düğmesi, kimsenin review etmediği bir katalog, kimsenin test etmediği bir log scrubber'ı ve herhangi bir okuyucunun filtrelerle yoklayabildiği maskeli bir kolon, sürtünme ya da evrak işi ekler ve hiçbir şeyi azaltmaz.

Bunların hepsini ne zaman kullanmamalı. Bir avuç tablosu olan ve müşteri kişisel verisi bulunmayan küçük bir sistemin bir katalog dosyasına, maskeli bir yüklemeye ya da bir anahtar servisine ihtiyacı yoktur. Hassas kolonların tek sayfalık bir listesi, onları dışarıda bırakan kolon düzeyinde grant'ler ve bir log allowlist'i değerin çoğunu verir. Verinin ikinci bir kitleye

ihtiyacı olduğunda maskeleme ekleyin, etrafından grant veremeyeceğiniz bir kopya bir kaygı olduğunda şifreleme ekleyin ve hangi olayın sizi sonraki adıma geçireceğini yazın.

Sık yapılan hatalar

- **Encryption at rest'i erişim kontrolü saymak.** Cazip gelir, çünkü tek bir ayardır ve bir kontrol listesini karşılar. Kaybolmuş bir diski korur ve bir sorguya karşı hiçbir şey yapmaz. Grant'leri koruyun ve bir kopyanın korunması gereken yerde uygulama düzeyinde şifreleme kullanın.
- **Maskelemeyi veritabanında değil client'ta yapmak.** Web arayüzündeki bir maske kurması kolaydır ve doğrudan bir bağlantı, bir export ya da bir API yanıtı onu atlar. Veri store'dan çıktığı yerde maskeleyin.
- **Kolonları değil tabloları sınıflandırmak.** Tablo başına tek etiket durum bayrağını fazla, telefon numarasını yetersiz korur. Her kolonu sınıflandırın ve tablonun etiketini onlardan türetin.
- **Yeni kolonları default olarak Internal yapmak.** Internal en az sürdürülmeli sınıftır; bu yüzden review edilmemiş her şeyin sınıfı olur. Default'u Confidential yapın ve review ile temizleyin.
- **Serbest metni desenlerle maskelemek.** Bir desen yazarının beklediği numaraları ve e-postaları bulur ve bir nota yazılmış bir adı kaçıır. Serbest metin kolonlarını geniş okuyuculardan hariç tutun.
- **Pseudonym olarak anahtarsız bir hash kullanmak.** Geri çevrilemez görünür. Az olasılığı olan bir değer için olasılıklar denenerek geri çevrilebilir. Anahtar veritabanının dışında kalan keyed bir pseudonym kullanın ya da kolonu düşürün.
- **Production'ı staging'e kopyalayıp "sonradan maskelemek".** Kopya ile maske arasındaki pencere, her şeyin bir kopyasıdır. Analytics yüklemesinin yaptığı gibi yolda maskeleyin ve staging'i Bölüm 13'teki kapılardan biri olarak ele alın.
- **Bir adımdan kazanmak için bir JIT isteğine Restricted kolon eklemek.** Tavan onu bir nedenle reddeder ve cazibe geceleri en güçlüdür. Restricted değerler yalnızca uygulama yolundan, four-eyes

onayıyla değişir ve kolon düzeyinde grant'i olmayan engine'lerde secret tablolarında durur.

- **On-call okumalarını otomatik onaylamak.** Bir kişinin hangi Confidential kolonun maskesinin kaldırıldığını kontrol ettiği duraklamayı kaldırır. Onaylayanı koruyun ve değere ihtiyaç duyulan sayıyı azaltın.
- **İstek gövdelerini error düzeyinde loglamak.** Gövde bir geliştirici için en yararlı şeydir ve bir müşterinin yazdığı her alanı tutar. Tanımlayıcıları ve durumları loglayın ve başka neyin görüneceğine bir allowlist karar versin.

Temel çıkarımlar

- Her kolonu sınıflandırın, serbest metni en kötü durum için ele alın ve sınıflandırılmamış kolonları default olarak Confidential yapın; böylece yeni veri kapalı başlar ve onu açma işi görünür olur.
- Sınıflandırma kataloğunu migration'ları tutan repository'de tutun ve sınıfsız bir kolon ekleyen merge'ü engelleyin, çünkü kontrol bir satıra mal olur ve yokluğu başarısızlığın kendisidir.
- Minimizasyon önce gelir: hiç saklanmamış veri bir maskeye ihtiyaç duymaz ve append-only tablolardan uzak tutulan kişisel veri bir silme problemine dönüşemez.
- Dinamik maskeleme `human_readonly`'ye uyar ve sıradan ifşaya karşı korur; statik maskeleme, ham değerlerin tamamen düşürülebildiği ve yüklemenin sınıflandırılmamış her şeyde fail closed davrandığı analytics kopyasına uyar.
- İnsanlara görevlerinin ihtiyaç duyduğu değer biçimini, bir görüntüleme string'i ya da bir evet ya da hayır gibi, verin ve Restricted kolonları her JIT isteğinin ve her ham SQL yazmasının dışında tutun.
- Sonuç akışındaki bir maske grant'in arkasındaki ikinci bir katmandır, yerine geçen bir şey asla değil; Restricted kolonları okuyamayan bir yükleme rolü de bir katalog hatasını fail closed yapar.
- Log'lar, hata mesajları ve statement kayıtları veri kopyalarıdır: tanımlayıcıları bir allowlist üzerinden loglayın, değerleri mesajların dışında tutun ve audit log'un kendisini sınıflandırın.

Bu bölümdeki kontroller veriyi veritabanında dururken ve bir sorgu onu döndürürken korur. Sonraki bölüm veriyi dışarı doğru, bu rollerin hiçbirinin geçerli olmadığı backup'lara, dump'lara, export'lara ve staging kopyalarına kadar izliyor.

Bölüm 13 — Backup'lar, Export'lar ve Diğer Yan Kapılar

Ledgerly'de bir cuma öğleden sonrasını düşünün. Robin, fatura hatırlatmalarında yalnızca gerçekçi veriyle ortaya çıkan bir bug'ın peşinde: alışılmadık vade tarihli faturalar, birden fazla iletişim kişisi olan müşteriler, kimsenin planlamadığı serbest metin taşıyan satır açıklamaları. Staging'de elle hazırlanmış bir düzine müşteri var ve bug bunların hiçbirinde görünmüyor. Birisi production'ın eski bir dump'ının paylaşımlı depolamada durduğunu hatırlıyor ve veritabanlarına bakan Alex, bütün ekip görebilsin diye onu staging'e restore ediyor.

Bug bir saat içinde bulunuyor. Dump klasörde kalıyor, restore edilen veritabanı da staging'de. Staging'de access gateway yok, çünkü production değil. Login'i ortak, her mühendis erişebiliyor, CI job'ları ona karşı çalışıyor ve geçen ay katılan bir yüklenicinin hesabı var. Staging'deki `invoice-worker`, production'la aynı giden e-posta ayarlarını kullanıyor ve hatırlatma job'ı açık.

Kimse müşteri verisini açığa çıkarmaya karar vermedi. Her adım, bir öğleden sonranın problemine verilmiş makul bir cevaptı; önceki bölümlerde kurulan kontroller ise artık içeriğinin tek kopyasını tutmayan bir veritabanını koruyor.

Bu bölüm, Kısım V'ten önceki hedef durumun son adımını ekliyor: backup'lar, export'lar ve staging kopyaları, sahibi ve süre dolumu olan kapılar olarak ele alınır. Soru şu: veri primary'den bilerek çıktığında (kurtarma için, bir vendor için, bir test için ya da bakması gereken biri için), kontrolleri verinin geldiği veritabanında bırakmak yerine verinin yanında nasıl tutarsınız?

Beş parçalı modelde (**Kimlik, Grant, Yol, Yürütme, Kayıt**) bir kopya, Grant'in ve Yol'un artık geçerli olmadığı yerdir; çünkü veri, ikisini de taşımayan bir güzergâhtan çıkmıştır. Bölümün işi, kopyanın kendisine bir

Grant taşıtmaktır: bir okuyucu listesi, bir sahip ve bir bitiş tarihi; ayrıca kopyayı kimin yaptığını ve kimin kaldırdığını gösteren bir Kayıt.

Problem ve neden zor olduğu

Bölüm 6, **yan kapıyı** primary'nin erişim kontrollerini aşan bir veri kopyası ya da export yolu olarak tanımlamıştı. Replica'lar ve analytics kopyası, bilerek kurduğunuz ve adını koyabildiğiniz kapılardır. Bu bölüm geri kalanı anlatıyor: backup'lar ve snapshot'lar, dump'lar, CSV dosyaları, staging veritabanları, bir laptop'taki birkaç bin satır, bir dashboard cache'i, bir vendor'ın gelen kutusundaki bir dosya. Çoğu iyi bir sebeple yapıldı. Sahibi olanı azdır.

Asıl güçlük, erişim kontrollerinin veride değil bir sistemde yaşamasıdır. Primary'de bir access gateway, session başına bir login, kolon düzeyinde grant'ler, timeout'lar ve bir audit log var. Primary'nin bir dump'ında bunların hiçbirisi yok, çünkü dump bir dosyadır. Dosyayı okuyabilen herkes içindeki her şeye sahiptir; rol şemasının `human_breakglass` dışındaki her insan rolünden sakladığı kolonlar da buna dahildir. Restricted kolonları tutan bir backup, onu okuyabilen kişiler için o kolonlara verilmiş kalıcı bir grant'tir; bu grant'i hiçbir gateway vermemiştir ve hiçbir review listelemez.

Bariz cevap bir politikadır: production verisini kopyalamayın. Bu, basit bir sebepten dolayı işe yaramaz. İş kopyalayarak yürür. Kurtarma tanımı gereği bir kopya ister. Bir vendor'ın dosyalara ihtiyacı vardır. Gerçek veri şekillerine bağlı bir bug, gerçek görünen bir şey ister. Bunların hepsini yasaklayan bir politikanın etrafından dolaşılır ve ortaya çıkan kopyalar yönetilmeyenler olur. Daha iyi soru şudur: hangi kopyalara ihtiyaç var, her birinin sahibi kim, kim okuyabilir ve ne zaman biter.

İkinci bir güçlük, bu kapıların farklı ekipler tarafından farklı sebeplerle açılması ve hiçbir tek sahibin hepsini görmemesidir. Platform ekibi backup'ların, data ekibi analytics kopyasının, backend ekibi staging'in sahibidir; vendor ilişkisi finansta, her laptop'un içeriği de o laptop'un mühendisindedir. İlk kontrol bir envanterdir ve o envanter bir politika dokümanında değil, storage bucket'larında, job zamanlamalarında ve konfigürasyonda bulunur.

Yan kapıların bir haritası

Kontrollerden önce bir harita. Tablo, bu bölümün ele aldığı kapıları, her birinin Ledgerly'de ne tuttuğunu ve kimse başka türlü tasarlamadıysa varsayılan olarak kimin ona eriştiğini listeliyor.

Kapı	Ne tutuyor	Varsayılan olarak kim erişir	İlk kontrol
Backup'lar ve volume snapshot'ları	Her şey; Restricted kolonlar ve secret tabloları dahil	Storage ya da altyapı erişimi olan herkes	Alma, restore ve silme yetkilerini ayırmak; kendi anahtarıyla encryption
Paylaşımlı depolamadaki dump'lar	Her şey, okunabilir bir dosya olarak	Klasörü açabilen herkes	Bul, bir sahip ata, backup kuralları uyarınca sil ya da taşı
İnsan export'ları (CSV, yapıştırılmış tablolar)	Sorgunun döndürdüğü ne varsa	Önce kişi, sonra kişinin gönderdiği herkes	Katmanlı onay, süre dolumu, Restricted veri yok
Staging ve test veritabanları	Production'ın bir kopyası ya da benzeri	Her mühendis, CI, yükleniciler	Maskelenmiş ya da sentetik veri, giden mesajlar kapalı
Laptop'lar	Sonuç setleri, yerel veritabanları, kaydedilmiş dosyalar	Laptop'un kullanıcısı ve backup'ları	İhtiyacı ortadan kaldır; yerine maskelenmiş örnekler
BI araçları	Cache'lenmiş extract'lar, zamanlanmış raporlar, paylaşılan dashboard'lar	Aracın paylaşıldığı herkes	Yalnızca analytics kopyasını oku, mümkünse kişi başına kimlik
Vendor dosyaları	Fatura ve defter export'ları	Vendor'ın çalışanları ve sistemleri	Dar rol, kolon listesi, retention, dosya süre dolumu

Her satırdaki problem aynı şekilde sahip: bir dosya sisteminin ya da bir SaaS ürününün kontrolleriyle duran bir veri kopyası ve kimsenin yazıya dökmediği bir okuyucu listesi. Bölüm 6'daki register tam da bu sebeple zaten var ve ben onu yukarıdaki her satıra genişletirdim. Register'da olmayan bir kapı, kimsenin review etmeyeceği bir kapıdır.

```

table data_copy:                                // extended from Chapter 6
  name      text primary key                    // "staging_refresh_current"
  kind      text not null                      // backup, dump, export,
staging, laptop,                                // bi_extract, vendor_file,
replica, analytics
  source     text not null                      // what it was made from
  contains   text not null                      // highest data class present
  readers    text not null                      // roles, logins or groups
  storage    text not null                      // where the bytes live
  owner      text not null                      // a team, never a person
  ends       timestamp null                    // null only for replicas and
analytics
  review_due timestamp not null                 // when the row is next reviewed
  deleted_by text null                         // who removed it, once removed

```

Yeni alanlar `storage` ve `deleted_by`; `kind` ise `backup`, `laptop`, `bi_extract` ve `vendor_file` değerlerini kazanıyor, `contains` de mevcut en yüksek veri sınıfını kaydediyor. İlk yeni alan, birisi kopyayı neyin okuyabildiğini sorduğunda nereye bakılacağını söyler, çünkü cevap genellikle bir veritabanı rolü değil, bir storage izni olur. İkincisi döngüyü kapatır: kimsenin işlemediği bir `ends` tarihi bir dilekten ibarettir, kopyayı kimin sildiğinin kaydı da o tarihin bir anlam taşıdığının kanıtıdır.

Backup'lar ve snapshot'lar

Backup, kimsenin karşı çıkmadığı tek yan kapıdır ve bu mutabakat, çoğu zaman kontrolsüz bırakılmasının sebeplerinden biridir. Her şeyi tutmak zorundadır, çünkü işi her şeyi geri getirmektir. Tasarımın tamamı budur ve açığın tamamı da budur.

Bir backup ne tutar

Backup'lar birkaç türde gelir. **Fiziksel backup** ya da volume snapshot'ı, veritabanının dosyalarını kopyalar; çoğu zaman da değişikliklerin sürekli bir arşivini tutar ki veritabanı belirli bir ana restore edilebilsin. **Mantıksal dump** ise veriyi veritabanı üzerinden okur ve statement ya da satır olarak yazar. İkisi de her tabloyu tutar. Fiziksel tür ayrıca cluster'ın rol tanımlarını ve engine'in onlar için sakladığı credential malzemesini de tutar; bazı mantıksal araçlar da kullanım biçimine göre bunu yapar.

Bundan iki sonuç çıkar ve ikisi de primary'nin bir diyagramında görünmez. Birincisi, Restricted kolonların backup'ta olması, onları kullanan engine'lerde secret tablolarının da orada olmasıdır (Bölüm 3'teki `bank_account_secrets` ve `payment_token_values`). Rol şemasının, bu değerlere yalnızca uygulama rollerinin ve `human_breakglass`'in eriştiği yönündeki kuralı veritabanını anlatır. Dosya hakkında hiçbir şey söylemez.

İkincisi, backup'ın alındığı günün durumunu tutmasıdır. Ortak admin login'i emekli edilmeden önce alınmış bir backup, o login'i hâlâ içerir. Sonradan ekip değiştirmiş kişilerin rol üyeliklerini tutar. Rotation ve revoke canlı sistemde olur, halihazırda var olan dosyalara geri uzanmaz. Biri primary'de bir rolü daralttığına geçen yılın restore'unda hiçbir şey olmaz: primary iyileşir, kopyalar yapıldıkları günün durumunu korur.

Restore bir okumadır

Backup tasarımının en başına tek bir cümle koyardım: restore etme yetkisi, her şeyi okuma yetkisidir. Bir backup'ı kontrol ettiği bir yere restore edebilen kişi, orada owner olarak login olup her kolonu okuyabilir; production'daki hiçbir gateway, kolon grant'i ya da maskeleyme kuralı geçerli olmaz, çünkü okuma production'da gerçekleşmez.

Bu, kontrolün kime ait olduğunu değiştirir. Backup erişimi genellikle altyapı rolleriyle verilir: biri bucket'ı okuyabilir, snapshot'ı mount edebilir ya da restore aracını çalıştırabilir. Bu roller, `human_readonly`'nin üstünde dursalar da nadiren onunla aynı özenle review edilir. Bir erişim review'u için soru bu yüzden yalnızca "primary'yi kim sorgulayabilir" değil, "kim restore edebilir ve nereye" sorusudur.

Üç yetkiyi ayırır ve farklı principal'lara verirdim:

Yetki	Anlamı	Kimde
Alma	Backup oluşturmak ve store'a yazmak	Backup job'ının kendi kimliği, store'a yalnızca yazma
Restore	Bir backup'ı okuyup bir yerde ayağa kaldırmak	Platform ekibi, istek başına, onaylayan ikinci bir kişiyle (four-eyes)
Silme	Backup'ları kaldırmak ya da süresini doldurmak	Retention süreci; insanlar yalnızca review edilmiş bir istekle

Ayrımın amacı, backup yazan job'ın onları geri okuyamaması ve hiç kimsenin hem bir backup'ı restore edip hem de kurtarma noktalarını silememesidir; böylece bir restore'u, restore edilen şeyin kaldırılması izleyemez. Storage belirli bir süre için write-once retention destekliyorsa aynı sebeple onu kullanırdım, bir bedelle: süre bitene kadar hata sonucu aldığınız bir backup'ı, örneğin içinde kısa süreliğine olmaması gereken bir şey bulunan bir veritabanından alınan, silemezsiniz. Bu durumun nasıl ele alınacağına önceden karar verin.

Encryption ve anahtar kimde

Backup'lar at rest şifrelenmelidir ve ilginç soru, anahtarı kimin tuttuğudur. Aynı administrator hem bucket'ı okuyup hem de anahtarı kullanabiliyorsa encryption yalnızca kaybolan bir diske karşı korur, başka bir şeye karşı değil. Anahtar storage administrator'larından farklı bir principal'da duruyorsa ve restore ikisini de gerektiriyorsa, storage ekibi tek başına veriyi okuyamaz, anahtarın sahibinin de tek başına okuyacak bir şeyi yoktur.

Bu ayrımın maliyeti belirli: restore artık iki tarafı gerektirir ve anahtarı kaybolmuş bir backup artık backup değildir. Bir anahtar kurtarma yolunun var olması ve test edilmesi gerekir; bu da platform ekibinin sürdüreceği bir prosedür daha demektir. Bölüm 12, uygulama düzeyinde encryption'ı ve anahtar yönetiminin sınırlarını ele alıyor; burada daha dar olan nokta, backup encryption'ının restore üzerinde bir gate olduğu ve bir gate'in ancak storage'ın gate'inden farklıysa değer taşıdığıdır.

Encryption, restore etmesine izin verilen kişiye karşı yardımcı olmaz. Dosyayı, restore yolunun parçası olmayan kişilerden ve sistemlerden transit ve at rest korur. Bunu tasarım dokümanında açıkça söyledim, çünkü "backup'lar şifreli" çoğu zaman "backup'lar güvende" diye duyulur ve bunlar farklı iddialardır.

Bir restore isteği

Restore, backup'ın canlı bir veritabanına dönüştüğü andır; kontrol de oraya aittir. Aşağıdaki blok, işlemlerin sırası yazılmış bir restore isteğini gösteriyor. Backup store'u, register ve audit log ayrı sistemlerdir; dolayısıyla hiçbir transaction hepsini kapsamaz.

```

function restoreBackup(request):
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.target not in isolatedTargets:
        raise TargetNotIsolated
    if request.ends is null or request.ends > now() + maxRestoreLife:
        raise RestoreNeedsEnd

    // the register comes first: a copy that exists with no record is
a side door
    registerCopy({ name: request.target, kind: "backup", source:
request.backup_id,
                    contains: "Restricted", readers: "none until access
applied",
                    owner: request.owner_team, ends: request.ends })

    try:
        auditLogAppend({ actor: request.approver, action: "restore",
                        subject: request.backup_id, target:
request.target })
        on AuditLogUnavailable:
            raise RestoreRefused

    // set before the copy exists, so a failure part-way still ends;
if this
    // call fails, the sweeper on `ends` removes the copy all the same
scheduleDeletion(request.target, at: request.ends)
    // reachable only by the restore tooling until the last step
startRestore(request.backup_id, into: request.target, reachable:
false)
    // every login returns with its backup-day credential, the
superuser and
    // ledgerly_access_admin included: nothing may connect before this
is done
    disableAllLogins(on: request.target)
    rotateSuperuserCredential(on: request.target)
    // the backup already holds a login with this name: drop it and
create it
    // again with a new credential, so the backup-day one never works
replaceLogin("ledgerly_access_admin", on: request.target,
credential: "new")
    applyAccessDefinition(on: request.target)
    openTarget(request.target, readers: request.readers)

```

Süreç `registerCopy`'den sonra ve restore başlamadan önce durursa, `register`'da var olmayan bir kopya için bir kayıt kalır; haftalık bir kontrol bunu bulup kaldırır. Restore başladıktan sonra durursa kopya vardır ve erişilemezdir, zamanlanmış silmesi de onu kaldırır; zamanlama bile başarısız oluyorsa `register`'daki `ends` üzerinde işleyen sweeper onu bulur. Ters sıra, var olan ama listelenmemiş bir kopya bırakırdı ve önemli olan

hata budur. Tasarımın maliyeti gecikmedir: biri tek bir satır görmeden önce bir onay ve izole bir hedef gerekir. Bir olay sırasında bu gecikme gerçektir; bu yüzden restore yolu, onay önceden ayarlanarak prova edilmelidir; Bölüm 15 bunu olaylar için ele alıyor. Amaç bütün bir veritabanı değil birkaç satırı kurtarmaksa, restore yine aynı izole hedefe yapılır, satırlar oradan alınır ve production'a Bölüm 10'un kuralları uyarınca bir veri düzeltmesi olarak geri girerler.

Restore edilen kopya kapalı başlar

Restore edilen bir veritabanı, backup gününün rollerini, üyeliklerini ve credential malzemesini geri getirir. Access definition yeniden uygulanana kadar onu düşman sayardım. Pratikte bu, servislerden hiçbir ağ rotası ve hiçbir insan login'i olmayan bir restore hedefi demektir; ilk adım da backup'ın geri getirdiği her login'i devre dışı bırakmaktır, superuser ve `ledgerly_access_admin` dahil, çünkü her biri backup gününde sahip olduğu credential'la geri döner. Sonra superuser credential'ı rotate edilir ve ancak ondan sonra Bölüm 3'teki access definition uygulanır: roller, üyelikler ve grant'ler, kodun bugün tarif ettiği hâliyle. Kopya bundan sonra erişilebilir olur, öncesinde değil.

Bu adım bir access administrator gerektirir ve bu, hedefe ait olan bir administrator olmalıdır. Backup zaten `ledgerly_access_admin` adlı bir login tutuyor ve o, backup'ın alındığı günün credential'ıyla geri geldi. Restore onu değiştirir: geri dönen login silinir ve aynı adla, yeni bir credential'la yeniden oluşturulur; bu credential restore için yapılır ve onunla birlikte kaldırılır. Production'ın bu rol için güncel credential'ı hiçbir zaman hedefe kopyalanmaz, backup'taki ise herhangi bir şey bağlanmadan önce gitmiştir; çünkü ikisinden birine sahip olan, yönettiği roller içinde her şeyi grant edebilir ve bir restore tatbikatı, credential'ların bir test ortamına yapıştırıldığı yerin ta kendisidir. Restore edilen kopyanın kendi administrator'ı ele geçirilirse blast radius o kopyadır.

Kopya ancak bundan sonra okuyucu alır ve okuyucular restore isteğindeki kişilerdir. Bir tatbikat için yapılan restore'un hiç insan okuyucuya ihtiyacı yoktur: tatbikat verinin ayağa kalktığını ve sayıların tuttuğunu kontrol eder, sonra kopya silinir.

Retention ve silme

Retention, iki maliyet arasında bir takastır. Backup'ları daha uzun tutarsanız geç fark edilen problemlerden kurtulabilirsiniz; bedeli, primary'nin çoktan sildiği müşteri verisi dahil her şeyi tutan daha büyük bir veri yığındır. Daha kısa tutarsanız daha azını tutarsınız ve bazı bozulmalar en eski backup'ınızdan daha eski olur. Şirketiniz için doğru süreyi bilmiyorum. Sessiz bir problemin genellikle ne kadar sürede fark edildiğine, kayıp bir aylık verinin ne kadara mal olacağına ve hukuk ile uyum ekibinizin backup'larda yaşamaya devam eden silinmiş veri hakkında ne dediğine bağlı. GDPR ya da KVKK gibi gizlilik rejimleri, bir kişinin verisi canlı sistemden silindikten sonra ne olduğunu sormaya meyillidir ve yorum onlara aittir, bu kitaba değil. Mühendisliğin sağlayabildiği şey uyum değil, kanıttır: sistemin ne yaptığını gösteren bir register ve bir silme kaydı.

Onlarla konuşmaya değer bir tasarım şu: backup'lar zamanlamaya göre eskir ve bir restore prosedürü, backup alındığından beri kaydedilmiş silmeleri yeniden uygular; böylece restore edilen bir kopya, bilerek kaldırılmış veriyi geri getirmez. Bu, her restore'a bir adım ekler ve silmelerin bir restore'un danışabileceği bir yere kaydedilmesini gerektirir; bu da silme isteklerini kalıcı bir kayıta tutmanın bir başka sebebidir. Adım, access definition uygulandıktan sonra ve kopya açılmadan önce çalışır.

```
function reapplyDeletions(target, backupTakenAt):
  // by execution time: a deletion recorded before the backup but
  run after
  // it is not in the backup's state, so it must be applied again
  for each request in deletionRecord.executedSince(backupTakenAt):
    auditLogAppend({ actor: "restore-tooling", action:
      "reapply_deletion",
                      subject: request.id, target: target }) //
  record first
    deleteRows(on: target, subject: request.subject_id)
  // a failure here leaves the copy closed: openTarget has not run
  yet
```

Tatbikatlar kopya üretir

Hiç restore edilmemiş bir backup'ın çalıştığı gösterilmemiştir. Restore tatbikatları, backup'ların çalıştığını öğrenme yoludur ve her tatbikat Restricted veri içeren bir veritabanı yaratır. Tatbikatları gerçek bir

restore'la aynı istek yolundan, aynı izole hedefle, aynı register kaydıyla ve gün içinde bir bitiş tarihiyle çalıştırdım. Tatbikat aynı zamanda olayın provasıdır: bitmesi için bir kestirme gerekiyorsa, o kestirme sabaha karşı üçte kullanılacaktır.

Uyarı: Restricted kolonları tutan ve geniş bir grubun okuyabildiği bir storage'da duran bir backup ya da dump, o kolonlara verilmiş, gateway'siz ve log'suz kalıcı bir grant'le aynı açıktır. Birini bulursanız önce storage iznini kısıtlayın, retention'ı sonra konuşun.

Dump'lar, export'lar ve CSV dosyaları

Backup'lar, bir amacı ve bir ekibi olan kapıdır. Dump'lar ve export'lar ise çoğunlukla zaman baskısı altındaki tek bir kişinin açtığı kapılardır ve farklı ele alınması gereken iki türe ayrılır: zamanlanmış olarak gerçekleşen export'lar ve bir kişinin yaptıkları.

Elle alınan dump'lar

Ledgerly'nin paylaşımlı klasöründe birkaç eski dump var. Her biri, birinin bir saatliğine bir kopyaya ihtiyacı olduğu için yapıldı ve kopya kaldı. Bir kişinin dump'ı; retention'ı, encryption politikası ve sahibi olmayan bir backup'tır. Dump'ın veritabanı üzerinden okuduğu engine'lerde ayrıca, birinin her tabloyu okuyabilen bir login'e sahip olduğunun kanıtıdır.

Üç şey yaptım. Birincisi, dump'ı normal araçlar kümesinden çıkarırdım: access gateway bütün veritabanı dump'ı sunmaz, insan rolleri bir dump'ın ihtiyaç duyacağı tabloları okuyamaz ve `human_breakglass`, Bölüm 8'deki break-glass logging setiyle bunu yapabilecek tek insan rolüdür. İkincisi, mevcut dump'ları bulurdum ve hiçbirini silmeden önce birinin tek backup'ı olup olmadığına bakardım. Bu olur: bir dump, kimsenin kopyalamadığı bir tablonun backup'ı hâline gelir. Önce gerçek backup'ları kurun, sonra dump'ları silin. Üçüncüsü, dump'a ihtiyaç duyan kişiye işi gören bir alternatif verirdim; bu genellikle maskelenmiş analytics kopyası ya da bir test veri setidir (aşağıdaki test verisi bölümüne bakın).

İnsan export'ları

İnsanların yaptığı export'lar en yaygın olanlar ve yasaklaması en zor olanlardır. Destek, tek bir müşterinin faturalarının listesine ihtiyaç duyar. Finans, geçen ayın ödemelerinin bir dosyasına ihtiyaç duyar. Data ekibi bir örnek ister. Onları dosyadaki en yüksek veri sınıfına göre katmanlara ayırırdım; çünkü hepsine uyan tek bir kural ya tehlikeli olanlar için fazla gevşek ya da zararsız olanlar için fazla ağır olur.

Katman	İçerik	Kontrol
Yalnız Internal	Aggregate'ler, sayımlar, durum kırılımları, kişisel içerik yok	Self-service, loglanır, onay yok
Confidential içerir	Müşteri iletişim bilgisi ya da serbest metin içeren satırlar	Gerekçeli ticket, bir onaylayan, dosya için bir bitiş tarihi; satırlar kolonları adlandıran bir JIT <code>SELECT</code> session'ından gelir (Bölüm 8), böylece onaylayan dosyanın tutacağı kolon listesini okur
Restricted içerir	Banka hesabı ya da token değerleri	Export edilmez; cevap bir görüntüleme biçimi ya da bir uygulama yoludur (Bölüm 12)

Katman, isteyen tarafından beyan edilmez, gateway'de hesaplanır ve kapalı yönde başarısız olur: sorgunun okuduğu kaynak kolonlar arasındaki en yüksek katalog sınıfıdır (Bölüm 12), sonuç kolonlarının adları arasındaki değil; çünkü bir alias ya da `lower(contact_email)` AS x gibi bir ifade kaynağı gizler. Kaynağı bir view, bir CTE ya da bir fonksiyon üzerinden çözilemeyen kolon, export edilebilir en yüksek katman sayılır; bu da kararlaştırılamıyorsa export reddedilir. Bölüm 9, statik analizin sınırlarını ele alıyor. Bu katmanlar bir şeklin örneğidir. Önemli olan, onayın riske bağlanmasıdır. Her seferinde, her şey için saniyeler içinde verilen bir onay, Bölüm 8'in anlattığı tiyatrodur. İnsan onaylayan gerektiren katman, onaylayanın isteği okuyacağı kadar küçük olmalıdır ve bunu, hiç kimsenin hayır deyip demediğiyle ölçerdim.

Ekiplerin atladığı kısım bitiş tarihidir. Bitiş tarihi olmayan bir export, sorusundan uzun yaşayan bir kopyadır. Dosyayı onaylayan ticket bir tarih taşır, dosya sahibi olan bir yere gider ve bir sweeper onu kaldırır. Bölüm 10'un ad-hoc işler kuralı en hafif durumu kapsıyor: kişisel içerik taşıyan

satırları değil, tanımlayıcıları ve sayıları yapıştırın. Bölüm 7, gateway'in toplu bir sonuç seti için yapabileceklerini anlatıyor: onu sınırlamak ve görünür kılmak, engellemek değil.

Vendor export'u

Harici muhasebe vendor'ı hiçbir Ledgerly veritabanına bağlanmaz. Dosya alır; bu da export adımını baştan sona tasarlayabileceğiniz bir kapı yapar, çünkü yolun iki ucunu da vendor'ın gelen kutusuna kadar siz kontrol edersiniz. Başlangıç durumunda `reporting-job`'ın export adımı bu dosyaları ortak admin login'ile yazar; dolayısıyla dosya, admin rolünün okuyabildiği her şeyi içerebilir. Hedef durumda `app_vendor_export` kullanır ve bu, dosyanın tutabileceklerini değiştirir, çünkü bir dosya, onu yazan login'in okuyabileceğinden fazlasını içeremez.

```
// illustrative column lists: the vendor's real needs decide them
GRANT SELECT (id, customer_id, total_minor, currency, status, due_on,
created_at)
    ON invoices TO app_vendor_export;
GRANT SELECT (id, amount_minor, currency, account_code, created_at)
    ON ledger_entries TO app_vendor_export;
```

Kontrol, sorgu değil grant'tir. Export sorgusundaki bir hata satırların eksik kalmasına ya da tekrarlanmasına yol açabilir, ama `contact_email`'i ekleyemez, çünkü rol onu okuyamaz. `customer_id` opak bir tanımlayıcıdır ve vendor'ın onu bağlayacağı bir tablosu yoktur. Vendor'ın işi müşteri adları gerektiriyorsa, bu birinin verip onayladığı bir karardır; bir sorgu genişletildi diye kendiliğinden gelmez.

Export adımı, Bölüm 6'daki bağlantı politikasının dediği gibi `ledgerly_replica_2`'ye bağlanır ve bir dosya yazar. Dosya sonra, onlarla anlaşılan kanaldan vendor'a gider. Adımlar üç ayrı sistemi içerir; bu yüzden aşağıdaki blok sırayı ve her hatanın ne bıraktığını gösteriyor.

```

function runVendorExport(period):
    file = writeExportFile(period, as: "app_vendor_export") // local,
not yet shared
    manifest = { file: file.name, rows: file.rowCount, checksum:
file.checksum,
                columns: file.columnNames, period: period }

    // the record exists before the file leaves: a sent file with no
    // record is the failure to avoid
    registerCopy({ name: file.name, kind: "vendor_file", source:
"ledgerly_replica_2",
                contains: "Internal", readers: "accounting vendor",
                owner: "platform", ends: now() + fileLife })

    try:
        sendToVendor(file, manifest)
    on SendFailed:
        // nothing has left; the file stays local and is removed with
its end date
        notify("platform", "vendor export not delivered: " +
file.name)
        raise ExportNotDelivered

    // the file has already left: a log outage delays the entry, never
the export
    auditOrQueue(manifest)

```

Buradaki `auditOrQueue`, kaydı audit log'a ekler ve log'a ulaşamıyorsa yeniden deneme için kuyruğa alıp platform ekibine bildirir; Bölüm 7'deki gibi. Log'un hatası sessiz değil, görünür ve yeniden denenen bir boşluğa dönüşür; log ile vendor kanalını kapsayan hiçbir transaction yoktur.

Manifest, insanların dışarıda bıraktığı kısımdır. İki tarafın da dosyanın kastedilen dosya olduğunu kontrol etmesini sağlar ve bir erişim review'una okuyacak bir şey verir: şu kadar satır, şu kolonlar, şu dönem. Bir kolon, geçen ayın manifest'inde olmadığı hâlde bir manifest'te görünüyorsa, bu birinin onaylamış olması gereken bir değişikliktir.

İki sınır kalıyor. Dosya vendor'a ulaştıktan sonra retention'lı bir sözleşme ve güven meselesidir: vendor'dan silmeyi teyit etmesini isteyebilirsiniz, ama doğrulayamazsınız. Ve checksum, dosyanın transit sırasında değiştirilmediğini gösterir; vendor'ın çalışanlarının onu okuyan tek kişiler olduğunu değil. Bu ikisi için sözleşmeye bir retention maddesi ve erişim review'una vendor'a en son ne zaman silmeyi teyit ettirildiğini soran bir satır koyardım.

Test verisi: staging, maskelenmiş kopyalar ve sentetik veri

Staging, Ledgerly senaryosunun başladığı yerdir ve çoğu zaman iyi niyetle açılan kapıdır. Mühendisler gerçekçi veri istiyor ve production, gerçek müşterilerin yarattığı şekillere sahip tek veri kaynağı. Bedeli, staging'in bir test sisteminin erişim modeline ve bir production sisteminin verisine sahip olmasıdır.

Production dışı verinin üç türü

Tür	Gerçekçilik	Maruz kalma	Maliyet
Production kopyası	En yüksek	Production'la aynı, daha zayıf kontrollerle	Yapması ucuz, sahiplenmesi pahalı
Maskelenmiş kopya	Yüksek, bazı şekiller kaybolur	Daha düşük, sıfır değil	Kurulacak ve bakımı yapılacak bir maskeleme pipeline'ı (Bölüm 12)
Sentetik veri	Daha düşük, modellenenle sınırlı	En düşük	Bakımı yapılacak bir generator ve modellenmemiş şekiller gerektiren bug'lar

Production verisini maskeleymeden staging'e kopyalayamazdım ve sebep soyut değil. Staging'in genellikle production'dan daha fazla okuyucusu, ortak bir login'i, log saklayan bir CI'ı ve production gibi davranacak şekilde yapılandırılmış entegrasyonları vardır. En geniş kitleyi en zayıf kimlik kontrolleriyle birleştirir ve en hassas veriyi tutar. Veri gerçek olmak zorundaysa kontroller production'ınki kadar güçlü olmalıdır ve o noktada staging, başka bir adla production'dır.

Maskelenmiş bir kopya hâlâ neyse o

Bölüm 12'de ele alınan maskeleme, şekli kullanılabilir tutarken değerleri değiştirir ya da bulanıklaştırır. Maskelenmiş kopya ham olandan çok daha iyidir ve yine de bir veri kopyasıdır; sebepleri, kimsenin "maskelenmiş"i "dağıtmak için güvenli" ile eşanlamı saymaması için söylemeye değer.

Birincisi, maskeleme yalnızca listelediğiniz kolonları kapsar.

`invoice_lines.description` gibi serbest metin, sınıflandırmanın hiç öngörmediği kişisel veri tutabilir; bu yüzden kitap onu en kötü durum için sınıflandırır ve onu atlayan bir maske, onu kopyada bırakır. İkincisi, şekli

koruyan maskeler kalan satırları ayırt edilebilir tutar: alışılmadık bir fatura tutarı, alışılmadık bir tarih ve küçük bir müşteri birlikte tek bir işletmeyi gösterebilir. Değerler maskelenmişti ve kombinasyon yine de bilgi taşıyor. Üçüncüsü, aynı girdinin aynı çıktıyı aldığı, böylece join'lerin çalışmaya devam ettiği tutarlı maskeleyme, aynı kişinin kopyalar ve tablolar arasında tanınmasına da izin verir.

Bunların hiçbirisi maskeleymenin yapmaya değmediği anlamına gelmez. Anlamı şudur: maskelenmiş bir kopya, geriye kalanı bir review edene kadar Confidential olarak ele alınır, kitlesi "bütün mühendislik"ten daha küçüktür ve her kopya gibi bir bitiş tarihi vardır; sürekli analytics kopyası istisnadır ve onun yerine `review_due` tarihinde review edilir. Bunun adı **yeniden tanımlama** (re-identification): bir kaydın kimin hakkında olduğunu geride bırakılmış alanlardan çıkarabilme yeteneği. Riski ekibe bu terimlerle anlatırdım, saldırı diliyle değil. Tasarım cevabı, daha uzun bir maskelenmiş alan listesi değil, daha küçük bir kitle ve bir süre dolumudur.

Sentetik veri

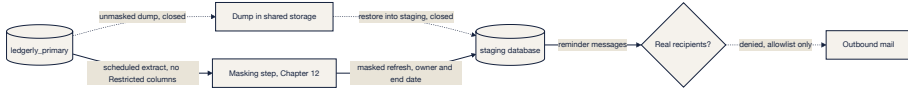
Sentetik veri hiçbir gerçek kişiyi tutmaz; bu yüzden en az maruz kalmayı taşır. Sınırı, yalnızca birinin modellemeyi akıl ettiği şekilleri tutmasıdır ve gerçek veri gerektiren bug'lar, kimsenin modellemediği bug'lardır. Derli toplu müşteriler üreten bir generator, altı iletişim kişisi olan ve iki dilde açıklaması bulunan müşteriyi üretmez.

Sentetik seti bug'lardan büyüttüm. Bir problemin yeniden üretilmesi için production şeklinde veri gerektiği her seferde, onu doğuran şekil, değerler değil, bir fixture olur: birden fazla iletişim kişisi olan bir müşteri, boş açıklamalı bir fatura satırı, ay sonuna denk gelen bir vade tarihi. Bir yıl içinde set derli toplu olmaktan çıkıp işe yarar türden tuhaflığa dönüşür. Böyle her bug'dan sonra bir mühendise bir saate mal olur ve bir sonraki production kopyasının ihtiyacını ortadan kaldırır.

Ledgerly'nin staging'i, yeniden

Robin'in cumasına dönelim. Kapıyı açan yol, paylaşımlı depolamadan staging'e bir dump'tı. Benim kuracağım yol onun yerine, maskeleyme adımından geçen ve kısa ömürlü, adı konmuş sahibi olan bir veritabanına

inen bir refresh koyar. Diyagramda kesik çizgili yollara bakın: tasarımın kapattığı yollar onlar.



Tasarımın sağ yarısı kolayca unutulur. Bir staging ortamı veri tutar ve ona göre hareket de eder. Staging'deki `invoice-worker` müşterileri okuyup hatırlatma gönderir ve giden ayarları production'la aynıysa, bunları verideki iletişim adreslerine gönderir. Maskelenmiş bir kopyada gerçek adres yoktur ve bu, tesadüfen bunu çözer. Ham bir kopya, bir test çalıştırmasını gerçek müşterilere giden mesajlara çevirir. Production dışı her ortamda giden yolu varsayılan olarak deny yapar ve veride ne olursa olsun yalnızca dahili test adreslerinden oluşan bir allowlist'e izin verirdim.

Payment token'ları da ilgili bir kuralı izler. Bir token, bir ödeme aracının yerini tutar ve onu veren sistemin dışında çalışıp çalışmadığı sağlayıcıya bağlıdır; sizinki için bunu ben yargılayamam. Varsayılan olarak `payment_tokens.token_value`, `bank_accounts.iban` ve `bank_accounts.account_holder_name`, maskelenmiş ya da maskelenmemiş, production dışı hiçbir kopyada bulunmaz (backup'lar istisnadır, çünkü her şeyi tutmak zorundadırlar) ve staging sağlayıcının test token'larını kullanır.

Not: Staging refresh'ine tek seferlik bir iş değil, bir hata yolu olan bir pipeline gibi davranın. Maskeleyme adımı başarısız olursa refresh durur ve staging önceki içeriğini korur. Bir çalıştırmalığına maskelenmemiş bir kaynağa düşen bir refresh, paylaşımlı klasördeki dump'ı ek adımlarla yeniden yaratır.

BI araçları, laptop'lar ve vendor'lar

Üç kapı daha kısa bir ele alışı hak ediyor, çünkü her biri veriyi veritabanından bir özellik gibi görünen bir yolla çıkarıyor.

BI araçları. Bir dashboard aracı tek bir login'le bağlanır ve dashboard'a erişimi olan herkes, o login'in okuyabildiğini görür. Bu, Bölüm 3'ün kimlik modelini tersine çevirir: veritabanı tek bir principal görür ve, diyelim, yüz

kişi onun verisini görür. İki kontrol yardımcı olur. Aracı bir replica'ya değil `ledgerly_analytics`'e yöneltin; böylece production rolleriyle değil, araç için kendi rolüyle, herhangi bir workload gibi bir `app_` rolüyle maskelenmiş veriyi okur (`human_analytics_reader`, data ekibinin insanlar için rolüdür). Ayrıca araç destekliyorsa her kişinin kimliğini veritabanına iletin; böylece satır ve kolon kuralları kişi başına uygulanır. Araç ayrıca kendi extract'larını, cache'lerini ve zamanlanmış e-posta raporlarını da tutar; bunlar register'a giren veri kopyalarıdır. Bir dashboard elverişli bir kapıdır ve cache'lenmiş bir extract, kontrolleri gitmiş aynı kapıdır.

Laptop'lar. Gateway, Bölüm 7'nin anlattığı gibi kayıtlı credential'ları kaldırır ve varsayılanları read-only yapar; ekrana ulaşan satırlarla kişinin ne yaptığını ise kontrol edemez. Pratik kontrol, yerel kopya tutma sebeplerini azaltmaktır: maskelenmiş bir örnek, production gibi davranan bir staging veritabanı ve bug'lara yol açan şekiller için fixture'lar. Bir ekip hiçbir laptop'ta production verisi olmadığını kanıtlayamaz ve ben bunu iddia etmezdim. Tam disk encryption'ı ve kaybolan bir cihazı silme yolu gibi bir taban, kaybolan bir laptop'un maliyetini azaltır ve bir taban olarak kalır, bir cevap değildir.

Diğer vendor'lar. Kitabın izlediği muhasebe vendor'ıdır, ötekiler ise veriyi yan etkiyle alır: bir request body'sini yakalayan bir hata izleyici, yapılandırılmış ekran görüntüleri alan bir destek masası, statement'ları saklayan bir log platformu. Kopya tutan her vendor, ne içerdiğini ve sözleşmenin ne zaman bittiğini gösteren bir register satırı alır. Bölüm 12, hassas değerleri log'lardan ve hatalardan uzak tutmayı ele alıyor; bu, onları bir vendor'ın store'undan uzak tutmanın en ucuz yoludur.

Ledgerly'de kapıları kapatmak

Alex'in Ledgerly'yi dağınık kopyalardan sahipli bir register'a nasıl taşıyacağını düşünün. Bölüm 6'daki okuma yolunda olduğu gibi sıra önemlidir ve altı adımda yapardım.

1. **Kopyaları envantere alın.** Backup'ları, snapshot'ları, dump'ları, export'ları, staging'i, BI extract'larını ve vendor dosyalarını storage, okuyucular ve bir sahiple listeleyin. Kimsenin hatırlamadığı kopyalar bulmayı bekleyin; register'ın ilk çalıştırması çoğunlukla keşiftir.

2. **Önce storage'ı kısıtlayın.** Restricted kolon tutan her şey için, neyin silineceğine karar vermeden önce storage'ı kimlerin okuyabileceğini daraltın. Bu hızlıdır ve geri alınabilir; gerisi çözülürken en geniş açığı kapatır.
3. **Backup yetkilerini kurun.** Backup job'ına yalnızca yazabilen bir kimlik verin, restore'u ikinci bir onaylayanın olduğu bir isteğin arkasına koyun, encryption anahtarını storage administrator'larından ayrı tutun ve istek yolundan bir tatbikat çalıştırın.
4. **Dump'ları emekli edin.** Gerçek backup'lar var olduğunda ve bir restore çalıştığında eski dump'ları silin ve her birini kimin sildiğini kaydedin. `ledgerly_analytics`'i besleyen gecelik maskelenmemiş restore, yerine Bölüm 12'nin `app_analytics_load` altındaki maskelenmiş yüklemesi çalıştığında biter; o zamana kadar en büyük açık kapıdır ve bir bitiş tarihi alır.
5. **Staging'i maskelenmiş ya da sentetik veriye taşıyın;** giden mesajlar varsayılan olarak kapalı olsun ve ortak staging login'ini kaldırın.
6. **Vendor export'unu kendi login'ine koyun.** Export adımını `app_vendor_export`'a taşıyın, manifest'i ekleyin ve bir dosya ömrü belirleyin.

Sonra register'ı zaten olan bir şeye bağlayın. Bölüm 14'ün erişim review'u hangi kopyaların var olduğunu ve her birini kimin okuyabildiğini sorar, bir kopya oluşturmak bir register satırı gerektirir ve bir sweeper, `ends`'ini geçmiş her şeyi, bir ek süre ve sahibe yapılan bir bildirimin ardından kaldırır. Restore istekleri ve export'lar da Bölüm 14'ün konusu olan audit log'a yazmalıdır. Restore ve export job'ları, onaylanmış yazmaları taşıyan runner'la aynı yürütme disiplini izler: kendi kimlikleri altında, onaylanmış bir istekten, kayıt önce yazılarak çalışırlar; yalnızca Internal katmanındaki bir export, yukarıdaki katman tablosunun self-service kuralını korur (bu disiplini Bölüm 9 anlatıyor).

Trade-off'lar ve maliyetler

İlk maliyet, bir olay sırasındaki hızdır. Onaylayan ve izole hedef gerektiren bir restore, bucket iznine sahip olan herkesin yaptığından yavaştır ve eklediği süre, en kötü anda on-call olan kişinin üzerine biner. Yolu prova

edin ve onayı önceden ayarlayın (Bölüm 15 olaylar sırasındaki erişimi ele alıyor); böylece kontrol insanların keşfettiği bir engel değil, bildikleri bir prosedür olur.

İkinci maliyet anahtar saklamadır. Backup anahtarını storage'dan ayırmak, kaybolan bir anahtarın kaybolan bir backup olması demektir ve anahtarın kurtarma yolu yazılması ve test edilmesi gereken bir prosedür daha demektir. Çok küçük bir ekip için bu ayırım getirdiğinden fazlasına mal olabilir; tek bir şifrelenmiş store, sıkı storage izinleri ve belgelenmiş bir anahtar kurtarma makul bir başlangıç noktası olabilir.

Üçüncü maliyet, daha yavaş bir debugging döngüsüdür. Mühendisler production'ı staging'e restore edemediğinde bazı bug'ların yeniden üretilmesi daha uzun sürer, çünkü onları tetikleyen şekil henüz test verisinde yoktur. Fixture alışkanlığı bunu zamanla geri öder ve başlangıçta gerçek bir masraftır. Maskelenmiş kopyalar, birinin sahip olduğu bir pipeline ekler ve bir kolon eklendiğinde bozulur; bu yüzden kolonu ekleyen migration, sınıfına karar verilen yerdir (Bölüm 11).

Dördüncü maliyet retention'dır. Daha uzun retention daha fazla geçmiş kurtarılabilir tutar ve daha fazla hassas veriyi daha fazla yerde tutar; daha kısa retention ikisini de tersine çevirir. Herkes için doğru olan bir değer yoktur ve birini seçmemiş bir ekip "sonsuz dek"i seçmiştir.

Son olarak, register ve sweeper bir overhead'dir. Birinin hâlâ ihtiyaç duyduğu bir kopyayı silen bir sweeper başlı başına bir olaydır; bu yüzden silme, bir bildirim ve bir ek süreden sonra gelmeli ve bir sahip, bir bitiş tarihini bir gerekçeyle bir kez uzatabilmelidir. Hiç açılmayan bir register hiçbir şeyi korumaz ve onu dürüst tutan şey erişim review'una bağlanmasıdır.

Hepsini ne zaman kurmamalı: tek veritabanı, bir avuç mühendisi olan ve adına değer Confidential verisi olmayan küçük bir şirketin katmanlı export onayına ya da bölünmüş bir anahtara ihtiyacı yoktur. Yine de envantere, backup'lar için kısıtlı storage'a ve production'dan hiçbir şeyin maskelenmeden staging'e inmeyeceği kuralına ihtiyacı vardır. Bu üçünü yapın, daha fazlasını yapmanızı neyin gerektireceğini yazın ve ekip ya da veri değiştiğinde o notu review edin.

Sık yapılan hatalar

- **Primary'ye erişimi daraltıp backup'larını görmezden gelmek.** Roller, bir gateway ve bir audit log yerinde, backup bucket'ı ise bütün altyapı grubunun okuyabildiği bir yer. Backup'lar "operasyon" olduğu için cazip gelir. Çare, restore'u kendi onaylayanı ve kendi log'uyla her şeyin okunması olarak ele almaktır.
- **"Şifreli"nin "güvenli" demek olduğuna inanmak.** At rest encryption dosyayı restore yolunun dışındaki kişilerden korur ve anahtar, aynı administrator'ın elinde bucket'ın yanında durur. Anahtarı ayrı tutun ve neye karşı korumadığını söyleyin.
- **Gerçek backup'lar yokken eski dump'ları silmek.** Dump, birinin bir tablonun tek kopyasıydı ve onu kaldırmak tek geri dönüş yolunu kaldırdı. Önce backup'ları kurup test edin, sonra dump'ları emekli edin.
- **Eski login'lerin çalıştığı bir ağa restore etmek.** Restore edilen kopya, kendi gününün rollerini ve credential'larını, o zamandan beri emekli edilmiş olanlar dahil geri getirir. İzolasyona restore edin, access definition'ı yeniden uygulayın ve backup'ın geri getirdiği `ledgerly_access_admin` login'ini eski credential'lardan birini tutmak yerine yenisiyle değiştirin.
- **Maskelenmiş bir kopyayı herkese açık saymak.** Değerler maskelenmiş ve kopya geniş çapta paylaşılmış, sonra serbest metin ya da alanların nadir bir kombinasyonu tek bir müşteriye gösteriyor. Review edilene kadar Confidential olarak, daha küçük bir kitleyle ve bir bitiş tarihiyle ele alın.
- **Staging'i production'dan tek seferlik bir istisna olarak yenilemek.** Maskeleme adımı başarısız oldu, release bekliyordu ve dump orada duruyordu. Refresh'i kapalı yönde başarısız olacak şekilde yapın; böylece eski veri kalır ve hiçbir şey ham bir kaynağa düşmez.
- **Bir BI aracının production rolüyle bağlanmasına izin vermek.** Diyelim yüz izleyicinin arkasında tek bir login ve kimsenin listelemediği bir cache. Onu analytics kopyasına kendi rolüyle bağlayın ve extract'larını register'a alın.
- **Her export'u saniyeler içinde onaylamak.** Onay var ve hiçbir şeyi azaltmıyor. Onaylayanı bir karar hak eden katman için ayırın ve kimsenin hiç reddedip etmediğini izleyin.

- **Bir kopyaya sahip verip bitiş vermemek.** Register eksiksiz ve hiçbir şey ondan çıkmıyor. Her satırın bir `ends` tarihine ve üzerinde işleyen, önce bildirim yapan bir `sweeper`'a ihtiyacı var.

Temel çıkarımlar

- Yan kapı, veriyi geldiği sistemin kontrolleri olmadan taşır, çünkü erişim kontrolleri dosyada değil veritabanında yaşar.
- Bir backup'ı restore etme yetkisi, içindeki her şeyi okuma yetkisidir; bu yüzden backup erişimi en ayrıcalıklı insan rolüyle aynı `review`'u hak eder.
- Backup'ları alma, restore etme ve silme farklı `principal`'larda olmalı ve encryption anahtarı, test edilmiş bir kurtarma yoluyla birlikte `storage administrator`'larından ayrı tutulmalıdır.
- Restore edilen kopya, alındığı günün rolleri ve `credential`'larıyla geri gelir; bu yüzden izole başlar, `access definition` yeniden uygulanır ve kendi `administrator login`'ini alır.
- Bir `vendor export`'u sorguyla değil `app_vendor_export`'un `grant`'iyle kontrol edilir; böylece dosya rolün okuyamayacağı bir kolonu tutamaz ve bir manifest dosyanın kontrol edilmesini sağlar.
- Test verisi maskelenmiş ya da sentetik olmalı, maskelenmiş bir kopya `review` edilene kadar `Confidential` olarak ele alınmalı ve `staging`, giden mesajları varsayılan olarak `deny` etmeli ve maskeleye başarısız olduğunda kapalı yönde başarısız olmalıdır.
- Her kopyanın, erişimle birlikte `review` edilen bir `register`'da bir sahibi ve bir bitiş tarihi olmalı, çünkü kimsenin sahip olmadığı bir kapıyı kimse kapatmaz.

Kopyaların, `export`'ların ve `restore`'ların artık sahibi ve bir bitiş tarihi var, ama bir tasarım ancak izlendiğine dair kanıt kadar iyidir. Sonraki bölüm `audit`'e dönüyor: `audit log` nerede yaşar, içine ne girer ve bu kitabın durmadan andığı erişim `review`'ları gerçekte nasıl yürür.

KISIM V

Kanıtlamak ve İşletmek

Bölüm 14 — Audit ve Hesap Verebilirlik

Ledgerly'de bir salı sabahını düşünün. Bir müşteri, geçen ay şirkette kimin ödeme hesabına baktığını sormuş ve güvenlik irtibat kişisi soruyu Alex'e iletiyor. Basit bir sorgulama gibi görünüyor. Alex, access gateway'in session listesiyle başlıyor; listede Robin'in primary üzerinde kırk dakika süren bir break-glass session'ı olduğu görünüyor. Sonra veritabanı log'unu açıyor; orada bir session login'i var ama session'ın sebebi yok. Ardından uygulamanın yaptığı iş eylemlerini kaydeden ve bir kişinin ne yazdığı hakkında hiçbir şey söylemeyen `audit_events`'te arıyor.

Üç kaynak hikâyenin çoğunda hemfikir, geri kalanında değil. Gateway'in saati veritabanınınkinden iki dakika ileride. Bir session login'i veritabanı log'unda görünüyor ama hiçbir gateway kaydı yok ve bunun listenin unuttuğu bir servis login'i mi, yoksa kimsenin tasarlamadığı bir yoldan gelen bir bağlantı mı olduğunu kimse söyleyemiyor. Alex muhtemelen ne olduğunu söyleyebiliyor ama gösteremiyor; veritabanı administrator yetkisi olan hiç kimsenin kanıtı düzenlemediğini de gösteremiyor.

Bu bölüm tezin hesap verebilirlik ayağına hizmet ediyor ve olaylardan önceki hedef durumun son adımını ekliyor: erişim, erişilen sistemin yeniden yazamayacağı bir store'a loglanır ve belirli aralıklarla review edilir. Beş parçalı modelde (**Kimlik, Grant, Yol, Yürütme, Kayıt**) bu, Kayıt hakkındaki bölümdür. Kimlik kimin eylemde bulunduğunu söyler; Kayıt geriye kalandır ve statement'ı kişiye, isteğe ve sonuca bağlayıp bağlamadığıdır. Buradaki soru şu: tek bir statement geride ne bırakmalı, nerede durmalı ve kanıtın eksik olduğunu, biri sormadan önce nasıl öğrenirsiniz?

Problem ve neden zor olduđu

Başlangıç durumunda sorunun cevabı yok. Veritabanı log'u ortak bir rolü adlandırıyor, bir kişiyi değil ve administrator'ların kontrol ettiđi host'larda duruyor. Önceki bölümler kimlik yarısını düzeltti: session başına bir login, grant başına bir istek, onaylanmış yazmalar için bir runner. Her biri kayda aynı çağrıyla, `auditLogAppend` ile ekledi ve girdilerin nereye gittiđini söylemedi. Bu bölüm onu cevaplıyor.

Güçlük şu: Ledgerly'nin zaten üç kaydı var ve her biri farklı bir şey biliyor. Hiçbiri, kitabın tanımladıđı anlamda audit log değil: kimin hangi veriye ne yaptıđının, eylemde bulunanın deđiştiremeyeceđi bir yerde tutulan kaydı.

Kayıt	Ne biliyor	Size neyi söyleyemez	Kim deđiştirebilir
Gateway session kaydı	Kişi, istek, sebep, gönderildiđi hâliyle statement	Veritabanının gerçekte ne yaptıđı; başka bir yolun olup olmadığı	Platform ekibi, gateway'in kendi store'u üzerinden
Veritabanı log'u	Login, zaman, loglandıysa statement metni, hatalar	Login'in arkasındaki kişi, istek, sebep	Veritabanı host'unu yöneten herkes
<code>audit_events</code>	İş eylemi, eylemde bulunan principal, konu	Onu üreten statement'lar; uygulamanın yazmadıđı her şey	UPDATE ile hiç kimse, ama <code>ledgerly_owner</code> tabloyu bir migration'da deđiştirebilir

Bariz cevap, logging'i artırmaktır: veritabanındaki her statement'ı loglayın ve uzun süre saklayın. Bu dördüncü bir kayıt ekler ve üç boşluğun hiçbirini cevaplamaz. Erişilen sistemin kendisi hakkında yazdıđı bir log, ancak administrator'ları kadar güvenilir değildir ve kimsenin okumadıđı bir log tiyatrodur. Bir statement log'u başlı başına hassas verinin bir kopyasıdır, çünkü insanlar koşullara literal yazar; Bölüm 12 bu konuda uyarmıştı ve bu bölüm mekanizmanın sahibidir.

Kayıt, sistemdeki boşluğu bulmak için vardır, içinden geçen insanları sıralamak için deđil. Robin'in session'ında bir alert tetiklendiğinde ilk soru Robin'in neyi yanlış yaptıđı deđil, tasarımın neyi gerekli kıldıđıdır.

Bir statement, bir kayıt

Bir kayıt, ancak önceden yazılmış bir soru listesini cevaplayabiliyorsa işe yarar. Önce listeyi yazar ve alanları ondan türetirdim; çünkü gateway'in tesadüfen ürettiği log satırlarından tasarlanmış bir audit store, o log satırlarının tesadüfen izin verdiği soruları cevaplar.

Gateway, statement başına iki girdi yazar; çünkü Bölüm 9 niyetin statement çalışmadan önce, sonucun sonra yazılmasını şart koşar. İkisi birlikte **statement kaydı**nı oluşturur; iki girdi bir `statement_id` paylaşır ve ikincisi başlangıçta bilinmeyen alanları doldurur. Tek bir şekil ikisine ve diğer principal'ların yazdığı kayıtlara hizmet eder.

```
table access_audit_record:
  stream          text not null      // one per producer instance
  sequence        integer not null   // gapless within a stream
  previous_hash   text not null      // hash of the previous entry
  entry_hash      text not null
  kind            text not null      // "intent", "outcome", "event"
  action          text null          // events only: grant, revoke,
breakglass, data-fix, contain, and so on
  statement_id    text null
  session_id      text null
  login           text not null      // what the database log shows
  principal       text not null      // the person or workload
identity
  on_behalf_of    text null          // the human behind an
automation principal
  roles           text not null      // role set at the time of the
statement
  request_id      text null          // always access_request.id; a
data fix has one too, and its fix_id resolves through it
  reason          text null
  approver        text null
  statement_class text null          // the Chapter 9 class the
classifier assigned
  fingerprint     text null          // statement shape, literals
redacted
  literals_hash   text null          // keyed hash of the literal
values
  tables          text null          // tables touched
  columns         text null          // columns touched, by data
class
  rows_affected   integer null       // outcome only
  rows_returned   integer null       // outcome only
  outcome         text null          // ok, refused, error, rolled-
back, row-cap-closed
  started_at      timestamp not null
  ended_at        timestamp null
```

event türü, statement olmayan her şeyi taşır: bir session'ın açılması, bir grant, bir revoke, bir rotation, bir restore, bir migration başlangıcı. Bunlar Bölüm 3, 4, 7, 8, 10, 11, 12, 13 ve 15'teki `auditLogAppend` ve `auditOrQueue` çağrılarıdır ve `action` kolonu bir grant'i bir revoke'tan ya da bir sınırlamadan ayırır. Aynı zinciri ve aynı kimlik alanlarını kullanırlar; böylece tek bir sorgu bir kişiyi bir istekten bir grant'e, onu kullanan statement'lara ve revoke'a kadar izleyebilir.

Tablo, ancak insanların sorduğu soruları cevaplıyorsa yapmaya değer. İkinci liste, şemanın yanına asacağım listedir.

Soru	Cevaplayan alanlar
Bu statement'ı kim çalıştırdı?	<code>principal</code> , <code>on_behalf_of</code> , <code>login</code>
Hangi grant'ler ve onayla?	<code>roles</code> , <code>request_id</code> , <code>approver</code> , <code>reason</code>
Ne istedi?	<code>statement_class</code> , <code>fingerprint</code> , <code>literals_hash</code> , <code>tables</code> , <code>columns</code>
Ne yaptı?	<code>outcome</code> , <code>rows_affected</code> , <code>rows_returned</code>
Restricted veriye dokundu mu?	Veri sınıfına göre sınıflandırılmış <code>columns</code> , <code>rows_returned</code> ile birlikte
Ne zaman ve ne kadar süreyle?	<code>started_at</code> , <code>ended_at</code>
Çevresinde eksik bir şey var mı?	<code>stream</code> , <code>sequence</code> , <code>previous_hash</code> , <code>outcome</code> 'u olmayan <code>intent</code>
Bu girdi yazıldığı gibi mi?	<code>entry_hash</code> , aşağıdaki <code>verifyChain</code> ile kontrol edilir

İki alan yükün çoğunu taşır. `principal` ve `on_behalf_of`, Bölüm 9'un otomasyon için şart koştuğu gibi, izi bir makinede son bulmaktan alıkoyar. `request_id`, Bölüm 8 ve 10'un kaydettiği her şeye giden join anahtarıdır.

`request_id`'si olmayan bir kayıt yine geçerlidir. Rutin bir `human_readonly` session'ının isteği yoktur ve kaydı bunu alanı boş bırakarak söyler. `human_jit_write` üçlülerini ya da `human_breakglass`'i tutan bir session'da boş bir `request_id` bir bulgudur ve bu bölümün ilerisindeki kontroller onu arar.

Erişilen sistemin yeniden yazamayacağı bir store

Girdilerin yaşadığı yer, biçimlerinden daha önemlidir. Bölüm 1, observability'yi principal'ın değiştiremeyeceği bir yerde duran bir kayıt olarak tanımlamıştı. Burada önemli olan principal'lar en çok yetkisi olanlardır: veritabanı administrator'ları, gateway'i işleten platform ekibi ve üç login'i, yönettiği roller içinde hemen her şeyi grant edebilen `ledgerly_access_admin`. Bunlardan biri kaydı düzenleyebiliyorsa, kayıt yalnızca onların geride bırakmayı seçtiğinin kanıtıdır.

Store'u beş özellikle, şu sırayla kurardım.

1. **Ayrı bir hesap.** Audit store'u, veritabanı administrator'larının, gateway operatörlerinin ve üç `ledgerly_access_admin` login'inin hiçbir yetkisinin olmadığı bir cloud hesabında, projede ya da host grubunda yaşar. Birinin onu yönetmesi gerekir; bu, hiçbir veritabanı administrator yetkisi taşımayan küçük bir gruptur ve store'u kendi kullanımı, grubun dışındaki biri tarafından görülebilir.
2. **Yalnızca yazabilen bir producer.** Her producer (gateway, runner, credential broker, pipeline, revoke job'ı, restore tooling'i), tek bir işlemi, append'i çağırabilen kendi credential'ını tutar. Okuyamaz, güncelleyemez ya da silemez. Sızmış bir producer credential'ı sahte girdi ekleyebilir; bir zincir bunu tespit edilebilir kılar. Gerçek girdileri kaldıramaz.
3. **Zinciri tutan bir append servisi.** Producer'lar hash hesaplamaz. Audit hesabının içindeki küçük bir servis stream'in başını yükler, yeni girdiyi bağlar ve ikisini de saklar. Kendi bağlantısını hesaplayan bir producer, ondan sonrasının hepsini yeniden de hesaplayabilirdi.
4. **Hesap administrator'ının kısaltamayacağı bir retention.** Storage write-once retention kilidi sunuyorsa, retention süresi için onu kullanın; böylece bir girdiyi bitişinden önce silmek, hesaptaki hiç kimse için mümkün olmaz. Sunmuyorsa dürüst ifade, administrator'ın silebileceğidir ve bir silmeyi görünür kılan şey, zincir ve aşağıdaki anchor'dır.
5. **Bir anchor.** Append servisi günde bir kez her stream'in baş hash'ini, farklı bir grubun sahip olduğu ikinci bir yere gönderir: güvenlik irtibat kişilerinin sistemi ya da ikinci bir store. Anchor olmadan store'un tam kontrolüne sahip biri bir stream'i belli bir noktadan itibaren yeniden

yazabilir ve ondan sonraki her hash'i yeniden hesaplayabilir. Anchor ile, yeniden yazılmış baş anchor'dan farklı olur.

```
function appendChained(stream, record):
  // runs inside the audit account; producers can call only this
  // one writer per stream at a time: the head is read and locked
  // inside the same transaction that stores the next entry
  transaction:
    head = lockHead(stream)
    entry = { stream: stream, sequence: head.sequence + 1,
              previous_hash: head.hash, body: record,
              stored_at: now() }
    entry.entry_hash = hashOf(entry.stream, entry.sequence,
                              entry.previous_hash, entry.body)
    // one system, so one transaction is allowed: the entry and
the
    // new head exist together or not at all
    storeEntry(entry)
    storeHead(stream, entry.sequence, entry.entry_hash)
    // commit point: the entry is durable and the chain has moved
    return { sequence: entry.sequence, hash: entry.entry_hash }
```

İki session aynı anda aynı stream'e ekleme yapabilir; bu yüzden baş, transaction içinde bir lock altında okunur; onsuz ikisi de aynı başı okur, aynı sequence numarasını yazar ve zinciri çatallar; verifier da hiç olmamış bir kurcalanmayı raporlardı. Kural, stream başına aynı anda bir yazardır ve bu lock'la sağlanır.

Verifier, aynı hesapta read-only bir credential'lı olan ayrı bir job'dır. Onun işi, bir store'u kanıta dönüştüren kısımdır:

```
function verifyChain(stream, from, to):
  previous = loadEntryBefore(stream, from)
  for each entry in entries(stream, from, to):
    if entry.sequence != previous.sequence + 1:
      notify("security", "hole in " + stream + " at " +
entry.sequence)
    if entry.previous_hash != previous.entry_hash:
      notify("security", "broken link in " + stream)
    if entry.entry_hash != hashOf(entry.stream, entry.sequence,
                                  entry.previous_hash, entry.body):
      notify("security", "entry altered in " + stream)
    previous = entry
  // compare at the anchor's own position, not at the end of the
range
  anchor = loadAnchor(stream, atOrBefore: to)
  if anchor is not null and anchor.hash != entryAt(stream,
anchor.sequence).entry_hash:
    notify("security", "head differs from anchor in " + stream)
```

Şimdi dürüst sınır; bunu tasarım dokümanına ayrı bir cümle olarak koyardım. Bu tasarım kanıt verir, önleme değil. Bir statement'ın çalışmasını engellemez ve audit hesabı üzerinde yeterli yetkisi olan birinin store'u yok etmesini de engellemez. Değiştirmeyi tespit edilebilir kılar ve onu değiştirebilecek kişileri kısa, ayrı bir liste yapar. Tespit ayrıca birinin `verifyChain`'in çıktısını okumasına bağlıdır; bu yüzden verifier, bir sonraki bölümde ele alınan kendi heartbeat'i olan zamanlanmış bir job'dır.

Maliyet gerçektir. Audit hesabı ayakta tutulacak bir sistem daha ve Bölüm 8 ile 9 en kötü sonucunu zaten kabul etti: intent yazılamadığında bir grant, bir break-glass beyanı ya da bir statement reddedilir. Dolayısıyla audit store'unun kesintisi ayrıcalıklı işi bloke eder ve bunu, birinin bir acil durumda beklediği anda yapar. Bu, iyi niyete bağlı olmayan bir kaydın bedelidir.

Uyarı: Audit store'unu, kendi sahibi olan bir şemada bile, tarif ettiği veritabanının içine koymayın. O veritabanını yönetebilen herkes girdileri değiştirebilir ve yalnızca `INSERT` yapabilen bir rol bunu değiştirmez. Kitabın zaten listelediği tek yapısal istisna, yani `ledgerly_owner`'ın bir migration'da `ledger_entries` ve `audit_events`'i değiştirebilmesi, `audit_events`'in uygulamanın iş kaydı olması, audit log olmamasının sebebidir.

Boşlukları bulmak

Bir zincir, mevcut girdilerin değiştirilmediğini kanıtlar. Her statement'ın bir girdi ürettiğini kanıtlamaz. İkinci kontrol ailesi, var olması gerekip olmayana arar. Dört tür boşluk vardır ve her birinin anlamı farklıdır; bu yüzden kontrol onları ayrı ayrı raporlar.

Sequence delikleri. Bir stream yapısı gereği boşluksuzdur; dolayısıyla bir delik, append servisinin bir girdiyi kaybettiği ya da birinin birini kaldırdığı anlamına gelir. En nadir bulgudur ve en ciddi olanıdır.

Outcome'u olmayan intent'ler. Gateway bir statement'tan önce bir intent, sonra bir outcome yazar. Timeout'un izin verdiği en uzun statement'tan sonra outcome'u olmayan bir intent, gateway'in ikisi arasında başarısız

olduğu, outcome'un yeniden deneme kuyruğunda beklediği ya da statement'ın hâlâ çalıştığı anlamına gelir. Yalnızca üçüncüsü zararsızdır ve timeout'tan uzun yaşamamalıdır.

Yalnızca bir tarafta görünen session'lar. Her veritabanı login'inin bilinen bir kaynağı olmalıdır. İnsanlar için kaynak bir gateway session'ıdır. Servisler için Bölüm 4'teki credential broker'dan bir lease'tir. Migration'lar için pipeline'ın bir çalıştırmasıdır. Veritabanı log'unda üçünden hiçbiri olmayan bir login, kimsenin tasarlamadığı bir yoldur; veritabanının hiç görmediği statement'ları iddia eden bir gateway session'ı ise olmamış bir şeyi anlatan bir kayıttır.

Sessiz producer'lar. Yazmayı bırakan bir stream sakın bir haftanın, ölü bir producer'ın ya da birinin kapattığı bir producer'ın işareti olabilir. Kontrol her stream'i beklenen bir kadansla karşılaştırır.

```
function detectGaps(window):
    gatewaySessions = sessionsFromAuditStore(window)
    databaseLogins = loginsInDatabaseLog(window)
    for each login in databaseLogins:
        if login not in gatewaySessions
            and not hasBrokerLease(login, window)
            and not isPipelineRun(login, window):
                notify("security", "login with no known origin: " + login)
    for each s in gatewaySessions:
        if s.statements > 0 and s.login not in databaseLogins:
            notify("platform", "gateway session with no database
login: " + s.id)
        for each i in intentsWithoutOutcome(olderThan: maxStatementAge):
            if not inRetryQueue(i):
                notify("platform", "intent with no outcome: " +
i.statement_id)
        for each stream in expectedStreams():
            if now() - lastEntry(stream) > expectedCadence(stream):
                notify("platform", "stream silent: " + stream)
```

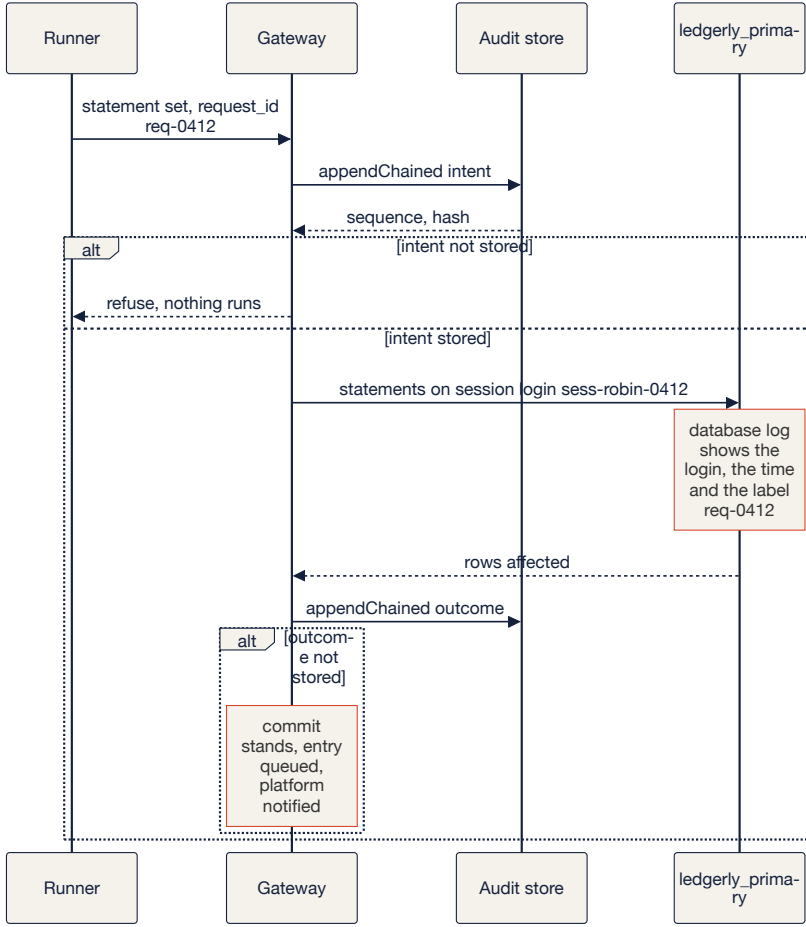
Veritabanı log'u, kendi yalnızca yazabilen credential'lı olan bir shipping agent aracılığıyla audit hesabına ulaşır. Host erişimi olan biri o agent'ı durdurabilir; sessiz-stream kontrolü bunun için vardır: durdurulmuş bir agent, girdisiz bir stream bırakır ve bu kendi bulgusunu doğurur.

Üç kaydı birleştirmek

Gateway kaydı, veritabanı log'u ve `audit_events`, hiçbir zaman bir transaction paylaşmayan üç sistem tarafından üretilir. Onları birleştirmek, Alex'in sabahındaki durumları tahminlerden bir zaman çizelgesine dönüştüren şeydir ve join anahtarları olaydan önce seçilmelidir, çünkü sonradan eklenemezler.

Üç anahtar işi yapar. **Login** veritabanı log'unda görünür ve bir session login'i tam olarak bir kişiye ya da bir workload instance'ına eşlendiği için, bir log satırını bir session kaydına bağlar. **Request id**, gateway tarafından audit kaydına ve runner tarafından audit store'unun kendi `request_id` alanına yazılır; statement'ı onu mümkün kılan onaya bağlar. Runner `audit_events`'e hiçbir zaman yazmaz; Bölüm 9'un matrisi bunu ona deny eder. Bölüm 3'ün getirdiği correlation id'yi taşıyan `audit_events` satırları, `admin-console`'un `payments-service`'i çağırdığı ürün yolundan gelir ve aynı isteğe o id üzerinden bağlanır. Bölüm 5'teki **bağlantı etiketi**, aracılı bağlantılar için request id'yi veritabanı log'una taşıyabilir. O etiketi bağlantıyı açan kişi koyar. Aracılı yürütmede bağlantıyı gateway, onaylanmış isteğin request id'siyle açar; bu yüzden güvenilirdir. Bir session handout'ta kişinin client'ı onu istediği şeye ayarlayabilir; bu yüzden anahtar login olarak kalır ve etiket bir ipucu olarak kalır.

Diyagram, onaylanmış tek bir veri düzeltmesini ve her kaydın nerede yazıldığını gösteriyor; runner primary'ye yalnızca gateway üzerinden ulaşır. Outcome yazımı başarısız olursa commit geçerli kalır ve boşluk tespit edilebilir.



Kaydın tutmaması gerekenler ve denetleyenleri kim denetler

Bir statement kaydı, insanların yazdığını tutar. Bir müşteriyi hesap numarasıyla arayan kişi, hesap numarasını statement'a koymuştur ve statement'ı saklayan bir log, Restricted bir değeri kolon düzeyinde koruması olmayan bir yere kopyalamıştır. Bölüm 12'nin uyarısı tüm gücüyle geçerlidir; bu yüzden kaydı kuran kişinin neyin saklanacağına karar vermesi gerekir.

Statement'ı iki parça olarak sakladım. **Fingerprint**, her literal'in bir placeholder ile değiştirildiği statement'tır; böylece `UPDATE invoices SET status = ? WHERE id IN (?)` şekli kaydeder, başka bir şeyi değil. **Literals hash**, literal değerlerinin keyed hash'idir; bir araştırmacının

şüpheli bir değeri, kayıt onu tutmadan kayda karşı test etmesini sağlar. Anahtar önemlidir: bir status ya da kısa bir hesap soneki gibi literal'lerin entropisi düşüktür ve anahtarsız bir hash, adaylar denenerek tersine çevrilebilir. Anahtar producer'lardan ayrı olarak audit hesabında durur ve araştırmacının aracı bir karşılaştırma ister, anahtarı hiçbir zaman istemez.

```
function buildStatementRecord(session, statement, analysis):
    // analysis comes from the gateway's query analysis, not from the
    client
    parsed = parse(statement)
    record = { kind: "intent", session_id: session.id,
              login: session.login, principal: session.principal,
              on_behalf_of: session.on_behalf_of,
              roles: session.roles, request_id: session.request_id,
              fingerprint: redactLiterals(parsed),
              literals_hash: keyedHash(auditKey, literalsOf(parsed)),
              tables: analysis.tables,
              columns: classifyColumns(analysis.columns),
              started_at: now() }
    // a statement that cannot be parsed is recorded as a refusal with
    no
    // text, never stored verbatim as a fallback
    if parsed is null:
        record.fingerprint = "unparsed"
        record.literals_hash = null
    return record
```

Son satırlar en önemlileridir. Parser başarısız olduğunda ham statement'ı saklayan bir fallback, bir literal'in en çok görünmesi muhtemel olan tek durumu saklardı; çünkü parser'ın başarısız olduğu statement'lar alışılmadık olanlardır. Hiçbir şey saklamaz ve reddin konuşmasına izin verirdim. Store'un kendisi en az Confidential olarak sınıflandırılır, çünkü kimin neye baktığını anlatır ve kendi okuyucu listesi ile retention'ı vardır. Bir şirketin kendi çalışanları hakkında ne kadar kayıt tutabileceği, hukuk ve uyum ekibinize sorulacak bir sorudur.

ledgerly_access_admin'i kim denetler

ledgerly_access_admin, login oluşturabilen ve yönettiği roller içinde her şeyi grant edebilen roldür ve üç login üzerinden kullanılır: access definition'ı uygulayan pipeline adımı (Bölüm 3), credential broker (Bölüm 4) ve gateway (Bölüm 7). Bölüm 3, her eylemin audit log'a gittiğini söylemişti; mekanizma, üçünün her birinin kendi stream'i olan bir

producer olmasıdır, böylece kayıt hangi login'in eylemde bulunduğunu söyler. Verdiği her `CREATE LOGIN`, `GRANT` ve `REVOKE`, değişiklikten önce, onu doğuran istek ya da çalıştırmayla birlikte yazılmış bir `event` girdisidir.

Bu, döngüyü kapatan bir kontrol verir. Veritabanının kataloğu hangi grant'lerin var olduğu hakkındaki doğrudur, audit store'u ise neyin grant edildiği hakkındaki doğrudur. Katalogda eşleşen bir girdisi olmayan bir grant, üç producer'ın dışında yapılmıştır ve Bölüm 3'ün drift kontrolünün bulmak için kurulduğu, gece 2'deki grant budur. İki kontrol iki uçtan görülen tek şeydir: drift kataloğu kodla karşılaştırır, bu ise kataloğu kayıtlarla karşılaştırır. İkisi de hiçbir şeyi onarmaz; ikisi de raporlar.

Migration'lar

Migration pipeline'ı, Bölüm 11'in şart koştuğu gibi, bir migration'ı çalıştırmadan önce ve bittiğinde ya da başarısız olduğunda tekrar audit store'una yazar. Harici girdinin sebebi, migration aracının uyguladığını `ledgerly_primary` içindeki bir tabloya da kaydetmesi ve `ledgerly_migrator`'ın o tabloyu değiştirebilen `ledgerly_owner`'ı üstlenmesidir. Eylemde bulunanın yeniden yazabildiği bir kayıt bir kolaylıktır, kanıt değil. Kural şudur: harici başlangıç girdisi ve commit hash'i audit store'unda varsa migration uygulanmış sayılır ve migration tablosunda eşleşen harici girdisi olmayan bir satır, aynı katalog-karşı-kayıt karşılaştırmasıyla, pipeline'ı atlamış bir değişiklik olarak raporlanır.

Birinin cevaplayacağı alert'ler

Bir kayıt olaydan sonra soruları cevaplar. Bu soruların birkaçı sürekli sorulmaya değer ve bunu yapan biçim bir alert'tir. Her kurala uygulayacağım test, birinin cevaplamak üzere adlandırılmış olması ve ne yapacağını bilmesidir.

Ledgerly üç kuralla başlıyor; her biri kaydın bildiği ve diğer kaynakların bilmediği bir şeyden yararlanır.

```

function evaluateAlertRules(window):
    for each r in outcomeRecords(window):
        // 1. large reads outside the person's working hours
        if isReadClass(r.statement_class) and isHuman(r.principal)
            and (r.rows_returned > nightRowThreshold or r.outcome ==
"row-cap-closed")
            and isOutsideHours(r.started_at, r.principal):
                notify("security", "large read outside hours: " +
r.statement_id)
        // 2. break-glass in the role set without a declared request
        if "human_breakglass" in r.roles and not
hasDeclaredRequest(r):
            notify("security", "break-glass role without a
declaration: " + r.session_id)
        // 3. Restricted columns returned to a human principal
        if touchesRestricted(r.columns) and r.rows_returned > 0
            and isHuman(r.principal) and not isDeclaredBreakGlass(r):
            notify("security", "Restricted data read by a human: " +
r.statement_id)
    for each s in sessionsOpened(window):
        if "human_breakglass" in s.roles:
            notify("security", "break-glass in use: " + s.id)

```

Eşikler örnektir; doğru `nightRowThreshold`, haftada bir avuç olayı bir insanın önüne getiren değerdir. İkinci kural, Bölüm 8'in beyanı üzerinde bağımsız bir kontroldür: o bölüm kişi acil durumu beyan ettiğinde alert verir, bu kural ise rolü tutan ve beyanı olmayan bir session'ı bulur; bu da onu atlamış bir grant'tir. Üçüncü kural neredeyse hiç tetiklenmemelidir. `human_readonly` Restricted kolonları okuyamaz ve bir JIT isteği birini adlandıramaz; dolayısıyla Restricted verinin bir insan tarafından okunması ya beyan edilmiş bir break-glass session'ıdır ya da matrisle uyuşmayan bir grant'tir ve ikincisi bir tasarım bulgusudur.

Her kuralın adlandırılmış bir sahibi ve bir ilk eylemi olmalıdır: güvenlik irtibat kişisi görevin ne olduğunu sorar, on-call platform mühendisi beyanı doğrular ve review kalemini açar, platform ekibi atlanmış bir grant'i revoke eder. Bakılmadan onaylanan bir kural güvenlik tiyatrosudur.

Erişim review'u

Alert'ler olayları izler. **Erişim review'u**, her principal'ın erişiminin hâlâ gerekli olup olmadığına dair zamanlanmış kontroldür ve hiçbir olayın göstermediği yavaş başarısızlığı yakalayan kontroldür: verildiğinde doğru olan ve sessizce doğru olmaktan çıkmış erişim. İnsanlar ekip değiştirir, bir

servis bir tabloyu okumayı bırakır, bir rol, workload'unun aylardır kullanmadığı bir ayrıcalığı korur. Bölüm 3, doğru beyan edilmiş yanlış bir grant'ın drift kontrolünden geçtiğini ve bu yargının yerinin erişim review'u olduğunu söylemişti.

Review, grant edilenle kullanılanı karşılaştırır. Kayıt ikinci listeyi sağlar ve onu imzalanacak bir forma değil bir karara dönüştüren budur.

```
function runAccessReview(period):
  proposals = []
  for each grant in declaredGrants():
    // role to table privileges, identity provider group to role,
    // and the logins of ledgerly_access_admin
    use = lastUse(grant, inRecords: true)
    if use is null or use.at < now() - period:
      proposals.append({ grant: grant, finding: "unused",
                        proposal: "revoke" })
    if grant.owner is null or not isActivePrincipal(grant.owner):
      proposals.append({ grant: grant, finding: "no owner",
                        proposal: "assign or revoke" })
  for each r in breakGlassUses(period):
    if reviewFor(r.id) is null:
      proposals.append({ grant: r, finding: "no review",
                        proposal: "review now" })
  // the review proposes; a named reviewer decides, and is never
  // the person whose access is under review
  return assignReviewers(proposals)
```

Çıktısını kullanırken üç kural uygulardım. Birincisi, fonksiyon önerir ve hiçbir zaman revoke etmez; Bölüm 3'ün verdiği sebeple: birinin bağımlı olduğu bir grant'ın otomatik revoke'u, production'ı bozmanın yeni bir yoludur. İkincisi, `period` workload'un en uzun döngüsünden daha uzun olmalıdır. Üç aylık bir rapor tarafından kullanılan bir grant, çoğu ayda kullanılmamış görünür; bu yüzden bir aylık bir review üç aydan ikisinde onu revoke etmeyi önerirdi. Üçüncüsü, kullanım yalnızca tabloya göre değil ayrıcalığa göre eşleştirilir. Bir tablodan her gece select yapan ve ona hiç update yapmamış bir rol, `S`, `U`'dan `S`'e daraltılmaya adaydır ve kayıt, gerekeni taşır; çünkü her kayıt statement sınıfını ve dokunduğu tabloları taşır.

Review'un kaydın sağlayamayacağı ikinci bir girdisi var: grant olmayan şeyler. Bölüm 13'ün kopya register'ı gündemde yer alır; JIT isteklerini kimin onaylayabileceğinin listesi ve audit store'unu kimin

okuyabileceğinin listesi de. Her biri için soru aynıdır: kişi ya da sistem buna hâlâ ihtiyaç duyuyor mu ve birinin hâlâ sahibi var mı? Ek'te bir şablon var.

Kanıt, uyum değil

Veri koruma ve ödeme kartı rejimleri erişim hakkında bir soru ailesi sorma eğilimindedir: veriye kim erişebilir, kim ne yaptı, erişim ne kadar sürer, kopyalar nerede ve acil durumda ne olur. Bu rejimlerden biri altında çalışan bir denetçi genellikle bir kontrolün var olduğunu, çalıştığını ve sonuçlarına bakıldığını görmek ister. Herhangi bir rejimin yorumu hukuk ve uyum ekibinize aittir. Bu bölüm, yukarıdaki mühendisliğin sağlayabileceğinden fazlasını iddia etmez; bu da kanıttır: soruları cevaplayan kayıtlar ve bunların değiştirilmediğini gösterme yolu.

Bu bölümdeki hiçbir şey bir şirketi herhangi bir şeye uyumlu yapmaz. Tablo, bir denetçinin sormaya meyilli olduğu soruları, her birini cevaplayan kontrolü, bıraktığı artifact'i ve onu kuran bölümü listeliyor; Ek bunu bir eşleştirme olarak tekrarlar.

Soru	Kontrol	Artifact	Bölüm
Production verisine kim ve hangi yoldan erişebilir?	Tek kapı olarak gateway, kod olarak rol şeması, drift kontrolü	Ağ kural listesi, rol tanımları, drift raporu	3, 7
Belirli bir statement'ı kim, hangi grant'ler ve onayla çalıştırdı?	Request id ve approver ile <code>access_audit_record</code>	Kayıt ve <code>verifyChain</code> raporu	14
Yükseltilmiş erişim ne kadar sürer ve kim onayladı?	Approver ve <code>expires_at</code> ile JIT isteği, revoke job'ı	İstek kayıtları, revoke log'u, gecikmiş grant kontrolü	8
Hassas veri nasıl tanımlanır ve sınırlanır?	Şemada sınıflandırma, maskeleme, saklanan kolonlar	Katalog, maskelenmiş yükleme doğrulaması	12
Production verisinin kopyaları nerede ve her birinin sahibi kim?	Sahip ve bitiş tarihli register, sweeper	Register, sweeper log'u	6, 13
Şemayı kim değiştirir ve kim review etti?	Migration pipeline'ı, korumalı branch, harici başlangıç girdisi	Commit'lere ve audit girdilerine eşlenmiş migration kayıtları	11

Soru	Kontrol	Artifact	Bölüm
Erişim hâlâ gerekli mi?	Adlandırılmış reviewer'larla erişim review'u	Kararlarıyla tamamlanmış review	14
Acil durumda ne olur?	Alert ve zorunlu review ile break-glass	Review kayıtları, alert geçmiş	8, 15
Erişimin kendisini kim yönetir?	Ayrı producer'lar olarak <code>ledgerly_access_admin</code> login'leri	Stream'leri ve katalog- karşı-kayıt karşılaştırması	3, 14

Trade-off'lar ve maliyetler

Kaydın bir faturası var ve çoğu iki yerde ödenir.

Storage ve dikkat. İki kez, statement başına bir kayıt trafikle birlikte büyür ve store'un kendi retention'ı, backup'ları ve okuyucu listesi gerekir. Daha büyük maliyet dikkattir. Kimsenin sorgulamadığı bir store pahalı bir güvenlik tiyatrosudur; bu yüzden Bölüm 7'deki yükseltilmiş session'ların haftalık örnekleme, alert'ler ve review, ekstra değil fiyatın parçasıdır.

Kritik yolda yeni bir bağımlılık. Intent statement çalışmadan önce yazıldığı için bir audit kesintisi ayrıcalıklı işi bloke eder. Alternatif, statement'ları çalıştırıp sonra loglamaktır ve bu, kaydı tam önem taşıdığı anda isteğe bağlı yapar. Grant'ler, break-glass ve yazmalar için reddi korurdum; ekip bir okuma kesintisinin bir boşluktan daha pahalıya mal olduğuna karar verirse, rutin `human_readonly` okumaları için kuyruğun ne kadar büyüyebileceğine bir sınır koyarak kuyruğa alınmış bir yazımı düşünürdüm. Bu, platform ekibinin bir kez verip yazıya dökeceği bir karardır.

Hepsini ne zaman kurmamalı: bir avuç mühendisi olan, tek veritabanlı ve henüz gateway'i olmayan bir şirket, bir append servisi, bir anchor ve ayrı bir hesapla başlamamalıdır. Üç şeyle başlayın: her kişinin session'ını kendi login'i altında loglayın, veritabanı log'unu host'tan administrator'ların silemeyeceği bir yere gönderin ve aylık bir review'a adlandırılmış bir kişi koyun. Zinciri ve anchor'ı, log'u değiştirebilecek insanların sayısı "güven"i yetersiz bir cevap yaptığında ekleyin.

Sık yapılan hatalar

- **Veritabanı log'unu audit log saymak.** En eksiksiz kaynaktır ve administrator'ların kontrolü altında durur. Onu başka yere gönderin ve join'e bir girdi olarak kullanın, kayıt olarak değil.
- **Ayrı bir store olmadan hash chain.** Zincir kanıt gibi görünür ve store'a yazabilen herkes tarafından yeniden hesaplanabilir. Store'u başka bir hesaba koyun, zinciri append servisinde tutun ve başı anchor'layın.
- **Ham statement'ı saklamak.** Araştırmayı kolaylaştırdığı için cazip gelir ve Restricted değerleri kolon koruması olmayan bir store'a kopyalar. Fingerprint'i ve keyed hash'i saklayın ve ham metne fallback'i reddedin.
- **Sahibi olmayan alert'ler.** Cevaplamak için kimsenin adlandırılmadığı bir kanal gürültüye dönüşür. Her kurala bir sahip ve bir ilk eylem verin.
- **Denetleyenler dışında her şeyi denetlemek.** Erişimi grant eden rol, eylemleri en çok önem taşıyan roldür. Ona kendi producer'larını verin ve işini katalogla karşılaştırın.

Temel çıkarımlar

- Audit log, kimin hangi veriye ne yaptığının, eylemde bulunanın değiştiremeyeceği bir yerde tutulan kayıdır; gateway kaydı, veritabanı log'u ve `audit_events` ona üç girdidir, kendisi değil.
- Bir statement kaydı, önce yazılmış bir soru listesinden kurulur ve kimlik alanları, eylemde bir makine bulunduğunda bile `principal` ve `on_behalf_of` ile bir kişide biter.
- Kurcalanma kanıtı ayrı bir hesaptan, yalnızca yazabilen producer'lardan, zinciri tutan bir append servisinde ve başkasının sahip olduğu bir anchor'dan gelir; kanıt verir, hiçbir zaman önleme değil.
- Bir zincir, mevcut girdilerin değiştirilmediğini gösterir; bu yüzden ayrı kontroller sequence deliklerini, outcome'u olmayan intent'leri, bilinen kaynağı olmayan veritabanı login'lerini ve sessizleşen producer'ları arar; üç kayıt login ve request id üzerinden birleşir, bağlantı etiketi ise yalnızca bir ipucudur.

- Kayıt, tarif ettiği verinin bir kopyasına dönüşmemelidir; bu yüzden statement'lar fingerprint olarak ve literal'lerin keyed hash'ile saklanır, store da veri gibi sınıflandırılır ve kısıtlanır.
- Alert'lerin bir sahibi ve bir ilk eylemi olmalıdır; erişim review'u ise grant'leri, workload'un döngüsünden daha uzun bir dönem boyunca kullanımla karşılaştırır, böylece revoke'lar kanıttan önerilir.
- Kanıt uyum değildir: soru, kontrol, artifact ve bölümden oluşan eşleştirme sistemin neyi kanıtlayabildiğini gösterir ve bu kanıtın herhangi bir rejim altında ne anlama geldiği hukuk ve uyum ekibinize aittir.

Kayıt artık var, birleştirilmiş ve izleniyor. Geriye, ihtiyaç duyulduğu gün kalıyor; soruyu soran artık bir denetçi değil, sızmış bir credential'lı olan bir on-call mühendisi olduğunda. Sonraki bölüm olaylar sırasındaki erişime dönüyor ve ele geçirilmiş bir principal'ın gerçekte neye dokunduğunu bulmak için burada kurulan kaydı kullanıyor.

Bölüm 15 — Olaylar Sırasında Erişim

Ledgerly'de bir cumartesi öğleden sonrasını düşünün. Destek ekibinden Sam, on-call kanalına yazıyor: park halindeki bir arabadan iş laptop'u çalınmış ve üzerinde bir gateway session'ı açık kalmış olabilir. Nöbetçi Robin mesajı iki kez okuyor. Yanlış giden bir şey olduğu bilinmiyor, ama "açık kalmış olabilir" tartışacak değil, cevaplanması gereken türden bir cümle.

Robin'in üç şeye ihtiyacı var ve sıra önemli. Birincisi, o session'ın ve arkasındaki her login'in hâlâ yapabileceği her şeyi durdurmak. İkincisi, erişim varken neler yaptığını öğrenmek. Üçüncüsü, ilk ikisi sürerken işi ayakta tutmak; yani başkalarının da veritabanına ihtiyacı var, bazılarının acilen, üstelik her zamanki onaylayanlar evlerinde.

Bu bölüm olay erişimini tasarlanıp prova edilecek bir şey olarak ele alıyor. Üç ayağı da aynı anda sınırlar: grant'ler aceleyle kaldırılıp eklenirken yetkilendirmeyi, onarımlar baskı altında çalışırken yürütmeyi, kayıt neyin olduğunu söylemek zorundayken hesap verebilirliği. Beş parçalı modelde Kimlik login'leri bulur, Grant ve Yol onları keser, Yürütme acil yazmaları taşır, Kayıt hasarı ölçer. Bu bölüm Ledgerly'nin hedef durumunun son adımını ekliyor: olay erişiminin prova edilmiş bir yolu vardır ve revoke rutindir. Soru şu: Ledgerly önceki bölümlerde kapattığı kapıları yeniden açmadan erişimi nasıl durdurur, nelere ulaşıldığını nasıl ölçer ve çalışmaya nasıl devam eder?

Problem ve neden zor olduğu

Olay, önceki her kontrolün engel gibi görüldüğü andır. Sakin bir salı gününde makul olan onay, cumartesi günü bir beklemeye dönüşür; her şeyi kaydeden gateway, çökebilecek bir şey daha olur; odadaki en temiz

kestirme de hepsini atlayan bir login'dir. Olağan tepki, o login'i "acil durumlar için" önceden kurmaktır. Bu, daha iyi bir adla yine o ortak admin parolasıdır.

Baskı altında üç şey değişir. Zaman kısadır, bu yüzden akılda tutması bir insanı gerektiren adımlar atlanır. Bilgi eksiktir, bu yüzden ilk kapsam tahmini bir yönde ya da öbür yönde yanlıştır. Aktörler de alışılmışın dışındadır: veritabanına nadiren dokunan insanlardan birden bunu yapmaları istenir, her gün dokunanlar ise yorgundur.

Tasarım sorusu bu yüzden hangi kontrollerin gevşeyeceği, hangilerinin gevşemeyeceğidir. Tek bir kuralı korurdum: aciliyet hızı değiştirir; kimliği, çalışan statement'ı ve kaydı asla değiştirmez. Bekleme süreleri ve onaylayan sırası kısalabilir. Principal'ın kendi login'ile hareket etmesi, bir statement hash'ine bağlı onay ve audit log'da bir kayıt kısalamaz. Bir kontrol bu üçünden birini kaldırmadan kısaltılamıyorsa yanlış olan kontroldür, kural değil.

Hız inşa edilmek zorundadır. Bir düzine elle adım gerektiren bir revoke yanlış sırayla ya da yarım yapılır. Sıra koddur, kapsam sorgusu koddur, olayın geride erişim bırakıp bırakmadığını söyleyen mutabakat da koddur. Bunlara işaret eden bir runbook prosedürdür; bunların yerine muhakemeyi koyan ise bir umuttur.

Bir ele geçirilmeyi sınırlamak

Burada **sınırlama**, tek bir principal'ın erişimini durdurmak demektir; sebep kaybolan bir laptop da olsa, sızan bir credential da olsa, ya da kimsenin beklemediği biçimde davranan bir workload da olsa. Sebebe karar vermez. Bölüm 4, panikte birleşmeye meyleden iki soruyu ayırmıştı: bir credential'ın çalışmasını nasıl durduracağız ve çalıştığı sürede ne yaptı. Sınırlama birincisini cevaplar, aşağıdaki sıra da ikincisinin cevaplanabilir kalmasını sağlar.

Sıra şöyle: grant'leri revoke et, session'ları kapat, credential'ları rotate et, hasarın kapsamını audit log'dan çıkar, sonra kaydet ve bildir. Her adım bulunduğu yerde bir sebeple durur.

Grant'leri revoke etmek ilk sırada, çünkü hiç yetkisi olmayan bir session hiçbir şey yapamaz ve yeniden bağlanan da hiçbir şey elde etmez. Session'ları kapatmak ikinci sırada; devam eden işler ve revoke edilmiş bir grant'ten daha uzun yaşayan açık bağlantılar için (Bölüm 4, süresi dolmuş credential'lar için aynı noktayı yapmıştı). Rotation üçüncü sırada, çünkü yeni login'leri durdurur ve zaten açık olanlar için hiçbir şey yapmaz. Kapsam çıkarma salt okunurdur ve erişime değil kayda ihtiyaç duyar, bu yüzden erişim gidene kadar bekler. Bildirim sonuncudur, ama her adımın kaydı sonuncu değildir: her adım biter bitmez eklenir.

```
function containCompromise(incident, principal, since):
    unfinished = []
    // every step runs even if the one before failed: a stuck revoke
    must
    // not leave sessions open and the credential valid
    for each step in ["revoke-grants", "close-sessions", "rotate-
    credentials"]:
        outcome = "done"
        try:
            runContainmentStep(step, principal)
        on StepFailed:
            outcome = "failed"
            unfinished.append(step)
            notify("platform", "containment step failed: " + step + "
            " + incident.id)
    // the log never delays containment; an unreachable log queues
    the entry
    audit0rQueue({ action: "contain", actor: incident.commander,
                    subject: principal, request: incident.id,
                    step: step, outcome: outcome })
    // read-only and last: it needs the record, not the access
    scope = blastRadius(principal, since)
    saveIncidentRecord(incident.id, { principal: principal, since:
    since,
                                scope: scope, unfinished:
    unfinished })
    notify("security", "contained, scope attached: " + incident.id)
    return unfinished
```

İki ayrıntı not edilmeye değer. Fonksiyon tamamlanmamış adımları döndürür ve exception fırlatmaz, çünkü yarım sınırlanmış bir principal izlenmesi gereken bir durumdur ve bir exception bunu bilen tek süreci sonlandırır. Ayrıca audit0rQueue, Bölüm 7'dekiyle aynı helper'dır: bir log kesintisi kaydı geciktirir, erişimin kaldırılmasını asla geciktirmez. Bu

asimetri, Bölüm 8'in revoke için seçtiği asimetridir: log kaydı olmayan bir grant bir açıklıktır, log kaydı olmayan bir revoke ise sonradan doldurulabilecek bir boşluktur.

Her adımın ne anlama geldiği principal'ın ne olduğuna bağlıdır. Tablo, Ledgerly'de bulunan dört durumu listeliyor.

Principal	Grant'leri revoke et	Session'ları kapat	Credential'ları rotate et
Kişi, session login'leri	JIT üçlülerini, <code>human_breakglass</code> ve <code>human_readonly</code> rollerini kişinin elindeki her session login'inden kaldır	Her login için <code>endSession</code> ; yeni session açılmasını diye kişiyi identity provider'da devre dışı bırak (Bölüm 7)	Kişinin identity provider credential'larını ve faktörlerini sıfırla; session login'leri düşürülür, dolayısıyla rotate edilecek bir veritabanı parolası yoktur
Servis, dinamik credential	Lease'i sonlandır; login onunla birlikte gider (Bölüm 4)	Broker, lease'i sonlandırmanın parçası olarak session'ları kapatır	Elle yapılacak bir şey yok; bir sonraki istek yeni bir değer alır
Servis, A/B çiftinde statik credential	<code>app_*</code> rolünü sızan login'den hemen kaldır, eski login'in boşalmasını beklemeden	Kalan session'larını kapat	Yeni bir değeri boştaki login'e yayımla ve pointer'ı değiştir; <code>secret</code> 'ını yeniden okuyamayan servisleri yeniden başlat
<code>ledgerly_access_admin</code> login'i	Yetkilerini kaldır; bu, onu kullanan principal için provisioning'i durdurur	Session'larını kapat	O principal'a yeni bir credential ver; kapsam sorusu, hiçbir isteğin açıklamadığı grant'ler ve login'ler haline gelir (Bölüm 3'ün drift kontrolü)

Statik satır, Bölüm 4'le örtüşen satırdır. Sızan bir statik credential planlı bir rotation değildir ve sızan login, yerine geçecek olan hazır olmadan kesilir; bu da onu kullanan servislerde bir kesintiye mal olur. Bölüm 4 bu maliyeti

zaten kabul etmişti ve ben bu kabulü korurdum: çalışmaya devam eden sızmış bir credential daha büyük sorundur. Yeniden başlatma gerektiren servisler, bunun önem kazandığı günden önce bir listede olmalıdır. Her servisin tek bir admin login'ini paylaştığı başlangıç durumunda, birinci adım bütün şirketin kesintisidir; rutin revoke'u önce kurmanın en güçlü argümanı budur.

Son satır ağır olandır. Erişimi yöneten login, yönettiği roller içinde her şeyi grant edebilir; dolayısıyla onu sınırlamak, yeni bir credential olana kadar gateway'in, pipeline'ın ya da broker'ın herhangi bir şey provision etmesini durdurur. En geniş erişimli login'i kesmenin bedeli budur ve bir ekip bunu kimin ödeyebileceğine önceden karar vermelidir.

Hasarın kapsamını kayıttan çıkarmak

Erişim gittikten sonra soru, onun ne yaptığına dönüşür. Cevap, tasarımı Bölüm 14'ün anlattığı `access_audit_record` üzerinde, o bölümün tanımladığı alanlarla çalıştırılan tek bir sorgudur. Sorgu, bir principal'in erişiminin ilk kez kötüye kullanılmış olabileceği T anından beri blast radius'udur: dokunduğu her tablo, kaç satırın döndüğü, kaçının değiştirildiği ve bu tablolardan hangilerinin Restricted veri tuttuğu.

```
// illustrative notation; field names as in Chapter 14
SELECT t.table_name,
       classOf(t.table_name)           AS data_class,
       COUNT(*)                       AS statements,
       SUM(r.rows_returned)           AS rows_returned,
       SUM(r.rows_affected)           AS rows_affected,
       SUM(CASE WHEN r.outcome = "refused" THEN 1 ELSE 0 END) AS
refused,
       MIN(r.started_at)              AS first_seen,
       MAX(r.ended_at)                AS last_seen
FROM access_audit_record r
     EXPAND r.tables AS t              // one row per table
touched
WHERE r.kind = "outcome"              // one row per statement;
intent rows have no counts
  AND r.login IN :logins
  AND r.started_at >= :since
GROUP BY t.table_name
ORDER BY (data_class = "Restricted") DESC, rows_returned DESC;
```

`:logins` ve `:since` değerlerini seçmek, muhakemenin çoğunun bulunduğu yerdir. Bir kişi için gateway her session'a yeni bir login oluşturur; dolayısıyla küme, principal'ın T'den beri sahip olduğu her session login'idir, tek bir tanesi değil. T için erişimin kötüye kullanılmış olabileceği en erken anı kullanın; örneğin laptop'un sahibinin elinde olduğu bilinen son zamanı, birinin fark ettiği zamanı değil. Çok dar bir pencere, yanlış soruya temiz bir cevap üretir.

Sonuç ancak kayıt kadar iyidir, bu yüzden kimse okumadan önce üç kontrol yapılır.

1. `verifyChain`, Bölüm 14'ün tanımladığı biçimde pencere boyunca geçer. Başarısız olursa sonuç, olaydan başka bir şeyin kanıtıdır ve olay ikinci bir kayıt alır.
2. Pencerede sıra boşluğu yoktur; veritabanı session'ı olmayan gateway session'ı da, tersi de yoktur. Bu, Bölüm 14'ün boşluk tespiti. Pencere içindeki bir boşluk, cevabın bilinmediği bir kesittir.
3. Principal'ın yan yolları ayrıca kontrol edilir: istediği restore'lar, export'lar ve kopyalar (Bölüm 13); bunlar bu sorgu üzerinden değil, kopya kayıt defteri üzerinden bulunur.

Kontrollerden biri başarısız olursa sonuç bir alt sınırdır ve olay kaydında bunu bu sözlerle söyledim. Sorgunun kendine ait sınırları da var.

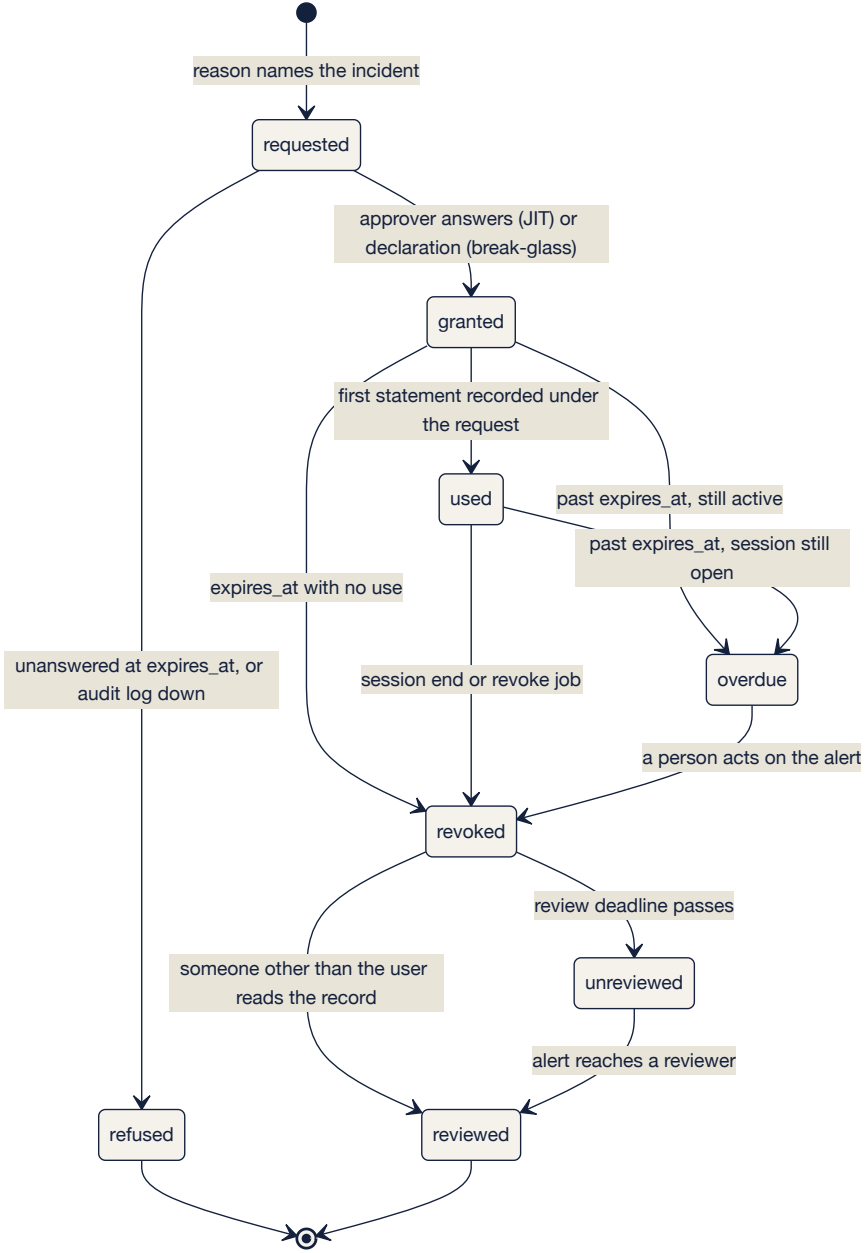
`rows_returned` bir sayıdır, liste değildir: bir Restricted tablonun pencerede şu kadar satır döndürdüğünü söyler, bunların hangi müşteriler olduğunu söyleyemez. `fingerprint` ve `literals_hash` alanları, bilinen bir statement'ın tekrarlanıp tekrarlanmadığını sormanıza izin verir, ama literal'ler tasarım gereği redact edilmiştir. Yani kayıt maruz kalmayı sınırlar, sıralamaz. Bir Restricted tabloda sıfırdan büyük bir `rows_returned`, güvenlik irtibatlarının ve okuyucunun hukuk ve uyum ekibinin ihtiyaç duyduğu çizgidir; bunun ne anlama geldiğinin yorumu, bildirim yükümlülüğü dahil, onlara aittir.

Olay erişiminin yaşam döngüsü

Sınırlama erişimi kaldırır. Bir olay ayrıca erişimin eklenmesini de gerektirir: birinin bakması, birinin onarması gerekebilir. O yöndeki risk, aceleyle oluşturulan ve pazartesi günü hâlâ orada duran grant'tir. Çare,

görünür bir sonu olan bir yaşam döngüsüdür; bir olay için yapılan her erişim parçası aynı beş durumdan geçer.

Diyagram, Bölüm 8'in zaten tuttuğu kayıtlar üzerinde bir görünümdür, yeni bir status alanı değildir. `granted`, istek kaydındaki `approved`'dır; `used`, istek altında bir statement'ın kaydedildiği anlamına gelir; `reviewed` ise kullanıcı dışında birinin kaydı okuduğu anlamına gelir. `reviewed`'a ulaşmayan iki çıkışa bakın: sonunu aşıp kalan bir grant ve hiç yapılmayan bir review.



Bunun işlemesi için olay sırasında yapılan her istek, gerekçesinde olayın tanımlayıcısını taşır. Gateway'in formu bunu zorunlu kılabilir ve açık bir olay varken tanımlayıcısız bir istek reddedilir. Bir job'ın "bu olayın grant ettiği her şeyi" sorabilmesini sağlayan bu tek konvansiyondur.

```

function reconcileIncidentAccess(incident):
    // watches the incident's own requests; never changes access
    (Chapter 8:
    // an automatic revoke of unknown grants is a way to break
    production)
    for each r in requestsWithReason(incident.id):
        if r.status == "approved" and r.expires_at < now() -
        graceMinutes:
            notify("platform", "incident grant past expiry: " + r.id)
            if r.status == "revoked" and reviewFor(r.id) is null
                and now() - r.expires_at > reviewDeadline:
                    notify("security", "incident access not reviewed: " +
                    r.id)
            // the database side: grants that no request explains
            for each g in unexplainedGrants(since: incident.opened_at):
                notify("platform", "grant with no request: " + g.login)

function closeIncident(incident):
    reconcileIncidentAccess(incident)
    // an incident is not over while any access it created is still
    open
    if openFindings(incident.id) > 0:
        raise IncidentHasOpenAccess
    markClosed(incident.id)

```

Bu job, Bölüm 8 ve 3'ün overdue kontrolünün ve drift kontrolünün yerini almaz. Aynı olgulara olayın tarafından bakar ve olaya bir gate verir: bir grant hâlâ canlıyken ya da bir break-glass kullanımı review edilmemişken kimse olayı kapalı olarak işaretleyemez. Ticket sistemi kullanan bir ekip `closeIncident`'ı ticket'a bağlayabilir; bu, "revoke etmeyi unuttuk" durumunu görünür bir engele çevirir.

Reviewed

Son durum en sık atlanan durumdur, çünkü olay bitmiş gibi hissettirir. Review, Bölüm 8'in break-glass için anlattığı review'dur ve olayın yaptığı her grant'e genişletilmiştir. Suçlayıcı değildir ve sistemi sorgular: erişim işi yapabilecek en küçük erişim miydi ve olması gereken daha hızlı bir yol var mıydı? Ek'te bir olay sonrası şablonu var. Karşılığını veren bulgu çoğunlukla eksik bir yoldur.

Olay sırasında yazmalar

Onarım, bir kontrolü gevşetmek için en cazip yerdir, çünkü zarar görünürdür ve veri şu anda yanlıştır. Bölüm 10, bir onarımın bir veri düzeltmesi olduğunu; Bölüm 8 ise break-glass'ın onarımlar için bir yol olmadığını kesinleştirmişti. Baskı altında her iki kararı da korurdum ve yanlış yolu erişilebilir kılmak yerine uyumlu yolu hızlı yapardım.

İlk seçim, ihtiyacın hangi yola ait olduğudur. Kontrollerin sırası en dar kapıdan en geniş kapıya doğru gider.

```
function chooseIncidentWritePath(need):
    // first match wins; Restricted values are never written by a
    person
    if need.touches_restricted_column or need.is_owner_link_change:
        return "product path: admin-console asks payments-service
        (Chapters 9, 10)"
    if need.is_schema_change:
        return "pipeline, expedited lane (Chapter 11)"
    if need.stops_ongoing_harm and need.is_small_limiting_action:
        // pausing, disabling or flagging: never the repair itself
        return "break-glass session, reviewed afterward (Chapter 8)"
    return "data fix: JIT request, runner, expedited review (Chapters
    8, 10)"
```

Break-glass'a yalnızca küçük sınırlayıcı eylem için ulaşılabilir: kötü satır yazan bir job'ı durduran bir flag, bir batch'i bekleten bir status. Verinin kalıcı onarımı her zaman bir veri düzeltmesidir ve gerekçesi öncekiyle aynıdır: ham bir break-glass session'ında çalıştırılan onarım, statement'ı çalışmadan önce okuyan tek kontrolü atlar ve sınırsız bir UPDATE'in hâlâ mümkün olduğu yerde çalışır (Bölüm 8 bunu kabul edilmiş bir risk olarak adlandırır). Bir olay sırasında cazibe daha güçlüdür, tek bir yanlış yazmanın bedeli de öyle.

Veri düzeltmesi şeklini korur. İstek, review edilmiş bir düzeltmeyi adlandırır ve merge kaydından alınmış bir `approved_hash` taşır; runner hash'i farklı olan bir script'i reddeder, grant düzeltmeden türetilir ve sayı koruması ile snapshot hâlâ geçerlidir (Bölüm 9 ve 10). Olayın değiştirdiği şey review'dur.

```

function expediteFixReview(fix, incident, reviewer):
    // relaxes: the waiting period, and who may be asked first
    // never relaxes: a reviewer who is not the author, the hash, the
runner
    if incident.status != "open":
        raise NoOpenIncident
    if reviewer == fix.author:
        raise SelfApprovalNotAllowed
    mergeRecord = mergeFix(fix.id, reviewedBy: reviewer, expedited:
true,
                        incident: incident.id)
    // the hash is read from this record by the gateway, not from the
runner
    queueFollowUpReview(fix.id, reviewer: "platform", due:
nextWorkingDay)
    notify("platform", "expedited fix merged: " + fix.id)
    return mergeRecord

```

Bu, Bölüm 11'in migration'lara verdiği hızlandırılmış şeridin aynısıdır: aynı job, aynı credential, aynı kayıt, farklı zamanlama. Kestirmenin bedelinin ödendiği yer takip review'udur ve hızlandırılmış bir yolu sayılan bir olaya çeviren de odur. Bir ekip bir çeyrekteki düzeltmelerin çoğunun bunu kullandığını görürse, normal lane iş için fazla yavaştır ve doğru tepki sayımı bırakmak değil, normal şeridi hızlandırmaktır.

Uyarı: Break-glass'a "ihtiyaç duyan" bir düzeltme, Bölüm 10'un dediği gibi başka bir şeyin işaretidir: bir kolonu olmayan bir tavan, saat için fazla küçük bir onaylayan havuzu ya da ürün yoluna ait olan bir Restricted kolon. Bir olayda bu işareti görmezden gelmek kolaydır ve iki kez görmezden gelmek pahalıdır.

Önceki bölümlerin ertelediği durumlar

Üç durum önceki bölümlerce ileriye aktarılmıştı. Tablo baştan alınan karardır; alt bölümler gerekçeyi verir.

Durum	Ne durur	Ne değişmez	Önceden kararlaştırılan
Gateway çalışmıyor	İnsan veri erişimi (ikinci bir kapı yok)	Servisler kendi credential'larıyla çalışmaya devam eder	Gateway'i kimin geri getireceği ve mühürlü aracın ne zaman kullanılabileceği

Durum	Ne durur	Ne değişmez	Önceden kararlaştırılan
Olay sırasında bir restore gerekiyor	Yol prova edilmişse hiçbir şey	Onaylayan, izole hedef, kayıt defteri girdisi, zamanlanmış silme	Adı belli bir onaylayan, hazırlanmış bir hedef, bir restore runbook'u
Geceleri onaylar yavaş	Acil olmayan bir onarım	Kendi kendine onay hâlâ reddedilir	Bir yükseltme sırası ve havuzu kimin genişletebileceği

Gateway çalışmıyor

Bölüm 7 ve 8, ikinci bir kapı olmadığına karar vermişti: insanlar veritabanına yalnızca gateway üzerinden ulaşır, dolayısıyla bir gateway kesintisi aynı zamanda bir break-glass kesintisidir. Oradaki gerekçe, gateway olmadan çalışan bir credential'ın logging, kimlik ve süre dolumu olmadan da çalışmasıydı. En kötü gecede de buna bağlı kalırdım ve bir kesintideki ilk hamle, neyin durmadığını fark etmektir. Servisler kendi credential'larıyla bağlanır ve çalışmaya devam eder; bu yüzden gateway'in çalışmıyor olduğu birçok olay, veritabanında bir insanı hiç gerektirmez. Deploy'u rollback et, job'ı duraklat, feature flag'i çevir, servisi ölçekle: bunların hiçbirisi gateway'den geçmez.

Olay veri erişimini gerçekten gerektiriyorsa, olay artık iki olaydır ve ilki gateway'dir. Onu geri getirmek veri sorusundan önce gelir ve geri getirenler platform ekibidir; Bölüm 7'nin istediği deploy yolunu ve yedekliliği kullanırlar.

Her instance çalışmıyorsa, hiçbir insan veri yolu yoktur ve geriye kalan tek araç, Bölüm 3'ün mühürlü saydığı superuser'dır. Bu bir engine seviyesi araçtır, bir veri yolu değil: append-only ve Restricted korumalarını atlar ve ağ rotası insanlara kapalıdır. Kullanımını gateway'i ya da gateway'in bağlı olduğu şeyi geri getirmekle sınırlardım; iş verisini okumak ya da yazmak için asla kullanmazdım ve kuralı yazıya dökerdim.

Uyarı: Gateway çalışmıyor diye mühürlü aracı veriye bakmak için kullanmayın. Append-only ve Restricted korumalarını atlar ve verdiği erişim, önceki her bölümün önlemek için kurulduğu erişimdir. Veri okunmak zorundaysa önce gateway'i geri getirin.

Kural çerçevesindeki kullanımın bir mekanizması var. Credential'ın alınması bir kayıt bırakır, ikinci bir kişi hazır bulunur ve adı yazılır, ağ rotasını açmak bir sahibi ve bitiş zamanı olan bir değişikliktir ve credential, bir insan onu gördüğü için sonradan rotate edilir. Gateway bu pencerede kayıt tutmaz, dolayısıyla engine log'u tek izdir. Gateway geri geldikten sonra platform ekibi, Bölüm 14'ün migration'lar için anlattığı harici audit girdisi gibi, elle bir girdi ekler ve kullanım, her break-glass kullanımının aldığı review'u alır.

Olay sırasında bir restore

Bölüm 13, restore'u bir istek haline getirmişti: isteyen olmayan bir onaylayan, izole bir hedef, bir kayıt defteri girdisi, zamanlanmış bir silme ve kapalı bir başlangıç. Orada adı konan maliyet, olay sırasındaki hızlı ve cevabı prova etmek ve onayı önceden ayarlamaktı. Mekanizmanın tamamı budur; bu yüzden "önceden" in ne demek olduğunu açmaya değer.

Olaydan önce runbook, geceleri bir restore'u kimin onaylayabileceğini adlandırır, hedef hazırlanmış ve izoledir ve restore isteğinin bir olay tanımlayıcısı alan bir formu vardır. Olay sırasında istek değişmeden ilerler: onaylayan isteyen değildir, kayıt defteri girdisi önce gelir, zamanlanmış silme kopya var olmadan ayarlanır, login'ler devre dışı bırakılır ve erişim tanımı kopya açılmadan önce uygulanır. Olay gerekçeyi ekler, kestirmeyi değil.

Kopyanın ne için olduğu, nasıl yapıldığından daha önemlidir. Bir olay sırasında restore edilmiş bir kopya bir okuma kaynağıdır, genellikle eski değerler için: yanlış bir yazmadan önceki bir kolonun durumu ya da yanlışlıkla silinmiş satırlar. Okuyucuları, istekte adı geçen kişilerdir. Ardından gelen onarım, primary'ye karşı bir veri düzeltmesi olarak çalışır; kopyadan okunan değerler review edilmiş bir script'in girdisidir ve Bölüm 13'ün reapplyDeletions fonksiyonu kopya açılmadan önce çalışır, böylece kopya bilerek silinmiş veriyi geri getirmez. Primary'yi restore edilmiş bir kopyayla değiştirmek bir kurtarma kararıdır ve platform ekibi ile olay komutanına aittir. İçindeki erişim sorusu yukarıdakidir: kopyayı kim okur ve ne zaman silinir.

Geceleri yavaş onaylar

Bölüm 8'in gece senaryosu, bu bölümün koruduğu bir kararla bitmişti: onaylayan havuzu cevap vermezse ve ihtiyaç bekleyebiliyorsa, bir not bırak ve dur. Bu iyi bir sonuçtur ve eski Ledgerly'nin üretemeyeceği bir sonuç. Bekleyemeyecek bir ihtiyaç için yükseltme isteğin içine yerleştirilmiştir; böylece cevaplanmayan bir alarm aynı istek içinde havuzda aşağı iner.

```
function escalateApproval(request, pool):
    // one request, one record: the page moves, the request does not
    restart
    for each approver in pool.inOrder():
        if approver == request.requester or approver ==
request.fix_author:
            continue
        page(approver, request.id)
        if waitForAnswer(request.id, within: escalationMinutes):
            return
    // pool exhausted, window still open: the incident commander
    decides
    notify("platform", "approval unanswered, pool exhausted: " +
request.id)
    // never widens the pool or approves by default; the request stays
    pending
    // until expires_at, when it is refused
```

Bunun çevresinde iki kural var. Birincisi, havuzu yalnızca ona sahip olan biri genişletir ve genişletme bir kayıttır: ilan edilmiş bir olay sırasında olay komutanı, olay için bitiş zamanlı, adı belli bir onaylayan ekleyebilir. İkincisi, nöbetçi kim olursa olsun ona otomatik onay hâlâ kurulmaz, çünkü her isteği saniyeler içinde grant eden bir kontrol kontrol değildir. Onaylar düzenli olarak grant penceresinden uzun sürüyorsa, bulgu nöbet çizelgesi, havuz boyutu ya da saat hakkındadır ve çare, havuza insan eklemektir; onaylayanı döngüden çıkarmak değil.

Yolları prova etmek

Hiç çalışmamış bir yol sınanmamıştır. Bölümün tamamı, kimsenin ilk kez bir olay sırasında keşfetmek istemediği adımlardan kurulu; bu yüzden her biri, burada game day denen bir provayı gerektirir: bir senaryosu, beklenen bir sonucu ve kayıt üzerinde bir kontrolü olan zamanlanmış bir

tatbikat. Dört senaryo, bu bölümün inşa ettiklerini kapsar. Tasarım gereği savunmacıdır; her biri, izlemenin zaten görmeyi bildiği bir tetikleyiciyle başlar ve her biri gözlemlenebilir bir geçme koşulu taşır.

Senaryo	Tetikleyici	Beklenen davranış	Geçme koşulu	Sınanan parçalar
Credential sızıntısı	Bir tatbikat principal'ının credential'ı ifşa olmuş ilan edilir	<code>containCompromise</code> sırayla; kapsam sorgusu olaya eklenir	Principal için canlı login yok, unfinished boş, kapsam tatbikatın dokunduğu her tabloyu adlandırır	Kimlik, Grant, Kayıt
Gateway çalışmıyor	Production dışı bir kopyada önce bir, sonra tüm gateway instance'ları durdurulur	İnsan erişimi durur; servisler çalışmaya devam eder; restore yolu çalışır	Platform ekibi gateway'i runbook'u kullanarak geri getirir; mühürlü araç kullanılmaz ya da kuralı çerçevesinde bir kayıtlı kullanılır	Yol, Kayıt
Şüpheli export	<code>evaluateAlertRules</code> 'tan (Bölüm 14) gelen, büyük <code>rows_returned</code> üzerine bir alarm	Principal'ı sınırla; hedefi kopya kayıt defterinde bul; sil	Kopya bulunur, sonlandırılır ve kaydedilir; kapsam sorgusu kayıt defteriyle uyuşur	Kimlik, Kayıt, Yürütme

Senaryo	Tetikleyici	Beklenen davranış	Geçme koşulu	Sınanan parçalar
Hatalı toplu UPDATE	Bir prova ortamında, row cap'e sığan ve yanlış satırları değiştiren bir yazma	Daha fazla yazmayı durdur; tersini snapshot'tan hesapla; hızlandırılmış bir veri düzeltmesi olarak çalıştır	Tersine çevirme, yazar olmayan bir reviewer tarafından merge edilir; takip review'u kuyruğa alınır	Yürütme, Grant, Kayıt

Bir game day yalnızca gözlenen sonuçla geçer, birinin ne yapacağını hatırlamasıyla asla. Bir tatbikat birisi bir kestirme bildiği için bitiyorsa başarısız olmuştur ve kestirme inşa edilecek şey olur. Her adımın gerçekte aldığı süreleri yazın; bu sayılar şirketinize aittir ve olmayan bir sayı vermezdim.

Mümkün olduğunda production dışı bir kopya, production tatbikatlarında ise sentetik principal'lar kullanın; böylece okumayı denemek için hiçbir gerçek müşteri verisi okunmaz. Tatbikat kopyaları diğer her kopya gibi kaydedilir ve kapatılır (Bölüm 13).

Trade-off'lar ve maliyetler

İlk maliyet provanın kendisidir. Game day'ler mühendisleri ürün işinden alır ve bir tatbikatın bir sahibi ve bir takvimi olmalıdır. Üç kişilik bir ekip için yılda bir, credential sızıntısını kapsayan tek bir tatbikat makul bir başlangıçtır ve diğer üç senaryo ekip büyüyene kadar bekleyebilir.

İkinci maliyet, sınırlamanın yol açtığı kesintidir. Bir servisin credential'ını hemen revoke etmek o servisteki bir kesintidir ve `ledgerly_access_admin`'i sınırlamak provisioning'i durdurur. İkisi de, bedelini ödeyecek insanlarla birlikte önceden verilecek kararlardır; çünkü gece 3'te karar veren kişi, seçimin neye mal olduğunu da orada öğrenmemelidir.

Üçüncü maliyet, olay tanımlayıcısı konvansiyonudur. Her isteğe bir alan ekler ve bir formun reddedilmesi için bir sebep daha ekler; ancak insanların onu doldurma isteği kadar iyidir. Ona bağlı mutabakat

doldurulmadığında sessizce başarısız olur, bu yüzden gateway alışkanlığına güvenmek yerine alanı zorunlu kılmalıdır.

Dördüncü maliyet, kapsam konusunda dürüstlüktür: bazı olaylar gerçeğinden büyük bir sınırla biter ve ben kayıta olmayan bir kesinliği iddia etmek yerine sınırı raporlardım.

Henüz audit log'u olmayan bir sisteme tüm düzeneği uygulamayın. `blastRadius` kayda ihtiyaç duyar ve `reconcileIncidentAccess` istek kayıtlarına ihtiyaç duyar; bunlar olmayan bir ekip önce onları kurmalıdır (Bölüm 8 ve 14) ve bu arada kimin revoke ettiğini, kimin session'ları kapattığını ve neye dokunulduğu sorusunun kime ait olduğunu adlandıran kısa bir runbook yazmalıdır.

Sık yapılan hatalar

- **Revoke etmeden rotate etmek.** Rotation secret'ı değiştirir ve açık session'ı ve verilmiş yetkileri yerinde bırakır. Cazip gelir, çünkü rotation ekibin zaten sahip olduğu bir rutindir. Önce revoke et, sonra session'ları kapat, sonra rotate et.
- **Sınırlamadan önce kapsam çıkarmak.** Anlama dürtüsü durdurma dürtüsünden önce gelir ve bir ekip, erişim canlı kalırken sorgu üzerinde bir saat harcayabilir. Önce sınırla. Kayıt sınırlamadan sağ çıkar, çünkü erişimin dokunamayacağı bir depoda yaşar.
- **Log'u kontrol etmeden ona güvenmek.** Boşluğu olan bir pencere üzerindeki kapsam sorgusu temiz bir cevap gibi okunur. Önce zinciri ve sırayı kontrol edin; biri başarısız olursa alt sınır raporlayın.
- **Onarım şeridi olarak break-glass.** Hızlıdır ve erişilebilirdir, ama statement review'u yoktur. Küçük sınırlayıcı eylem için kullanın ve onarımı hızlandırılmış bir veri düzeltmesi olarak çalıştırın.
- **Olay erişimini açık bırakmak.** Sebep düzeltildiğinde olay bitmiş gibi hissettirir ve grant'ler kalır. Olayın kapatılmasını mutabakata bağlayın, böylece erişim açıkken ticket kapanamaz.

Temel çıkarımlar

- Aciliyet bir kontrolün hızını değiştirir; hareket eden kimliği, çalışan statement'ı ya da geride kalan kaydı asla değiştirmez.

- Sınırlamanın bir sırası vardır: grant'leri revoke et, session'ları kapat, credential'ları rotate et, kapsamı audit log'dan çıkar, sonra kaydet ve bildir; önceki adım başarısız olsa bile her adım çalışır.
- Revoke, rotation değildir: rotation yeni login'leri durdurur ve zaten açık olanlar için hiçbir şey yapmaz; bu yüzden sıranın bir statik credential önce kesilir, sonra değiştirilir.
- Blast radius, `access_audit_record` üzerinde tek bir sorgudan gelir ve zincirde ya da sırada bir boşluk olduğunda her zaman bir alt sınırdır.
- Olay erişimi requested, granted, used, revoked ve reviewed durumlarından geçer ve bir olay, yarattığı herhangi bir grant hâlâ açıkken kapatılamaz.
- Olay sırasındaki onarımlar, hızlandırılmış review'lu veri düzeltmeleridir ve break-glass zararı durduran küçük eylem için kalır.
- Prova edilmemiş bir yol bir varsayımdır; bu yüzden credential sızıntısı, gateway kesintisi, şüpheli export ve hatalı toplu `UPDATE`'in her biri gözlemlenebilir bir geçme koşuluyla bir game day alır.

Bölüm 16, kitabın tamamındaki kontrolleri bir kontrol listesinde ve bir olgunluk merdiveninde toplar ve her bölümün hizmet ettiği cümleye geri döner.

Bölüm 16 — Bir Erişim Kontrol Listesi ve Kapanış Düşünceleri

Ledgerly'yi, Bölüm 1'deki güvenlik anketinden bir yıl sonra düşünün. Başka bir müşteri adayı aynı iki satırı gönderiyor: "Production veritabanınıza bağlanabilen her kişiyi ve sistemi listeleyin" ve "Ne yaptıklarını nasıl biliyorsunuz?" Alex parola yöneticisi yerine erişim envanterini açıyor, kapıların listesini okuyor ve bir öğleden sonrada geri gönderiyor. Cevapta etkileyici hiçbir şey yok. Kısa, her satırın bir sahibi ve bir sonu var ve hiçbirisi için eşelenmek gerekmedi.

O hafta nöbetçi olan Robin sohbetten soruyor: "Elimizde bir gateway, bir runner, bir pipeline, bir audit store ve üç review job'ı var. Bunlardan hangilerine ihtiyacımız vardı, hangilerini kitap öyle dediği için kurduk?" Alex'in derli toplu bir cevabı yok. Bazı parçalar, bir grant'in kendi kendine süresi dolduğu ilk seferde yerini hak etti; bazıları ise hiç gerçek bir ihtiyaçta kullanılmadı ve Alex bunun çalıştıkları anlamına mı geldiğini, yoksa boşta durdukları anlamına mı geldiğini söyleyemiyor.

Bu bölüm üç ayağa birden hizmet eder: yetkilendirme, yürütme ve hesap verebilirlik. Yeni bir kontrol eklemeyiz. Hedef durumun aşamalarını bir ekibin ulaşabileceği sırayla, örüntüleri yan yana puanlanmış halde ve satırları görüş değil gözlem olan bir kontrol listesini ekler. Cevapladığı soru, Robin'in sorduğu sorudur: bir ekip bundan sonra neyi inşa edeceğine nasıl karar verir ve bir kontrolün çalıştığını nasıl bilir?

Problem ve neden zor olduğu

On beş bölüm okuyucuya çok sayıda kontrol bırakıyor, ama bir sıra bırakmıyor. Her biri kendi gerekçesiyle tanıtıldı ve her gerekçe kendi bölümünde sağlandı. Birlikte okunduklarında bir yıl ya da daha uzun süren bir projeye benziyorlar ve bu listeyi karşılaştıran bir ekip genellikle iki

şeyden birini yapar. Ya en ucuz olandan başlar ve iş siyasilleşince durur, ya da en korkutucu olandan başlar ve ortak parola beş konfigürasyon dosyasında dururken tasarımın ayrıntılı bir parçasını inşa eder.

Zorluk şu ki erişim işinde "bitti"yi görmek zordur. Kâğıt üzerinde var olan bir rol şeması ile uygulanan bir rol şeması, bir review toplantısında aynı görünür; bu yüzden ekipler neyin kurulduğuna göre ölçer ve kurulmuş olmak yanlış birimdir. Yararlı birim, birinin çalıştırabileceği bir testtir: izini sürebilecekleri bir statement, reddedilen bir bağlantı, ortadan kalkmış bir grant.

Ayrıca bu, politikanın ne dediğinden çok sistemin nasıl kurulduğuyla ilgili bir konudur. Politika ne olması gerektiğini söyler; tasarım ise ne olabileceğini ve olduğunda neyin kayıt bıraktığını belirler. Aşağıdaki her satır bir sisteme, bir log'a ya da bir tabloya bakılarak cevaplanabilir; yalnızca birine ne niyet ettiğini sorarak cevaplanabilen bir satır listede yer almaz.

Tez, yerine oturdu

Kitap tek bir cümleyi savundu ve burası, o cümlemin bölümlere karşı sınıandığı yer:

Production veritabanı erişimi bir credential dağıtım problemi değildir; bir yetkilendirme, yürütme ve hesap verebilirlik problemidir.

Cümlemin üç ayağı var ve her ayak belirli bölümler üzerinde durur. Yetkilendirme, grant artı yoldur ve istek başına karara bağlanır. Yürütme, her statement'a olandır. Hesap verebilirlik, kimlik artı kayıttır. Bir parola bunlardan yalnızca birincisine dokunur, o da yalnızca kapıda.

Ayak	Modelin parçaları	Onu kuran bölümler	Okuyucunun artık cevaplayabildiği soru
Yetkilendirme	Grant, Yol	3, 5, 6, 8, 12	Bu principal nerede, ne kadar süreyle ve hangi kolonlar gizliyen ne yapabilir?
Yürütme	Yürütme	7, 9, 10, 11, 13	Bu statement veriye nasıl ulaştı, kim ya da ne inceledi ve onu ne sınırladı?

Ayak	Modelin parçaları	Onu kuran bölümler	Okuyucunun artık cevaplayabildiği soru
Hesap verebilirlik	Kimlik, Kayıt	4, 7, 14, 15	Hangi kişi ya da workload instance'ı, kimin isteğiyle hareket etti ve sonuçta hangi kanıt bağlıdır?

Bölüm 1 ve 2'yi de çerçeveler, Bölüm 15 ise onları baskı altında sınar. Bölüm 7 iki kez geçer, çünkü access gateway hem yoldur hem de bir kişinin, bir log'un adlandırabileceği bir session login'ine dönüştüğü yerdir; Bölüm 13 yürütmenin altında durur, çünkü reporting-job'un export adımı bir statement kümesi çalıştırır ve diğer yan kapılar aynı kontrollerin etrafından dolanan yollardır.

Credential çalışması neden gerekliydi ve neden cevap değildi

Bölüm 4 credential'larla ilgilidir ve çoğu okuyucunun kitabın merkezi olmasını beklediği bölümdür. Gereklidir. Parolasını ortak bir dosyadan okuyan bir servisin kendine ait bir kimliği yoktur ve ayrı kimlikler olmadan kayıt kimseyi adlandıramaz. Kısa ömürlü ve dinamik credential'lar revoke'u rutin bir adım yapar. Workload identity dosyayı ortadan kaldırır.

Bunların hiçbiri, bir principal içeri girdikten sonra ne yapabileceğine, bir statement'ın nasıl kontrol edildiğine ya da neyin yazıya geçirildiğine karar vermez. Bir ekip Bölüm 4'ü kusursuz bitirip yine de her tabloya sahip bir role, her statement'ı yazıldığı gibi commit eden bir laptop session'ına ve her satırın ayrı bir adı olduğu ama hiçbirinin bir gerekçesi olmadığı bir log'a sahip olabilir. Bir parola deposu, Kimlik parçasının ihtiyaç duyduğu ayrı adları ve revoke yoluyla Grant'in bir parçasını sağlar; bir parola yalnızca birinci ayağa dokunur, o da yalnızca kapıda. Diğer parçalar başka yerde kurulur ve kitabın geri kalanı o yerdir.

Bir öneri "X'i iyileştirerek erişimi düzelteceğiz" dediğinde, onaylamadan önce X'i modelin üzerine yerleştirin:

```
function placeControl(proposal):
    parts = partsOfModelTouched(proposal) // Identity, Grant, Path,
    Execution, Record
    if "Execution" not in parts and "Record" not in parts:
        // a control that touches only the door leaves two of three
        pillars as they were
        note("improves how access is obtained; says nothing about what
        it does or what remains")
        for each part in ["Identity", "Grant", "Path", "Execution",
        "Record"]:
            if part not in parts:
                note("unchanged: " + part)
    return notes
```

Fonksiyon bir gate değildir. Bir önerinin hangi parçaları olduğu gibi bıraktığını adlandırır, çünkü boş bir parça bir ihmal değil, bir bulgudur.

Ayaklara göre puanlanmış bir örüntü kataloğu

Bölüm 1 içeri giden yolların aralığını çizdi ve Bölüm 7 somut yolları karşılaştırdı. Aşağıdaki tablo o aralığın beş örüntüsünü yan yana koyar ve her birini üç ayağa göre puanlar. Puanlar benim kanaatimdir, ölçüm değil. Önceki bölümlerin grant'leri ve logging'i eşit tutulduğunda, örüntünün bir ayağın ne kadarını kendi başına kanıtlayabildiğini söylerler. "Güçlü"yi "örüntü bunu kanıtlamak için kurulmuştur", "kısmi"yi "bir kısmını kanıtlar ya da yalnızca başka bir şey eklenirse", "yok"u ise "örüntü burada sessizdir" diye okuyun.

Örüntü	Yetkilendirme	Yürütme	Hesap verebilirlik
Doğrudan erişim (client'tan veritabanına)	Kısmi: yalnızca grant'ler, erişim alanı ağın izin verdiği kadar	Yok	En iyi ihtimalle kısmi: biri provision ederse kişi başına login'ler, yoksa hiçbir
Bastion	Kısmi: ağ yolunu daraltır	Yok: iletir, bir SELECT ile bir DROP arasındaki farkı söyleyemez	Kısmi: host'a kimin login olduğu, neyin çalıştığı değil
Session proxy ya da access gateway	Güçlü: session başına login'ler, süre dolumu, istek başına grant'ler	Kısmi: grant'ler ve guarded setting'lerle sınırlı, statement'lar serbestçe yazılır	Kim ve ne zaman konusunda güçlü, ne konusunda kısmi

Örüntü	Yetkilendirme	Yürütme	Hesap verebilirlik
Statement'a aracılık eden executor	Güçlü	Güçlü: çalışmadan önce incelenir, onaylanmış bir hash'e bağlıdır	Güçlü: statement, onaylayan ve sonuç tek bir kayıta
Ürün yolu (SQL yok)	Güçlü: işlem izindir	Güçlü: işlem sabittir	Kısmi: kişi aracın audit izinde yer alır, veritabanı log'u servis login'ini gösterir

Doğrudan erişim

- **Kanıtladığı:** bir login'in var olduğunu ve grant'lerinin olduğu gibi olduğunu.
- **Kanıtlayamadığı:** ortak bir login'i hangi kişinin kullandığını ya da bir statement'ın ne yaptığını.
- **Maliyet:** kurması sıfır, cazibesi de bu. Her maliyet sonradan, bir haftalık eşelenme olarak gelir.
- **Başarısızlık biçimi:** Bölüm 1'in ortak admin parolası.
- **Şu durumda kullanmayın:** login'e birden fazla kişi ya da workload ulaşabiliyorsa. Tek mühendisli, tek tüketicisi sahibi olan bir sistem için katlanılabilir ve bunun artık doğru olmamasına neyin yol açacağını yazıya dökerdim.

Bastion

- **Kanıtladığı:** bir bağlantının kontrol edilen tek bir host'tan geldiğini ve üzerindeki hesaplar kişi başıysa orada kimin login olduğunu.
- **Kanıtlayamadığı:** herhangi bir statement'ın ne olduğunu, çünkü host veritabanı protokolünü konuşmaz.
- **Maliyet:** yama yapılacak ve erişilebilir tutulacak bir host; karşılığında ağ yolunu kapatabileceğiniz tek bir yer.
- **Başarısızlık biçimi:** ortak bir bastion hesabı, host'u bir sıçrama önceden yeniden ortak parolaya çevirir.
- **Şu durumda kullanmayın:** statement düzeyinde kanıta ihtiyacınız varsa. Kayıt tutan bir şeyin önünde ağ erişimi olarak yararlı kalır.

Session proxy ya da access gateway

- **Kanıtladığı:** adı belli bir kişinin, iki zaman damgası arasında, belirli grant'ler altında belirli bir session login'ine sahip olduğunu.
- **Kanıtlayamadığı:** o session'da yazılan bir statement'ın iyi bir fikir olduğunu. Session bir handout'tur (Bölüm 9) ve grant'lerin izin verdiği her şey yazılabilir.
- **Maliyet:** biri ihtiyaç duyduğunda erişilebilir olması gereken yeni bir sistem ve onunla gelen onay yolu (Bölüm 7 ve 8).
- **Başarısızlık biçimi:** yanında ikinci bir kapı bulunan gateway; bu, daha iyi bir adla ortak paroladır.
- **Şu durumda kullanmayın:** görevler tekrar ediyorsa ve sabit bir biçimi varsa. Onlar, kimsenin session'a ihtiyaç duymadığı bir ürün yoluna aittir.

Statement'a aracılık eden executor

- **Kanıtladığı:** hash'yle tanımlanan bu statement kümesinin adı belli bir kişi tarafından onaylandığını, incelendiğini, bir sınır içinde çalıştığını ve sayılmış bir sonuç ürettiğini.
- **Kanıtlayamadığı:** analizin statement'ı anladığını. Sorgu analizi hataları azaltır ve yetkilendirme sınırı değildir; o sınır grant'lerde kalır (Bölüm 9).
- **Maliyet:** her görev review'u bekler, hiçbir şey etkileşimli değildir ve birinin runner'ı kurması ve ayakta tutması gerekir.
- **Başarısızlık biçimi:** her zaman saniyeler içinde verilen bir onay; bu, ek adımlarla güvenlik tiyatrosudur.
- **Şu durumda kullanmayın:** iş keşif amaçlıysa. Bir durumu anlamaya çalışan kişinin önce bir okuma session'ına ihtiyacı vardır ve `human_readonly` bunun için vardır.

Ürün yolu

- **Kanıtladığı:** tanımlı bir işlemin, bir araçtaki bir rol için, aracın kendi audit iziyle çalıştığını.
- **Kanıtlayamadığı:** kimsenin özellik geliştirmediği görevler hakkında hiçbir şeyi. Serbest biçimli sorular için yol yoktur.
- **Maliyet:** görev başına bir özellik ve beşin en yavaş değişiklik döngüsü.

- **Başarısızlık biçimi:** ilk acil vakanın özelliği olmadığı için "herhangi bir SQL çalıştır" kaçış kapısı biriktiren ürün yolu.
- **Şu durumda kullanmayın:** görev henüz tekrar etmediyse. Bölüm 10, tekrar eden bir düzeltmeyi bir özelliğe çevirmenin sinyalini anlatır ve bir kez olmuş bir görev bu sinyal değildir.

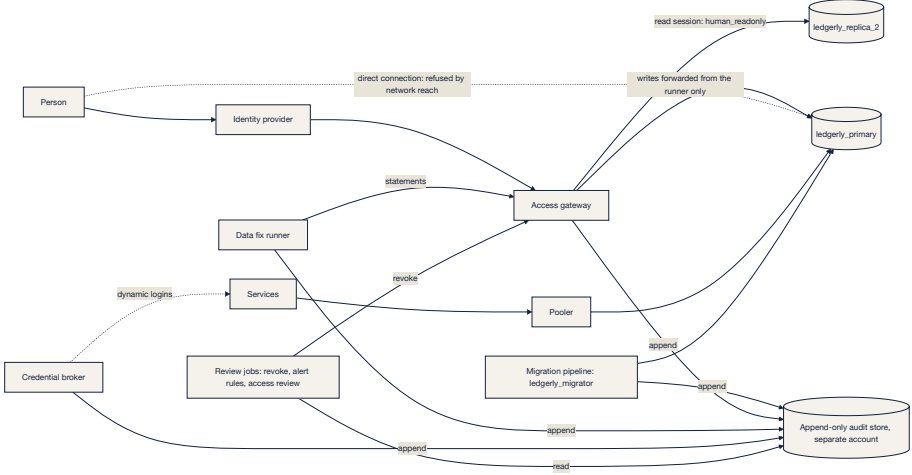
En sağdaki örüntü her zaman en iyisi değildir ve statement'a aracılık eden bir executor, bir müşterinin faturaları hakkındaki bir sorunun ihtiyacından fazlasıdır. Görev başına seçerdim: görevin riske attığı ayakları hâlâ kanıtlayan en ucuz örüntüyle:

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
    not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

Son satır asıl noktadır. Uymayan bir yol, Bölüm 1'in envanter bloğunun istediği gibi, bir sahip ve bir bitiş tarihiyle envantere girer ve ikisinden biri olmayan bir istisna artıktır.

Referans mimari

Diyagram, Ledgerly'nin hedef durumunu kitabın her kontrolüyle tek bir resimde gösterir. Üç şeye bakın: hiçbir insan okunun gateway dışında bir veritabanına ulaşmadığına, bir kişiden gelen her yazmanın runner'dan geçtiğine ve audit store'un veritabanı yöneticilerinin yazamadığı bir hesapta durduğuna.



Resim basitleştirir: Bölüm 6'nın uygulama read replica'ları yazarlarıyla aynı kutuyu paylaşır ve maskelenmiş analitik yük, vendor export'u ve backup'lar (Bölüm 13) dışarıda bırakılmıştır, çünkü her biri bir sahibi ve bir süre dolumu olan bir kopyadır ve her kopyanın üzerinde olduğu bir çizimin okunabilir bir merkezi olmazdı.

Her kutu modele eşlenir: identity provider ve broker Kimlik'i sağlar; roller, session login'leri ve süre dolumu job'ı Grant'i sağlar; noktalı okun arkasındaki ağ kuralı ve gateway Yol'u sağlar; runner ve pipeline Yürütme'yi sağlar; audit store ve review job'ları Kayıt'ı sağlar. Hiçbir parça sağlamayan bir kutu kaldırılmaya adaydır ve Robin'in sorusu bu testi hak eder.

Çıkış testli bir olgunluk merdiveni

Hedef durum bir yön, bir bitiş çizgisi değil; bu yüzden kitabın adımları burada beş aşamada gruplanıyor. Gruplama benim. Bir ekibin, bir sonraki değişiklik kolaylaşmadan önce ne kadarını değiştirebileceğini izliyor ve farklı kuruluşlar onu yeniden sıralayacaktır; savunacağım tek sıralama, envanterin ilk gelmesidir, çünkü onsuz kimse sonraki her adımın neyi kapsaması gerektiğini bilmez. Kayıt tutmak da basit biçimde erken başlamak en iyisidir, çünkü sonraki her aşama kanıtlarla daha kolay değerlendirilir.

Bir aşama, çıkış testi geçtiğinde biter; "kurduk" bir test değildir. Her test, birinin tekrarlayabileceği bir gözlemdir.

Aşama	Ne ekler	Çıkış testi	Ekipler neden burada takılır
1. Kapıları bil	Principal'ların, kapıların, kopyaların envanteri (Bölüm 1, 2, 13)	Tek bir doküman her kapıyı bir sahip ve bir sonla listeler ve yeni bir mühendis "primary'ye kim ulaşabilir?" sorusunu yalnızca ondan cevaplayabilir	Envanter bir proje gibi ele alınır, bu yüzden hiç bitmez ve hiç güvenilmez
2. Her principal'a bir ad ver	Principal başına kendi login'i ve rolü, dinamik credential'lar, rol başına bütçeler (Bölüm 3, 4, 5)	Veritabanı log'unun bir günü ortak bir login içermez ve hiçbir servis owner ya da admin olarak bağlanmaz	Geçiş, her servis için bir kesinti riski gibi görünür, bu yüzden ortak parola "şimdilik" kalır
3. Tek yol, sınırlı	Gateway, replica'lar, JIT ve break-glass, sınıflandırma ve maskeleme (Bölüm 6, 7, 8, 12)	Bir laptop'tan primary'ye doğrudan bağlantı başarısız olur, bir JIT grant'i kimse bir şey yapmadan <code>expires_at</code> anında ortadan kalkar ve bir okuma session'ı hiçbir Restricted kolon döndürmez	Gateway'in tek hata noktası olmasından korkmak ve geceleri cevap veremeyen onaylayanlar
4. Aracılı ve yazılı	Yürütme politikası, runner, review edilmiş düzeltmeler, yalnızca pipeline ile migration'lar (Bölüm 9, 10, 11)	Bir kişinin kendi client'ından yazma reddedilir ve runner, hash'i <code>approved_hash</code> 'ten farklı bir statement kümesini reddeder	Review darboğaz olur, bu yüzden istisnalar ürün özelliklerinden daha hızlı birikir
5. Kanıtlanabilir ve prova edilmiş	Hash zincirli audit store, alarm kuralları, erişim review'u, game day'ler (Bölüm 14, 15)	Dünden seçilen, duyurulmamış bir statement, kimseye sormadan bir kişiye, bir isteğe ve bir sonuca kadar izlenebilir; <code>verifyChain</code> geçer; bir game day, kayıttan doğrulanan revoke edilmiş erişimle biter	Log var ve kimse okumuyor ve bir release gecikince ilk kesilen şey game day oluyor

Bir aşama, bitti ilan edildikten sonra testini geçemeyebilir, çünkü konfigürasyon kayar; bir takvime göre yeniden çalıştırılan test, kayma küçükken onu gösterir.

Not: Tek veritabanı ve dört mühendisi olan küçük bir ekip makul biçimde üçüncü aşamada durabilir ve devam etmesi için neyin gerektiğini yazıya dökebilir. Disiplin yazılı sebeptir, yükseklik değil.

Kontrol listesi

Kontrol listesi ayağa göre gruplanmıştır ve her satır bir gözlemdir. Kullanım konvansiyonu çıkış testleriyle aynıdır: bir "hayır", bir sahibi ve bir tarihi olan bir bulgudur; savuşturulacak bir başarısızlık değil.

```
function runChecklist(items):
  findings = []
  for each item in items:
    answer = observe(item.check)    // read a system, a log or a
    table, never ask what is intended
    if answer != "yes":
      findings.add({ item: item.id, owner: nominate(item), due:
        dateByWhichToDecide })
    // a finding may end in "accepted, with reason, until a date"; it
    may not end in silence
  return findings
```

Yetkilendirme

- ☐ Her tablo `ledgerly_owner`'a aittir ve hiçbir kişi ya da servis onun login'ini tutmaz.
- ☐ Her servis, kendi `app_*` rolünün üyesi olan bir login'le bağlanır ve hiçbir uygulama rolü DDL tutmaz.
- ☐ Hiçbir kişi için kalıcı yazma erişimi yoktur; `human_readonly` varsayılandır ve `human_jit_write` bir tavandır, grant edilmiş bir rol değil.
- ☐ Her JIT ve break-glass grant'inin bir `expires_at` değeri vardır ve `revoke job`'ı ile bağımsız overdue kontrolü yakın zamanda çalışmıştır.
- ☐ Bir laptop'tan `primary`'ye doğrudan bağlantı reddedilir ve `replica`'lar admin login'ini reddeder.
- ☐ Restricted kolonlar hiçbir JIT isteğinde, hiçbir okuma session'ında ve hiçbir analytics kopyasında görünmez ve sınıflandırma kataloğu var olan her kolonu listeler.
- ☐ Her backup, dump, export ve staging kopyasının adı belli bir sahibi ve bir süre dolumu vardır.

Yürütme

- ☐ İnsanlardan ve otomasyondan gelen statement'lar veriye yürütme politikası üzerinden ulaşır ve `deny` / `gate` / `allow` matrisi bir sahiple birlikte version control'dadır.

- ☐ Yazma session'ları auto-commit kapalı ve bir row cap ile çalışır ve guarded setting'ler bir session'ın içinden değiştirilemez.
- ☐ Bir JIT yazması review edilmiş bir düzeltmeyi adlandırır ve runner, hash'i `approved_hash`'ten farklı olan her statement kümesini reddeder.
- ☐ Owner-link değişiklikleri ve Restricted kolonlara yazmalar her lane'de ürün yoluna gider.
- ☐ Şema değişiklikleri pipeline'dan `ledgerly_migrator` olarak çalışır ve hiçbir kişi bu login'i tutmaz.
- ☐ Export adımı dar bir rolle çalışır ve dosyasının adı belli bir sahibi vardır.

Hesap verebilirlik

- ☐ Her login tam olarak bir kişiye ya da bir workload instance'ına eşlenir ve veritabanı log'u bunu gösterir.
- ☐ Her yazma ve her yükseltilmiş session kaydı isteği, onaylayanı, fingerprint'i ve döndürülen satırları taşır ve literal'ler redact edilmiştir.
- ☐ Audit store, veritabanı yöneticilerinin ve `ledgerly_access_admin`'in yazamadığı bir hesapta durur ve `verifyChain` geçer.
- ☐ Bir sıra boşluğu, veritabanı session'ı olmayan bir gateway session'ı ya da tersi bir bulgu üretir.
- ☐ `evaluateAlertRules` ve `runAccessReview` son döngüde çalıştı ve önerdikleri revoke'lar uygulandı ya da bir gerekçeyle kabul edildi.
- ☐ `containCompromise` bir game day'de prova edildi ve blast radius sorgusu kayıttan cevaplandı.

Kültür: güvenli yol kolay yol olmalı

Kitaptaki her kontrol, onu kullanan kişiye bir maliyettir ve o kişi, haftanın en kötü gecesinde bunu ödeyip ödemeyeceğine karar verir. Yavaş ya da sık çöken bir kontrol atlanır, çünkü tasarım güvenli yolu zor yol yapmıştır ve bir olay beklemes. Yararlı tepki, atlamayı tasarımdaki bir kusur olarak ele almaktır.

Kültürün merkezine tek bir kural koyardım ve onu ölçülebilir yaptım:

```
function reviewBypass(signal):
    // signal: someone used a second door, a shared login, or an
    unplanned break-glass
    cause = ask("what made the intended path unusable at that
    moment?")
    if cause in ["slow approval", "gateway unavailable", "missing
    product feature", "grant too narrow"]:
        // a design defect: repair the path, and note the date by
        which it should be easy
        openFinding(cause, owner: "platform", due: nextReview)
    else if cause == "unknown":
        // an unexplained bypass stays open until it is understood
        keepOpen(signal)
    // nothing here assigns the signal to a person
    return cause
```

İki alışkanlık kuralı destekler. Break-glass rolünün ne sıklıkla ilan edildiğini sayın ve her seferinde onun yerine bir JIT yolunun ya da bir ürün özelliğinin hizmet etmesi gerekip gerekmediğini sorun; artan bir sayı normal yolun fazla yavaş olduğunu söyler, bir yıl boyunca sıfır sayısı ise rolün çalıştığı ya da kimsenin buna izin verileceğine inanmadığı anlamına gelebilir; bunu bir game day çözer. Ve onaylayanlara geceleri cevaplayabilecekleri bir şey verin: saatler süren bir onay isteyenlere ihtiyaçlarından fazlasını istemeyi öğretir, her zaman saniyeler içinde verilen onay ise kimseye onu okumayı öğretmez (Bölüm 8, onaylayanın kontrol listesini kapsar).

Trade-off'lar ve maliyetler

Yalnızca tavsiye eden bir kapanış bölümü, kitabın uyardığı hatayı tekrarlardı; o yüzden tüm tasarımın neye mal olduğu burada. Referans mimarideki sistemin çok sayıda parçası var ve her birinin, birinin ihtiyaç duyduğu anda erişilebilir olması gerekir. Gateway, runner, pipeline, broker, audit store ve review job'larının her birinin bir sahibi, bir on-call yolu ve bir hata biçimi vardır. Küçük bir platform ekibi zamanını kontrolleri çalışır tutmakla geçirdiğini görebilir ve bu, kaldırdığı maruz kalmaya karşı konulması gereken gerçek bir maliyettir.

Kontrol listesinin ve merdivenin de kendi maliyetleri var. Üç ayda bir okunan bir kontrol listesi, bulgular kapatılmadıkça bir ritüele dönüşür ve ekipler aşama adı için optimize ettiğinde merdiven bir durum raporuna dönüşür; çıkış testi bunun panzehiridir, çünkü yeniden çalıştırması

ucuzdur ve taklit etmesi zordur. "Güçlü" puanı, bir örüntünün neyi kanıtlamak için kurulduğunu söyler, belirli bir kurulumun bunu yaptığını değil.

Bu bölümün düzeneğinin ne zaman kullanılmayacağı: tek veritabanı olan ve vendor'ı olmayan tek mühendisli bir sistemin ihtiyacı, kimsenin kullanmadığı bir owner login'i, salt okunur bir rol, restore edilebilen bir backup ve bir sayfalık bir envanterdir. Kitap, principal sayısının ya da bir hatanın maliyetinin bunu aştığı nokta içindir ve o noktanın şirketiniz için nerede olduğunu bilmiyorum; veriye kaç kişinin ulaşabildiğine, ne kadarının Restricted olduğuna ve bir hatanın ne kadar süre görünmez kaldığına bağlıdır.

Sık yapılan hatalar

- **Kontrol listesini sertifika saymak.** Bir turu bitirip sonucu dosyalamak cazip gelir. Kontrol listesi, okunduğu günde kanıtın nerede eksik olduğunu gösterir. Onu bir takvime göre yeniden çalıştırın ve bulguları, her biri çözülene ya da bir gerekçe ve tarihle kabul edilene kadar açık tutun.
- **En sağdaki örüntüyü önce kurmak.** Statement'a aracılık eden bir executor, en çok kontrol gibi görüldüğü için çekicidir. Altında envanter ve ayrı principal'lar olmadan, ortak bir login'in statement'larına aracılık eder. Merdiveni sırayla çıkın ya da farklı bir sıranın size neden uyduğunu yazın.
- **Acil durumlar için ikinci bir kapı eklemek.** Dayanıklılık gibi görünür ve ortak parolayı yeniden kurar. Bir ekip gateway olgunlaşırken buna ihtiyaç duyuyorsa, bir sahibi, kullanımda bir alarmı ve bir emeklilik koşulu olmalıdır (Bölüm 8).
- **Bir atlamayı davranış sorunu olarak okumak.** Biri bir kontrolün etrafından dolaştığında, tasarım bunu amaçlanan yoldan ucuz yapmıştır. O anda amaçlanan yolun neye mal olduğunu sorun ve onu değiştirin.

Temel çıkarımlar

- Production veritabanı erişimi bir yetkilendirme, yürütme ve hesap verebilirlik problemidir ve bir parola bunlardan yalnızca birincisine

dokunur, o da yalnızca kapıda.

- Bölüm 4'ün credential çalışması gereklidir, çünkü kimlik kaydı mümkün kılar; yeterli değildir, çünkü bir principal'ın ne yapabileceği ya da bir statement'ın ne yaptığı hakkında hiçbir şey söylemez.
- Doğrudan erişim, bastion, access gateway, statement'a aracılık eden executor ve ürün yolu arasında görev başına seçin; görevin riske attığı ayakları hâlâ kanıtlayan en ucuz örüntüyle.
- Bir olgunluk aşaması çıkış testi geçtiğinde biter ve birinin yeniden çalıştırabileceği bir test, kaymadan sağ çıkan tek "bitti" biçimidir.
- Kontrol listesinin her satırı bir sistemin, bir log'un ya da bir tablonun gözlemidir; bir "hayır", bir sahibi ve bir tarihi olan bir bulgudur, bir kişi hakkında asla bir hüküm değil.
- Güvenli yol kolay yol olmalıdır ve her atlama, tasarımın amaçlanan yolu nerede pahalı yaptığı hakkında bilgidir.
- Kontroller blast radius'u azaltır ve davranışı görünür kılar; availability'ye, dikkate ve beklemeye mal olurlar, bu yüzden bu maliyetleri benimsemeden önce ve ödendiklerinde yeniden adlandırın.

Alex, Robin'in sorusunu ertesi sabah bir savunmayla değil bir testle cevaplıyor: bir parça, modelin başka hiçbir şeyin sağlamadığı bir parçasını sağlıyorsa ve çıkış testi geçiyorsa kalır. Anket yeniden gelecek, başka bir müşteriden ya da başka bir denetçiden ve aynı iki satır üzerinde olacak. Kitabın size bırakabileceği şey, ilk satırı bir dokümandan, ikincisini bir kayıttan cevaplayan ve nöbetçi kişinin kahramanca bir çaba göstermeden doğru olanı yapmasına izin veren bir sistemdir.

Ek

Bu ek, kitabın dayandığı üç ayak için başvuru malzemesidir: yetkilendirme, yürütme ve hesap verebilirlik. Altı kısım içerir: bir sözlük, Bölüm 16'da puanlanan örüntülerin hızlı başvurusu, Bölüm 9'un deny / gate / allow matrisi, denetçi sorularından artifact'lara bir eşleştirme ve iki şablon; biri Bölüm 14'ün erişim review'u, biri Bölüm 15'in olay sonrası review'u için.

Burada yeni bir şey yok. Bir tablo bir bölümdekini tekrar ediyorsa hücreler bölümün kendi hücreleridir ve ikisi farklıysa kaynak bölümdür. Herhangi bir rejimin yorumu hukuk ve uyum ekibinize aittir; eşleştirme mühendislik artifact'larını listeler ve fazlasını iddia etmez.

A. Sözlük

Terimler alfabetik olarak, her birini kuran bölümle birlikte listelenmiştir. Bu, kitabın kendi sözcük dağarcığı için son başvurusudur: her girdi kısa biçimdir ve adı geçen bölüm tam anlatımı taşır.

Terim	Anlamı
access gateway	Bir insanı doğrulayan, kısa ömürlü bir credential veren ve session'ı kaydeden, kimliğin farkında bir aracı. Bir bastion değildir ve bir veritabanı proxy'si değildir (Bölüm 7).
access_audit_record	Her statement için yazılan girdiler; biri intent türünde, biri outcome türünde olmak üzere, principal'ı, login'i, isteği, statement fingerprint'ini, dokunulan tabloları ve kolonları ve outcome üzerinde satır sayılarını ve sonucu tutar. Modelin Kayıt parçasıdır (Bölüm 14).
ad-hoc sorgu	Bir kişinin yazıp bir kez çalıştırdığı, herhangi bir uygulamanın dışındaki sorgu (Bölüm 10).
allowlist, denylist	İzin verilenlerin listesi ve reddedilenlerin listesi. Asla "whitelist" ya da "blacklist" değil (Bölüm 7).
analytics kopyası	ledgerly_analytics deposu. Production verisinin bir kopyasıdır, replica değildir (Bölüm 6).

Terim	Anlamı
approved hash	Bir erişim isteğinin <code>approved_hash</code> değeri: onaylanan statement kümesinin hash'i; review edilmiş düzeltmenin merge kaydından ya da onaylanmış istek kaydından alınır, runner'ın girdisinden asla (Bölüm 9, 10).
aracılı yürütme	Principal'ın hiçbir zaman bir bağlantı tutmadığı bir mod: bir statement gönderir, gateway ya da bir runner onu analiz eder ve runner çalıştırır (Bölüm 9).
audit log	Kimin hangi veriye ne yaptığının, aktörün değiştiremeyeceği bir yerde tutulan kaydı. <code>audit_events</code> tablosu uygulamanın iş kaydır, audit log değildir (Bölüm 14).
bastion	Korunan bir sisteme giden tek ağ yolu olan, sertleştirilmiş bir host. Trafiki iletir ve veritabanı protokolünü anlamaz (Bölüm 7).
blast radius	Bir principal, credential ya da eylem ters giderse ne kadarının etkilenebileceği. Her zaman belirli bir şey hakkında söylenir (Bölüm 1, 2).
break-glass	Ağır logging ve zorunlu review'lu acil erişim. Rol <code>human_breakglass</code> 'tır ve istek kendi kendine ilan edilir (Bölüm 8).
credential	Bir principal'ın kimliğini doğrulamak için sunduğu şey: bir parola, bir sertifika, bir token. Credential, bir tür secret'tır (Bölüm 4).
deny / gate / allow	Yürütme politikasının cevapları, her zaman bu sırayla. Deny bu yol üzerinden asla çalışmaz. Gate yalnızca ek onaydan sonra ya da yalnızca ürün yolu üzerinden çalışır. Allow session'ın grant'leri içinde çalışır (Bölüm 9).
dinamik credential	İstek üzerine tek bir principal için üretilen kısa ömürlü bir credential; böylece iki istek asla aynı değeri almaz (Bölüm 4).
en az yetki	Bir görevin ihtiyaç duyduğu şeyi, ihtiyaç duyduğu süre boyunca, bunu yapmanın maliyeti sayılarak grant etmek (Bölüm 1, 3).
engine'den bağımsız	Kitabın, hiçbir veritabanı engine'ine ait olmayan notasyon ve tavsiye için kullandığı sözcük (Önsöz).
erişim review'u	Her principal'ın erişiminin hâlâ gerekli olup olmadığına dair zamanlanmış bir kontrol. Grant edilenle kullanımları karşılaştırır ve adı belli bir reviewer karar verir (Bölüm 14).
four-eyes	Riskli bir eylem çalışmadan önce ikinci bir kişinin review ettiği onay. Kimin onayladığını söyler, neyin onaylandığını değil (Bölüm 8).
grant	Bir role bir yetki vermek, bir login'e ya da başka bir role bir rol vermek ya da yalnızca JIT için, adı verilmiş kolonlar üzerindeki bir yetkiyi doğrudan bir session login'ine vermek. Ortaya çıkan kayıt da (Bölüm 3).

Terim	Anlamı
guarded settings	Bir principal'ın bir session'ın içinden değiştiremediği session ayarları: statement timeout, row cap, auto-commit modu, salt okunur varsayılan, rol. Yalnızca bir mekanizma zorunlu kıldığında non-overridable olarak adlandırılır (Bölüm 5, 9).
güvenlik tiyatrosu	Maruz kalmayı azaltmadan sürünme ekleyen bir kontrol (Bölüm 2, 8).
hash chain	Her girdinin bir öncekinin hash'ini taşıdığı bir kayıt biçimi; böylece değiştirilmiş ya da kaldırılmış bir girdi verifyChain'i bozar. Kurcalanma kanıtı verir ve kurcalamayı önlemez (Bölüm 14).
just-in-time (JIT) erişim	Belirli bir görev için ve sınırlı bir süre boyunca verilen erişim (Bölüm 8).
kalıcı erişim	Kullanılıp kullanılmadığına bakılmaksızın sürekli var olan erişim (Bölüm 1).
kayıt	Hangi kanıtın kaldığını ve onun statement'ı kişiye, isteğe ve sonuca bağlayıp bağlamadığını soran model parçası (Bölüm 14).
kısa ömürlü credential	Kısa, tanımlı bir süre içinde kendi kendine süresi dolan bir credential (Bölüm 4).
kimlik	Bir principal'ın söylediği kişi olduğuna dair doğrulanmış iddia. Bir rol, bir kimlik değildir (Bölüm 1).
kolon düzeyinde grant	Bir tablonun adı verilmiş kolonları üzerinde bir yetkinin grant edilmesi. Engine'de bu yoksa Restricted kolonlar tavanın dışındaki ayrı tablolarda yaşar ve JIT grant'leri tablo düzeyine iner (Bölüm 3, 8).
login	Bir principal'ın bağlandığı veritabanı düzeyindeki hesap (Bölüm 1, 3).
maskeleme	Gerçek değer görünmesin ama biçim kullanılabilir kalsın diye değerleri değiştirmek ya da gizlemek. Statik maskeleme bir kopyayı değiştirir, dinamik maskeleme bir sorgunun döndürdüğünü değiştirir (Bölüm 12).
migration	ledgerly_migrator tarafından çalıştırılan, versiyonlu bir şema değişikliği. "Şema değişikliği" genel fikirdir, "migration" artifact'tır (Bölüm 11).
minimizasyon	Kimsenin ihtiyaç duymadığı veriyi toplamamak, kopyalamamak ya da tutmamak (Bölüm 12).
otomasyon principal'ı	Kişi olmayan bir principal: bir job, bir runbook, bir ChatOps komutu, bir AI agent. Bir workload identity'si ve salt okunur bir varsayılanı vardır ve kayıt hem onu hem de onu tetikleyen kişiyi adlandırır (Bölüm 9).

Terim	Anlamı
owner-link değişikliği	Bir satırı sahibi olan müşteriye bağlayan bir kolonun UPDATE'ı; örneğin bank_accounts üzerindeki customer_id. Gate'lenir ve ürün yoluna yönlendirilir (Bölüm 9).
PII	Kişiyi tanımlayabilen bilgi. "Hassas veri", Restricted ve Confidential verinin şemsiye terimidir (Bölüm 2).
primary, replica	Yazılabilir veritabanı ve onun okuma kopyaları. Analytics kopyası bir replica değildir (Bölüm 6).
principal	Kimlik doğrulayan her şey: bir kişi, bir servis, bir job, bir vendor entegrasyonu (Bölüm 1).
redaction	Bir değeri bir çıktıdan ya da bir log'dan tamamen kaldırmak (Bölüm 12).
revoke, rotation	Revoke, bir credential'ın ya da grant'in çalışmasını hemen durdurur. Rotation, bir credential'ı bir takvime göre ya da şüphe üzerine değiştirir ve zaten açık olan session'ları kapatmaz (Bölüm 4, 15).
rol	Veritabanında yetkilerin adlandırılmış bir demeti. Bir rol asla bir kişi değildir (Bölüm 3).
runbook	Tekrarlanabilir bir görev için yazılı, izlenen prosedür (Bölüm 10).
runner	Review edilmiş veri düzeltmesi runner'ı. Bir statement kümesini yalnızca hash'i approved_hash'e eşitse çalıştırır ve gateway JIT yazmalarını yalnızca ondan iletir (Bölüm 10).
secret	İfşa edilirse erişim sağlayan her değer (Bölüm 4).
session	Bir principal'ın kimliği doğrulanmış tek bir bağlantısı ya da gateway aracılığıyla yapılan bir etkileşimi (Bölüm 5, 7).
session handout	Gateway'in principal'a bir session login'i ve bir bağlantı verdiği ve statement'ları principal'ın kendi client'ının gönderdiği bir mod. Login üzerindeki grant'ler her şeyi sınırlar (Bölüm 9).
sınırlama	Tek bir principal'ın erişimini sabit bir sırayla durdurmak: grant'leri revoke et, session'ları kapat, credential'ları rotate et, hasarın kapsamını çıkar, kaydet ve bildir (Bölüm 15).
sorgu analizi	Bir statement'ı çalışmadan önce incelemek: türü, dokunduğu tablolar ve kolonlar, bir yazmanın sınırlı bir satır korumasının olup olmadığı ve beklenen satır sayısı. Hataları azaltır ve yetkilendirme sınırı değildir (Bölüm 9).
statement'a bağlı onay	Yalnızca bir zaman penceresine ya da bir yetki kümesine değil, çalışacak tam statement kümesine, approved_hash ile tanımlanarak bağlanan onay (Bölüm 8, 9, 10).
statik credential	Biri değiştirene kadar aynı kalan bir credential (Bölüm 4).

Terim	Anlamı
tavan	Bir JIT isteğinin isteyebileceği azami şey; <code>human_jit_write</code> ile tanımlanır. Asla bir Restricted kolon değil (Bölüm 3, 8).
tokenization	Bir değeri, yalnızca kontrollü bir servisin geri eşleyebildiği bir vekille değiştirmek. Ne şifrelemedir ne maskeleye (Bölüm 12).
ürün yolu	Bir araçtaki tanımlı bir işlem; örneğin <code>admin-console</code> 'un <code>payments-service</code> 'ten bir Restricted kolonu değiştirmesini istemesi, kişi için SQL olmadan (Bölüm 9, 10).
veri düzeltmesi	Production verisinde, uygulamanın onaramayacağı bir durumu onarmak için normal uygulama yolunun dışında bir kişi tarafından yapılan değişiklik (Bölüm 10).
veri kopyası	Production verisinin çoğaltıldığı her yer: replica'lar, analytics kopyası, backup'lar, dump'lar, export'lar, staging verisi (Bölüm 6, 13).
veri sınıflandırması	Veriyi, işleme kurallarına karar veren adlandırılmış sınıflara ayırmak (Bölüm 12).
veri sınıfları	Verinin dört adlandırılmış sınıfı: Restricted (ifşasının geri alınması zordur; örneğin <code>iban</code> ve <code>token_value</code>), Confidential (müşteriler ve kişiler hakkındaki kişisel ya da ticari veri), Internal (kişisel içeriği olmayan operasyonel veri), Public (yayınlanması güvenli). Sınıf adı olarak kullanıldığında büyük harfle yazılır (Bölüm 2, 12).
veritabanı proxy'si	Veritabanı bağlantılarını havuzlayan, yönlendiren ya da filtreleyen, protokolü bilen bir aracı. Konu yalnızca havuzlamaysa pooler denir (Bölüm 5, 7).
workload identity	Platformun çalışan bir workload'a verdiği, saklanmış bir parola olmadan kimlik doğrulayabilmesini sağlayan kimlik (Bölüm 4).
yan kapı	Primary'nin erişim kontrollerini atlayan bir veri kopyası ya da export yolu (Bölüm 6, 13).
yetki	Bir grant'ın izin verdiği belirli bir şey; örneğin bir tablo üzerinde <code>SELECT</code> (Bölüm 1).
yürütme	Bir statement'ın veriye nasıl ulaştığı: onu kim ya da ne gönderir, çalışmadan önce bir şey inceler mi, etkisini ne sınırlar ve nasıl commit edilir (Bölüm 1, 9).

Bu çiftlerden birkaçı birbirine karıştırılır ve kısa bir kural, tartışma başlamadan her birini çözer.

Soru şuysa	Kullan	Kullanma
Statement'lardan habersiz, ağ trafiğini ileten bir host	bastion	access gateway
Kişi başına session login'leri veren ve session'ları kaydeden bir aracı	access gateway	veritabanı proxy'si
Bağlantıları havuzlamak ya da yönlendirmek, insanlarla ilgisi yok	pooler ya da veritabanı proxy'si	access gateway
Kişinin kendi client'ı, gateway'in verdiği bir bağlantı üzerinden statement gönderir	session handout	aracılı yürütme
Kişi bir statement gönderir ve hiçbir zaman bir bağlantı tutmaz	aracılı yürütme	session handout
Bir credential'ın çalışmasını hemen durdurmak	revoke	rotation
Riskli bir eylemi kimin onayladığı	four-eyes	statement'a bağlı onay
Tam olarak neyin, hash ile onaylandığı	statement'a bağlı onay	four-eyes
Uygulamanın iş kaydı	audit_events	audit log

B. Örüntü Hızlı Başvurusu

Bölüm 16, beş örüntüyü üç ayağa göre puanlar. Puanlar kitabın kanaatidir, ölçüm değil: "güçlü", örüntünün ayağı kanıtlamak için kurulduğu; "kısmi", bir kısmını kanıtladığı ya da yalnızca bir şey eklenirse kanıtladığı; "yok" ise sessiz olduğu anlamına gelir. Maliyet sütunu, Bölüm 16'nın verdiği sütundur, kısaltılmıştır.

Örüntü	Neyi çözer	Maliyet	Yetkilendirme	Yürütme	Hesap verebilirlik	Bölümler
Doğrudan erişim (client'tan veritabanına)	Bir login vardır ve grant'leri olduğu gibiydi	Kurması sıfır; her maliyet sonradan, bir haftalık eşelenme olarak gelir	Kısmi: yalnızca grant'ler, erişim alanı ağın izin verdiği kadar	Yok	En iyi ihtimalle kısmi: biri provision ederse kişi başına login'ler, yoksa hiçbiri	1

Örüntü	Neyi çözer	Maliyet	Yetkilendirme	Yürütme	Hesap verebilirlik	Bölümler
Bastion	Kontrol edilen tek bir host, ağ yolunu kapatmak için tek bir yer	Yama yapılacak ve erişilebilir tutulacak bir host	Kısmi: ağ yolunu daraltır	Yok: iletir, bir <code>SELECT</code> ile bir <code>DROP</code> arasındaki farkı söyleyemez	Kısmi: host'a kimin login olduğu, neyin çalıştığı değil	7
Session proxy ya da access gateway	Adı belli bir kişi, iki zaman damgası arasında, belirli grant'ler altında belirli bir session login'ine sahiptir	Biri ihtiyaç duyduğunda erişilebilir olması gereken yeni bir sistem ve onunla gelen onay yolu	Güçlü: session başına login'ler, süre dolumu, istek başına grant'ler	Kısmi: grant'ler ve guarded setting'lerle sınırlı, statement'lar serbestçe yazılır	Kim ve ne zaman konusunda güçlü, ne konusunda kısmi	7, 8
Statement'a aracılık eden executor	Bu tam statement kümesi, hash'iyile, onaylandı, incelendi, sınırlandı ve sayıldı	Her görev review'u bekler, hiçbir şey etkileşimli değildir, biri runner'ı kurar ve ayakta tutar	Güçlü	Güçlü: çalışmadan önce incelenir, onaylanmış bir hash'e bağlıdır	Güçlü: statement, onaylayan ve sonuç tek bir kayıttadır	9, 10
Ürün yolu (SQL yok)	Bir araçtaki bir rol için tanımlı bir işlem çalıştı	Görev başına bir özellik ve beşin en yavaş değişiklik döngüsü	Güçlü: işlem izindir	Güçlü: işlem sabittir	Kısmi: kişi aracın audit izinde yer alır, veritabanı log'u servis login'ini gösterir	9, 10

Aralarından seçmek görev başına bir karardır ve kural Bölüm 16'nın verdiği kuraldır: görevin riske attığı ayakları hâlâ kanıtlayan en ucuz örüntü.


```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
    not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

C. Tehlikeli İşlemler Matrisi

Aşağıdaki matris, Bölüm 9'un yürütme politikasıdır ve hücreleri değiştirilmeden tekrarlanmıştır. Satırlar, sınıflandırıcının ürettiği statement sınıflarıdır. Sütunlar, bir statement'ın gelebileceği lane'lerdir. Her hücrenin üç cevaptan biri vardır: deny (bu yol üzerinden asla çalışmaz), gate (henüz sahip olmadığı bir onaya ihtiyaç duyar ya da farklı bir güzergâha aittir), allow (döngünün sınırları içinde çalışır ve bir hücre "bound" dediğinde, yalnızca adı geçen kaynaktan gelen sınırın içinde).

Statement sınıfı	İnsan okuma session'ı	JIT yazma	Break-glass	Veri düzeltmesi runner'ı	Migration
Okuma, row cap dahilinde	Allow	Allow	Allow	Allow	Allow
Sınırlı yazma, onaylanan sayı dahilinde	Deny	Allow, düzeltmeye bağlı (yalnızca runner)	Allow, bir row cap'e bağlı (not gerekli)	Allow, onaylanan hash'e ve sayıya bağlı	Allow, backfill kuralına bağlı
Sınırsız yazma	Deny	Deny	Deny	Deny	Gate: adı belli onaylayan

Statement sınıfı	İnsan okuma session'ı	JIT yazma	Break-glass	Veri düzeltmesi runner'ı	Migration
Owner-link değişikliği	Deny	Gate: ürün yolu	Gate: ürün yolu	Gate: ürün yolu	Gate: adı belli onaylayan
Restricted bir kolona yazma	Deny	Gate: ürün yolu	Gate: ürün yolu	Gate: ürün yolu	Gate: adı belli onaylayan
Export	Gate: kademeye göre	Deny	Deny	Deny	Gate: adı belli onaylayan
Şema değişikliği (DDL)	Deny	Deny	Deny	Deny	Allow: review edilmiş lane
Yetki değişikliği	Deny	Deny	Deny	Deny	Gate: platform review'u
ledger_entries ya da audit_events tablosuna yazma	Deny	Deny	Deny	Deny	Gate: platform review'u

Üç okuma kuralı aramaların çoğundan kurtarır.

- **Bir kişinin elindeki yazma ya denied'dır ya da bound'dur.** İnsan okuma session'ının yazma yolu yoktur, JIT yazmaları yalnızca runner'dan gelir ve break-glass, bir kişinin doğrudan yazdığı tek lane'dir; bir row cap altında ve bir notla.
- **Bir gate, statement'ın sırada nereye gittiğini adlandırır.** "Ürün yolu", admin-console'un payments-service'ten istekte bulunması, ikinci bir kişinin onaylaması ve değişikliğin audit_events'e kaydedilmesi demektir. "Adı belli onaylayan" ve "platform review'u", migration şeridinin statement'ı, evet demesi gereken bir kişi için işaretlemesi demektir.
- **Matris yetkilendirme sınırı değildir.** Sorgu analizi hataları azaltır. Login üzerindeki grant'ler sınır olarak kalır ve sınıflandırıcının yanlış okuduğu bir statement yine onlarla sınırlanır (Bölüm 9).

Tek bir statement için karar, bir kural olarak yazılmış hâliyle:

```

function answerFor(statementClass, lane):
    // the matrix above is the only source; no lane has its own rule
    set
        cell = matrix[statementClass][lane]
        if cell starts with "Deny":
            refuse and record the refusal
        else if cell starts with "Gate":
            // the text after the colon names the route or the approver
            hold the statement; route it as the cell says; do not run it
    as sent
        else:
            // "bound" cells compare the real count with the bound before
    COMMIT
        run within the loop's limits

```

D. Uyum Eşleştirmesi

Veri koruma ve ödeme kartı rejimleri, erişim hakkında bir soru ailesi sormaya eğilimlidir. Bu rejimlerden birinin altında çalışan bir denetçi genellikle bir kontrolün var olduğunu, çalıştığını ve sonuçlarına bakıldığını görmek ister. Bu eşleştirme soru türlerini, her birini cevaplayan kontrolü, bıraktığı artifact'ı ve onu kuran bölümü listeler. Bölüm 14'ün tablosudur, tekrarlanmıştır ve hiçbir kontrolün hiçbir yükümlülüğü karşıladığı iddiasında bulunmaz.

Soru	Kontrol	Artifact	Bölüm
Production verisine kim, hangi yoldan ulaşabilir?	Tek kapı olarak gateway, kod olarak rol şeması, drift kontrolü	Ağ kuralı listesi, rol tanımları, drift raporu	3, 7
Belirli bir statement'ı kim, hangi grant'ler ve onayla çalıştırdı?	İstek id'si ve onaylayanla <code>access_audit_record</code>	Kayıt ve <code>verifyChain</code> raporu	14
Yükseltilmiş erişim ne kadar sürer ve onu kim onayladı?	Onaylayan ve <code>expires_at</code> içeren JIT isteği, revoke job'ı	İstek kayıtları, revoke log'u, overdue-grant kontrolü	8
Hassas veri nasıl tanımlanır ve sınırlanır?	Şemada sınıflandırma, maskeleme, saklanan kolonlar	Katalog, maskelenmiş yükün doğrulaması	12
Production verisinin kopyaları nerede ve her birinin sahibi kim?	Sahip ve bitiş tarihi olan kayıt defteri, sweeper	Kayıt defteri, sweeper log'u	6, 13

Soru	Kontrol	Artifact	Bölüm
Şemayı kim değiştirir ve kim review etti?	Migration pipeline'ı, korumalı branch, harici başlangıç girdisi	Commit'lerle ve audit girdileriyle eşleştirilmiş migration kayıtları	11
Erişim hâlâ gerekli mi?	Adı belli reviewer'larla erişim review'u	Kararlarıyla tamamlanmış review	14
Acil durumda ne olur?	Alarm ve zorunlu review'lu break-glass	Review kayıtları, alarm geçmişi	8, 15
Erişimin kendisini kim yönetir?	Ayrı üreticiler olarak <code>ledgerly_access_admin</code> login'leri	Akışları ve katalog-kayıt karşılaştırması	3, 14

Var olan ama hiç çalışmamış, ya da çalışıp hiç bakılmamış bir kontrol, sorunun yalnızca üçte birini cevaplar. Aşağıdaki kontrol, bir artifact'ı onu isteyen birine vermeden önce çalıştırılacak olandır.

```
function evidenceStatus(row):
  // three things an auditor tends to want, from Chapter 14
  if row.artifact is null:
    return "control not evidenced: build or find the artifact"
  if row.artifact.lastProducedAt is null:
    return "control exists, no evidence it ran"
  if row.artifact.lastReviewedBy is null:
    return "control ran, results not looked at"
  return "evidenced: attach artifact and its review"
```

E. Erişim Review Şablonu

Bu, Bölüm 14'ün `runAccessReview` fonksiyonunun arkasındaki şablondur. Her review döngüsü için kopyalayın. Review önerir ve adı belli bir reviewer karar verir; kimse kendi erişimini review etmez ve dönem, review edilen workload'un en uzun döngüsünden uzundur (bir aylık bir review, üç ayın ikisinde üç aylık bir raporun kullandığı bir grant'in revoke edilmesini önerirdi).

Üst bilgi

Review cycle: <period start> to <period end>
 Reviewer(s): <names; none is the owner of an item they review>
 Record queried on: <date>, verifyChain result: <passed | failed: details>
 Period chosen and why: <longest workload cycle covered?>

Kesim 1: kullanıma karşı ilan edilmiş grant'ler

Grant (rol, yetki, tablo ya da login)	Sahip	Kayıttaki son kullanım	Bulgu	Karar	Karar veren	Tamamlanma tarihi
<role>, <privilege>, <table>	<named principal>	<date or none>	kullanılmıyor / sahibi yok / kullanım grant'ten dar / sorun yok	koru / <privilege> düzeyine daralt / revoke et / sahip ata	<reviewer>	<date>

Kullanımı yalnızca tabloya göre değil, yetkiye göre eşleştirin. Her gece select yapan ve hiç update yapmamış bir rol, S, U'dan S'ye daraltılmaya adaydır. Identity provider grup-rol eşlemelerini ve ledgerly_access_admin login'lerini dahil edin.

Kesim 2: grant olmayan şeyler

Madde	Soru	Cevap	Sahip	Karar
Veri kopyalarının kayıt defteri (Bölüm 13)	Her kopyanın hâlâ bir sahibi ve gelecekte bir bitiş tarihi var mı?	<yes / no, which>	<name>	<keep / end by date>
JIT isteklerini kimin onaylayabileceği	Her onaylayan hâlâ bunu haklı çıkaran bir rolde mi?	<yes / no, which>	<name>	<keep / remove>
Audit store'u kimin okuyabileceği	Her okuyucunun buna hâlâ ihtiyacı var mı?	<yes / no, which>	<name>	<keep / remove>
Otomasyon principal'ları (job'lar, runbook'lar, AI agent'lar)	Her biri hâlâ bir workload identity'si ve salt okunur varsayılanla çalışıyor mu?	<yes / no, which>	<name>	<keep / retire>

Kesim 3: dönemdeki break-glass kullanımları

İstek id'si	Kullanan	Olay ya da gerekçe	Review eden	Review tarihi	Sistem hakkındaki bulgu
<id>	<principal>	<reason>	<other than user>	<date or MISSING>	<for example: missing path>

Review tarihi olmayan bir satır, kendi başına bir bulgudur.

Onay kuralları

```
function closeAccessReview(review):
  for each row in review.rows:
    // a row without a decision is not a review, it is a form
    if row.decision is null or row.decidedBy is null:
      raise ReviewIncomplete
    if row.decidedBy == row.owner:
      raise ReviewerIsOwner
  // decisions become tracked work with an owner and an end date
  for each row in review.rows where row.decision != "keep":
    createChange(row, owner: row.owner, endsOn:
review.nextCycleStart)
markClosed(review.id)
```

F. Olay Sonrası Review Şablonu

Bu şablon Bölüm 15'i izler. Suçlayıcı değildir: her soru, bir eyleme izin veren ya da onu görünmez kılan sistem hakkındadır ve hiçbirisi kimin daha iyi bilmesi gerektiğini sormaz. Hafızadan değil, kayıttan doldurun. Kayıt bir soruyu cevaplayamıyorsa bunu yazın, çünkü boşluk bir bulgudur. Zamanlar, kaydın gösterdiği zamanlardır; tahmin etmeyin.

1. Özet

```
Incident id:          <id carried in every request reason>
Opened / closed:     <timestamps from the incident record>
Trigger:             <alert, report or declared exposure>
Principal(s) in scope: <logins, services, automation principals>
Pillars touched:     <authorization / execution / accountability>
```

2. Sırasıyla sınırlama

Adım	Tamamlanma zamanı	Sonuç	Yolda neyin durduğuna dair notlar
Grant'leri revoke et	<timestamp>	done / failed	
Session'ları kapat	<timestamp>	done / failed	
Credential'ları rotate et	<timestamp>	done / failed	
Hasarın kapsamını audit log'dan çıkar	<timestamp>	done / failed	
Kaydet ve bildir	<timestamp>	done / failed	

Sorular: Önceki adımlardan biri başarısız olsa da her adım çalıştı mı? Tamamlanmamış kalan adımlar oldu mu ve izlendiler mi? Log'a ulaşılabilir miydi; değilse girdiler, erişimin kaldırılmasını geciktirmeden kuyruğa alındı mı?

3. Blast radius

```
Query run:          <everything the principal touched since T,
Restricted rows per table>
Tables touched:     <list, with class>
Restricted columns: <touched or not, rows returned per table>
Agreement check:    <does the scope agree with the register of copies
and the gateway session list?>
```

4. Olay erişiminin yaşam döngüsü

İstek id'si	Tür	İstendi	Grant edildi	İlk kullanım	Revoke edildi	Review eden
<id>	jit / breakglass	<time>	<time>	<time or none>	<time>	<other than user, or MISSING>

Sorular: Herhangi bir grant `expires_at` değerinden sonra kaldı mı? Hiçbir isteğin açıklamadığı bir erişim grant edildi mi? `closeIncident` açık erişim buldu mu ve olaydan önce kapatıldı mı?

5. Olay sırasındaki yazmalar

Yazma	Güzergâh (hızlandırılmış review'lu runner, ürün yolu, ham break-glass)	Yazar olmayan reviewer	Approved hash eşleşti
<description>	<route>	<name>	evet / hayır

Soru: Runner'ın işe yarayacağı bir yazma için ham break-glass kullanıldı mı ve neden?

6. Ertelenen durumlar, olduysa

- **Gateway kullanılamıyor:** insan erişiminin ne kadar süre durduğu, servislerin çalışmaya devam edip etmediği ve restore runbook'unun yazıldığı gibi işleyip işlemediği.
- **Olay sırasında restore:** restore edilen kopyanın bir sahip ve bir bitiş tarihiyle kaydedilip kaydedilmediği ve sonlandırılıp sonlandırılmadığı.
- **Geceleri yavaş onaylar:** hangi isteğin, kimi beklediği ve cevaplanmamış istek kuralının ne yaptığı.

7. Bulgular ve takip

Sistem hakkındaki bulgu	Parça (Kimlik, Grant, Yol, Yürütme, Kayıt)	Değişiklik	Sahip	Bitiş tarihi
<for example: a path that should have existed>	<part>	<change>	<name>	<date>

Tek bir kontrolle bitirin: her bulgu, sahibi ve bitiş tarihi olan bir tasarım değişikliği mi, yoksa daha dikkatli olunması gerektiğine dair bir hatırlatma mı?

KİTAP HAKKINDA

Production veritabanı erişimi bir credential dağıtım problemi değildir. Bir yetkilendirme, yürütme ve hesap verebilirlik problemidir.

Daha iyi bir parola kasası yalnız sırrın kişiye nasıl ulaştığını değiştirir; o kişinin ne yapabileceğine, veriye karşı gerçekte neyin çalıştığına ve sonradan hesabı kimin vereceğine karar vermez. Bu kitap Ledgerly'yi paylaşılan bir admin parolasından bilinçle tasarlanmış bir erişim modeline taşır: en az yetkili kimlikler, just-in-time ve break-glass erişim, tehlikeli işlemler için aracılı yürütme ve kimin neyi neden yaptığını söyleyebilen bir audit izi. Desenler veritabanı motorundan bağımsızdır ve her biri taşıdığı bedelle birlikte anlatılır.

YAZAR HAKKINDA

Muhammet Şafak 2008'den bu yana yazılım geliştiriyor. Uzun yıllar boyunca sağlık, finans ve telekomünikasyon domainlerinde aktif görev aldı. Teknolojiye değil, probleme odaklanarak geliştirme yapılması gerektiğini savunuyor. 2024'ten bu yana yüksek trafikli sistemleri uçtan uca ölçeklenebilir biçimde tasarlayıp geliştiriyor.

Küçük ve orta ölçekli yazılım geliştirme ekiplerine mentorluk yapıyor. Açık kaynak projeler yürütüyor. Kararlarını gerekçesi ve bedeliyle yazıyor.

Güvenilir sistemler güvenilir parçalardan kurulmaz. Bozulan parçaların etrafına kurulur.

I

ERİŞİM

II

EVENT'LER

III

AGENT'LAR